

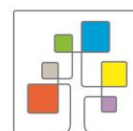


# **JenNet-IP Smart Home Application Note**

JN-AN-1162

v2004

28/01/2015



JenNet-IP



---

## Contents

<b>About this Manual</b>	<b>6</b>
Organisation	6
Conventions	7
Acronyms and Abbreviations	7
Compatibility	8
Related Documents	9
Trademarks	10
Certification	10
<b>1 Introduction</b>	<b>11</b>
<b>2 System Concepts</b>	<b>12</b>
2.1 Gateway System Topology	13
2.1.1 Gateway GUIs	15
2.1.2 Gateway Hardware	20
2.2 Coordinator System Topology	22
2.3 Standalone System Topology	23
2.4 Gateway/Coordinator Failure	24
2.5 MIBs and Variables	24
2.6 Custom Protocols	25
2.7 Identifiers	26
2.7.1 Device ID (32 bits)	26
2.7.2 Device Type IDs (16 bits)	27
2.7.3 MIB IDs (32 bits)	28
2.8 Message Transmission	29
2.8.1 Unicast Messaging	29
2.8.2 Multicast Messaging	29
<b>3 Device Concepts</b>	<b>30</b>
3.1.1 Bulbs	30
3.1.2 Remote Controls	33
3.1.1 Low Energy Switches	34
3.1.2 Sensors	35
<b>4 System Operation</b>	<b>41</b>
4.1 Gateway System Operation	42
4.1.1 Gateway System Operation Overview	43
4.1.2 Setting Up the Gateway System	49
4.1.3 Operating the Bulb Devices	60
4.1.4 Operating the Remote Control	75
4.1.5 Operating the Low Energy Switch	85
4.1.6 Operating the Occupancy Sensor	91
4.1.7 Operating the Illuminance Sensor	102
4.1.8 Co-operating Occupancy and Illuminance Sensors	110
4.1.9 Operating the Combined Occupancy/Illuminance Sensor	118
4.2 Standalone System Operation	123
4.2.1 Standalone System Operation Overview	123
4.2.2 Setting Up the Standalone System	126
4.2.3 Operating the Bulb Devices	129
4.2.4 Global Bulb Control from Remote Control	132
4.2.5 Group Bulb Control from Remote Control	132

<b>5 MIB Variable Reference</b>	<b>133</b>
5.1 Bulb MIBs	134
5.1.1 BulbConfig MIB (0xFFFFFE01)	134
5.1.2 BulbStatus MIB (0xFFFFFE00)	141
5.1.3 BulbScene MIB (0xFFFFFE03)	144
5.1.4 BulbControl MIB (0xFFFFFE04)	148
5.2 Colour MIBs	156
5.2.1 ColourConfig MIB (0xFFFFFE09)	156
5.2.2 ColourControl MIB (0xFFFFFE0C)	161
5.3 Sensor MIBs	176
5.3.1 OccupancyConfig MIB (0xFFFFFE31)	176
5.3.2 OccupancyStatus MIB (0xFFFFFE30)	184
5.3.3 OccupancyControl MIB (0xFFFFFE34)	186
5.3.4 IlluminanceConfig MIB (0xFFFFFE39)	187
5.3.5 IlluminanceStatus MIB (0xFFFFFE38)	191
5.3.6 IlluminanceScene MIB (0xFFFFFE3B)	196
5.3.7 IlluminanceControl MIB (0xFFFFFE3C)	200
5.3.8 OccllIBulbConfig MIB (0xFFFFFE3F)	207
5.3.9 OccupancyMonitor MIB (0xFFFFFE32)	211
5.4 Remote MIBs	217
5.4.1 RemoteConfigGroup MIB (0xFFFFFE25)	217
5.5 Device MIBs	219
5.5.1 DeviceConfig MIB (0xFFFFFEA1)	219
5.5.2 DeviceStatus MIB (0xFFFFFEA0)	219
5.5.3 DeviceControl MIB (0xFFFFFEA2)	220
5.5.4 DeviceScene MIB (0xFFFFFEA3)	222
<b>6 Software Reference</b>	<b>228</b>
6.1 Standard Device Software Features	229
6.1.1 Standard Device <i>Type</i> Folder Features	230
6.1.2 Common Module Features	242
6.1.3 Standard MIB Module Features	243
6.2 DeviceBulb Folder	246
6.2.1 DeviceBulb Makefile	247
6.2.2 DeviceDefs.h	248
6.2.3 DeviceBulb.c	249
6.2.4 DeviceScene MIB	251
6.2.5 BulbScene MIB	253
6.3 MibBulb Folder	254
6.3.1 BulbConfig MIB	254
6.3.2 BulbStatus MIB	255
6.3.3 BulbControl and Device Control MIBs	256
6.4 MibColour Folder	262
6.4.1 ColourConfig MIB	262
6.4.2 ColourControl MIB	263
6.4.3 Colour Modules	267
6.5 DriverBulb Folder	275
6.5.1 DriverBulb.h, DriverBulb_ <i>Type</i> .c	275
6.6 DeviceSensor Folder	278
6.6.1 DeviceSensor Makefile	279
6.6.2 DeviceDefs.h	280
6.6.3 DeviceSensor.c	281
6.7 MibSensor Folder	283
6.7.1 OccupancyConfig MIB	283

6.7.2 OccupancyStatus MIB	283
6.7.3 OccupancyControl MIB	285
6.7.4 IlluminanceConfig MIB	286
6.7.5 IlluminanceStatus MIB	286
6.7.6 IlluminanceControl MIB	288
6.7.7 IlluminanceScene MIB	289
6.7.8 OccIIIbBulbConfig MIB	290
6.7.9 OccupancyMonitor MIB	292
6.8 DriverSensor Folder	293
6.8.1 DriverOccupancy.h, DriverOccupancy_Type.c	293
6.8.2 DriverIlluminance.h, DriverIlluminance_Type.c	294
6.9 DeviceRemote Folder	296
6.9.1 DeviceRemote Makefile	296
6.9.2 RemoteDefault.h	297
6.9.3 DeviceRemote.c	297
6.9.4 DriverCapTouch.h, DriverCapTouch.c, DriverCapTouch_DIO.c	303
6.9.5 DriverLed.h, DriverLed.c	304
6.9.6 Key.h, Key.c	304
6.9.7 Mib.h, Mib.c	305
6.9.8 ModeCommission.h, ModeCommission.c	307
6.9.9 JipCallbacks.c	307
6.10 MibRemote Folder	308
6.10.1 RemoteConfigGroup MIB	308
6.11 LowEnergySwitch Folder	310
6.11.1 LowEnergySwitch Makefile	311
6.11.2 LowEnergySwitch.c	311
<b>Appendices</b>	<b>314</b>
A Revision History – JN-SW-4141 Toolchain	314
B Revision History – JN-SW-4041 Toolchain	318

---

## About this Manual

This manual provides information about the *JenNet-IP Smart Home Application Note (JN-AN-1162)*. 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	<a href="#">Colour Controlled Temperature</a>
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 <a href="#">CIE xyY colour space</a> .
xyY	Colour representation in the <a href="#">CIE xyY colour space</a> .
XYZ	Colour representation in the <a href="#">CIE XYZ colour space</a> .

## 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 as 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](http://www.JenNet-IP.com)**.

---

## 1 Introduction

This Application Note provides software for a Smart Home network allowing the control and monitoring of Smart Devices via standard Internet Protocol messages over a low power radio network.

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

The Application Note includes software for the following Smart Devices:

- Bulb software allowing the bulb to be turned on, off and dimmed remotely. Plus monitoring of on-time and counter for bulbs being turned on. White, colour controlled temperature and full colour bulbs are all supported.
- Sensor software allowing the monitoring of occupancy and/or illuminance with optional control of bulb devices.
- Remote control providing control of Smart Devices from a small touch sensitive device.
- Low energy switch providing limited control of Smart Devices from a coin cell or energy harvesting device.

## 2 System Concepts

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

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

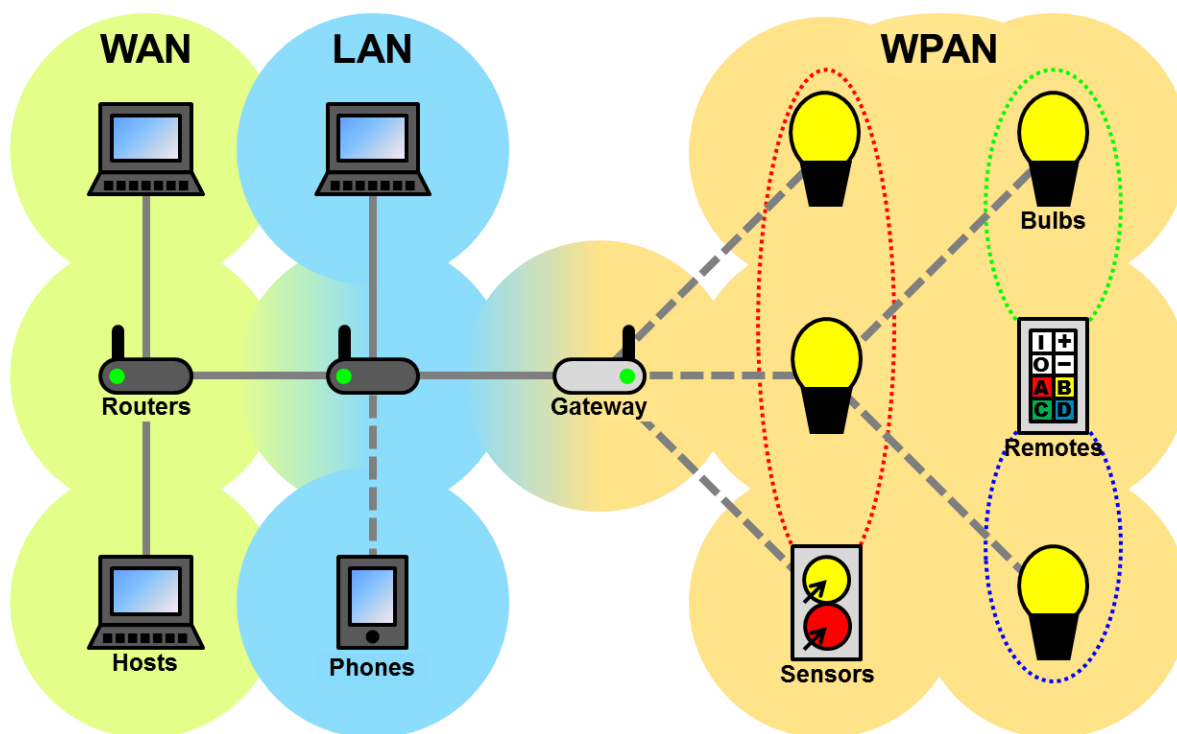
- Gateway Mode includes a gateway device allowing access to the low-power wireless Smart Devices from other Internet Protocol devices connected via the local IP network, Wi-Fi or even from the external internet. Smart Devices can also be controlled by other Smart Devices within the low power wireless 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

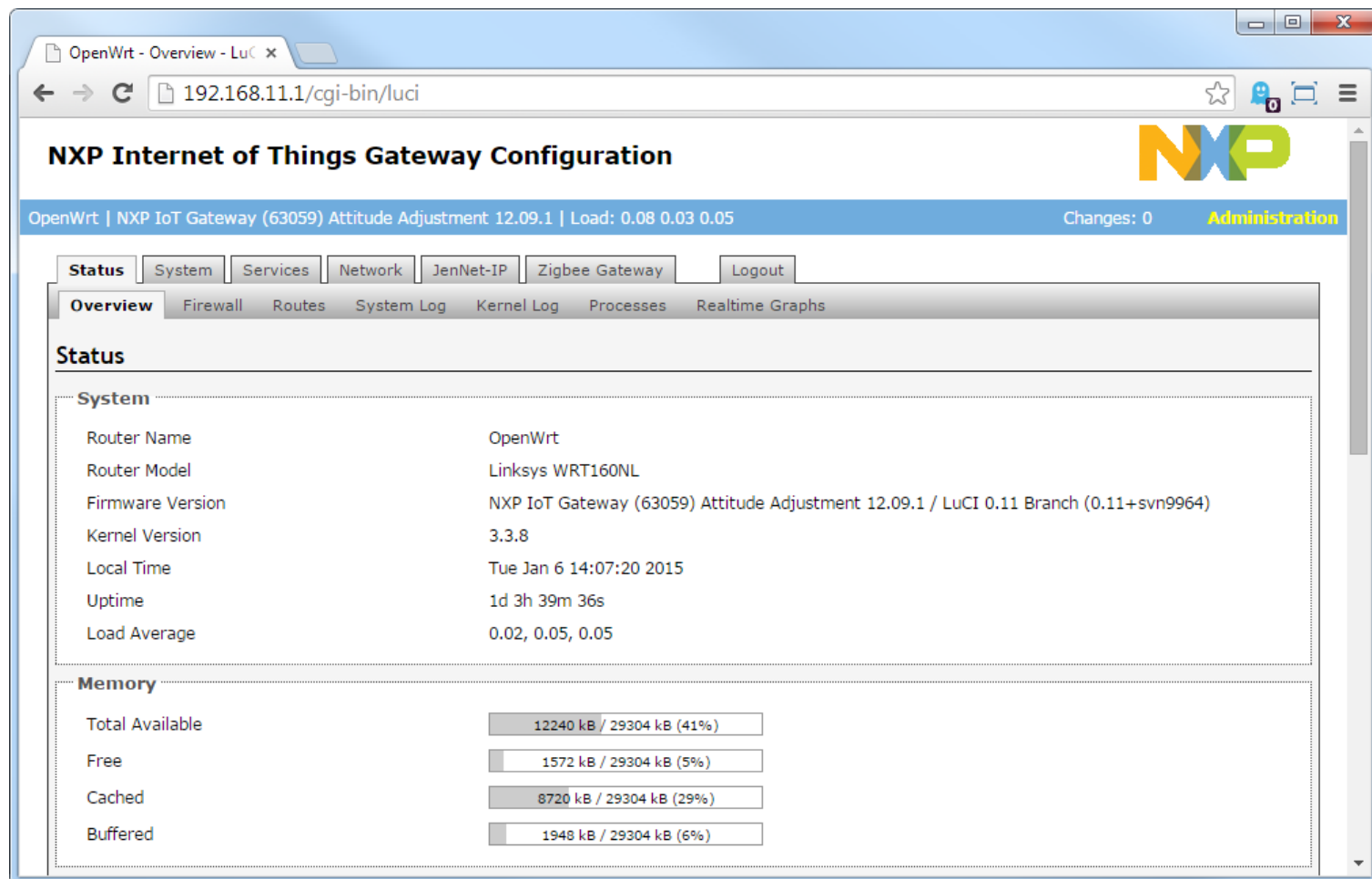
The gateway provides a number of GUIs, served as webpages to allow administration, monitoring and control of the Smart Devices.

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.



OpenWrt - Overview - LuCI x

192.168.11.1/cgi-bin/luci

## NXP Internet of Things Gateway Configuration

OpenWrt | NXP IoT Gateway (63059) Attitude Adjustment 12.09.1 | Load: 0.08 0.03 0.05 Changes: 0 Administration

**Status** System Services Network JenNet-IP Zigbee Gateway Logout

**Overview** Firewall Routes System Log Kernel Log Processes Realtime Graphs

### Status

#### System

Router Name	OpenWrt
Router Model	Linksys WRT160NL
Firmware Version	NXP IoT Gateway (63059) Attitude Adjustment 12.09.1 / LuCI 0.11 Branch (0.11+svn9964)
Kernel Version	3.3.8
Local Time	Tue Jan 6 14:07:20 2015
Uptime	1d 3h 39m 36s
Load Average	0.02, 0.05, 0.05

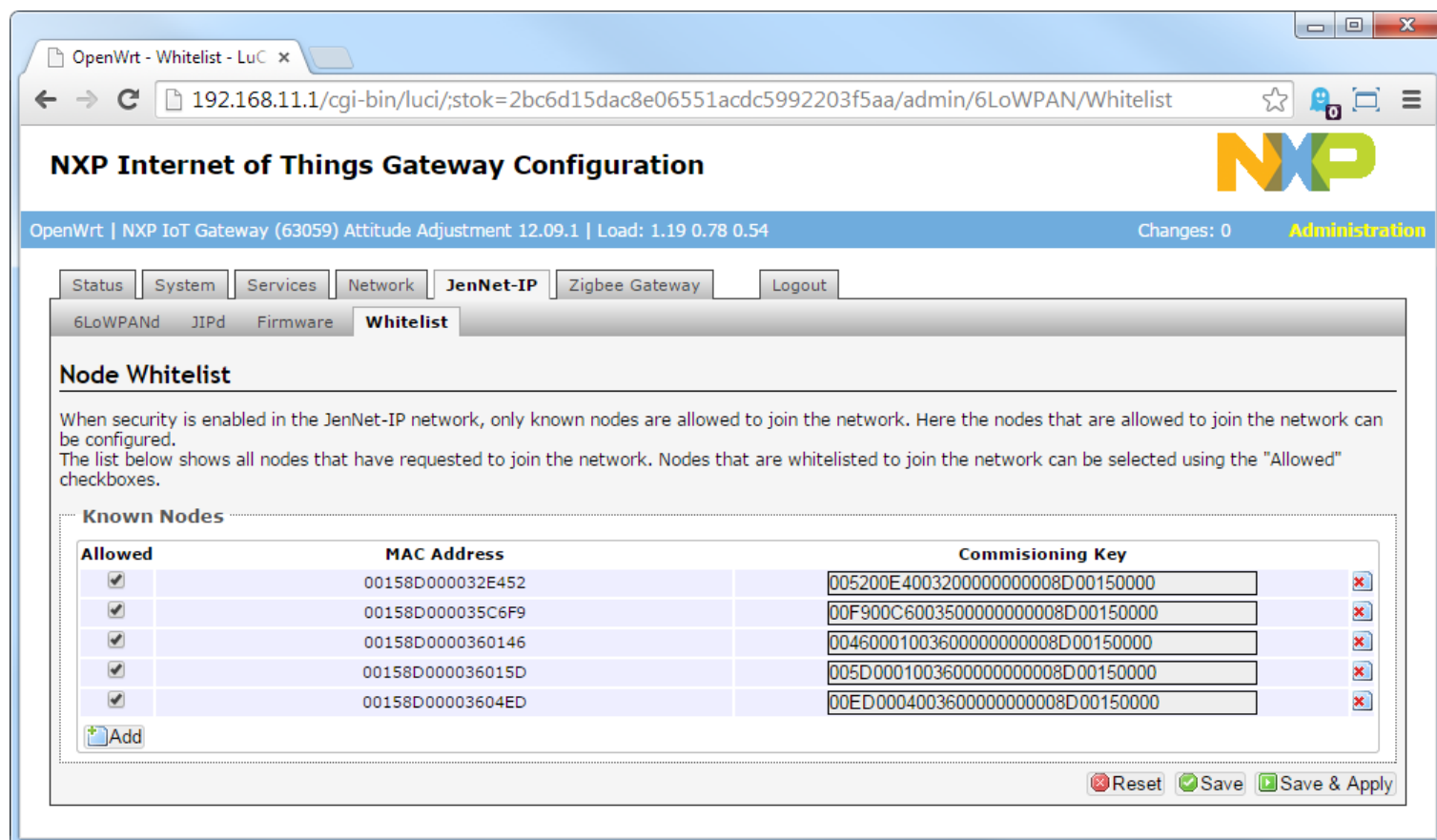
#### Memory

Total Available	12240 kB / 29304 kB (41%)
Free	1572 kB / 29304 kB (5%)
Cached	8720 kB / 29304 kB (29%)
Buffered	1948 kB / 29304 kB (6%)



## JenNet-IP Smart Home Application Note

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.

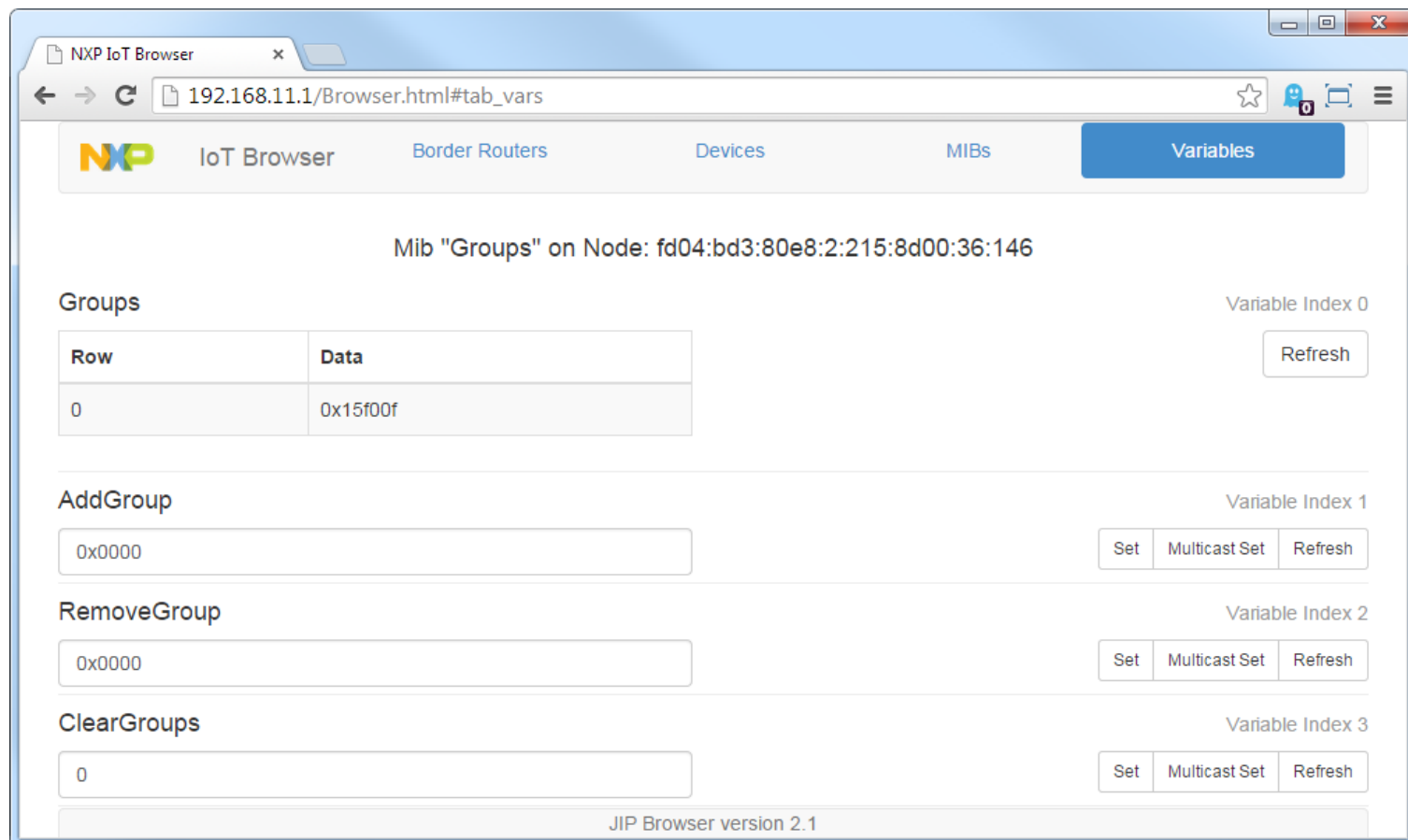


The white list includes the MAC addresses of the devices allowed into the network and a commissioning key for each device. The commissioning key in the white list must match the key programmed into the device attempting to join the network or it will not be able to join.

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

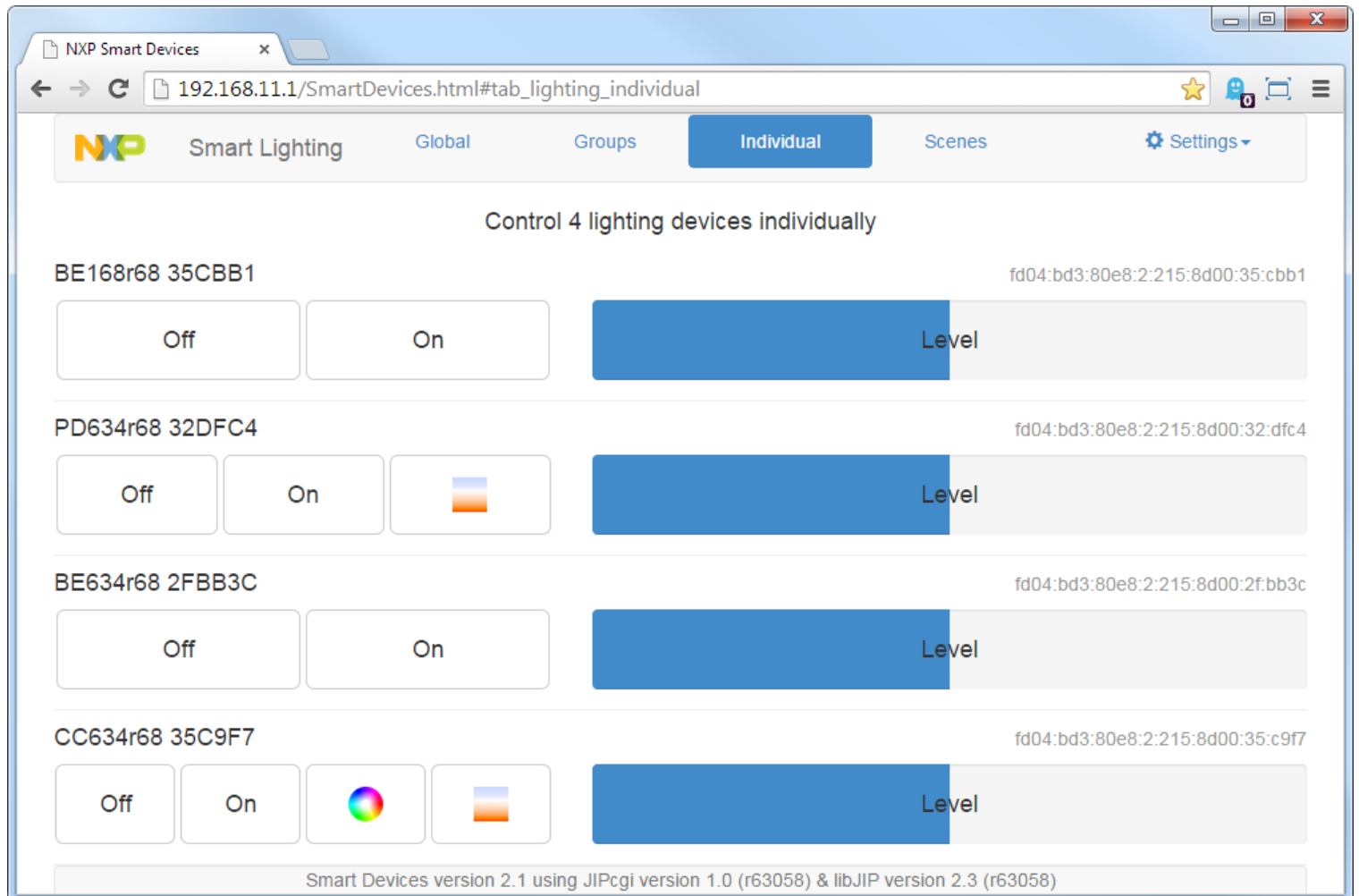
The gateway contains a generic engineering JenNet-IP Browser interface. This allows the devices in the WPAN to be discovered. The MIBs and variables in each device can be accessed allowing them to be viewed and edited.



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)*:

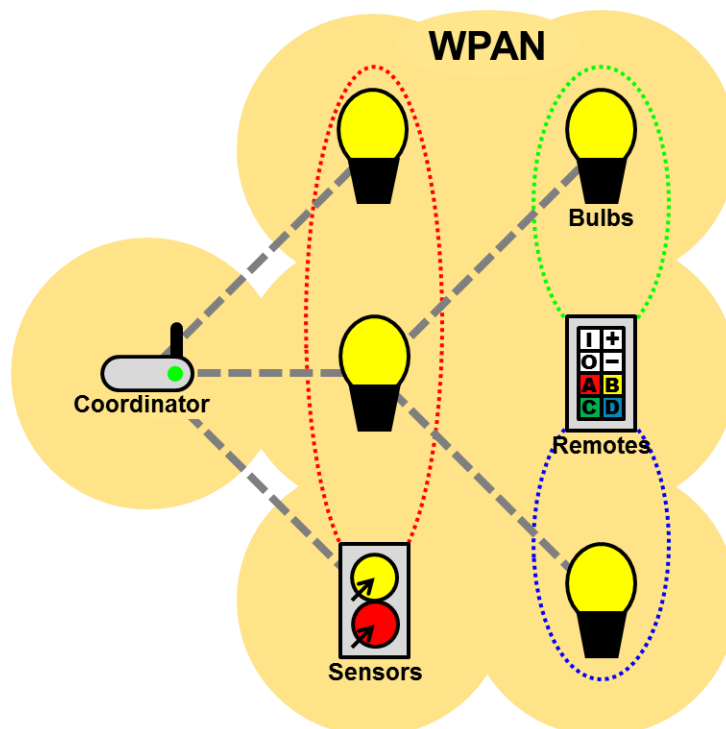
- The internet router is a standard off-the-shelf router running the manufacturer's stock firmware.
- The border router host and node are combined in a single device with the host OpenWRT firmware running on an LPC3240 microcontroller and the node firmware running on a JN5168. The border router device 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)*:



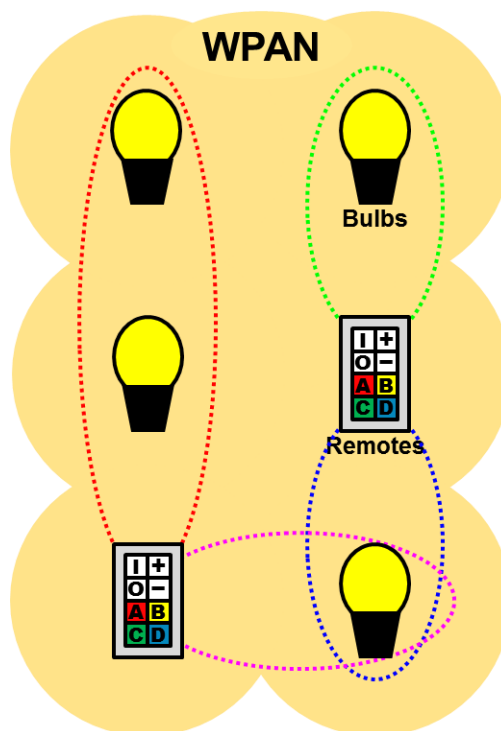
This topology replaces the gateway with a Coordinator device that does not allow connection to an existing IP network. The Coordinator device independently creates the network for the other devices to join and accepts any device attempting to join its network.

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.



**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 be the source or destination of 6LoWPAN level messages it will be necessary to adapt the border router firmware. However, it will router 6LoWPAN messages for other devices in the network unaltered.

---

## 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.

---

### 3.1.1 Bulbs

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.

Three types of bulbs are implemented in the Application Note:

1. Dimmable white bulbs, these bulbs can be turned on and off and also have their brightness levels adjusted.
2. Colour controlled temperature (CCT) bulbs, include the features of dimmable white bulbs but can also have their colour temperature adjusted.
3. Colour bulbs, include the features of CCT bulbs but can also have their colour fully adjusted.

---

#### 3.1.1.1 Basic Control

Bulbs can be turned on or off and also have their brightness level set. Additionally the colour temperature and colour can be set on bulbs that support such features.

A variety of methods are provided for controlling the brightness and colour of the bulbs, these allow different interfaces to control these elements in different ways by writing to different MIB variables:

- Explicit level: The values can be set to an explicit level. This is the method used by the gateway's Smart Devices GUI and sets the level based upon the position clicked in the brightness bar. Target explicit levels can be set which the bulb transitions to over time or a current explicit level can be set which the bulb changes to instantly.
- Change over time: The bulbs can be placed into a mode where the value is increased or decreased over time, when the desired level is reached the mode can be cleared to leave the bulbs at the set brightness. This is the method used by the remote control, touching the up or down button places the bulb into the up or down mode, releasing the button takes the bulb out of the up or down mode thus setting the desired level.
- Change by value: The bulbs can have their current values altered up or down by a fixed amount. This method of control is not used in the Application Note but would be useful for devices featuring jog controls where each jog alters the level by a set amount.

---

### 3.1.1.2 Group Control

Bulbs can be placed into groups, when a command is broadcast to a group all the bulbs in that group will carry out the command allowing many bulbs to be turned on or off and their levels changed together. The remote control always uses this method to control bulbs while the Smart Devices GUI provides group control for a limited number of groups.

All bulbs are placed into “All Devices” and “All Bulbs” groups by default, (though the user may choose to remove them later). The JenNet-IP Browser GUI can be used to place bulbs into additional groups.

Each bulb can be a member of up to 16 groups in the default configuration provided in this Application Note.

Group control is most useful where many devices need to have their states set to the same value activated by a single broadcast command. Any MIB variable can be controlled this way. Group control may be used with almost all other JenNet-IP devices as it is a feature provided by the JenNet-IP stack.

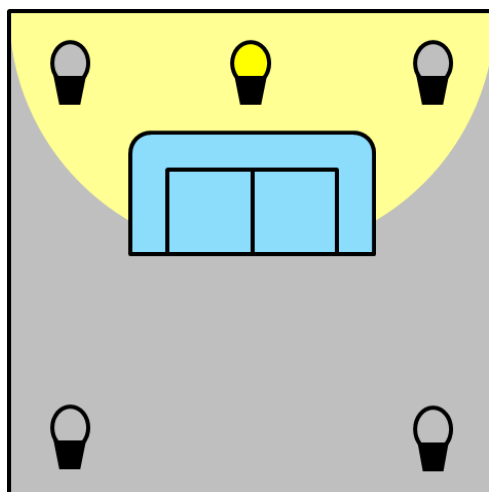
---

### 3.1.1.3 Scene Control

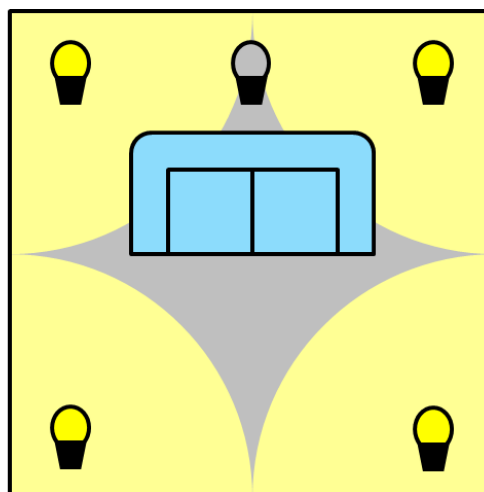
Bulbs can also be placed into scenes using one of two methods:

1. Using the current settings: First the state and levels of the bulbs taking part in the scene should be configured, different bulbs may have different settings. The bulbs are then issued with a command to place themselves into a specific scene and store their current settings for use when the scene is activated.
2. Using specific settings: A single command can be issued to place the bulb into a specific scene with the state and levels specified as part of the command. The advantage of this method is that the current settings are not altered in order to create the scene.

To activate a scene a command is transmitted to all bulbs, the bulbs taking part in the scene will then apply their configuration for the scene. This allows many bulbs to be placed into different states using a single broadcast command, for example a “reading” scene may set a reading light to full brightness while dimming all other bulbs in a room to minimum brightness. The Smart Devices GUI provides controls to activate a limited number of scenes, (but no controls to configure scenes – the JenNet-IP Browser must be used for scene configuration).



**Reading Scene**



**General Scene**

Each bulb can participate in up to 8 scenes in the default configuration provided in this Application Note.

Scene control is most useful where many devices need to have their states set to different values activated by a single broadcast command. Only the MIB variables that form the state of the bulb can be controlled this way. Scene control can only be used with JenNet-IP devices that have implemented scene support as it is a feature provided by the application.



### 3.1.2 Remote Controls

Remote controls, allow control of other Smart Devices in the low power JenNet-IP wireless network. These devices operate as sleeping broadcaster devices in order to allow mobile operation and preserve battery life. To do this they spend most of their time asleep preserving power and only waking to read button inputs.

Commands are always broadcast to a group of devices so the remote does not need to maintain a full connection to the network, this allows the remote to be freely moved around the area covered by the WPAN.

#### 3.1.2.1 Bulb Control

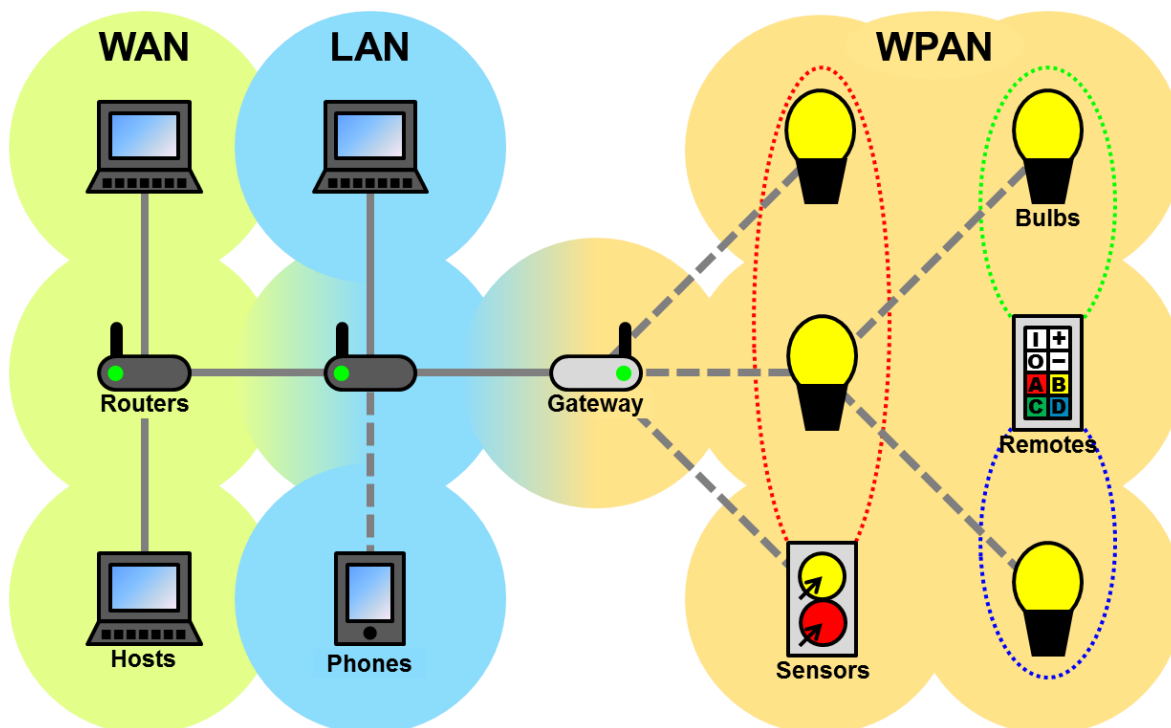
The remote control allows bulbs to be turned on or off and to alter their brightness and colour.

#### 3.1.2.2 Group Configuration

The remote control is able to broadcast its commands to the “All Bulbs” group, (which bulbs are placed into by default on start-up).

The remote control can also broadcast to four other groups unique to each remote control. A key sequence on the remote control can be used to add and remove bulbs that are within range of a low power transmission into and out of its groups.

The blue and green dotted ovals attached to the remote control in the gateway system topology diagram (below) represent two of the remote control's groups each containing a single bulb.



---

### 3.1.1 Low Energy Switches

Low energy switches, allow control of other Smart Devices in the low power JenNet-IP wireless network. These devices operate as broadcasting devices consuming minimal amounts of power using coin cells or energy harvesting devices as a power source.

Commands are always broadcast to a group of devices. The low energy switches do not join the network, instead the devices in the network must be enabled to receive commands from specific low energy switches and placed into the unique group each low energy switch transmits to.

Low energy switches are only able to operate in a gateway system as the gateway is responsible for configuring other devices to receive commands from the low energy switches.

Low energy switches are configured to operate on only a single channel, (as they cannot scan for a network), the devices to be controlled by the switch must therefore be in a network on the same channel.

---

#### 3.1.1.1 Bulb Control

The low energy switch allows bulbs to be turned on or off using the provided software.

The software may be extended to allow control of the bulb brightness and colour where additional inputs on the switch can be used to select an appropriate operation.

---

### 3.1.2 Sensors

Sensors, allow control of other Smart Devices in the low power JenNet-IP wireless network. These devices operate as either Router nodes extending the network for other Smart Devices to join or End Devices sleeping to preserve power and able to run on batteries.

The readings taken by the sensors can be monitored by other devices within the system. The sensors can also be configured to control other devices, such as bulbs within the network.

The following sensor types are available in the Application Note:

---

#### 3.1.2.1 Occupancy Sensors

Occupancy sensors detect whether a room or area is occupied using Passive Infra-Red (PIR) or other methods of detection. The occupancy state of the sensor can be monitored remotely.

##### Bulb Control from Occupancy Sensors

The default configuration of the occupancy sensors allows them to broadcast group commands to control bulbs based upon the occupancy state.

- When the state is unoccupied bulbs are turned off.
- When the state is occupied bulbs are turned on at full brightness.

While the occupancy state remains unchanged the occupancy sensor regularly re-sends its last command in order to bring any bulbs that get powered on back under the sensor's control.

The group the bulb control commands are broadcast to is unique to each occupancy sensor, in order to control a bulb from an occupancy sensor it simply needs to be added to the group the sensor transmits to. It is useful to remove a bulb being controlled in this way from any other groups, (including the "All Devices" and "All Bulbs" groups), to avoid different devices conflicting with each other over the state of the bulb they are both trying to control.

##### Group Control

Occupancy sensors can be placed into groups, when a command is broadcast to a group all the occupancy sensors in that group will carry out the command allowing many occupancy sensors to be controlled together.

All occupancy sensors are placed into "All Devices" and "All Occupancy Sensors" groups by default, (though the user may choose to remove them later). The JenNet-IP Browser GUI can be used to place occupancy sensors into additional groups.

Each occupancy sensor can be a member of up to 16 groups in the default configuration provided in this Application Note.

Group control is most useful where many devices need to have their states set the same value activated by a single broadcast command. Any MIB variable can be controlled this way. Group control may be used with almost all other JenNet-IP devices as it is a feature provided by the JenNet-IP stack.

### 3.1.2.2 Illuminance Sensors

Illuminance sensors detect the amount of light in a room or area using a Photo-Diode or other methods of detection. The illuminance state of the sensor can be monitored remotely.

#### Bulb Control from Illuminance Sensors

The default configuration of the illuminance sensors allows them to broadcast group commands to control bulbs based upon the illuminance state in order to bring the measured illuminance within a target band:

- When the illuminance is below the target band bulbs are turned on and brightened over time until the measured illuminance is within the target band.
- When the illuminance is above the target band the bulbs are dimmed over time and turned off if the measured illuminance is still too bright at the minimum brightness or until the measured illuminance is within the target band.

While the illuminance state remains within the target band the illuminance sensor regularly re-sends it last command in order to bring any bulbs that get powered on back under the sensor's control.

The group the bulb control commands are broadcast to is unique to each illuminance sensor, in order to control a bulb from an illuminance sensor it simply needs to be added to the group the sensor transmits to. It is useful to remove a bulb being controlled in this way from any other groups, (including the "All Devices" and "All Bulbs" groups), to avoid different devices conflicting with each other over the state of the bulb they are both trying to control.

#### Illuminance Sensor Control

The target band of the illuminance sensor can be controlled in a similar way to the brightness of bulbs. A variety of methods are provided for this:

- **Explicit values:** Explicit values can be set for the LuxTarget variable, (which sets the middle point of the target band), and the LuxBand variable, (which sets the width of the target band). This method would be most useful when setting the values using slider controls.
- **Change over time:** The illuminance sensor can be placed into different modes that allow the LuxTarget variable to be moved up or down over time and the LuxWidth variable to be widened or narrowed over time. Exiting these modes will cause the illuminance sensor to stop altering the affected variable. This method would be most useful when changing a value on a button press and stopping the change on release, however due to the latency in the bulbs reaching the new target band it provides little feedback to users when the level is set to the required value.
- **Change by value:** The illuminance sensor can have the current LuxTarget variable raised or lowered by a fixed amount and the LuxWidth variable widened or narrowed by a fixed amount. This method would be most useful for devices featuring jog controls where each jog alters the target band by a set amount.

## **Group Control**

Illuminance sensors can be placed into groups, when a command is broadcast to a group all the illuminance sensors in that group will carry out the command allowing many illuminance sensors to be controlled together.

All illuminance sensors are placed into the “All Devices” and “All Illuminance Sensors” groups by default, (though the user may choose to remove them later). The JenNet-IP Browser GUI can be used to place illuminance sensors into additional groups.

Each illuminance sensor can be a member of up to 16 groups in the default configuration provided in this Application Note.

Group control is most useful where many devices need to have their states set the same value activated by a single broadcast command. Any MIB variable can be controlled this way. Group control may be used with almost all other JenNet-IP devices as it is a feature provided by the JenNet-IP stack.

## **Illuminance Sensor Scene Control**

Illuminance sensors can also be placed into scenes. First the state and target band of the illuminance sensors taking part in the scene should be configured, different illuminance sensors may have different settings. The illuminance sensors are then issued with a command to place themselves into a specific scene and store their current settings for use when the scene is activated. Alternatively an illuminance sensor may be placed directly into a scene along with the scene’s settings in with a single command, this method avoids having to alter the current settings to configure a scene.

To activate a scene a command is transmitted to all illuminance sensors, the illuminance sensors taking part in the scene will then apply their configuration for the scene. This allows many illuminance sensors to be placed into different states using a single broadcast command, for example a “night” scene may set all sensors to maintain a certain level of illumination while an “emergency” scene may set all sensors to enforce maximum illuminance.

Each illuminance sensor can participate in up to 16 scenes in the default configuration provided in this Application Note.

Scene control is most useful where many devices need to have their states set to different values activated by a single broadcast command. Only the MIB variables that form the state of the illuminance sensor can be controlled this way. Scene control can only be used with JenNet-IP devices that have implemented scene support as it is a feature provided by the application.

---

### 3.1.2.3 Occupancy/Illuminance Sensors

Occupancy/illumination sensors combine an occupancy sensor and illumination sensor in a single device. They include the MIBs from both the occupancy sensor and illumination sensor devices and so provide almost identical functionality.

#### Bulb Control from Occupancy/Illuminance Sensors

The default configuration of the occupancy/illumination sensors allows them to broadcast group commands to control bulbs based upon the both the occupancy and illumination state combined:

- When the occupancy state is unoccupied bulbs are turned off.
- When the occupancy state is occupied bulbs are controlled to bring the measured illumination within the target band in the same way as the Illuminance Sensor.

It is possible to configure the combined occupancy/illumination sensor to control bulbs based only on the occupancy state, (in the same way as the occupancy sensor), or only on the illumination state, (in the same way as the illumination sensor).

The group the bulb control commands are broadcast to is unique to each occupancy/illumination sensor, in order to control a bulb from an occupancy/illumination sensor it simply needs to be added to the group the sensor transmits to. It is useful to remove a bulb being controlled in this way from any other groups, (including the “All Devices” and “All Bulbs” groups), to avoid different devices conflicting with each other over the state of the bulb they are both trying to control.

#### Group Control

Occupancy/illumination sensors can be placed into groups, when a command is broadcast to a group all the occupancy/illumination sensors in that group will carry out the command allowing many occupancy/illumination sensors to be controlled together.

All occupancy/illumination sensors are placed into “All Devices”, “All Occupancy Sensors” and “All Illuminance Sensors” groups by default, (though the user may choose to remove them later). The JenNet-IP Browser GUI can be used to place bulbs into additional groups.

Each occupancy/illumination sensor can be a member of up to 16 groups in the default configuration provided in this Application Note.

Group control is most useful where many devices need to have their states set the same value activated by a single broadcast command. Any MIB variable can be controlled this way. Group control may be used with almost all other JenNet-IP devices as it is a feature provided by the JenNet-IP stack.

#### Scene Control

The illumination sensor settings can be stored and activated as a scene in the same way as illumination sensors.

---

### 3.1.2.4 Co-operating Occupancy Sensors

It is possible to configure the system so that multiple occupancy sensors can co-operate to work together.

Each occupancy sensor can be configured to broadcast its occupancy state to other devices in the system. These broadcasts are transmitted to a group address unique to the transmitting sensor and different to the group address used to control bulbs.

Other sensors in the system can be configured to receive the broadcasts from the occupancy sensor configured in this way. The receiving sensors can then take the occupancy state of all the transmitting sensors into account when controlling bulbs. Up to 16 additional occupancy sensors can be monitored in this way on a single device.

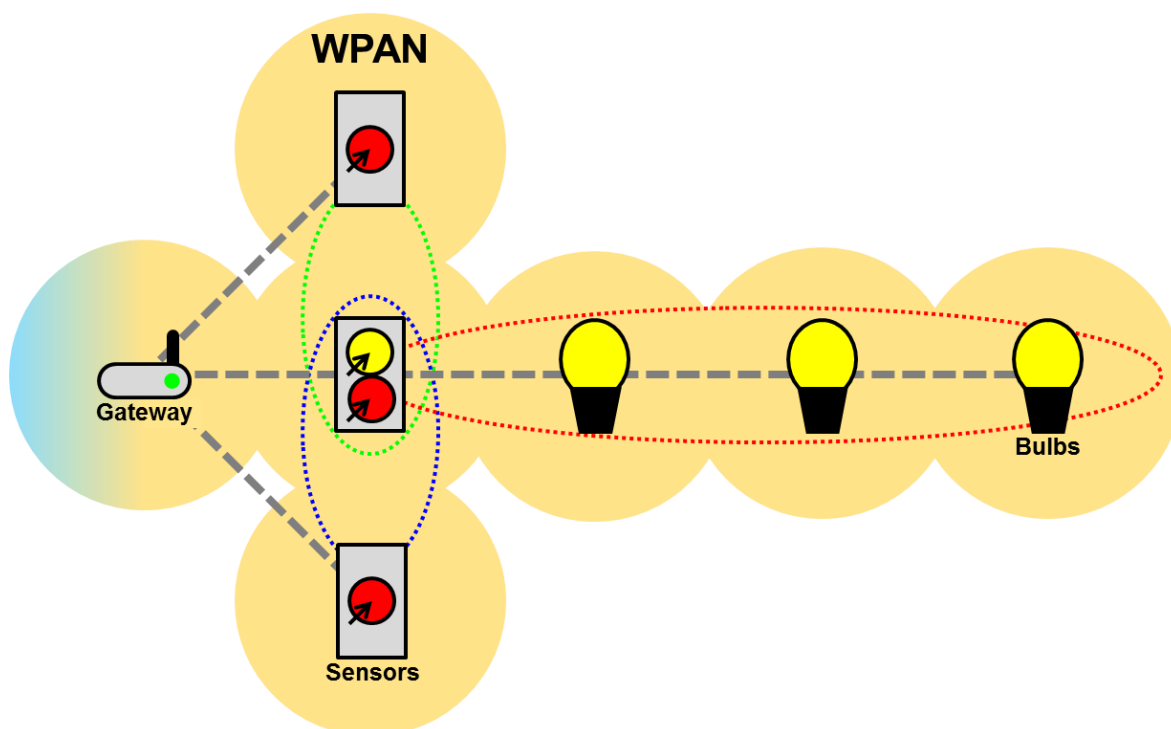
End Devices cannot receive broadcast messages so this system works best when the device receiving the occupancy status messages is a router within the WPAN. The broadcast address used by the transmitting occupancy sensors can be altered to be the unicast address of the receiving device allowing the receiver to operate as an end device, however this introduces latency into the system, as there will be a delay between the transmission of the occupancy status and the receiving device waking from sleep to receive the message. Using unicast messages in this way also removes the ability for one-to-many transmissions of the occupancy state that the use of broadcasts allows.

When an occupancy sensor is monitoring additional occupancy sensors the bulbs will be turned on when *any* of the occupancy sensors report occupied and the bulbs will be turned off when *all* of the occupancy sensors report unoccupied.

When an illuminance sensor is monitoring additional occupancy sensors it can be configured to take the states of the occupancy sensors into account when controlling bulbs. When *all* the occupancy sensors report unoccupied the bulbs are turned off, when *any* of the occupancy sensors report occupied the bulbs will be controlled to meet the illuminance target band.

When a combined occupancy/illuminance sensor is monitoring additional occupancy sensors the bulbs will be turned off when *all* of the occupancy sensors report unoccupied, the brightness of the bulbs will be controlled to meet the illuminance target band when *any* of the occupancy sensors report occupied.

The diagram below shows two occupancy sensors being monitored by a combined occupancy/illuminance sensor which is controlling three bulbs. The green and blue dotted ovals represent the groups the occupancy sensors are transmitting their status to, with the occupancy/illuminance sensor being a member of these groups. While the red dotted oval represents the group the combined occupancy/illuminance Sensor is transmitting to in order to control the bulbs:





---

## 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 section describes how to use the contents of an evaluation kit to set up and run *JenNet-IP Smart Home (JN-AN-1162)* demonstration using a gateway to allow access to the WPAN from PCs connected to the gateway's LAN. This demonstration is based on a WPAN with nodes containing lights and sensors, which may be monitored and controlled as follows:

- via an IP connection from a remote device on a LAN or WAN (e.g. from a PC)
- wirelessly from a remote control or low energy switch within the WPAN



A 'standalone' version of the *JenNet-IP Smart Home (JN-AN-1162)* demonstration is also available which does not provide IP connectivity, allowing only wireless control of the nodes from a remote control within the WPAN. If you prefer to set up and run the standalone version, go to [Section 4.2 "Standalone System Operation"](#).

---

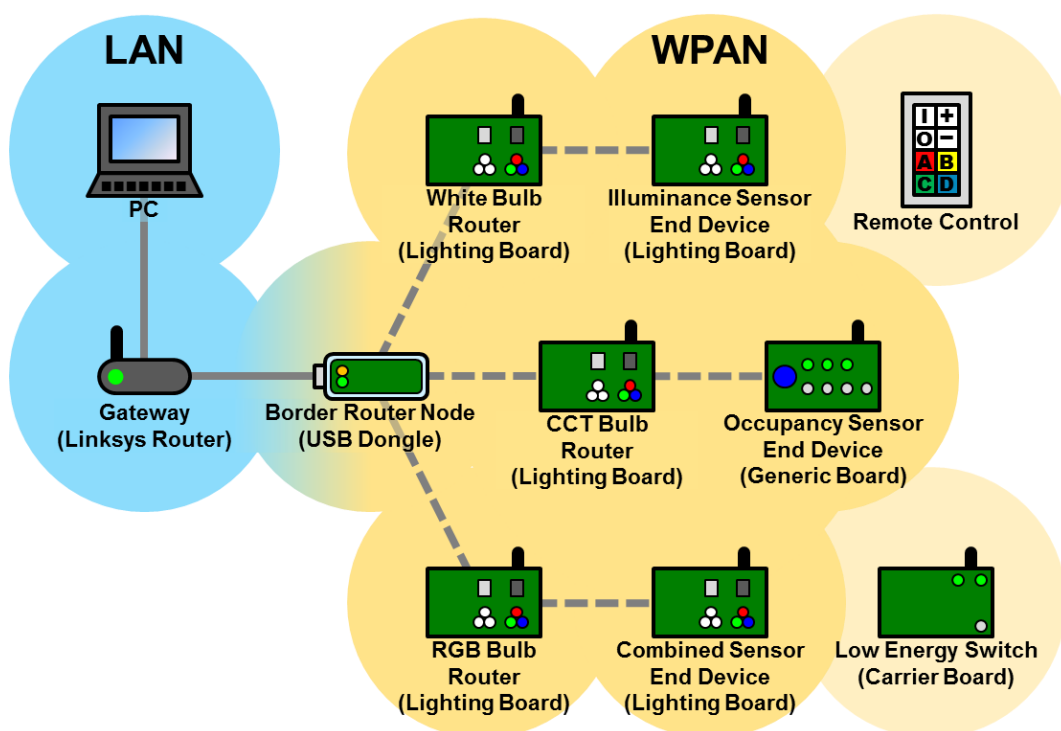
### 4.1.1 Gateway System Operation Overview

In the *JenNet-IP Smart Home (JN-AN-1162)* demonstration, a set of lights and sensors form a WPAN which can be accessed either from a remote control within the WPAN or from a PC located on an Ethernet bus. The components of an evaluation kit are used in the demonstration as follows:

- **Carrier Boards with Lighting/Sensor Expansion Boards:** The four *Carrier Boards (DR1174)* 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. These boards are used to run the following devices:
  - **Bulb device:** The white LEDs or the RGB LED on the *Lighting/Sensor Expansion Boards (DR1175)* or LED2 on the *Generic Expansion Board (DR1174)* are the lights to be controlled.
  - **Occupancy Sensor:** A Parallax Passive Infra-red Module can easily be fitted to the Carrier Board or Lighting/Sensor Expansion Board to act as the occupancy sensor. Alternatively this software can be recompiled to use one of the buttons on the Carrier or Generic Expansion Boards.
  - **Low Energy Switch:** The Carrier Board is used to operate as the Low Energy Switch.
  - **Illuminance Sensor:** The photo-diode on Lighting/Sensor Expansion Boards is used to measure illuminance.
  - **Combined Occupancy/Illuminance Sensor:** A Parallax Passive Infra-red Module can easily be fitted to the Carrier Board or Lighting/Sensor Expansion Board to act as the occupancy sensor (details of this module are provided in [Section 4.1.6.1 "Setting up the Occupancy Sensor"](#)). Alternatively this software can be recompiled to use one of the buttons on the Carrier Board. The photo-diode on Lighting/Sensor Expansion Boards is used to measure illuminance.
- **Remote Control:** The remote control acts as a node of the WPAN. In the demo described in this chapter, the unit initially behaves as a WPAN Router until it has joined the network and then acts as a 'sleeping broadcaster'. In the latter mode, the device sleeps and only wakes when it is needed to broadcast control commands (it does not have the role of a conventional WPAN node). If sleeping, the unit must be activated using the Wake (circle) button below the keypad before any other keys are pressed.

- **USB Dongle:** This demonstration uses one of the supplied USB dongles programmed as a border router 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.
- **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 IP router handles the LAN/WAN side of the interface and connects to the Ethernet bus on which the controlling PC is located.

A complete Smart Home demonstration system is illustrated below, (there are not enough boards in a single evaluation kit to build this system so a subset should be used when working with a single evaluation kit).



#### 4.1.1.1 Lighting 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.

Two web applications are provided on the Linksys router these allow a PC user to monitor and control the lights in the WPAN. These applications run on the border router host and serve web pages to a normal web browser on the PC, allowing the user to interact with the WPAN nodes through the border router. The applications are:

- Smart Devices Demonstration: This application provides a high-level interface for monitoring and controlling the lights in the WPAN, through easy-to-use graphical controls (on-screen buttons and a controls). The application can be accessed by entering the following (case-sensitive) IP address into the web browser: <http://192.168.11.1/SmartDevices.html>
- 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, the Smart Devices Demonstration and the Gateway Configuration interfaces are all provided on the Linksys router. All these interfaces 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/>



### 4.1.1.2 Lighting Control from the Remote Control Unit

In this demonstration, control commands can be entered into the remote control and wirelessly broadcast (in JenNet-IP packets) to the WPAN nodes. A command can be addressed to all nodes or to a pre-defined group of nodes.

A complete list of the operations that can be performed from the keypad is provided in [Section 4.1.4.4 "Remote Control Command Tables"](#). The table below provides a summary of the use of individual keys by the demonstration and the figure below shows the keypad of the remote control.

The group (\*, A, B, C, D) is selected first, all operations (I, O, +, -) are then broadcast to the selected group until a different group is selected. All lights are automatically placed into the All (\*) group further actions must be taken to add lights to the other (A, B, C, D) groups used by the remote control.

The parameter controlled by the up and down (+, -) keys is selected using the numeric key (1, 2, 3, 4) all subsequent uses of the up and down keys will affect that parameter until a different parameter is selected.



Note: In this manual, operations are generally described as function sequences followed by the key sequences in square brackets - for example:

PRG OFF DOWN OFF [# O - O]

Function	Key	Description
Group Selection	*	Select "All Bulbs" group
	A	Select group A
	B	Select group B
	C	Select group C
	D	Select group D
Level Parameter Selection	1	Select brightness parameter
	2	Select saturation parameter
	3	Select hue parameter
	4	Select colour temperature parameter
State Control	I	Switch on light(s)
	O	Switch off light(s)
Level Control	+	Increase level of selected parameter
	-	Decrease level of selected parameter
Program	#	Programming mode
Wake	•	Wake Remote Control Unit from sleep



The remote control normally operates as a 'sleeping broadcaster'. Thus, the unit sleeps until it is needed. If sleeping, the unit can be activated using the Wake (circle) button below the keypad. Once woken, the unit remains active for 10 minutes following the last key press before going back to sleep. When the unit is active, pressing any key will cause the left LED to momentarily illuminate (if this does not happen, you must first activate the unit using the Wake button).

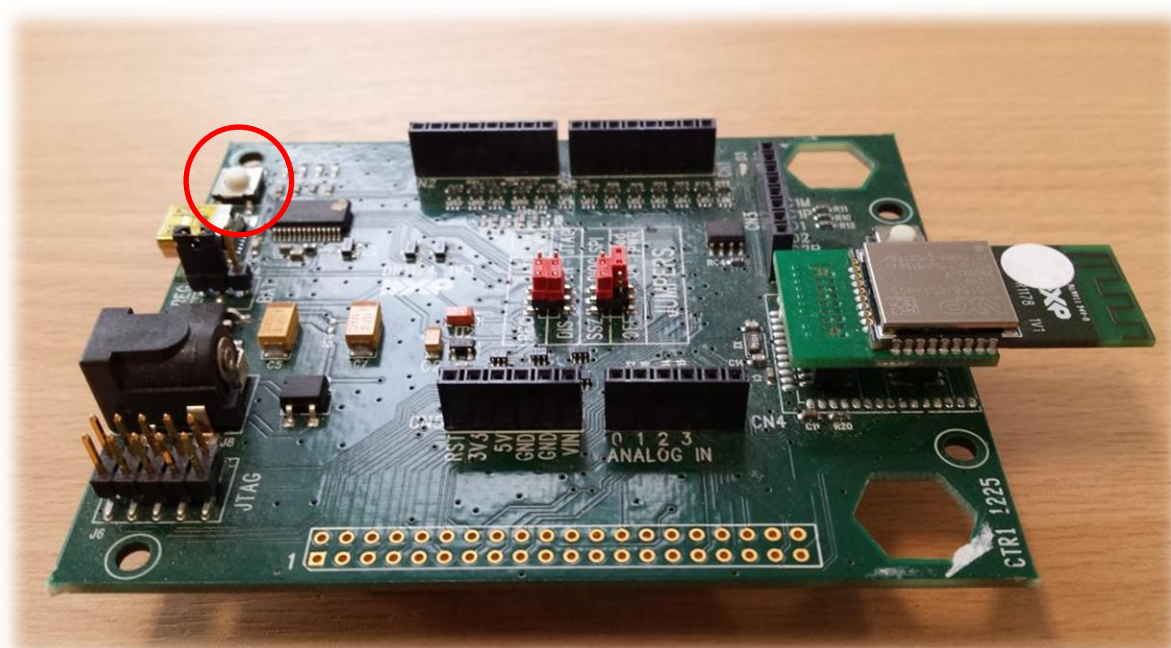


#### 4.1.1.3 Lighting Control from a Low Energy Switch

In this demonstration control commands are broadcast to a pre-defined group of nodes.

The software broadcasts a command each time the software is run and then enters deep sleep mode to preserve power (when running from a coin cell). The command toggles each time the software is run between turning bulbs off and on.

To operate the software on the *Carrier Board (DR1174)* the reset button (marked RST) should be used to run the software and toggle bulbs off and on.





---

## 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.



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)*.



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



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 dongle is not already programmed with the latest border router node firmware it should be updated prior to running the application.



The USB dongles provided in the evaluation kit are pre-loaded with an older version of the border router node 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 border router node firmware.



Instructions on how to connect the USB dongles 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 border router host firmware it should be updated before continuing to run the application.



The Linksys router provided in the evaluation kit is pre-loaded with an older version of the border router host 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 border router host firmware and instructions to update the Linksys router.

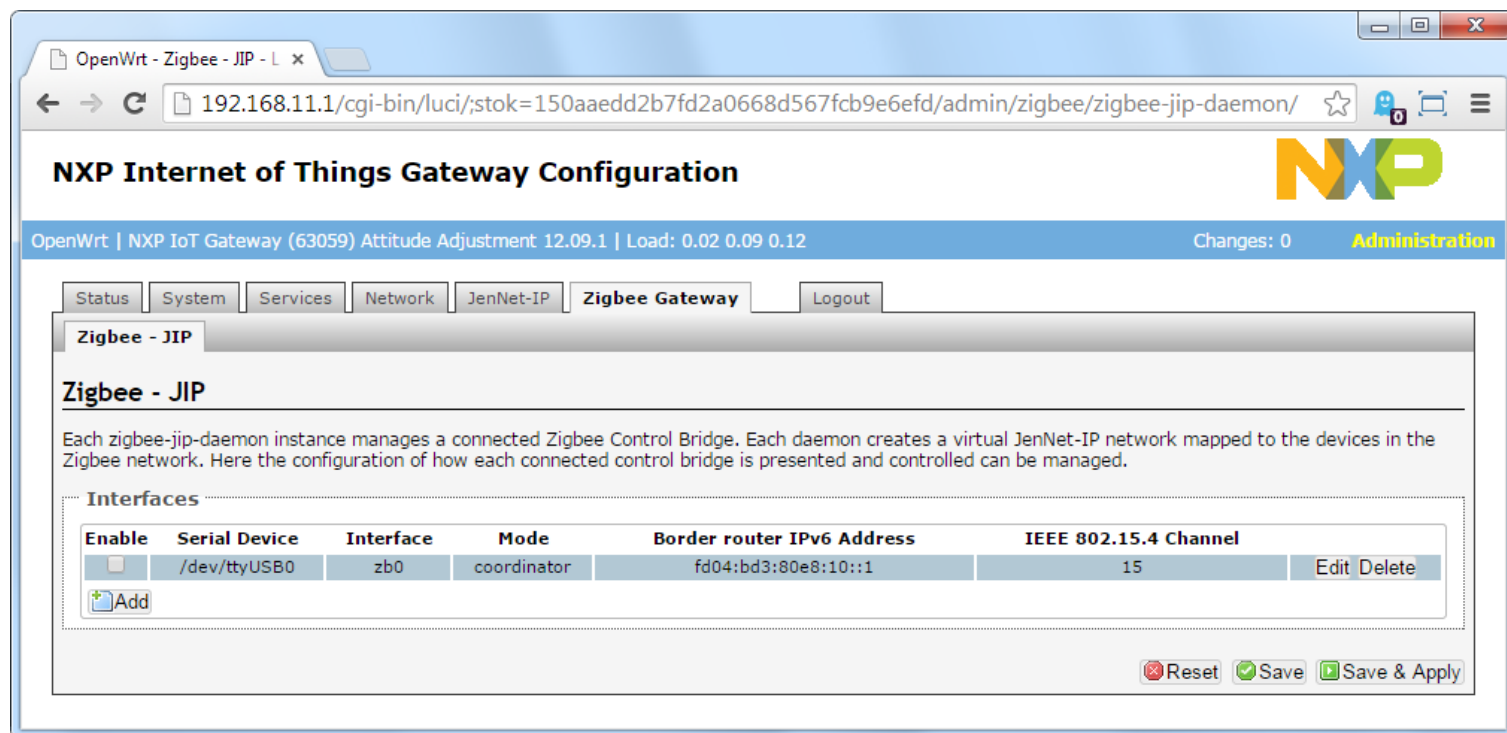
**Step 6 Enable JenNet-IP in the Linksys router from the PC (once only)**

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.

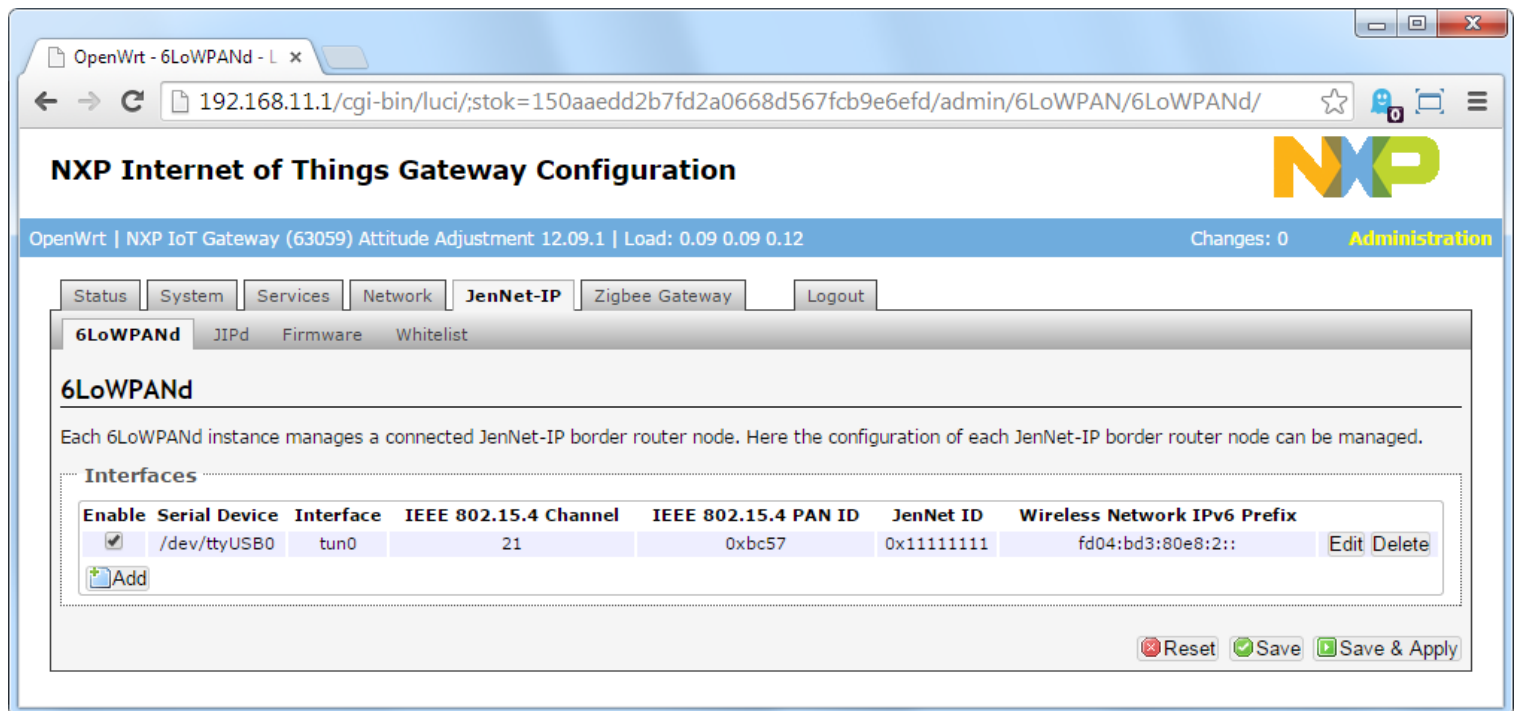
The **ZigBee – JIP** sub-tab is illustrated in the screenshot below:



- 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.



**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 Coordinator 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 Coordinator 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:

The screenshot shows a web browser window with the URL `192.168.11.1/cgi-bin/luci/stok=150aaedd2b7fd2a0668d567fcb9e6efd/admin/6LoWPAN/6LoWPANd_edit/t`. The page title is "NXP Internet of Things Gateway Configuration". The navigation bar includes tabs for Status, System, Services, Network, **JenNet-IP**, Zigbee Gateway, and Logout. Below the navigation bar, there are sub-tabs for 6LoWPANd, JIPd, Firmware, and Whitelist. The main content area is titled "6LoWPAN" and contains the text "Configuration of 6LoWPAN daemon to route IPv6 packets to IEEE 802.15.4 network". The "6LoWPANd Configuration" section has three sub-tabs: General Setup (selected), Security, and IEEE802.15.4 Radio Hardware. The General Setup tab contains the following fields:

Field	Value	Description
Enable Interface	<input checked="" type="checkbox"/>	
Enable 15.4 Bandwidth Throttling	<input type="checkbox"/>	IEEE802.15.4 is a low data rate network. Enabling this option reduces the potential for overloading the network with excessive traffic.
Serial Port	<code>/dev/ttyUSB0</code>	Specifies the serial port to which the IEEE 802.15.4 device is connected to. Examples: USB FTDI= <code>/dev/ttyUSB0</code> , USB LPC1343= <code>/dev/ttyACM0</code>
Baudrate	<code>1000000</code>	Specifies the baud rate that IEEE802.15.4 communicates at.
Network interface	<code>tun0</code>	Specifies the name of the network interface that will be created to host the JenNet-IP network. Eg <code>tun0</code>
IEEE 802.15.4 Channel	<code>21</code>	The 802.15.4 channel that the network will operate on
IEEE 802.15.4 Pan ID	<code>0xbc57</code>	The 802.15.4 Pan ID that the network will operate on
JenNet Network Id to start	<code>0x11111111</code>	ID of the JenNet network to start. Other nodes looking for this ID will join
JenNet Profile	<code>0 - Large dense network (Default)</code>	JenNet network profile tailors network configuration to its size and the topology of its environment.
6LoWPAN Network Prefix	<code>fd04:bd3:80e8:2::</code>	IPv6 Prefix for the 6LoWPAN network

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

- 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.

OpenWrt - LuCI

192.168.11.1/cgi-bin/luci/stok=150aaedd2b7fd2a0668d567fcb9e6efd/admin/6LoWPAN/6LoWPANd\_edit/t

## NXP Internet of Things Gateway Configuration

OpenWrt | NXP IoT Gateway (63059) Attitude Adjustment 12.09.1 | Load: 0.00 0.03 0.09 Changes: 0 Administration

Status System Services Network **JenNet-IP** Zigbee Gateway Logout

6LoWPANd JIPd Firmware Whitelist

### 6LoWPAN

Configuration of 6LoWPAN daemon to route IPv6 packets to IEEE 802.15.4 network

#### 6LoWPANd Configuration

General Setup Security IEEE802.15.4 Radio Hardware

JenNet Security Enabled ☒

JenNet Security Key   
JenNet security key. Specified like an IPv6 address e.g. 0:1:2::3:4

Node Authorisation Scheme

RADIUS Server IPv6 Address   
IPv6 Address of RADIUS server that will authorise nodes. This should usually be set to the IPv6 address of the lan interface: 'fd04:bd3:80e8:1::1'

[Back to Overview](#) [Reset](#) [Save](#) [Save & Apply](#)

- 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



*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)*.



**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.

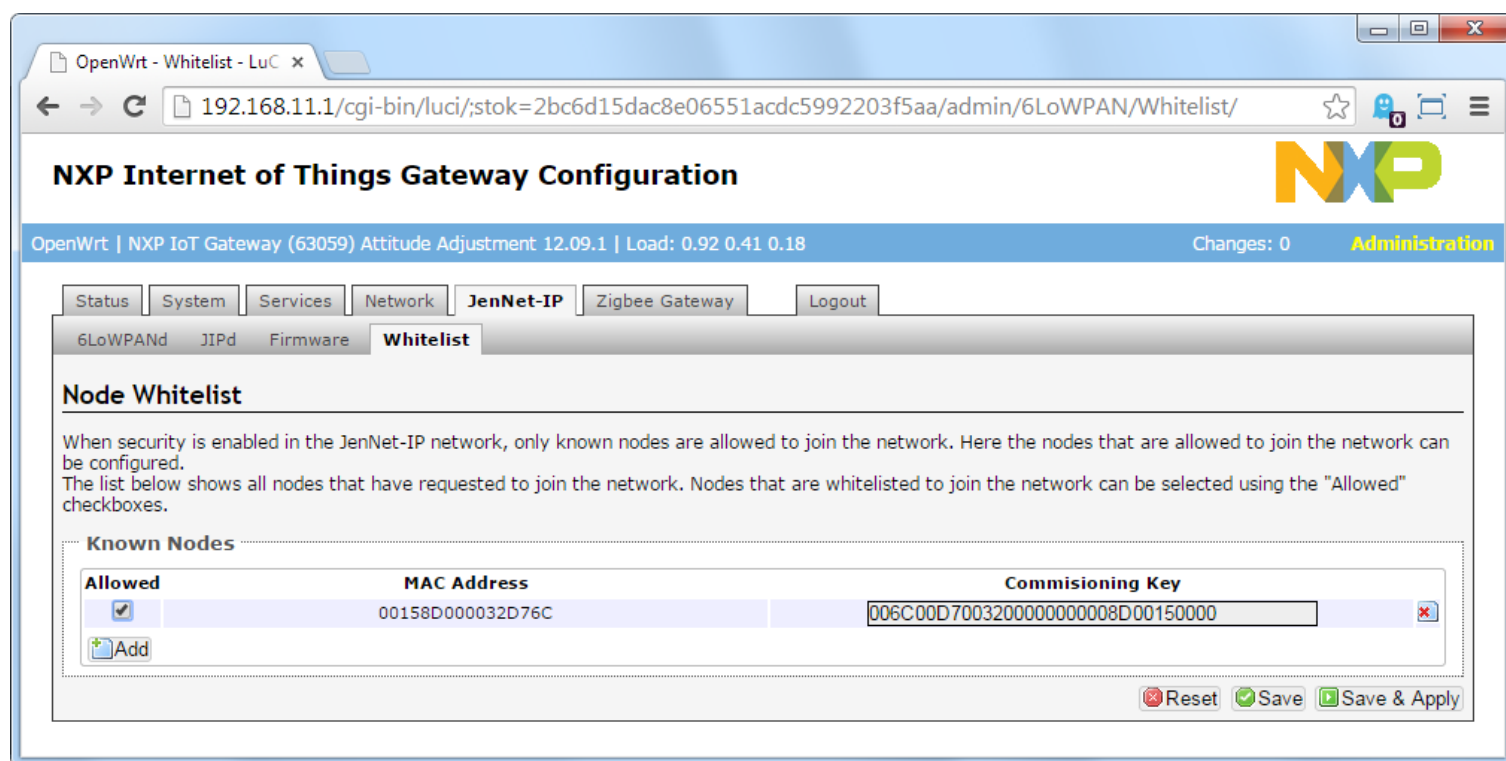
## Step 2 Access the Gateway Configuration interface from the PC

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 MAC 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.



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.

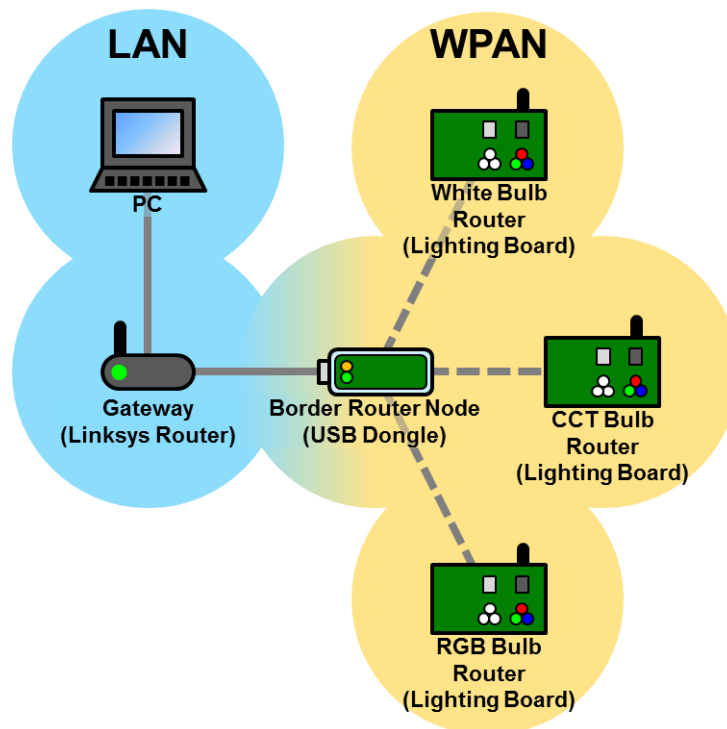
The screenshot shows a web browser window with the URL `192.168.11.1/cgi-bin/luci/stok=2bc6d15dac8e06551acdc5992203f5aa/admin/6LoWPAN/Whitelist`. The page title is "NXP Internet of Things Gateway Configuration". The navigation bar includes tabs for Status, System, Services, Network, JenNet-IP, Zigbee Gateway, and Logout. The "JenNet-IP" tab is active, and the "Whitelist" sub-tab is selected. The "Node Whitelist" section contains a description: "When security is enabled in the JenNet-IP network, only known nodes are allowed to join the network. Here the nodes that are allowed to join the network can be configured. The list below shows all nodes that have requested to join the network. Nodes that are whitelisted to join the network can be selected using the 'Allowed' checkboxes." Below this is a table with three columns: "Allowed", "MAC Address", and "Commissioning Key". The table lists five nodes, all of which are checked in the "Allowed" column. Each row has a red 'X' icon in the "Commissioning Key" column. An "Add" button is located at the bottom left of the table. At the bottom right of the page, there are buttons for "Reset", "Save", and "Save & Apply".

Allowed	MAC Address	Commissioning Key
<input checked="" type="checkbox"/>	00158D000032E452	005200E4003200000000008D00150000
<input checked="" type="checkbox"/>	00158D000035C6F9	00F900C6003500000000008D00150000
<input checked="" type="checkbox"/>	00158D0000360146	00460001003600000000008D00150000
<input checked="" type="checkbox"/>	00158D000036015D	005D0001003600000000008D00150000
<input checked="" type="checkbox"/>	00158D00003604ED	00ED0004003600000000008D00150000

### 4.1.3 Operating the Bulb Devices

This section describes how to setup and operate the bulb devices included in the Application Note using a PC outside of the WPAN (via IP) control the lights (white or RGB LEDs) on the nodes in the WPAN.

This section sets up a network with a white, CCT and colour bulb which can be controlled from a PC connected to the gateway as shown in the image below:



#### 4.1.3.1 Setting Up the Bulb Devices

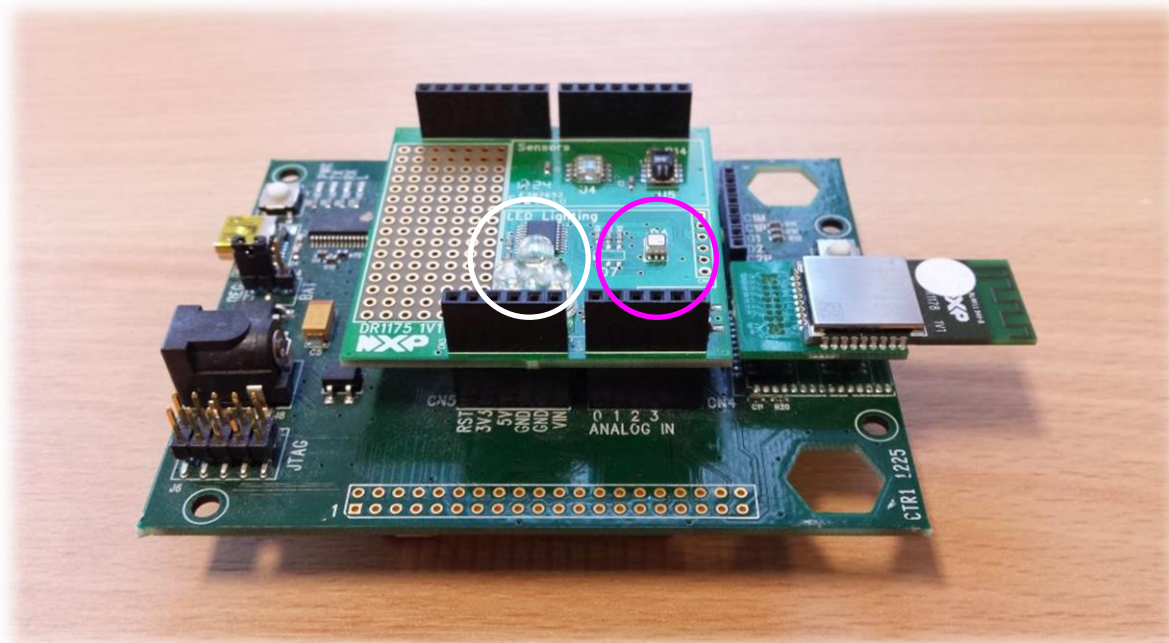
In setting up the bulb part of the demo system, you will need the following components:

- LAN part of the system (set up as described in [Section 4.1.2.2 "Setting Up the Border Router"](#)).
- Three *Carrier Boards (DR1174)* fitted with JN516x modules and *Lighting/Sensor Expansion Boards (DR1175)*, antennae and batteries programmed with the required firmware.

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

### Step 1 Setup bulb hardware

The bulb software runs on a *Carrier Board (DR1174)* fitted with the *Lighting/Sensor Expansion Board (DR1175)* as shown below:



The setup procedure is identical for the white, CCT and colour bulbs which differ only in the LEDs used to represent the bulb, (circled in the image above).

It is recommended that three boards are configured this way for this part of the demonstration.

### Step 2 Program the bulb software

The following pre-built binaries are provided in the Application Note for use on the evaluation kit boards. It is recommended that the three types of bulb software are used to create a white, CCT and colour bulb, (either of the white bulb builds may be used).

#### **0x11111111s\_DeviceBulbWhite\_JN516X\_Router\_JN5168\_v0000.bin**

This binary file is for a white bulb driving the three white LEDs, shown with a white circle in the image above, on the Carrier Board and Lighting/Sensor Expansion Board combination.

#### **0x11111111s\_DeviceBulbWhite\_DR1175\_Router\_JN5168\_v0000.bin**

This binary file is for a white bulb driving the RGB LED, shown with a magenta circle above, (displaying white only), on the Carrier Board and Lighting/Sensor Expansion Board combination.

#### **0x11111111s\_DeviceBulbTemperature\_DR1175\_Router\_JN5168\_v0000.bin**

This binary file is for a CCT bulb driving the RGB LED, shown with a magenta circle above, (displaying CCT colours only), on the Carrier Board and Lighting/Sensor Expansion Board combination.

#### 0x11111111s\_DeviceBulbColour\_DR1175\_Router\_JN5168\_v0000.bin

This binary file is for a colour bulb driving the RGB LED, shown with a magenta circle above, (displaying a full range of colours), on the Carrier Board and Lighting/Sensor Expansion Board combination.



*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 bulbs 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"](#).

The three white LEDs or the RGB LED on the *Lighting/Sensor Expansion Board (DR1175)* are controlled by the application and act as the bulb's light source.

On power-up, the node will attempt to join the WPAN (for which the USB dongle is the Coordinator). While the node is trying to join the network, the LEDs on the expansion board will be fully illuminated (*they are very bright and, to avoid eye damage, you must not look directly into them for an extended period of time*).

### **Step 4     Bulb feedback**

Once the node has joined the network, the LEDs will flash twice to indicate this and then remain fully illuminated.

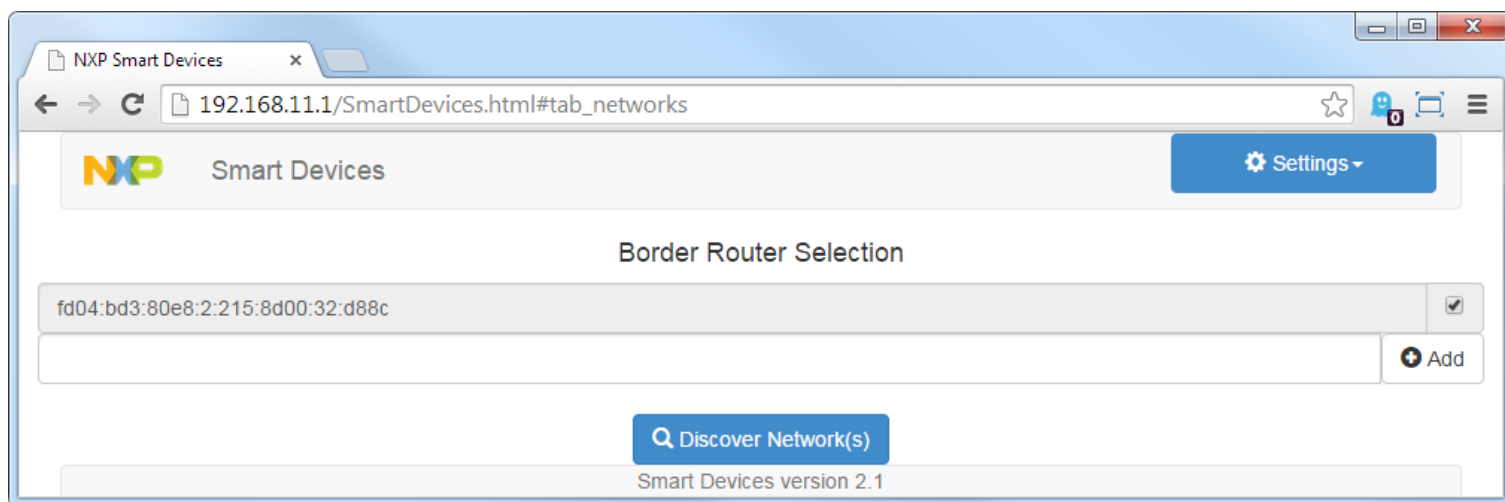
#### 4.1.3.2 Global Bulb Control from PC

The lights in the WPAN can be controlled from the PC via IP. This method of control allows the lights to be switched on, switched off and to alter the levels individually, as a group or globally (all lights).

You can control the lights in the WPAN from the Smart Devices Demonstration interface which is accessed by directing the web browser on the PC to the following (case-sensitive) IP address (or via the gateway's landing page):

<http://192.168.11.1/SmartDevices.html>

The Border Router Selection page is displayed first when accessing this interface:



More than one border router may be connected to a LAN and the Smart Devices interface allows the devices in many WPANs to be controlled together by selecting multiple border routers in this interface.

When working with the evaluation kit there should only be a single border router in the table. The IPv6 address is shown on the left of the table. To select the border router's devices ensure that the tick box on the right of the table is ticked and click the **Discover Networks** button.

The Border Router Selection page can be returned to from the other pages in this interface by clicking the **Settings** drop-down then the **Border Router Selection** option.



The **Global** tab should be displayed by default once the border routers have been selected, (or it can be selected from the tabs along the top of each page).

In the **Global** tab, there is a set of controls as depicted in the figure below:



Using the controls, you can control all the bulbs in the network together:

- Switch the lights on by clicking on the **On** box
- Switch the lights off by clicking on the **Off** box
- Vary the brightness of the lights (dimmer operation) using the horizontal control with the blue bar (to the right to brighten, to the left to dim) - this is achieved by clicking the desired location in the bar (rather than moving the vertical slider)
- Alter the colour temperature of CCT and colour bulbs by clicking the control showing a square fading from blue at the top to red at the bottom. This will open a new control window, clicking in the coloured area will set the bulb to that colour. The **Close** button will close the window. Some CCT bulbs cannot display the full range of colours and will get as close as they can to the selected colour when an out-of-range value is selected.
- Alter colour of colour bulbs by clicking the control showing a circular colour wheel. This will open a new window, clicking the colour circle will set the bulb to that colour. Clicking the **Start Loop** button will cause the bulb to loop around the colour wheel at its current saturation, the **Stop Loop** button will end this, while the **Close** button closes the window.

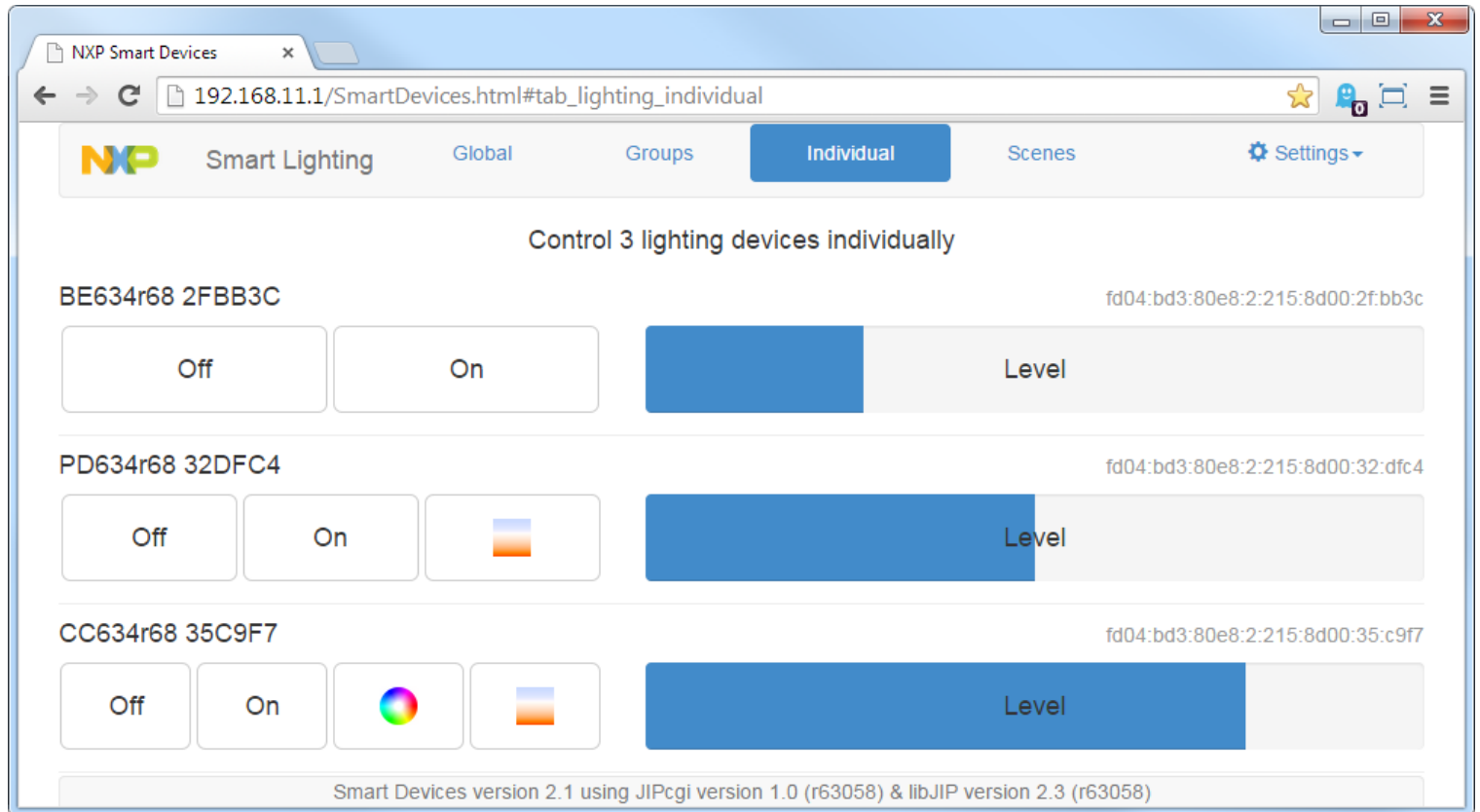
Bulbs that do not support the colour features will not be affected by the use of the colour controls. So white bulbs will not change when the CCT or colour is set and white or CCT bulbs will not change when the colour wheel is used. The colour controls will only be displayed if there are devices that support those features in the WPAN.

The global control operates by using a multicast to the “All Devices” group that all devices are placed into when started from the factory state. The group address takes the form of a special IPv6 address, the address for the “All Devices” group is FF15::F00F and is on the middle right of the image above.



### 4.1.3.3 Individual Bulb Control from PC

The **Individual** tab contains sections for all the nodes in the WPAN which have lights that can be controlled (a node is identified by a name and its IPv6 address). For each node, there is a set of controls as depicted in the figure below.



The controls work in the same as the controls in the **Global** tab but only affect the individual nodes.

The **Individual** tab only displays devices that have been discovered in the network if devices are being powered cycled or started one at a time it may be necessary to re-discover the devices in the network. Selecting the **Refresh Devices** option from the **Settings** drop-down will re-discover the individual devices in the network.

#### 4.1.3.4 Group Bulb Control from PC

The **Group** tab contains a section for two groups of lights (“Hall” and “Lounge”). For each group, there is a similar set of controls as described above for the **Individual** tab:



The group controls will have no effect on the bulbs currently in the network as they have not yet been enrolled into the Hall or Lounge groups.

The lights 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 light 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.

The JenNet-IP Smart Home demonstration transmits to the following fixed groups from the Groups tab:

- “Hall” with multicast address FF15::A00A - this group is initially empty
- “Lounge” with multicast address FF15::B00B - this group is initially empty

The interface uses a group identifier which is derived from the group’s multicast address - for an IPv6 multicast address of the format FF15::gggg, the group identifier is of the format 0x15gggg (for example, FF15::A00A is abbreviated to 0x15A00A).

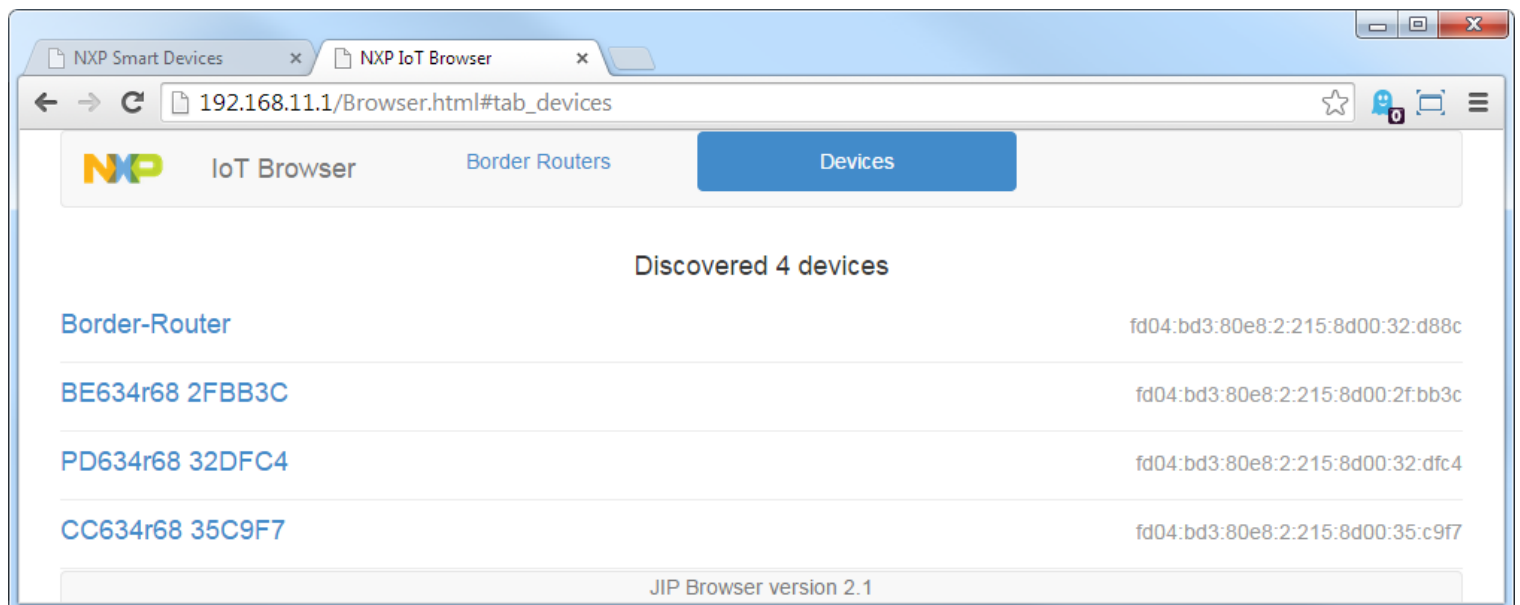
Groups of lights can be set up from the JenNet-IP Browser, which runs on the Linksys router and is accessed via a normal web browser on the PC. The JenNet-IP browser is accessed by directing the web browser on the PC to the following (case-sensitive) IP address (or via the gateway's landing page):

<http://192.168.11.1/Browser.html>



Opening the JenNet-IP Browser in a new tab will make switching between the two interfaces easier.

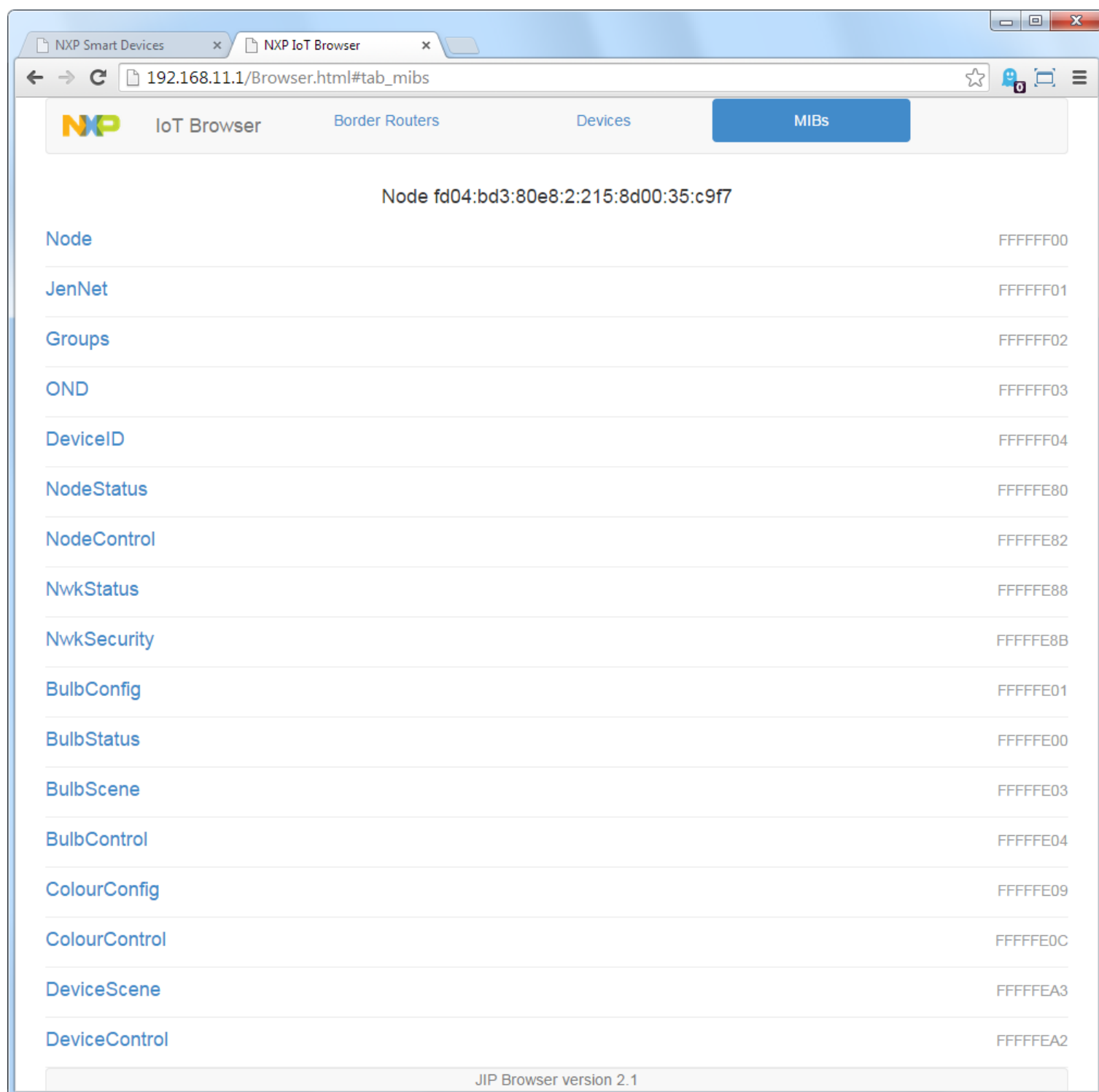
The first page displayed is a Border Router Selection page that operates identically to the Border Router Selection page in the Smart Devices Demonstration interface. Select the **Border Router** and then the **Discover Networks** button to display the devices in the network:



In the above example each node is listed by name. The default names are assigned automatically according to the device type, (the user can change the names to more meaningful values by editing the Node MIB's DescriptiveName variable):

- 'Border-Router' refers to the USB dongle attached to the Linksys router
- The entry beginning with "B" refers to the WPAN nodes that are running the white bulb device firmware - for example, 'BE634r68 2FBB3C' where 'B' indicates a White Bulb, 'E634' is the Product ID of the device, 'r' indicates a router node, '68' indicates that a JN5168 chip is being used. The final '2FBB3C' is the least significant part of the MAC address of the node.
- Entries beginning 'P' indicate CCT Bulbs.
- Entries beginning 'C' indicate colour Bulbs.

Clicking on a network node displays a Node MIBs page for that particular node, containing a list of the Management Information Bases (MIBs) on the node, as illustrated in the screenshot below for a colour bulb device:



The IPv6 address of the relevant node is shown on the near the top of the page. The name of each MIB in the device is presented on the left and the MIB ID on the right.

Clicking on the **Devices** tab will take you back to the devices page.

To configure the node's group memberships, click on the Groups MIB on the Node MIBs page. This takes you to the Groups MIB page that lists the variables contained in the Groups MIB, as illustrated in the screenshot below.

The screenshot shows the NXP IoT Browser interface. The address bar displays `192.168.11.1/Browser.html#tab_vars`. The navigation bar includes links for IoT Browser, Border Routers, Devices, MIBs, and Variables. The main content area is titled "Mib 'Groups' on Node: fd04:bd3:80e8:2:215:8d00:35:c9f7".

**Groups**

Row	Data
0	0x15f00f
1	0x15fe04
2	0x15fe0c

Variable Index 0

**AddGroup** Variable Index 1

**RemoveGroup** Variable Index 2

**ClearGroups** Variable Index 3

JIP Browser version 2.1

The **Groups** table variable lists the groups the device is currently a member of. The 0x15F00F group is the "All Devices" groups that all devices place themselves into by default, the 0x15FE04 group is the "All Bulbs" group that all bulb devices place themselves into by default, the 0x15FE0C is the "Colour Bulbs" group that CCT and colour bulbs place themselves into by default.

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

- **AddGroup:** To add the node to a group:
  - a) Enter the identifier of the group in this field (removing the curly bracket, if necessary), e.g. 0x15A00A for the "Hall" group.
  - b) Click on the **Set** button for **AddGroup**.
  - c) Refresh the **Groups** table by clicking the **Refresh** button. The new group should now appear in the **Groups** section of the page.

- **RemoveGroup:** To remove the node from a group:
  - a) Enter the identifier of the group in this field (removing the curly bracket, if necessary), e.g. 0x15B00B for the “Lounge” group.
  - b) Click on the **Set** button for **RemoveGroup**.
  - c) Refresh the **Groups** table by clicking the **Refresh** button. The group should now disappear from the **Groups** section of the page.
- **ClearGroups:** To remove the node from all groups:
  - a) Enter any value in this field.
  - b) Click on the **Set** button for **ClearGroups**.
  - c) Click on the orange **MIB Groups** tab to refresh the page. All groups should now disappear from the **Groups** section of the page.

The group addresses are displayed and entered using a shortened version of the IPv6 group address. A value of 0x15A00A is expanded into a full IPv6 address of FF15::A00A. The leading FF is assumed for all values displayed and entered into the variables in this MIB and the 15 is shifted to the most significant position (after the assumed FF). The remaining digits are right shifted into the least significant position of the IPv6 address with zeros padding out the middle

Once bulbs have been added to the “Hall” group (0x15A00A) or “Lounge” group (0x15B00B) they can be controlled from the **Group** page of the gateway’s Smart Devices Demonstration interface.

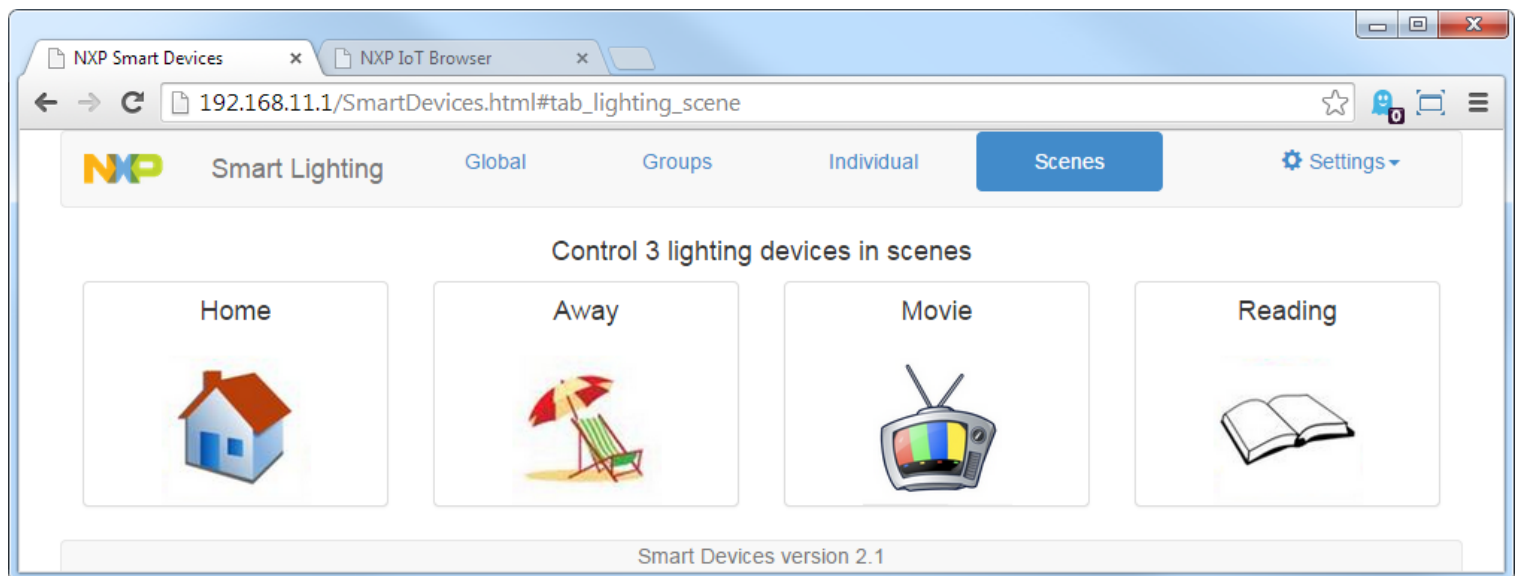
#### 4.1.3.5 Scene Bulb Control from PC

The lights on the nodes in the WPAN can be enrolled into scenes. The lights participating in a scene can be set to different states by broadcasting a command to activate a scene to the network. For example in a real situation a “night” scene might consist of a single bulb turned on at minimum brightness while turning off all other lights. A “reading” scene could dim all lights in a room to the minimum brightness while setting a single reading lamp to full brightness.

Note that a light can be enrolled into more than one scene (or into no scenes).

Scenes must be configured using the JenNet-IP Browser. The Smart Devices interface may be used to activate a limited number of scenes, alternatively full control is available via the JenNet-IP Browser.

The **Scene** tab in the Smart Devices Demonstration interface presents four fixed scenes that can be activated by clicking the appropriate button:



Each scene is identified by a 16-bit Scene ID, the Scene IDs allocated in the Smart Home Demonstration interface are shown below:

Scene Name	Scene ID
Home	0xA00A
Away	0xB00B
Movie	0xC00C
Reading	0xD00D

The following instructions describe how to configure and activate scenes. The steps below operate by first changing the current settings of the bulbs to match the settings they should display in the scene, then telling the bulb to use the current settings for a particular scene. An alternative method that allows the settings to be set in a single operation, without altering the current settings, is also available.

To configure the bulbs to take part in a scene follow the instructions below:

### Step 1 Apply bulb settings to be used in the scene

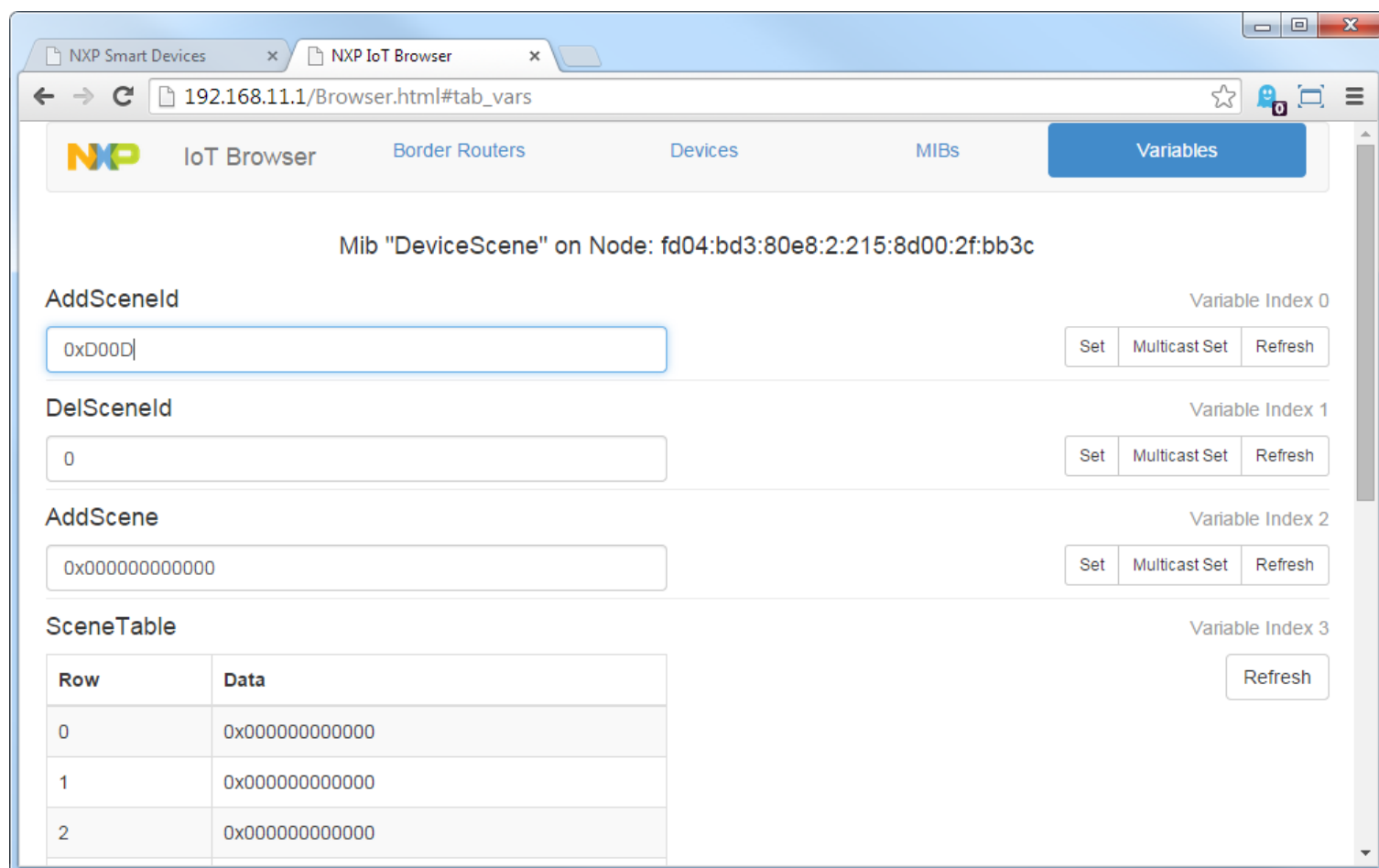
The first step in configuring a scene is to place the bulbs taking part in the scene into the state that they should be in when the scene is activated.

Use the Smart Devices Demonstration interface to navigate to the **Individual** tab and use the controls to alter the state of the bulbs as they should appear when the scene is activated.

For example, to set the bulbs into appropriate state for a “reading” scene ensure the white bulb is on at full brightness and turn off the CCT and colour bulbs.

### Step 5 Enroll bulbs into the scene

Next the bulbs need to be told to apply their current state when placed into the scene. Use the JenNet-IP Browser to navigate to the DeviceScene MIB for the first bulb as shown in the screenshot below:



Mib "DeviceScene" on Node: fd04:bd3:80e8:2:215:8d00:2f:bb3c

Variable Index 0

AddScenelId

0xD00D

Set Multicast Set Refresh

Variable Index 1

DelScenelId

0

Set Multicast Set Refresh

Variable Index 2

AddScene

0x000000000000

Set Multicast Set Refresh

Variable Index 3

SceneTable

Row	Data
0	0x000000000000
1	0x000000000000
2	0x000000000000

Refresh

Each scene is identified by a 16-bit Scene ID. The Smart Devices interface provides controls to activate 4 different scenes, including a “reading” scene with an ID of 0xD00D. To configure the bulb to apply its current settings when the reading scene is activated enter “0xD00D” into the **AddScenelId** variable and click the **Set** button.



The enrolment of the bulb into the scene can be checked by refreshing the **SceneTable** variable:

NXP IoT Browser

Mib "DeviceScene" on Node: fd04:bd3:80e8:2:215:8d00:2f:bb3c

Variable Index 0

AddSceneId

0xD00D

Set Multicast Set Refresh

Variable Index 1

DelSceneId

0

Set Multicast Set Refresh

Variable Index 2

AddScene

0x00000000000000

Set Multicast Set Refresh

Variable Index 3

SceneTable

Row	Data
0	0x00e1d00d01ff
1	0x00000000000000
2	0x00000000000000

Refresh

The first row of data should be non-zero. The highlighted section of the data is the Scene ID which should match that entered into the **AddSceneId** variable previously. The remaining data identified the device type the scene applies to and the settings for that scene. The above screenshot is for a white bulb, the CCT and colour bulbs will contain additional data used to specify the colour to be applied when the scene is activated.

The device can be added to additional scenes by repeating this step.

Additional bulbs can be added to the scene by repeating this step on those devices. For example when configuring the "reading" scene the process should be repeated on all three bulbs.

### **Step 6     Activating a scene**

In order to better observe the effects of activating the scene the bulbs should be set to medium brightness and turned on from the Smart Devices Demonstaration interface using the **Global** tab.

The “reading” scene configured in the previous section can be easily activated from the Smart Devices interface on the **Scenes** tab. Click the “Reading” scene control to activate the scene. The white bulb will turn on (if off) and adjust to the brightness set when the bulbs were placed into the “reading” scene. The CCT and colour bulbs will turn off (if on).

The scene activation works by writing the Scene ID to the BulbControl MIB’s Scene ID variable.

### **Step 7     Removing bulbs from a scene (optional)**

A bulb can be removed from a scene by entering the appropriate Scene ID into the **DelSceneld** variable and clicking the **Set** button.

Bulbs not taking part in a scene are unaffected when that scene is activated.

This can be observed by removing one of the colour bulbs from the reading scene, ensuring it is on, and observing that it no longer turns off when the scene is activated.

### **Step 8     Updating a scene (optional)**

The settings for a scene can be updated by altering the state of the bulb then re-adding the bulb to the scene by re-entering the Scene ID into the **AddSceneld** field again.

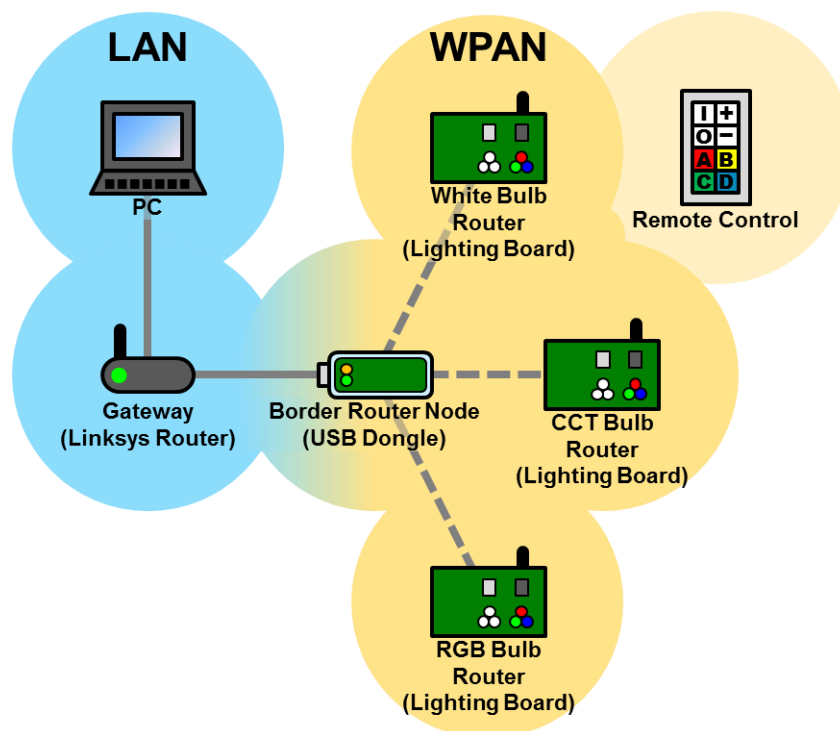
This can observed by altering the remaining CCT or colour bulb enrolled into the reading scene so it is on a full brightness, then repeating the entry of the Scene ID. When the scene is activated this bulb should turn on a full brightness in addition to the white bulb.

---

#### 4.1.4 Operating the Remote Control

The lights on the nodes of the WPAN can be controlled from within the network using the remote control. This method of control allows the lights to be switched on, switched off and the levels controlled.

This section adds a remote control to the system created earlier to create the system shown below:



---

##### 4.1.4.1 Setting Up the Remote Control

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

- LAN part of the system
- Bulbs operating in the network.
- Remote control programmed with appropriate firmware
- Batteries for the above board

To set up the remote control part of the system follow the instructions below:

### Step 1 Program the remote control software

The following pre-built binary is provided in the Application Note for use on the remote control.

**0x11111111s\_DeviceRemote\_RD6035\_JN5168\_v0000.bin**

This is the binary file for the remote control device running on the remote control.



*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 2 Set up the remote control hardware

Remove the battery compartment slide-cover on the rear of the remote control and insert two of the supplied AAA batteries (the required polarities are indicated inside the battery compartment). Then replace the cover.



On installing the batteries, the remote control will automatically power up. The unit will then attempt to join the WPAN that has been created by the USB dongle (Coordinator) attached to the Linksys router. The left LED on the remote control will flash twice per second while the unit is trying to join the network, *but the unit will not be able to join until it has been white listed.*



Note 1: If the remote control has been previously used, it will remember the last network to which it belonged. To clear this information and return to the factory settings, either erase the EEPROM during programming or enter the following key sequence into the unit: PRG OFF DOWN OFF [# O - O].

Note 2: While the remote control will initially join the WPAN as a Router node, it will then discard its Router functionality and become a 'sleeping broadcaster' (see Section 3.1). In order to wake the unit from sleep at any time, press the Wake (circle) button located below the keypad.

#### **Step 9 Add the remote control 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"](#).

#### **Step 10 Remote control feedback**

Once the unit has joined the network, the left LED will illuminate solidly for 5 seconds before being extinguished (note that the unit does not sleep and wake while trying to join a network, and should therefore not be left in this state for a long period of time in order to conserve battery power).

After joining the network, the remote control enters a configuration mode for about 30 seconds, during which the left LED will flash slowly. This mode provides an opportunity for configuration to be performed via the border router, but is not used in this demonstration. Once out of this mode, the left LED is extinguished and the unit enters 'sleeping broadcaster' mode (when required, the unit can be brought out of the sleep state by pressing the Wake (circle) button).

#### 4.1.4.2 Global Bulb Control from Remote Control

The remote control allows the control of a group of nodes or all the nodes in the network. Nodes can only be controlled individually if they have been assigned to separate groups, one node per group. Therefore, if groups of nodes have not been configured, the remote control can only be used to control all lights (synchronously).

The main operations that can be performed on the lights from the remote control are described below.



**Note 1:** All possible operations and their associated key sequences are summarised in [Section 4.1.4.4 "Remote Control Command Tables"](#).



**Note 2:** If the remote control is sleeping (the left LED does not illuminate when a key is pressed), the unit must be activated using the Wake (circle) button below the keypad before entering any command sequences.

The main operations that can be performed on the lights from the remote control are described below:

##### **Step 1 Select the Global Group**

The remote control broadcasts commands to a group address. To select the global "All Devices" group the \* key should be pressed. This group address will be used for all future transmissions until a different group address is selected using the A, B, C or D group keys.

##### **Step 2 Turn off bulbs**

The bulbs (in the selected group) can be turned off by pressing the key:

OFF [O]

##### **Step 3 Turn on bulbs**

The bulbs (in the selected group) can be switched on by pressing the key:

ON [I]

If a light is off when this key is pressed, it will be illuminated in the same state that it had before it was last switched off.

#### **Step 4 Changing the brightness of lights**

First select the brightness level parameter by pressing the 1 key. All future level commands will affect the brightness level until a different level parameter is selected using the 2, 3 or 4 keys.

The bulbs (in the selected group) can be increased in brightness by pressing the key:

UP [+]

- The brightness will only increase while the key is being pressed, until the key is released or maximum brightness is reached (*to avoid eye damage, do not look directly into the LEDs when they are at or near maximum brightness*).
- If a light is off when this key is pressed, the command will switch on the light.

The lights (in the selected group) can be decreased in brightness by pressing the key:

DOWN [-]

- The brightness will only decrease while the key is being pressed, until the key is released or the minimum brightness is reached (the lights cannot be completely switched off with this key).
- If a light is off when this key is pressed, the command will have no effect.

#### **Step 5 Changing the colour temperature of lights**

First select the colour temperature level parameter by pressing the 4 key.

The Up (+) and Down (-) keys will control the colour temperature of CCT and colour bulbs in a similar way to controlling the brightness level.

#### **Step 6 Changing the saturation of lights**

First select the saturation level parameter by pressing the 2 key.

The Up (+) and Down (-) keys will control the saturation of colour bulbs in a similar way to controlling the brightness level.

#### **Step 7 Changing the hue of lights**

First select the hue level parameter by pressing the 3 key.

The Up (+) and Down (-) keys will control the hue of colour bulbs in a similar way to controlling the brightness level.

The effect is only noticeable when the colour of the bulb is somewhat saturated, changing the hue of a bulb while it is white in colour will have no visible effect.

When changing the hue the colour cycles round the colour wheel, in order to allow colour bulbs displaying different colours to be brought to the same colour using the remote control the hue up and down controls stop at red at the minimum and maximum hue values.

## **Step 8    Test modes**

Each of the level parameters has a test mode that is accessed by pressing the > key, it has the following effects depending upon the selected level parameter:

<b>Level Parameter</b>	<b>Effect</b>
Brightness	Toggles the bulbs between on and off
Saturation	Displays random colours and saturations. Pressing the button ends the mode and returns to the original settings.
Hue	Loops continuously around the colour wheel at the current saturation. Pressing the button ends the effect and returns to the original settings.
CCT	Displays random colours and brightness in the orange region of the CCT band. Pressing the button a second time ends the effect and returns to the original settings.



---

#### 4.1.4.3 Group Bulb Control from Remote Control

This section describes how to set up groups of lights (WPAN nodes) for control from the remote control in the WPAN.

On the remote control, there are keys for four default groups: A, B, C and D. These groups are initially empty. In addition, there is a key for the All group (the \* key). The running nodes in the WPAN will be automatically put into the All group. To assign one or more nodes to another group (other than All), follow the instructions below:



**Note 1:** When assigning a node to a group, the radio transmitter of the remote control operates in a low-power mode. This requires the unit to be brought near to the node to be added to a group.



**Note 2:** Group memberships set up using the remote control are particular to that unit and do not automatically apply to other remote controls or to control from a PC via the Smart Devices interface.



**Note 3:** Grouping also provides a method of controlling individual nodes from the remote control, by assigning one node per group.

##### **Step 1 Remove the nodes that are not required in the group**

The nodes that are not required in the group should be powered off (e.g. by removing their batteries) or positioned some distance from the nodes that are required in the group.

##### **Step 2 Locate the remote control near to the node to be added**

Place the remote control close to the node to be added to the group.

##### **Step 3 Add the node to the group**

Add the nearby node to the group by entering the following key sequence into the remote control: PRG UP ON GRP [# + I A/B/C/D]

- For group A, enter: # + I A
- For group B, enter: # + I B
- For group C, enter: # + I C
- For group D, enter: # + I D

**Step 4 Repeat for another node**

If another node is to be added to the group, repeat the procedure from **Step 2**.



Note: A node can be removed from a group using the command PRG DOWN OFF GRP - for example, for group A the required key sequence is # - O A.

Once bulbs have been added to a remote control's group they can be controlled from the remote control by first pressing the key (A, B, C or D) for the group to be controlled and then using the command keys (O, I, +, -).

#### 4.1.4.4 Remote Control Command Tables

The following tables show all the commands that can be used on the remote controls:

##### Generic Expansion Board (DR1199) Remote Control

The table below summarises the key sequences and associated operations that can be performed from the *Carrier Board (DR1174)* fitted with the *Generic Expansion Board (DR1199)*.

Note that this remote control transmits commands to a group address unique to each remote control – devices must be added to the remote control's group before they can be controlled.

Operation	Key Combination	
<b>Switch on lights</b> in selected group	ON	SW1
<b>Switch off lights</b> in selected group	OFF	SW2
<b>Increase brightness</b> of lights in selected group (increase occurs while key UP [SW3] key is pressed). If a light is off, it is switched on.	UP	SW3
<b>Decrease brightness</b> of lights in selected group (decrease occurs while key DOWN [SW4] key is pressed). If a light is off, command has no effect.	DOWN	SW4
<b>Add nodes</b> (within radio range) to the remote's group (the radio is put in a <b>low-power mode</b> for this operation)	ON + UP	SW1 + SW3
<b>Commission and then add bulbs</b> (within radio range) to the remote's network.	PRG + ON	DIO8 + SW1
<b>Remove nodes</b> (within radio range) from the remote's group (the radio is put in a <b>low-power mode</b> for this operation)	ON + DOWN	SW1 + SW4
<b>Decommission nodes</b> (within radio range) from the specified group.	PRG + DOWN	DIO8 + SW4
<b>Commission nodes</b> (within radio range) to learn network settings in extending standalone WPAN to full JIP system, use for devices that do not have specific commissioning sequences. (Bulbs and Remote Controls should be commissioned using the device specific sequences)	PRG + UP	DIO8 + SW3
<b>Commission an additional Remote Control Unit</b> (within radio range) to act as <b>independent</b> Remote Control Unit for WPAN.	PRG + OFF	DIO8 + SW2
<b>Try to join an existing WPAN.</b>	ON + Power	SW1 + RESET
<b>Create a standalone WPAN.</b>	PRG + Power	DIO8 + RESET
<b>Perform a factory reset.</b>	OFF + Power	SW2 + RESET
<b>Table 7: DR1199 Remote Control Key Sequences and Associated Operations</b>		

SW1 = ON SW2 = OFF SW3 = UP SW4 = DOWN DIO8 = PRG Power = RESET

### Touch Remote Control (RD6035/DR1159 5v4)

The table below summarises the key sequences and associated operations that can be performed from the evaluation kit remote control. Note that a selected group or level parameter is used for all future operations, the group and level do not need to be re-selected prior to every command.

Operation	Key Sequence	
<b>Switch on lights</b> in selected group	[GRP/ALL] ON	A/B/C/D/* I
<b>Switch off lights</b> in selected group	[GRP/ALL] OFF	A/B/C/D/* O
<b>Increase level</b> of lights in selected group (increase occurs while key UP [+] key is pressed).	[GRP/ALL] [LVL] UP	A/B/C/D + * +
<b>Decrease level</b> of lights in selected group (decrease occurs while key DOWN [-] key is pressed).	[GRP/ALL] [LVL] DOWN	A/B/C/D - * -
<b>Select Brightness Level Parameter</b>	LVL	1
<b>Select Saturation Level Parameter</b>	LVL	2
<b>Select Hue Level Parameter</b>	LVL	3
<b>Select CCT Level Parameter</b>	LVL	4
<b>Add nodes</b> (within radio range) to the specified group (the radio is put in a <b>low-power mode</b> for this operation)	PRG UP ON GRP/ALL	# + I A/B/C/D/*
<b>Commission and then add bulbs</b> (within radio range) to the specified group as well as to the 'All' group.	PRG ON OFF ON GRP	# I O I A/B/C/D
<b>Commission and then add bulbs</b> (within radio range) to the 'All' group only.	PRG ON OFF ON ALL	# I O I *
<b>Remove nodes</b> (within radio range) from the specified group (the radio is put in a <b>low-power mode</b> for this operation)	PRG DOWN OFF GRP/ALL	# - O A/B/C/D/*
<b>Decommission nodes</b> (within radio range) from the specified group.	PRG OFF ON OFF GRP	# O I O A/B/C/D
<b>Decommission nodes</b> (within radio range) from the 'All' group only.	PRG OFF ON OFF ALL	# O I O *
<b>Commission an additional Remote Control Unit</b> (within radio range) to act as <b>independent</b> Remote Control Unit for WPAN.	PRG ON OFF ON DOWN	# I O I -
<b>Commission an additional Remote Control Unit</b> (within radio range) to act as <b>cloned</b> Remote Control Unit for WPAN.	PRG ON OFF ON UP	# I O I +
<b>Commission nodes</b> (within radio range) to learn network settings. (Bulbs, Remote Controls and Low Energy Devices should be commissioned using the device specific sequences)	PRG ON OFF ON OFF	# I O I O
<b>Join an existing WPAN.</b>	PRG UP DOWN UP	# + - +
<b>Create a standalone WPAN.</b>	PRG DOWN UP DOWN	# - + -
<b>Perform a factory reset.</b>	PRG OFF DOWN OFF	# O - O
<b>Perform a software reset.</b>	PRG ON UP ON	# I + I
<b>Table 7: RD6035 Remote Control Key Sequences and Associated Operations</b>		

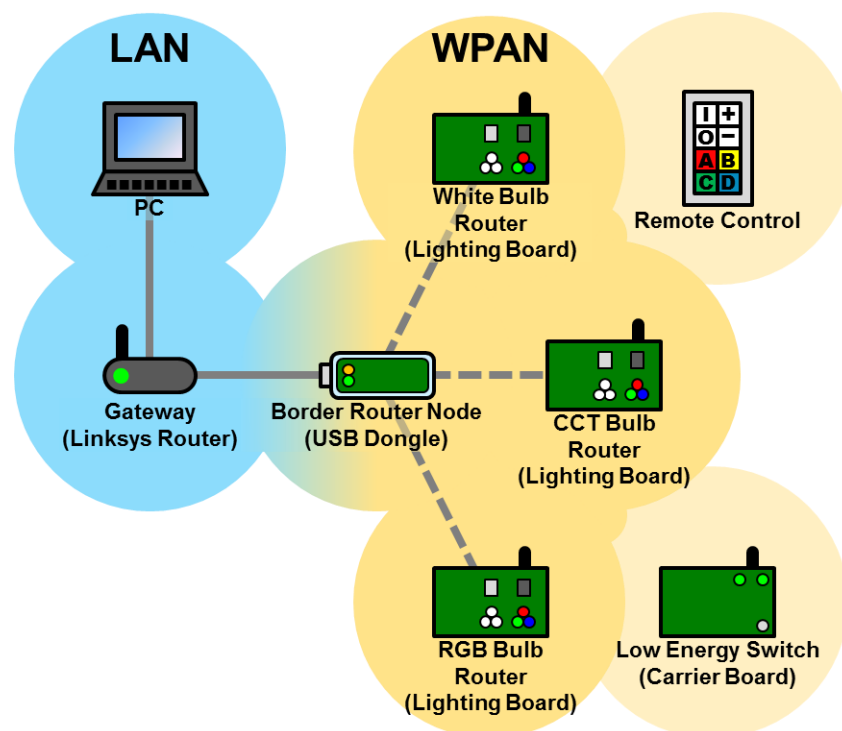
I = ON    O = OFF    + = UP    - = DOWN    # = PRG    \* = ALL  
GRP = A, B, C or D    LVL = 1, 2, 3 or 4

---

### 4.1.5 Operating the Low Energy Switch

The lights on the nodes of the WPAN can be controlled from within the network using the low energy switch. This method of control allows the lights to be switched on and off. With suitable hardware featuring additional inputs it could be extended to allow control of the bulb levels.

This section adds a low energy switch to the system created earlier to create the system shown below:



---

#### 4.1.5.1 Setting up the Low Energy Switch

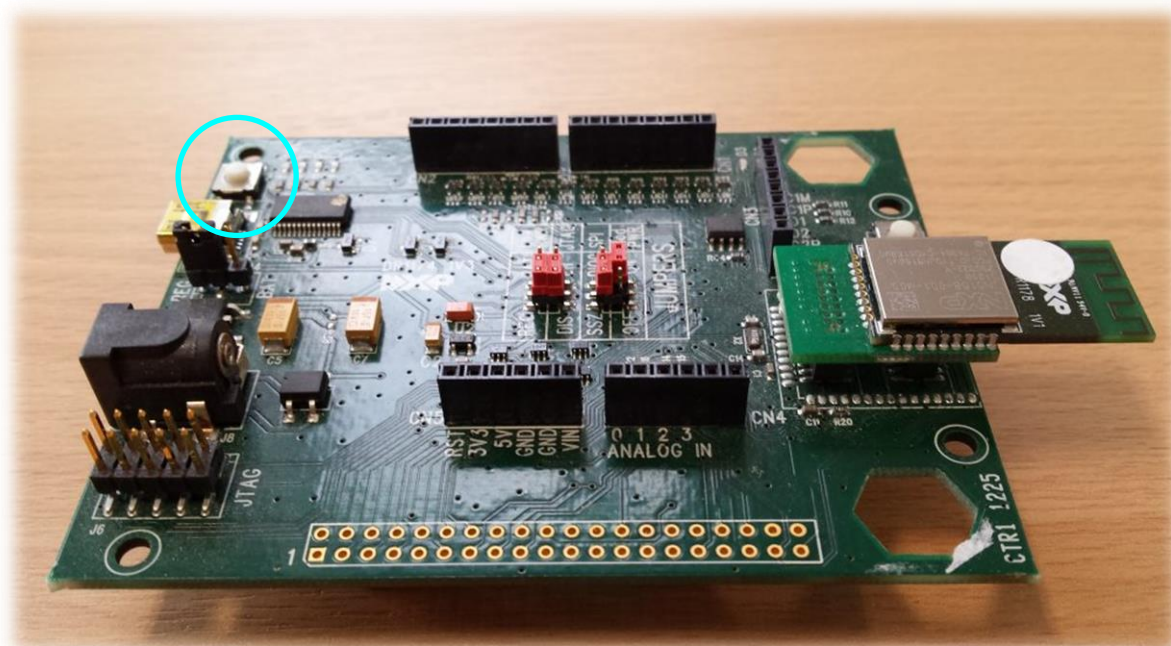
In setting up the low energy switch part of the demo system, you will need the following components:

- LAN part of the system
- Bulbs operating in the network.
- Carrier Board programmed with appropriate firmware
- Batteries for the above board

To set up the low energy switch part of the system follow the instructions below:

### Step 1 Setup low energy switch hardware

The low energy switch software runs on a *Carrier Board (DR1174)* as shown below:



The reset button marked RST, (circled in the image above), is used to wake the switch from deep sleep mode and transmit commands to the bulbs.

### Step 2 Program the low energy switch software

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

#### **CH21\_LowEnergySwitch\_DR1174\_v0000.bin**

This binary file is compiled to operate on channel 21 only. The channel used for the network can be found in the Gateway Configuration Interface accessed via the JenNet-IP > 6LoWPANd tab. If you are using a different channel you can perform one of the following actions:

- Compile the low energy switch software for a different channel by changing the CHANNEL variable in the **makefile** for the low energy switch.
- Move the network to channel 21 by first changing the channel in the Gateway Configuration interface, then factory resetting the devices in the network so they re-join the network on the new channel. (The devices in this application note lock themselves onto the channel and PAN ID of the network they join to reduce the re-join time – this behaviour can be altered by changing the application code so they scan all channels and PAN ID for use in systems where the channel may be regularly changed.)

### Step 3 Add the low energy switch to the network

Unlike the other devices in this Application Note the low energy switch does not join the network in the normal way, it simply broadcasts commands to bulbs whenever it is powered up or reset.

However the low energy switch must still be added to the white list in the gateway for its commands to be recognised by the network.

Once the node has been powered up press the reset button a few times. This will generate some commands and make the nodes in the network aware of its presence. The bulbs send commands to the gateway indicating that there is a low energy switch attempting to communicate with them that they do not recognise. The gateway in turn will grey list the low energy switch adding its MAC address to the table but leaving the tickbox unticked.

Add node to white list in the JenNet-IP Border Router Configuration interface following the general instructions. The low energy switch uses a fixed security key of 0x10203040506070801121314151617181, (instead of one derived from the MAC address), which is which must be entered into the white list as shown in the following screenshot:

OpenWrt - Whitelist - LuC x

192.168.11.1/cgi-bin/luci/stok=1fdb84dd17cd21f2310de9abc95e8284/admin/6LoWPAN/Whitelist/

## NXP Internet of Things Gateway Configuration

OpenWrt | NXP IoT Gateway (63059) Attitude Adjustment 12.09.1 | Load: 0.32 0.09 0.07 Changes: 0 Administration

Status System Services Network **JenNet-IP** Zigbee Gateway Logout

6LoWPANd JIPd Firmware **Whitelist**

### Node Whitelist

When security is enabled in the JenNet-IP network, only known nodes are allowed to join the network. Here the nodes that are allowed to join the network can be configured. The list below shows all nodes that have requested to join the network. Nodes that are whitelisted to join the network can be selected using the "Allowed" checkboxes.

Allowed	MAC Address	Commissioning Key
<input checked="" type="checkbox"/>	00158D00002FBB3C	003C00BB002F00000000008D00150000
<input checked="" type="checkbox"/>	00158D000032C17C	007C00C1003200000000008D00150000
<input checked="" type="checkbox"/>	00158D000032D79D	10203040506070801121314151617181
<input checked="" type="checkbox"/>	00158D000032DFC4	00C400DF003200000000008D00150000
<input checked="" type="checkbox"/>	00158D000032E452	005200E4003200000000008D00150000
<input checked="" type="checkbox"/>	00158D000035C6F9	00F900C6003500000000008D00150000
<input checked="" type="checkbox"/>	00158D000035C9F7	00F700C9003500000000008D00150000
<input checked="" type="checkbox"/>	00158D000035CBB1	00B100CB003500000000008D00150000
<input checked="" type="checkbox"/>	00158D0000360146	00460001003600000000008D00150000
<input checked="" type="checkbox"/>	00158D000036015D	005D0001003600000000008D00150000
<input checked="" type="checkbox"/>	00158D00003604ED	00ED0004003600000000008D00150000

Add

Reset Save Save & Apply



**Step 4** Once the low energy switch has been white listed devices receiving a command from a switch for the first time will query the gateway and be informed if it is a recognised device. Once authorised by the gateway devices in the network can decode the commands and react to them as appropriate.



At this stage the bulbs will not react to commands from the low energy switch, the bulbs must be placed into the group the low energy switch broadcasts its commands to as described in the next section.



#### 4.1.5.2 Group Bulb Control from Low Energy Switch

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

The process is the same as one used in [Section 4.1.3.4 "Group Bulb Control from PC"](#) but the unique group address used by each low energy switch must be entered. This address is derived from the MAC address of the low energy switch as follows:

FF15::MMMM:MMMM:MMMM:MMMM:0000

With the *M*s replaced by the MAC address of the low energy switch. The following screenshot shows the entry for a low energy switch with a MAC address of 0x00158D000032D79D the group for which is entered into the **AddGroup** variable as 0x**15**00158D000032D79D0000 (with an added 15 specifying a group address shown in bold). The MAC address of the low energy switch can be easily copied from the white list entry set up in the previous section.

The screenshot shows the NXP IoT Browser web interface in a browser window. The address bar shows the URL 192.168.11.1/Browser.html#tab\_vars. The interface has a navigation bar with tabs: IoT Browser, Border Routers, Devices, MIBs, and Variables (which is active). Below the navigation bar, the title is "Mib "Groups" on Node: fd04:bd3:80e8:2:215:8d00:2f:bb3c".

Under the "Groups" section, there is a table with 2 columns: Row and Data.

Row	Data
0	0x15f00f
1	0x15fe04
2	0x15158d000032c17c0001

To the right of the table is a "Refresh" button and the text "Variable Index 0".

Below the table is the "AddGroup" section. It has a text input field containing "0x1500158D000032D79D0000". To the right of the input field are three buttons: "Set", "Multicast Set", and "Refresh". The text "Variable Index 1" is to the right of these buttons.

Below the "AddGroup" section is the "RemoveGroup" section. It has a text input field containing "0x0000". To the right of the input field are three buttons: "Set", "Multicast Set", and "Refresh". The text "Variable Index 2" is to the right of these buttons.

Below the "RemoveGroup" section is the "ClearGroups" section. It has a text input field containing "0". To the right of the input field are three buttons: "Set", "Multicast Set", and "Refresh". The text "Variable Index 3" is to the right of these buttons.

At the bottom of the interface, it says "JIP Browser version 2.1".

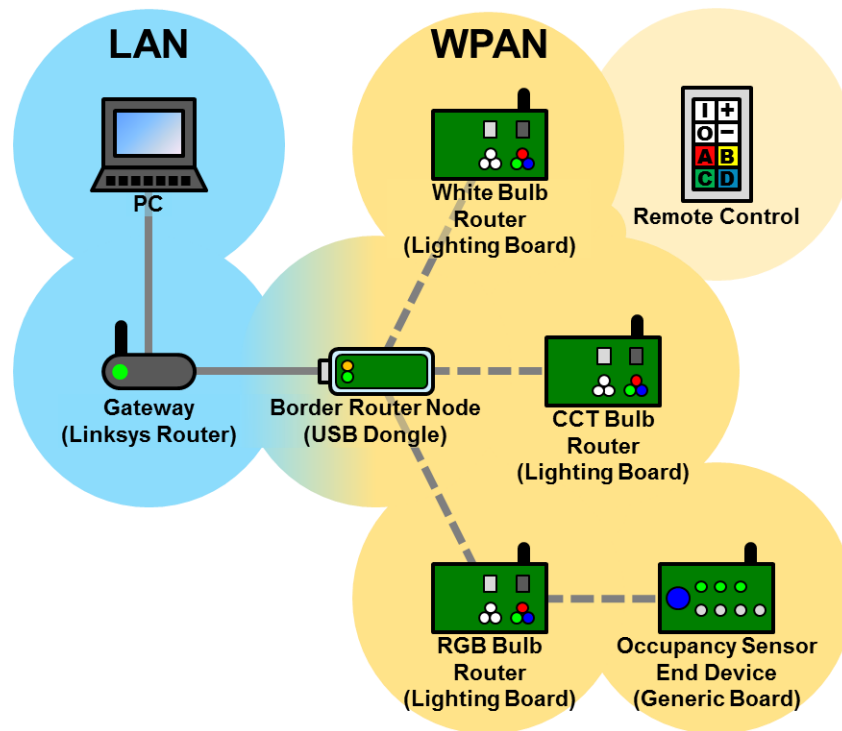
Once the switch has been white listed and lights added to the low energy switch's group those lights can be toggled on and off each time the low energy switch is reset by pressing and releasing the RST button on the *Carrier Board (DR1174)*.

---

### 4.1.6 Operating the Occupancy Sensor

The lights on the nodes of the WPAN can be controlled from within the network using the occupancy sensor. This method of control allows the lights to be switched on when the sensor detects an area is occupied and off when unoccupied.

This section replaces the low energy switch in the system created earlier with an occupancy sensor to create the system shown below:



---

#### 4.1.6.1 Setting up the Occupancy Sensor

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

- LAN part of the system
- Bulbs operating in the network.
- *Carrier Board (DR1174)* fitted with the *Generic Expansion Board (DR1199)* programmed with appropriate firmware

or

*Carrier Board (DR1174)* fitted with Parallax PIR module programmed with appropriate firmware

- Batteries for the above board

When using a single evaluation kit it is suggested that the board used as the low energy switch be re-used as the occupancy sensor.

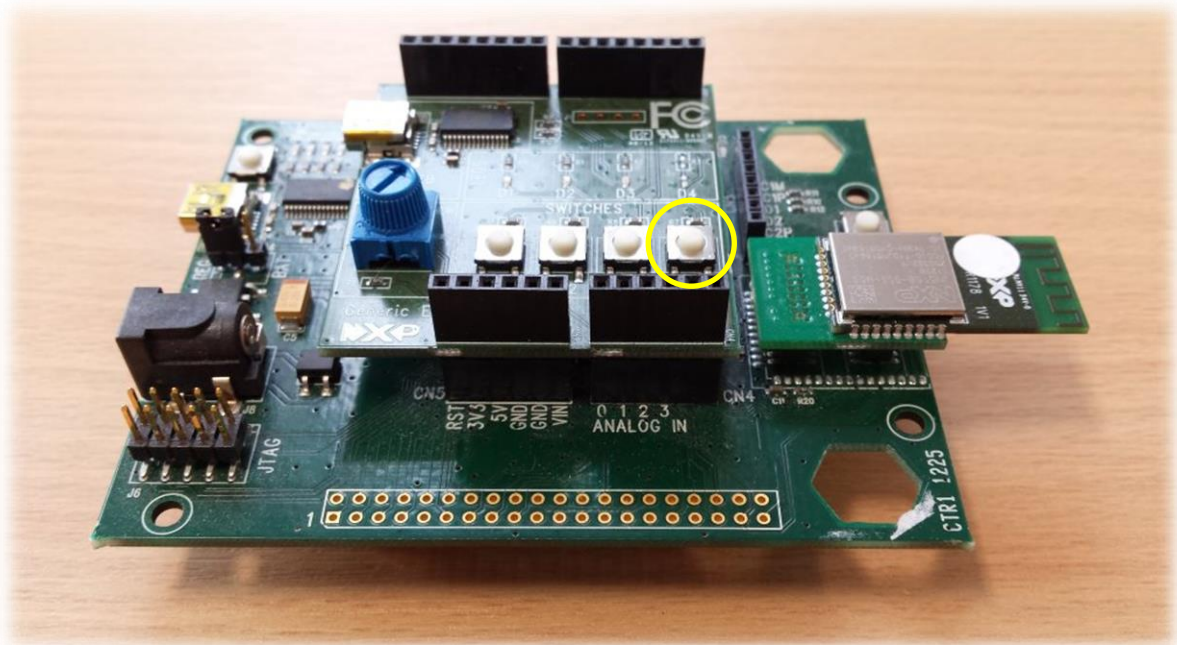
To set up the occupancy sensor part of the system follow the instructions below:

**Step 1 Setup occupancy sensor hardware**

Two hardware configurations are available for the occupancy sensor:

**a) Occupancy Sensor using Generic Expansion Board**

A Carrier Board fitted with the Generic Expansion Board may be used when only the evaluation kit hardware is available for use. The button marked SW4 on the Generic Expansion Board is monitored by the application and acts as the occupancy sensor. The button registers as unoccupied when pressed and occupied when released.



**b) Occupancy sensor using PIR module**

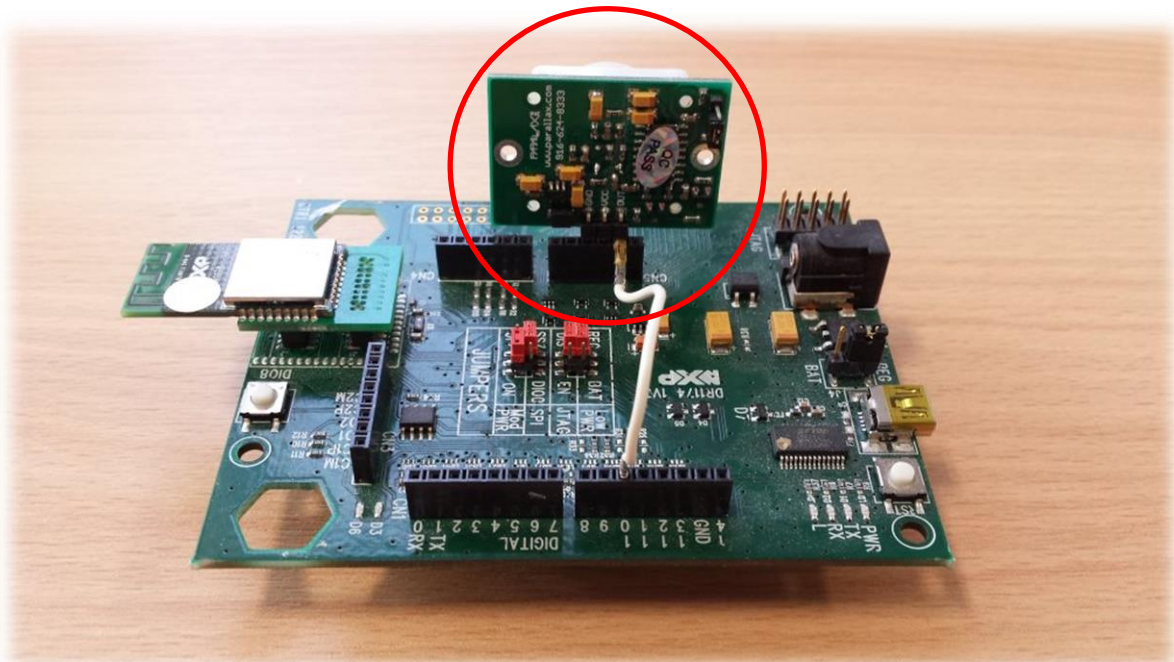
A Carrier Board fitted with the following Parallax PIR module (item code: 555-28027) may be used to create an Occupancy Sensor using a real PIR sensor, (the Parallax PIR module is not supplied as part of the evaluation kit):

<http://www.parallax.com/Store/Sensors/ObjectDetection/tabid/176/CategoryID/51/List/0/SortField/0/Level/a/ProductID/83/Default.aspx>

Install a Parallax PIR Module onto the Carrier Board as follows:

- Bend the OUT pin of the PIR module back 90 degrees.
- Insert the GND and VCC pins into the GND and 5V sockets of the CN3 connector of the Carrier Board.
- Connect the OUT pin to socket 10 (DIO 1) on the CN2 connector of the Carrier Board.

The following image shows the Parallax PIR module fitted to the Carrier Board:



## **Step 2 Program the occupancy sensor software**

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

**0x11111111s\_DeviceSensorOccupancy\_DR1174\_Router\_JN5168\_v0000.bin**  
**0x11111111s\_DeviceSensorOccupancy\_DR1174\_EndDevice\_JN5168\_v0000.bin**

The Router build operates as a Router node in the network, extending the network for other devices to join and so must be permanently powered.

The End Device build operates as an End Device node in the network. End Devices spend the majority of their time asleep and thus are suitable for battery powered operation.

Either build may be used for this part of the demonstration, it is recommended to use the End Device build in order to introduce an End Device node into the network.



*It may be necessary to remove the Parallax PIR module during programming, depending upon the power supply used, as it draws additional power.*

## **Step 3 Add the occupancy sensor 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"](#).

## **Step 5 Occupancy sensor feedback**

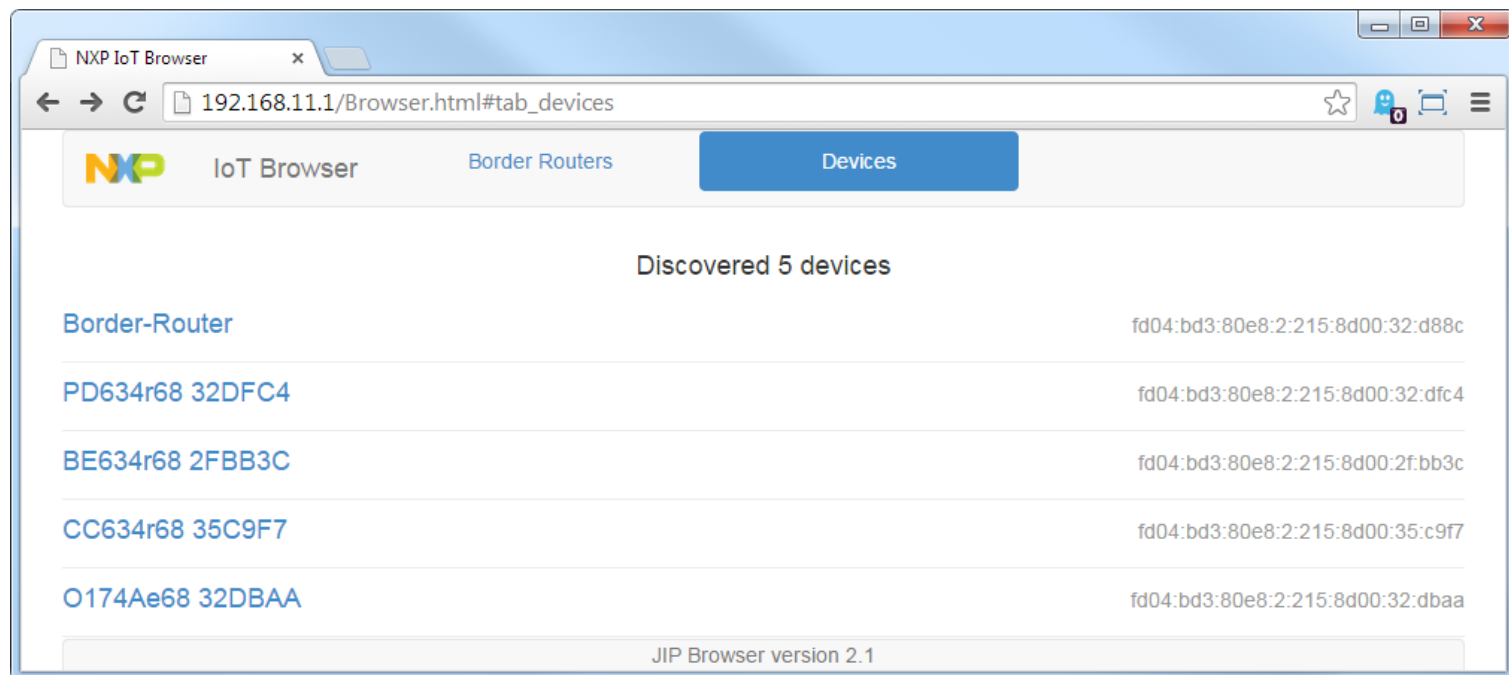
The occupancy sensor does not provide any feedback when it joins a network. The JenNet-IP Browser interface provided by the gateway can be used to check if the occupancy sensor has joined the network.

#### 4.1.6.2 Occupancy Sensor Monitoring from PC

The sensors in the WPAN can be monitored from the PC via IP. This method of monitoring allows the state of the sensor devices to be read.

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

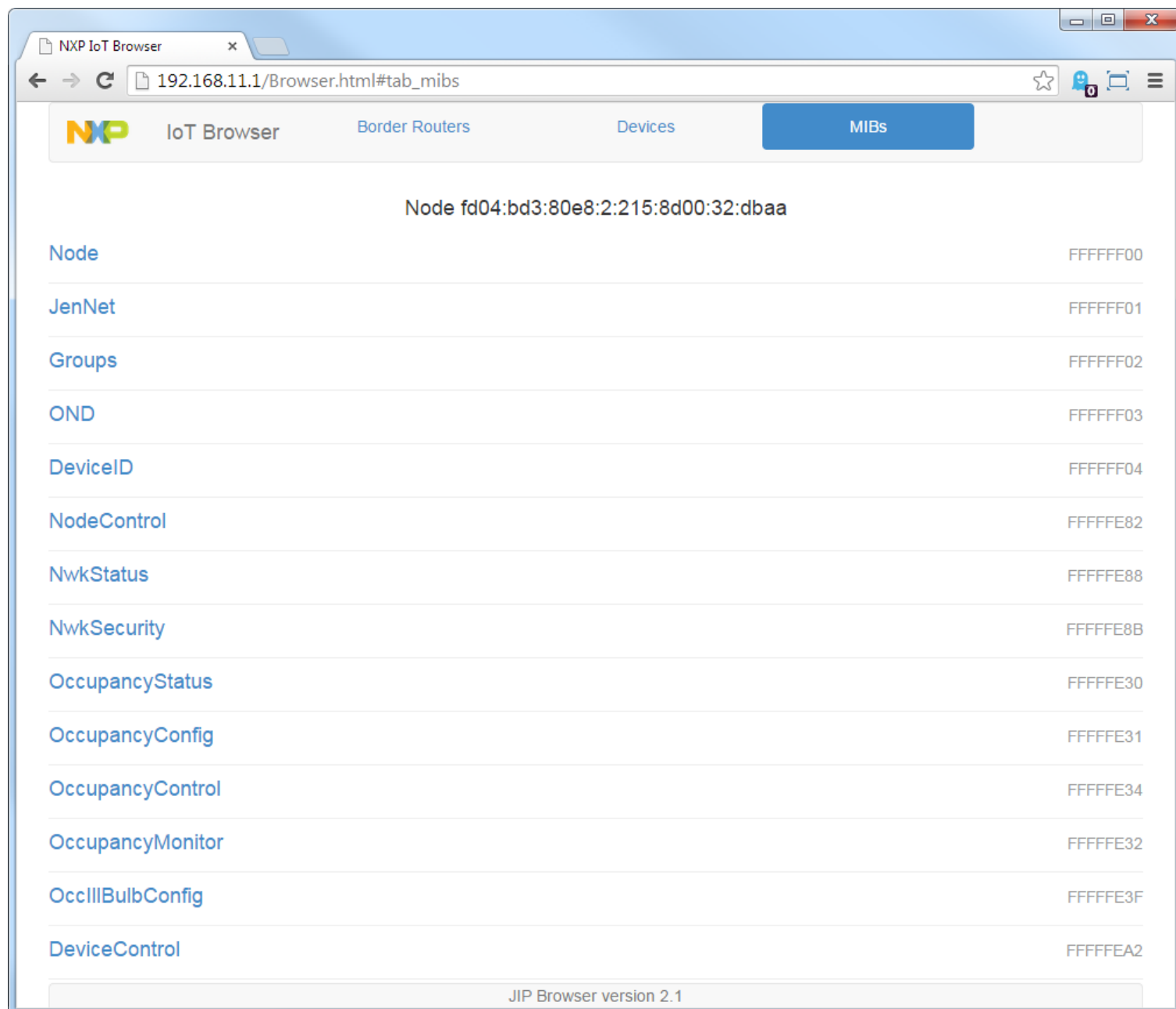
You can monitor the state of the sensors using the JenNet-IP Browser. Navigate through the Border Router Selection page to display the **Devices** tab to display the devices in the network as shown in the screenshot below:



Occupancy sensors use a default name beginning 'O', illustrated by the last entry in the screenshot above.

## Step 2 Select the occupancy sensor

Clicking on a network node displays a Node MIBs page for that particular node, containing a list of the Management Information Bases (MIBs) on the node, as illustrated in the screenshot for the occupancy sensor below:

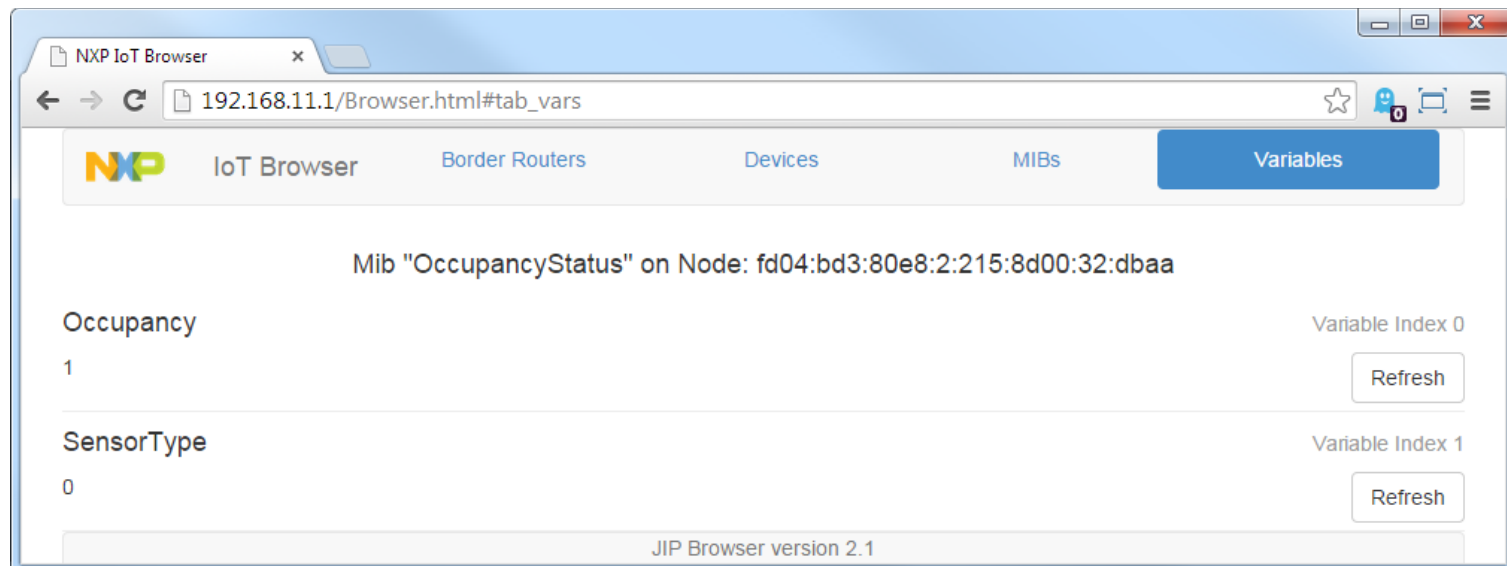


A number of the MIBs are dedicated to the operation of the occupancy sensor.



### Step 3 Select the OccupancyStatus MIB

To monitor the status of the occupancy sensor, click on the OccupancyStatus MIB on the MIBs page. This takes you to the OccupancyStatus MIB page that lists the variables contained in the OccupancyStatus MIB, as illustrated in the screenshot below.



The **Occupancy** variable indicates the occupancy state of the occupancy sensor. A value of 0 indicates unoccupied while a value of 1 indicates occupied.



The web page browser does not update in real time to read a changed value the variable needs to be refreshed by pressing the **Refresh** button.

### Step 4 Monitor the occupancy state

The state will be updated to occupied as soon as the sensor hardware indicates occupation, however the state is only updated to unoccupied when the hardware sensor has reported unoccupied for a period of 30 seconds. The configuration of the occupancy sensor, including the guard periods, can be altered by changing the variables in the OccupancyConfig MIB, these variables are described in detail in [Section 5.3.1 "OccupancyConfig MIB"](#).

The occupancy state is also indicated by the LEDs on the Carrier Board:

- When unoccupied LEDs D3 and D6 are off.
- When occupied LEDs D3 and D6 are on.

---

#### 4.1.6.3 Group Bulb Control from Occupancy Sensor

The lights on the nodes in the WPAN can be controlled from the occupancy sensor devices. The bulbs are turned on and off based upon the measurements taken from the sensors.

When unoccupied the bulbs are turned off, when occupied the bulbs are turned on at full brightness.

The sensors are configured by default to broadcast bulb control commands to a group address unique to each sensor. To configure a bulb to be controlled by a sensor it simply needs to be added to the group the sensor transmits to using the JenNet-IP Browser.

The sensor devices regularly re-transmit their most recent bulb control commands in order to bring any bulbs that are powered cycled back under the control of the device. These commands may conflict with any commands issued through the gateway or from a remote control. It is best practice to remove such bulbs from all groups except the group of sensor that should be controlling the bulb.

Similar issues may arise when placing a bulb under the control of more than one sensor device by adding it to multiple groups, for example controlling a bulb from two occupancy sensors may cause conflicts if one sensor is reporting occupied while the other reports unoccupied. It is possible to configure multiple occupancy sensors to work together and/or with an illuminance sensor to provide combined control, this mode of operation is covered later in the documentation.

### Step 1 Check the bulb control mode

Identify the sensor that should control the bulbs in the Browser's device list and navigate to the OccIIIIBulbConfig MIB for the occupancy sensor device:

NXP IoT Browser

192.168.11.1/Browser.html#tab\_vars

IoT Browser Border Routers Devices MIBs Variables

Mib "OccIIIIBulbConfig" on Node: fd04:bd3:80e8:2:215:8d00:32:dbaa

Mode	Variable Index 0
<input type="text" value="1"/>	<input type="button" value="Set"/> <input type="button" value="Multicast Set"/> <input type="button" value="Refresh"/>
LuminanceDelta	Variable Index 1
<input type="text" value="1"/>	<input type="button" value="Set"/> <input type="button" value="Multicast Set"/> <input type="button" value="Refresh"/>
AdjustInterval	Variable Index 2
<input type="text" value="32"/>	<input type="button" value="Set"/> <input type="button" value="Multicast Set"/> <input type="button" value="Refresh"/>
RefreshInterval	Variable Index 3
<input type="text" value="3000"/>	<input type="button" value="Set"/> <input type="button" value="Multicast Set"/> <input type="button" value="Refresh"/>
Address	Variable Index 4
<input type="text" value="0xff150000000000158d000032dbaaf3f"/>	<input type="button" value="Set"/> <input type="button" value="Multicast Set"/> <input type="button" value="Refresh"/>

JIP Browser version 2.1

This MIB is also used in the illuminance and combined occupancy/illuminance sensor devices and is used to configure the way the bulbs should be controlled by the sensor readings.

The **Mode** variable determines which sensors the sensor device should take into account when controlling bulbs. The default value is set to a value appropriate to the type of sensor device. The following values are available:

- 0: Disable bulb control
- 1: Control bulbs based upon the occupancy sensor, this is the default for occupancy sensor devices, and should be used for this part of the demonstration.
- 2: Control bulbs based upon the illuminance sensor, this is the default for illuminance sensor devices.
- 3: Control bulbs based upon both the occupancy and illuminance sensor, this is the default for combined occupancy/illuminance sensor devices.

## Step 2 Copy the group address the occupancy sensor broadcasts commands to

The **Address** variable contains the group address the sensor device broadcasts its bulb control commands to. The address is derived from the sensor device's MAC address and the MIB ID of the OccIIBulbConfig MIB forming a unique address for each sensor device. Users may change the address if required to manipulate the operation of the system, writing a value of 0 will revert the address back to the default value.

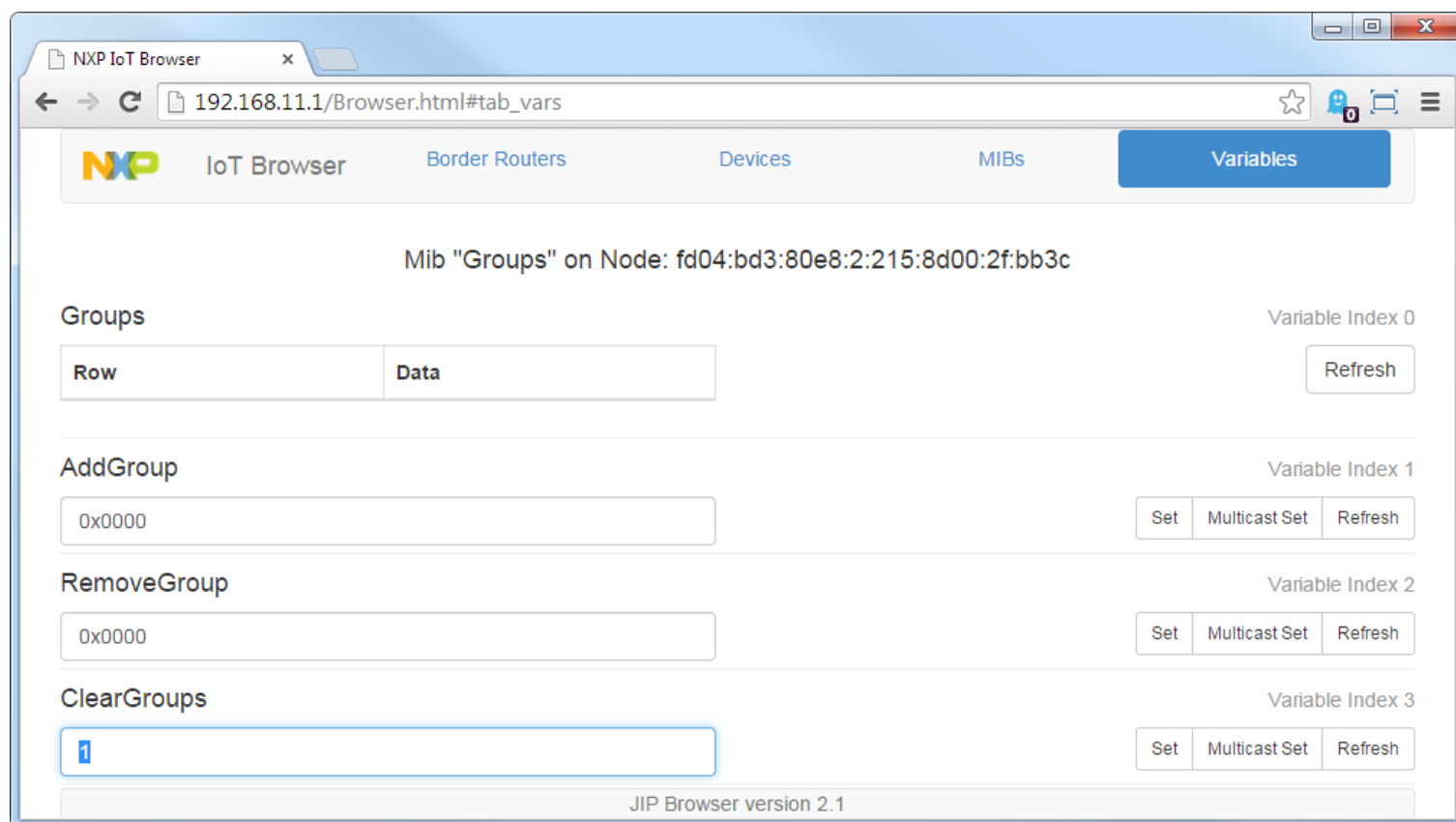
Select and copy the address from the variable's edit box.

## Step 3 Remove bulb from existing groups

Next navigate to the Groups MIB for the bulb to be controlled from the sensor.

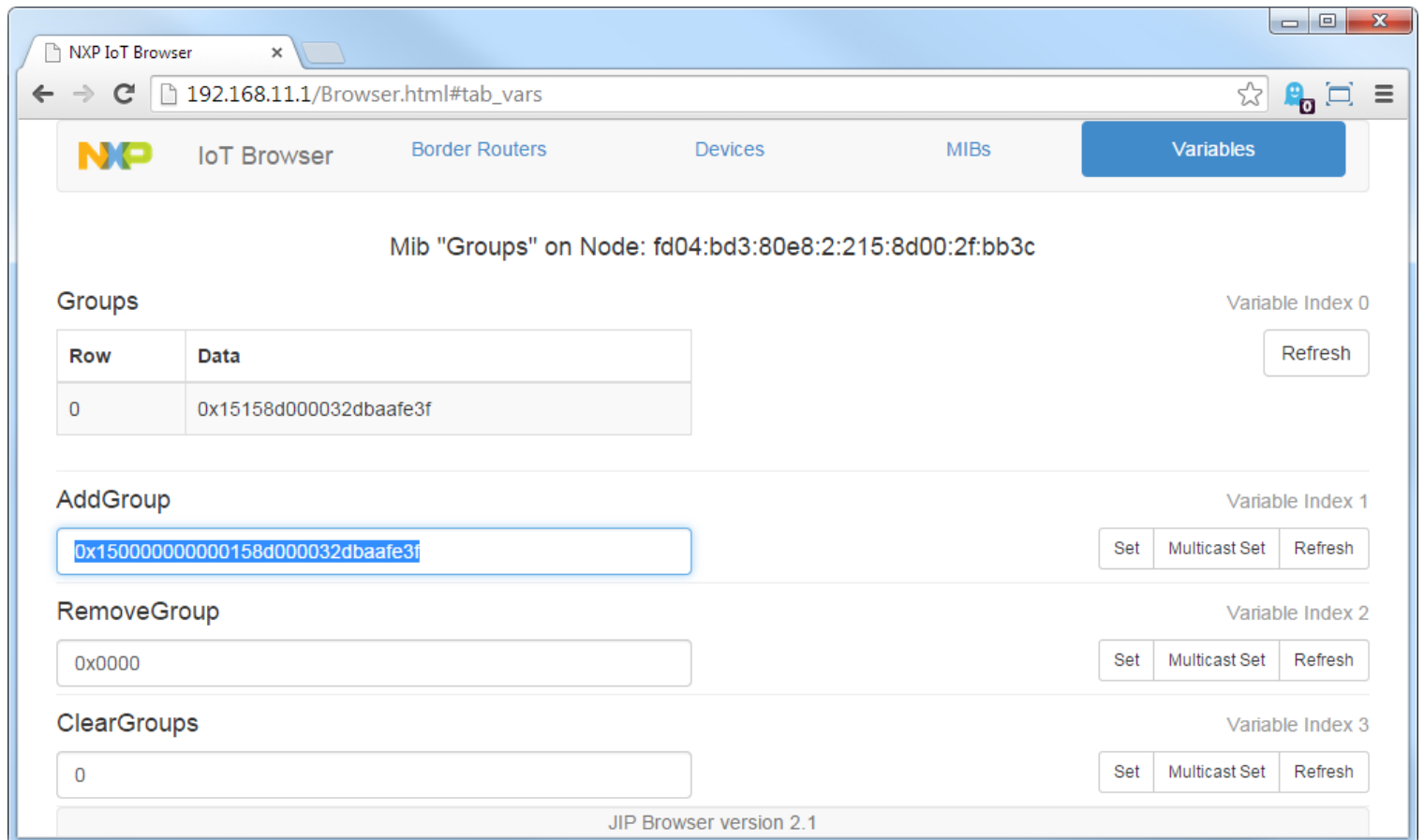
As the sensor regularly transmits commands to its group address it is best to remove the bulb from any existing groups before adding it to the occupancy sensor's control group.

This can be achieved by entering a non-zero value in the **ClearGroups** variable and clicking the **Set** button. Refreshing the **Groups** table should indicate that the bulb has been removed from all groups, as shown in the screenshot below:



#### Step 4 Add bulb to occupancy sensor's control group

Paste the sensor's group address into the **AddGroup** variable's edit box. Delete the "FF" immediately following the "0x" in the pasted value (this is the expected format for group addresses used in the Groups MIB). Click the **Set** button. Check that the bulb has been added to the group by refreshing **Groups** table, (it is normal for intermediate 0s between the initial 15 and the rest of the group address to be stripped out in this table). This final state is shown in the following screenshot:



#### Step 5 Bulb control from occupancy sensor

The bulb should then be under the control of the sensor. Holding down SW4 on the *Generic Expansion Board (DR1199)* or remaining stationary when using the PIR module for approximately 30 seconds should cause the sensor to turn off the light. The light should be turned back on immediately when SW4 is released or motion detected.

These timings and other configuration settings can be adjusted by altering the variables described in [Section 5.3.1 "OccupancyConfig MIB"](#).

The way the sensor interacts with the bulbs can be adjusted by altering the variables described in [Section 5.3.8 "OccIIIBulbConfig MIB"](#).

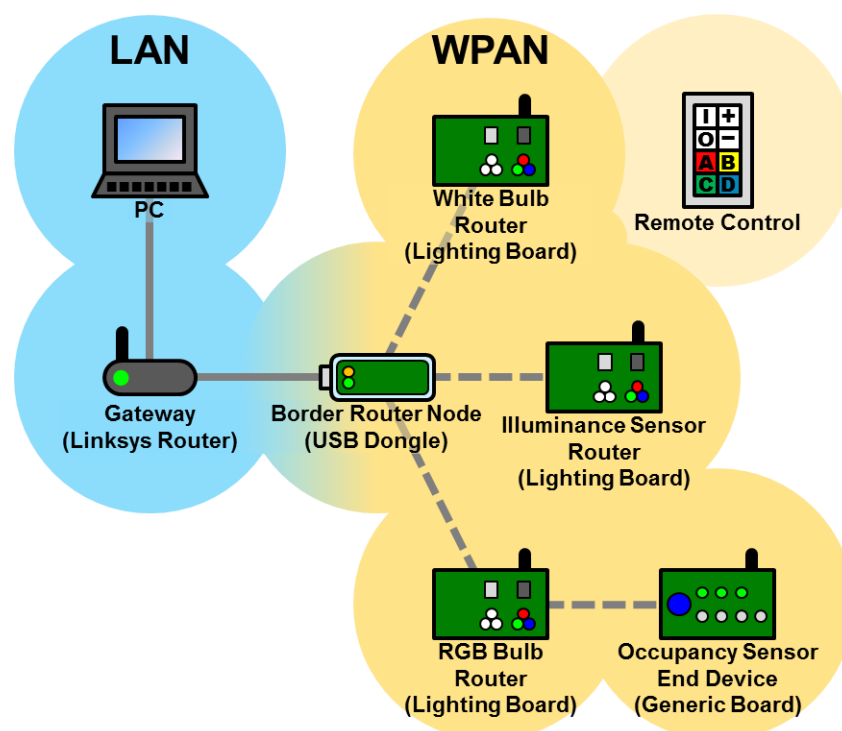
#### Step 6 Add additional bulbs to the occupancy sensor's control group (optional)

Additional bulbs can be added to a sensor's group by repeating the above steps.

### 4.1.7 Operating the Illuminance Sensor

The lights on the nodes of the WPAN can be controlled from within the network using the illuminance sensor. This method of control allows the brightness of the lights to be adjusted to specified level automatically.

This section replaces a bulb in the system created earlier with an illuminance sensor to create the system shown below (the occupancy sensor from the previous section may be left in the network):



#### 4.1.7.1 Setting up the Illuminance Sensor

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

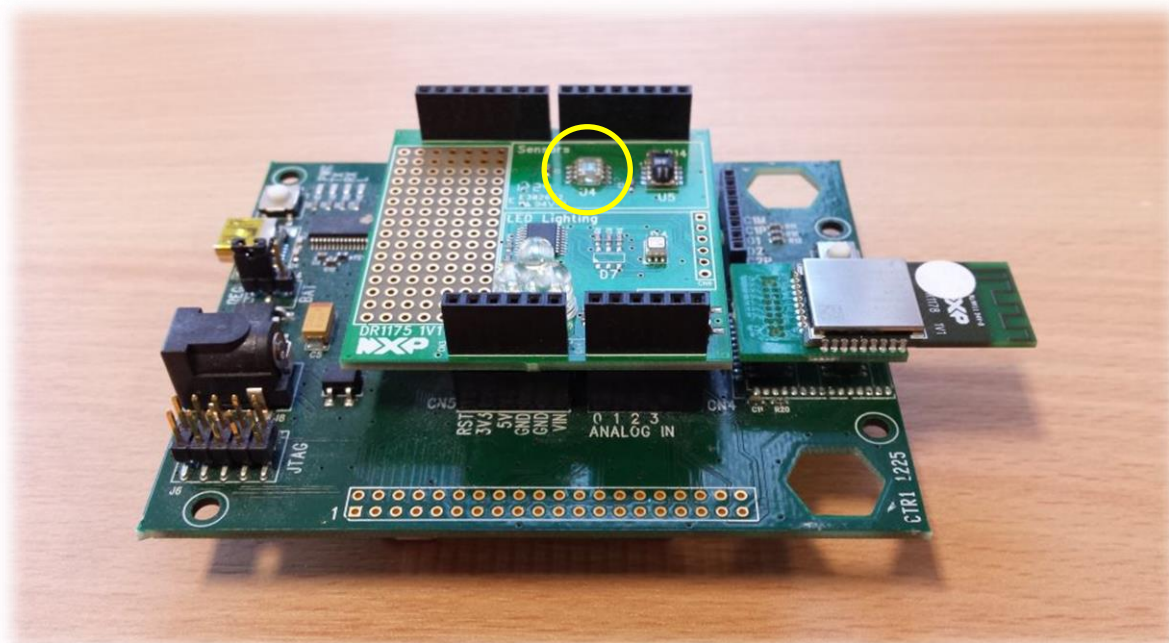
- LAN part of the system
- Bulbs operating in the network.
- Carrier Board fitted with the Lighting/Sensor Expansion Board programmed with appropriate firmware
- Batteries for the above board

When using a single evaluation kit it is suggested that the one of the boards used as a bulb be re-used as the illuminance sensor. In this case it is recommended to erase the contents of the EEPROM during programming and also note that the device will already be white listed in the gateway.

To set up the illuminance sensor part of the system follow the instructions below:

### Step 1 Setup the illuminance sensor hardware

The illuminance sensor software runs on a *Carrier Board (DR1174)* fitted with the *Sensor/Lighting Expansion Board (DR1175)*. The photo-diode marked U4 on the Lighting/Sensor Expansion Board (circled in the image below) is monitored by the application and acts as the illuminance sensor:



### Step 2 Program the illuminance sensor software

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

**0x11111111s\_DeviceSensorIlluminance\_DR1175\_Router\_JN5168\_v0000.bin**  
**0x11111111s\_DeviceSensorIlluminance\_DR1175\_EndDevice\_JN5168\_v0000.bin**

The Router build operates as a Router node in the network, extending the network for other devices to join and so must be permanently powered.

The End Device build operates as an End Device node in the network, End Devices spend the majority of their time asleep and thus are suitable for battery powered operation.

Either build may be used for this part of the demonstration, it is recommended to use the Router build if you intend configure the occupancy sensor and illuminance sensor to co-operate with each other.

### Step 3 Add the illuminance sensor 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"](#).

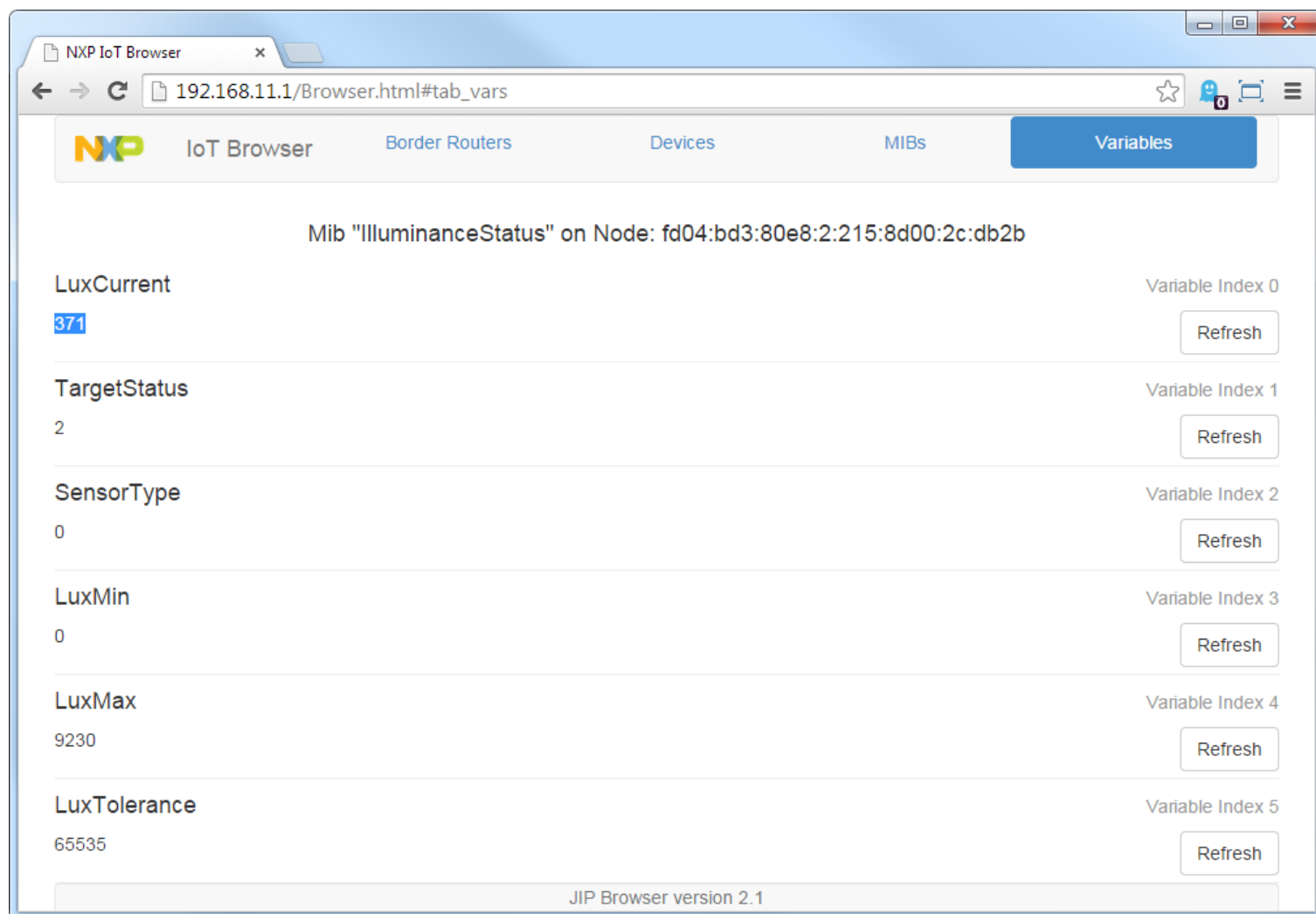
### Step 4 Illuminance sensor feedback

The illuminance sensor does not provide any feedback when it joins a network. The JenNet-IP Browser interface provided by the gateway can be used to check if the illuminance sensor has joined the network.



#### 4.1.7.2 Illuminance Sensor Monitoring from PC

To monitor the status of the illuminance sensor, navigate to the IlluminanceStatus MIB page for the illuminance sensor in the JenNet-IP Browser. The default name for the illuminance sensor begins with an 'I'. This page lists the variables contained in the IlluminanceStatus MIB as illustrated in the screenshot below:



The **LuxCurrent** variable indicates the light level measured by the illuminance sensor in Lux. This value can be changed by shading the sensor, shining a torch on the sensor or altering the ambient lighting levels.



The web page browser does not update in real time use the **Refresh** button to update the **LuxCurrent** value to the most recent reading.



The illuminance sensor also provides a simple state in the form of the **TargetStatus** variable. This value is the result of a comparison between the measured illuminance and a target illuminance level, (set in the IlluminanceControl MIB). The **TargetStatus** variable indicates the following states:

- 0: Sensor disabled
- 1: On target
- 2: Below target
- 3: Above target

The illuminance state is also indicated by the LEDs on the Carrier Board:

- When on target both LEDs D3 and D6 are on
- When below target LED D3 is on and LED D6 is off
- When above target LED D3 is off and LED D6 is on

### 4.1.7.3 Illuminance Sensor Control from PC

To alter the target illuminance - navigate to the IlluminanceControl MIB using the JenNet-IP Browser as shown in the following illustration:

The screenshot shows the NXP IoT Browser interface. The address bar displays '192.168.11.1/Browser.html#tab\_vars'. The main navigation bar includes 'IoT Browser', 'Border Routers', 'Devices', 'MIBs', and 'Variables'. The title of the page is 'Mib "IlluminanceControl" on Node: fd04:bd3:80e8:2:215:8d00:2c:db2b'. The variables are listed as follows:

Variable Name	Value	Variable Index	Set	Multicast Set	Refresh
Mode	1	Variable Index 0	Set	Multicast Set	Refresh
Sceneld	0	Variable Index 1	Set	Multicast Set	Refresh
LuxTarget	750	Variable Index 2	Set	Multicast Set	Refresh
LuxBand	50	Variable Index 3	Set	Multicast Set	Refresh
LuxAdjust	0	Variable Index 4	Set	Multicast Set	Refresh
LuxTargetChange	0	Variable Index 5	Set	Multicast Set	Refresh
LuxBandChange	0	Variable Index 6	Set	Multicast Set	Refresh

The footer of the browser window indicates 'JIP Browser version 2.1'.

The **LuxTarget** variable is used to set the target illuminance in Lux that the sensor should compare its readings against. The **LuxBand** variable is used to set the width of the target band in Lux. Additional variables may be used to adjust the target band in different ways, these variables are described in detail in [Section 5.3.7 "IlluminanceControl MIB"](#).

The comparison of the measured light level against the target band is used when controlling the brightness of lights in order to illuminate an area to a specified brightness.

#### 4.1.7.4 Group Bulb Control from Illuminance Sensor

The lights on the nodes in the WPAN can be controlled from the illuminance sensor devices. When the illuminance is lower than the target band the bulbs are turned on and their brightness increased over time. When the illuminance is too high the bulbs brightness is decreased over time and if the minimum brightness is reached they are turned off.

In order to demonstrate the control of bulbs by an illuminance sensor it may be necessary to shine a torch on the sensor to simulate bright conditions or cover the sensor to simulate dark conditions and so cause a change in the bulb's brightness. In a true lighting system an illuminance sensor should be placed so that the light from the bulbs being controlled falls upon the illuminance sensor thus allowing the target band to be reached.

The illuminance sensors control the bulbs in exactly the same way as the occupancy sensor, broadcasting commands to the bulbs using a group address unique to the illuminance sensor.

The procedure for controlling bulbs from the illuminance sensor is the same as that for the occupancy sensor, though a different control mode is specified that also adjusts the bulbs brightness based upon the sensor reading.

##### Step 1 Check the bulb control mode

Identify the sensor that should control the bulbs in the Browser's device list and navigate to the OccIIIIBulbConfig MIB for the illuminance sensor device:

The screenshot shows the NXP IoT Browser interface. The address bar displays '192.168.11.1/Browser.html#tab\_vars'. The main navigation bar includes 'IoT Browser', 'Border Routers', 'Devices', 'MIBs', and 'Variables'. The current view is for the MIB 'OccIIIIBulbConfig' on Node: fd04:bd3:80e8:2:215:8d00:2c:db2b. The configuration fields are as follows:

Field	Value	Variable Index	Buttons
Mode	2	Variable Index 0	Set, Multicast Set, Refresh
LuminanceDelta	1	Variable Index 1	Set, Multicast Set, Refresh
AdjustInterval	32	Variable Index 2	Set, Multicast Set, Refresh
RefreshInterval	3000	Variable Index 3	Set, Multicast Set, Refresh
Address	0xff150000000000158d00002c:db2bfe3f	Variable Index 4	Set, Multicast Set, Refresh

JIP Browser version 2.1

The default **Mode** variable value should be set to 2, indicating that bulbs should be controlled based upon readings from the illuminance sensor.

**Step 2 Copy the group address the illuminance sensor broadcasts commands to**

The **Address** variable contains the group address the sensor device broadcasts its bulb control commands to. The address is derived from the sensor device's MAC address and the MIB ID of the `OccllIBulbConfig` MIB forming a unique address for each sensor device. Users may change the address if required to manipulate the operation of the system, writing a value of 0 will revert the address back to the default value.

Select and copy the address from the variable's edit box.

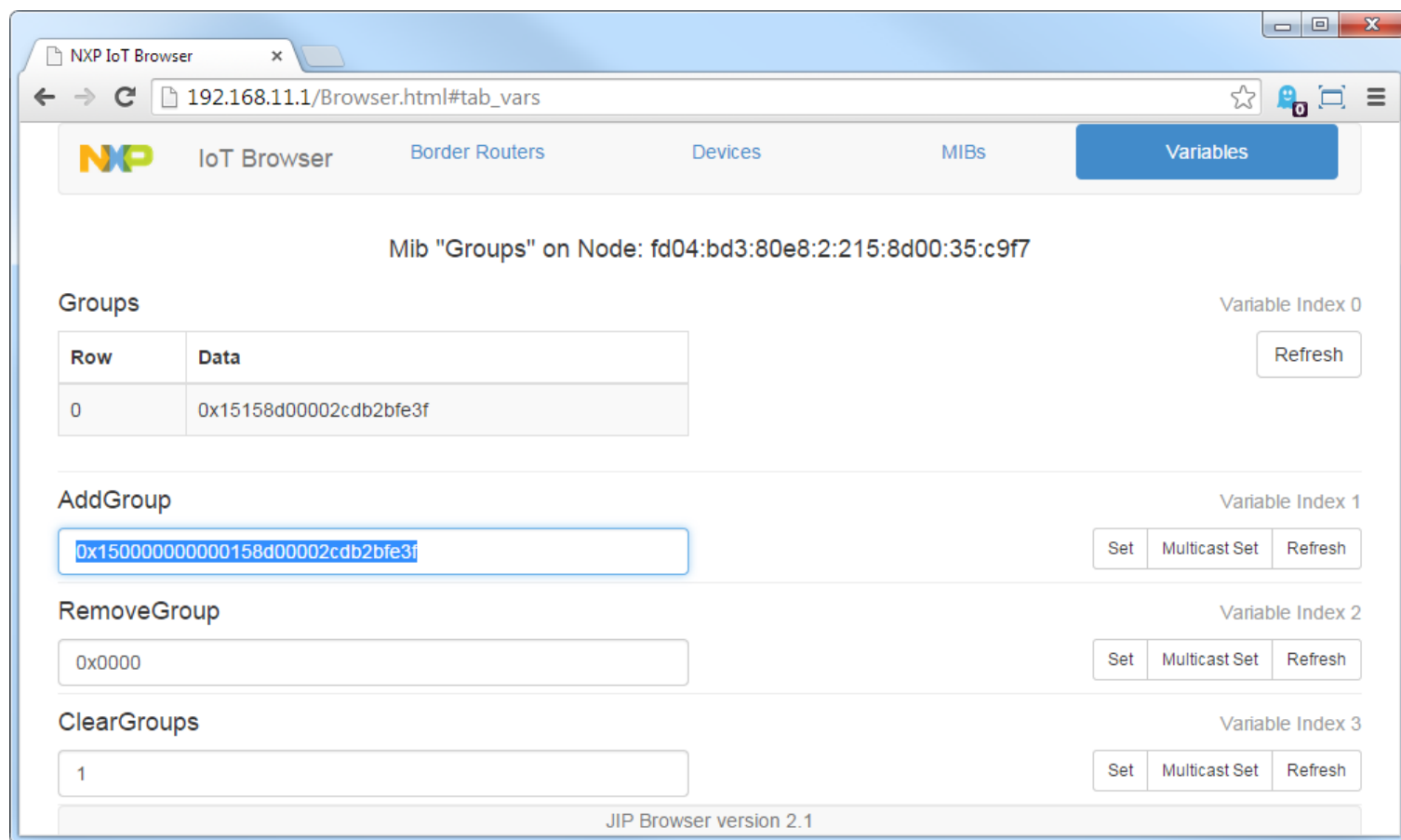
**Step 3 Remove bulb from existing groups**

Next navigate to the Groups MIB for the bulb to be controlled from the sensor.

As the sensor regularly transmits commands to its group address it is best to remove the bulb from any existing groups before adding it to the Illuminance Sensor's control group. This can be achieved by entering a non-zero value in the **ClearGroups** variable and clicking the **Set** button.

**Step 4 Add bulb to occupancy sensor's control group**

Paste the sensor's group address into the **AddGroup** variable's edit box. Delete the "FF" immediately following the "0x" in the pasted value (this is the expected format for group addresses used in the Groups MIB). Click the **Set** button. Check that the bulb has been added to the group by refreshing **Groups** table, (it is normal for intermediate 0s between the initial 15 and the rest of the group address to be stripped out in this table). This final state is shown in the following screenshot:



### Step 5 Bulb control from illuminance sensor

The bulb should then be under the control of the sensor.

The default target brightness is set quite high, it may be necessary to shine a torch on the light sensor, move it closer to light source or lower the target level to observe the light sensor altering the brightness of the bulb device.

The default configuration also adjusts the brightness of the light very slowly it may be necessary to illuminate the sensor for some time before the effect is noticed. This behaviour can be altered by adjusting the variables in the sensor's `OccIIIIBulbConfig` MIB, see [Section 5.3.8 "OccIIIIBulbConfig MIB"](#) for full details.

In a properly set up system a brighter light source than the LEDs used on the Lighting/Sensor Expansion Board would be used and the sensor sited so that the light falls upon the illuminance sensor.

### Step 6 Add additional bulbs to the illuminance sensor's control group (optional)

Additional bulbs can be added to a sensor's group by repeating the above steps.

---

### 4.1.8 Co-operating Occupancy and Illuminance Sensors

The occupancy sensors can be configured to send their occupancy state to other occupancy sensors, illuminance sensors or combined occupancy/illuminance sensors. The receiving sensors can then take the occupancy state of many external occupancy sensors into account when controlling bulbs:

- When *any* of the monitored occupancy sensors (or the local occupancy sensor) report occupied the whole area covered by all the co-operating sensors is considered occupied and bulbs are controlled appropriately.
- When *all* of the monitored occupancy sensors (including the local occupancy sensor) report unoccupied the whole area covered by the co-operating sensors is considered unoccupied and bulbs are controlled appropriately.

The following steps describe how to configure an occupancy sensor to provide its occupancy state to an illuminance sensor in order to control bulbs based upon a combination of the two sensors.

The same steps may be followed to configure the monitoring of additional occupancy sensors by the same illuminance sensor or to configure the occupancy sensor reporting to another occupancy sensor or an occupancy/illuminance sensor.

---

#### 4.1.8.1 Setting up the Co-operating Sensors

In setting up the co-operating sensors part of the demo system, you will need the following components:

- LAN part of the system
- Bulbs operating in the network
- Occupancy sensor operating in the network
- Illuminance sensor operating in the network

To set up the co-operating sensors part of the system follow the instructions below:

## Step 1 Occupancy sensor transmission configuration

First the occupancy sensor needs to be configured to broadcast its occupancy state to a group address. Navigate to the OccupancyConfig MIB of the occupancy sensor in the JenNet-IP browser, as shown in the following screenshot (with the default settings):

NXP IoT Browser

192.168.11.1/Browser.html#tab\_vars

NXP IoT Browser Border Routers Devices MIBs Variables

Mib "OccupancyConfig" on Node: fd04:bd3:80e8:2:215:8d00:32:dbaa

Sensitivity	Variable Index 0
<input type="text" value="255"/>	<input type="button" value="Set"/> <input type="button" value="Multicast Set"/> <input type="button" value="Refresh"/>
UnoccupiedDelay	Variable Index 1
<input type="text" value="3000"/>	<input type="button" value="Set"/> <input type="button" value="Multicast Set"/> <input type="button" value="Refresh"/>
OccupiedDelay	Variable Index 2
<input type="text" value="0"/>	<input type="button" value="Set"/> <input type="button" value="Multicast Set"/> <input type="button" value="Refresh"/>
OccupiedEvents	Variable Index 3
<input type="text" value="0"/>	<input type="button" value="Set"/> <input type="button" value="Multicast Set"/> <input type="button" value="Refresh"/>
StateMibId	Variable Index 4
<input type="text" value="0"/>	<input type="button" value="Set"/> <input type="button" value="Multicast Set"/> <input type="button" value="Refresh"/>
StateVarIdx	Variable Index 5
<input type="text" value="255"/>	<input type="button" value="Set"/> <input type="button" value="Multicast Set"/> <input type="button" value="Refresh"/>
StateAddress	Variable Index 6
<input type="text" value="0xff150000000000158d000032dbaafe31"/>	<input type="button" value="Set"/> <input type="button" value="Multicast Set"/> <input type="button" value="Refresh"/>
StateRefresh	Variable Index 7
<input type="text" value="3000"/>	<input type="button" value="Set"/> <input type="button" value="Multicast Set"/> <input type="button" value="Refresh"/>

JIP Browser version 2.1

The occupancy sensor's default configuration disables the transmission of the occupancy state in order to prevent unneeded data being transmitted in the network.

To enable the transmission a non-zero value needs to be entered into the **StateMibId** variable which defines the MIB ID to write the occupancy state into when transmitting. All of the sensor devices include an OccupancyMonitor MIB to receive such updates with an MIB ID of 0xFFFFFE32. Enter "0xFFFFFE32" into the **StateMibId** variable and click the **Set** button to enable transmission of the occupancy state, (as shown in the screenshot below).

The occupancy sensor's default configuration also uses an invalid variable index to write the data to. To correctly configure the transmission the variable index of the OccupancyMonitor MIB's **Occupancy** variable's index needs to be specified which has an index value of 2. Enter "2" into the into the **StateVarIdx** variable and click the **Set** button to enable transmission, (also shown in the screenshot below).

The screenshot shows the NXP IoT Browser interface. The address bar displays '192.168.11.1/Browser.html#tab\_vars'. The 'Variables' tab is selected in the top navigation bar. The main content area is titled 'Mib "OccupancyConfig" on Node: fd04:bd3:80e8:2:215:8d00:32:dbaa'. It lists several variables with their current values and control buttons (Set, Multicast Set, Refresh):

Variable Name	Value	Variable Index
Sensitivity	255	0
UnoccupiedDelay	3000	1
OccupiedDelay	0	2
OccupiedEvents	0	3
StateMibId	0xfffffe32	4
StateVarIdx	2	5
StateAddress	0xff150000000000158d000032dbaaf31	6
StateRefresh	3000	7

The bottom of the interface shows 'JIP Browser version 2.1'.



This same configuration should be used in all the external occupancy sensor devices to be monitored. The receiving device will take the source address of the received message into account in order to monitor multiple occupancy sensors.

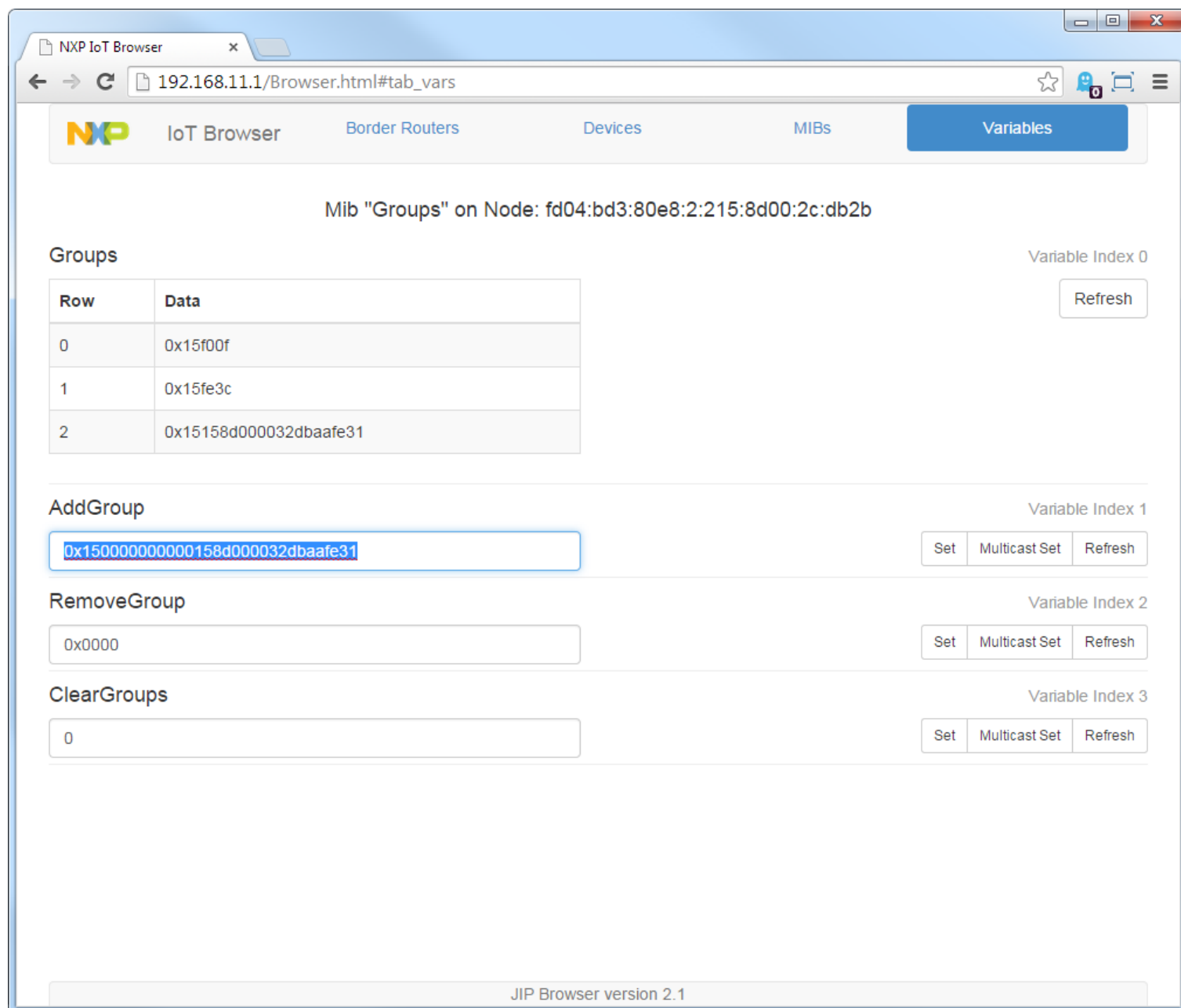
Finally the occupancy sensors are configured to transmit their occupancy status to a group address derived from the MAC address of the device and the MIB ID of the OccupancyConfig MIB, thus creating a group address unique to the sensor. This group address is specified in the **StateAddress** variable. Select this value and copy it for use in configuring the receiving sensor.

The use of a group address allows many sensors to receive the occupancy status of a single occupancy sensor if required. However the transmission address may also be changed by the user to adapt the behaviour of this feature.

## Step 2 Illuminance sensor group configuration

The receiving illuminance sensor needs to be configured to receive the transmissions from the occupancy sensor.

The first step is to add the illuminance sensor to the group the occupancy sensor is transmitting its command to. Navigate to the Groups MIB for the illuminance sensor as shown in the following screenshot:

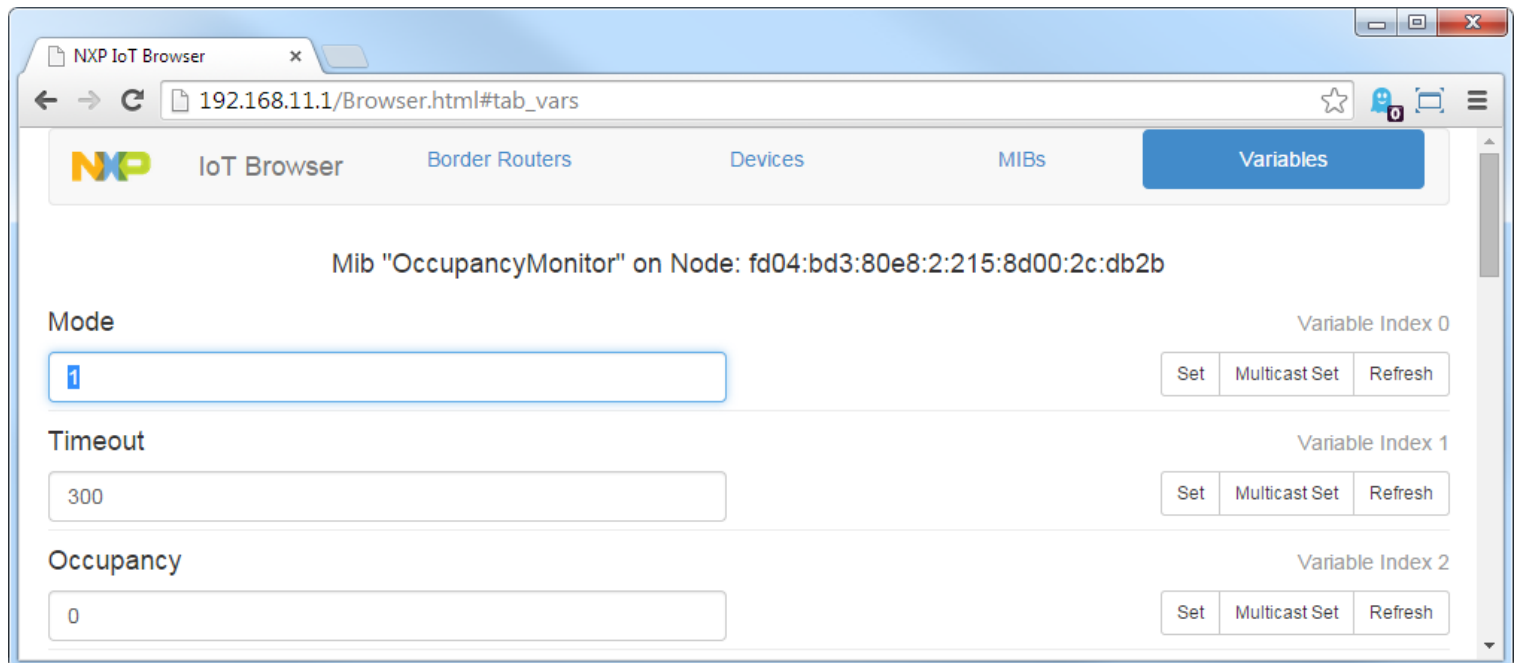


Paste the occupancy sensor's group address into the **AddGroup** variable's edit box. Delete the "FF" immediately following the "0x" in the pasted value to enter the group address with the correct formatting. Click the **Set** button.

The illuminance sensor should now be able to receive the broadcasts from the occupancy sensor.

### Step 3 Illuminance sensor monitoring configuration

The next step is to enable the illuminance sensor to monitor the external occupancy sensor, (as this is disabled by default). Navigate to the OccupancyMonitor MIB for the receiving sensor as shown in the screenshot below:



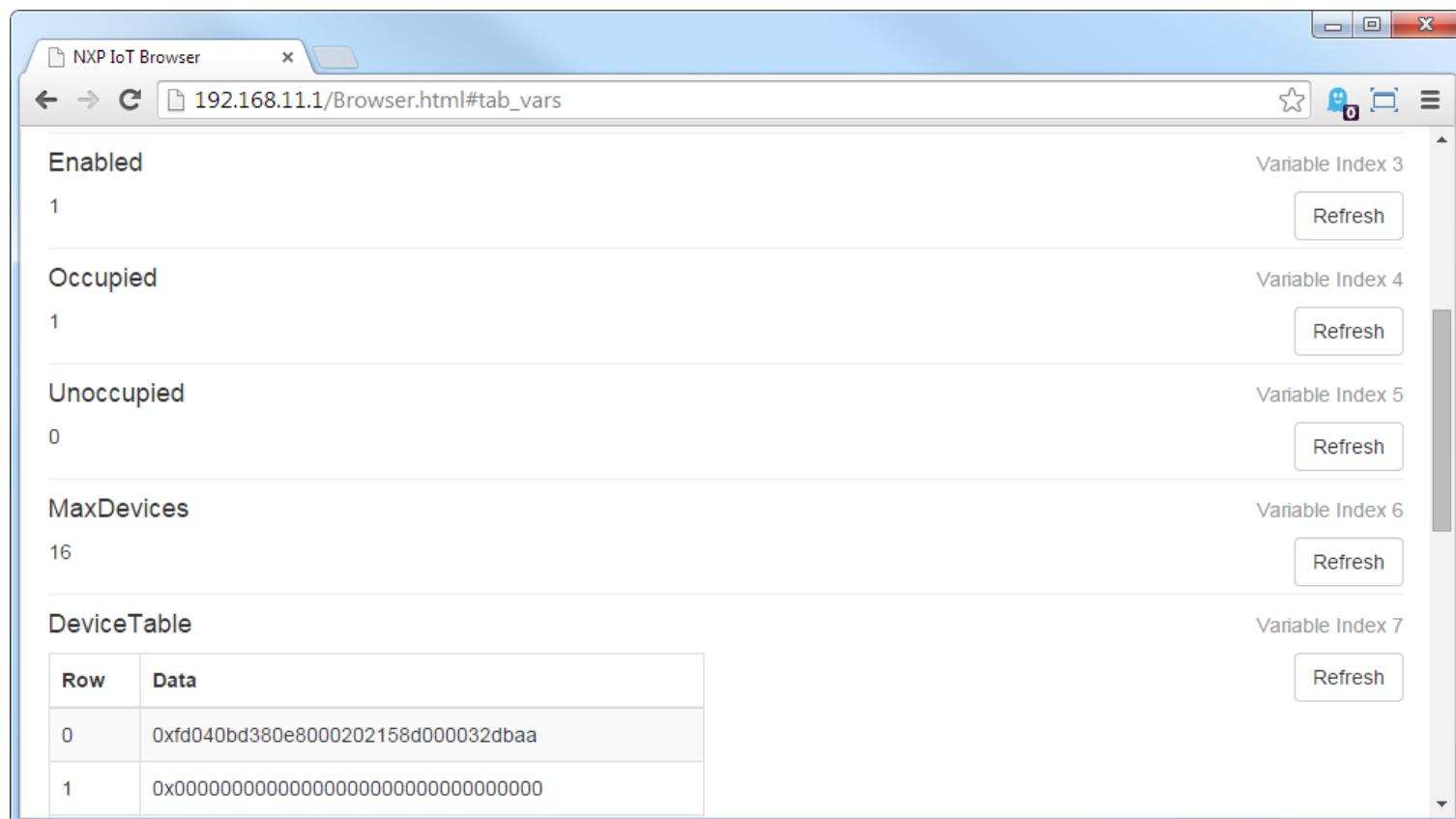
Change the value of the **Mode** variable from “0” to “1” then click the **Set** button to enable monitoring of external occupancy sensors.

The occupancy sensors regularly re-transmit their occupancy status even when it remains unchanged. If a monitored occupancy sensor fails to send an update within the period defined by the **Timeout** variable (in seconds) it is considered lost and no longer taken into account when calculating the occupancy of the area covered by the sensors.

The **Occupancy** variable is used to receive the occupancy state from all the occupancy sensors being monitored.

#### Step 4 Illuminance sensor monitoring operation

Once the OccupancyMonitor MIB is enabled it should start receiving the occupancy status from the external occupancy sensors. The status of the external occupancy sensors can be monitored using the remaining variables of the OccupancyMonitor MIB as shown in the screenshot below:



The **Enabled** variable is a bitmask, with each bit indicating if an external occupancy sensor is being monitored.

The **Occupied** and **Unoccupied** variables are also bitmasks with each bit indicating if an external occupancy sensor state is occupied or unoccupied respectively.

The **MaxDevices** variable indicates the maximum number of external occupancy sensors that can be monitored.

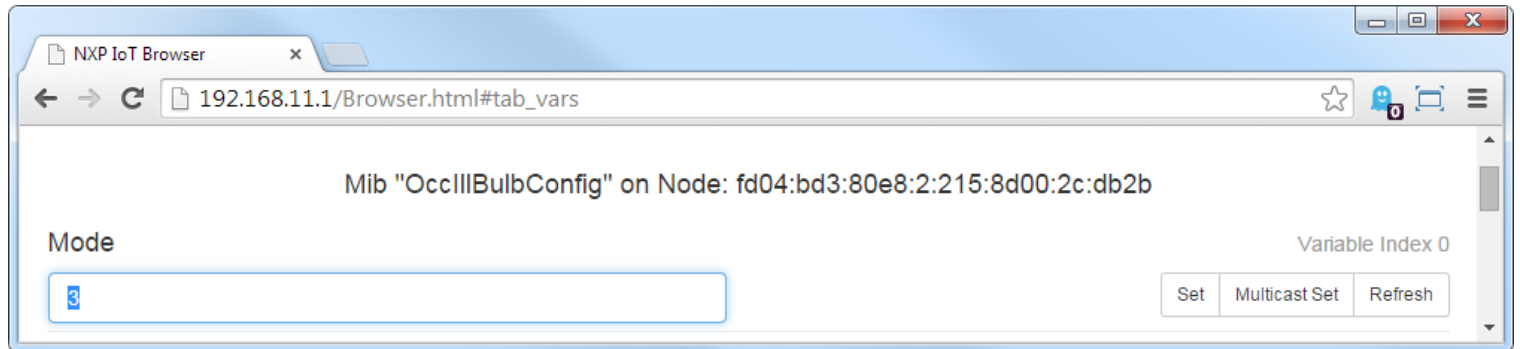
The **DeviceTable** variable lists the IPv6 Addresses of the external occupancy sensors currently being monitored. Entry 0 corresponds to bit 0 in the bitmasks used for the **Enabled**, **Occupied** and **Unoccupied** variables.

It may be necessary to refresh the variables or the entire page to update these variables.

### Step 5 Illuminance sensor bulb control configuration

The final step, for a receiving illuminance sensor, is to configure bulb control to take the occupancy state into account when controlling bulbs, (as the default configuration only uses the illuminance state).

Navigate to the OccIllBulbConfig MIB for the receiving illuminance sensor as shown in the screenshot below:



Change the default “2” value, (illuminance only control), to a “3” to enable control of the bulbs based upon both the local illuminance state and external occupancy states.

The bulb configured for control by the illuminance sensor in the previous section should now be controlled by a combination of the occupancy and illuminance sensors readings as follows:

- When *all* of the occupancy sensors report unoccupied the bulbs are turned off.
- When *any* of the occupancy sensors report occupied the brightness of the bulbs is controlled to bring the illuminance into the target illuminance band, (which may include turning the bulbs on or off).

The bulb originally configured to be controlled by the occupancy sensor will continue to be controlled by the occupancy sensor alone.

### Step 6 Extending the system with additional bulbs and sensors

Additional bulbs can be brought under the control of appropriate sensors by altering the groups they are enrolled in.

If the control of bulbs by the occupancy sensor alone is no longer required the bulb control commands can be disabled to avoid unnecessary radio communications.

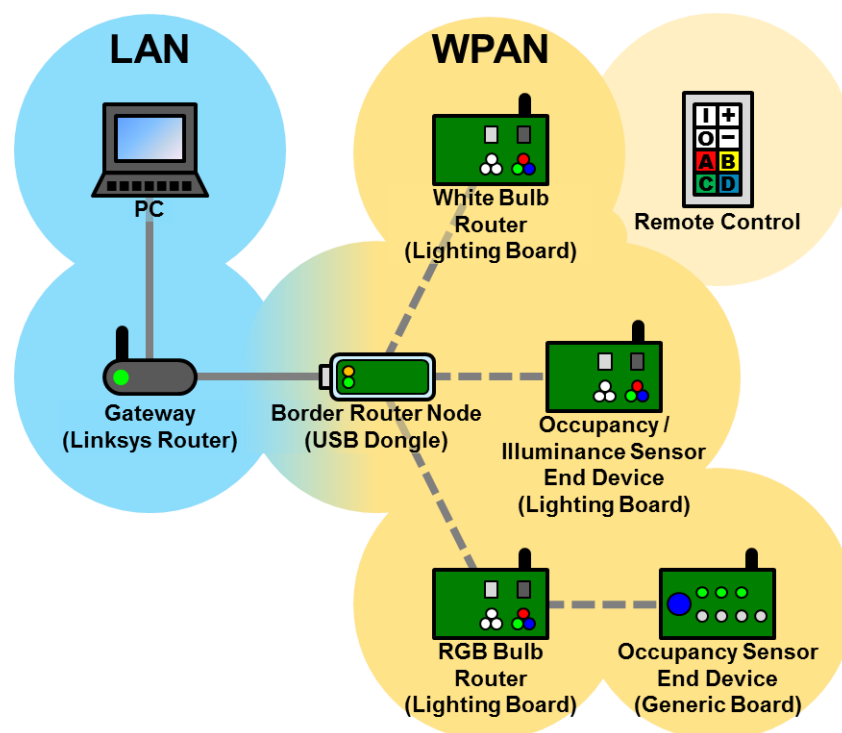
Additional occupancy sensors can provide their readings to the illuminance sensor by repeating the appropriate steps above.

Occupancy sensors are also able to provide their readings to other occupancy sensors and also the combined occupancy/illuminance sensor described in the following section.

### 4.1.9 Operating the Combined Occupancy/Illuminance Sensor

The combined occupancy/illuminance sensor device contains all the MIBs that are in the individual occupancy sensor and illuminance sensor devices, thus providing the functionality of both devices in a single device.

This section replaces the illuminance sensor in the system created earlier with a combined occupancy/illuminance sensor to create the system shown below (the occupancy sensor from the previous section may be left in the network):



#### 4.1.9.1 Operating the Combined Sensor

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

- LAN part of the system
- Bulbs operating in the network.
- *Carrier Board (DR1174)* fitted with the *Lighting/Sensor Expansion Board (DR1175)*
- (Optional) Parallax PIR module
- Batteries for the above board

When using a single evaluation kit it is suggested that the board used as the illuminance sensor be re-used as the combined sensor.

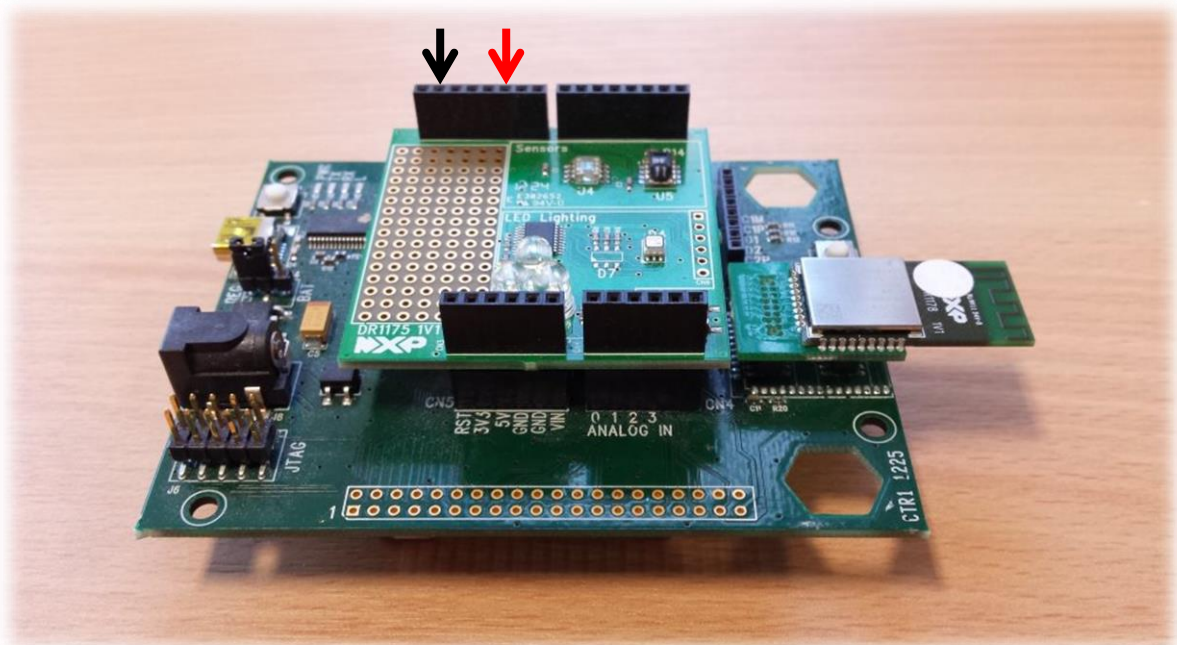
To set up the combined sensor part of the system follow the instructions below:

### Step 1 Setup combined sensor hardware

Two hardware configurations are available for the combined sensor:

#### a) Combined sensor using Lighting/Sensor Expansion Board only

A Carrier Board fitted with the Lighting/Sensor Expansion Board may be used when only the evaluation kit hardware is available for use. As there are no buttons on this expansion board the appropriate input on the DIO header must be used instead. Inserting a wire into the input on CN2 marked 10 (indicated by a red arrow in the photo below) will allow the input to be connected to ground (for unoccupied marked by a black arrow below) or left to float high (for occupied).





**b) Combined sensor using Lighting/Sensor Expansion Board and PIR module**

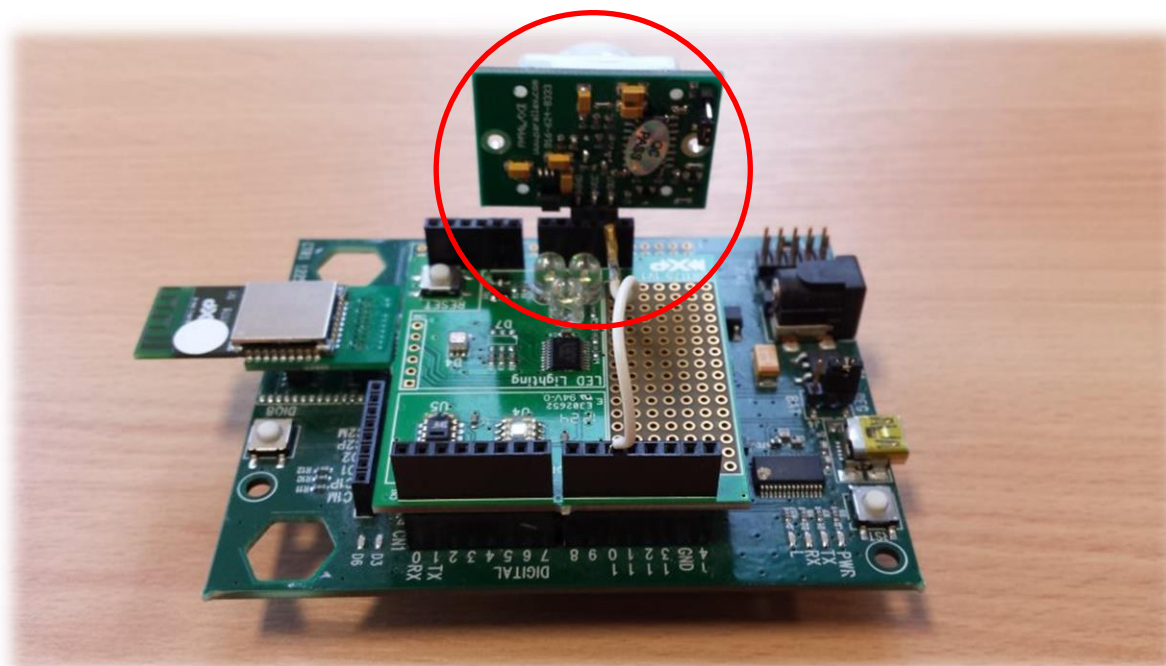
A Carrier Board fitted with both a Lighting/Sensor Expansion Board and the following Parallax PIR module (item code: 555-28027) may be used to create an combined sensor using a real PIR sensor, (the Parallax PIR module is not supplied as part of the evaluation kit):

<http://www.parallax.com/Store/Sensors/ObjectDetection/tabid/176/CategoryID/51/List/0/SortField/0/Level/a/ProductID/83/Default.aspx>

Install a Parallax PIR Module onto the Carrier Board as follows:

- Bend the OUT pin of the PIR module back 90 degrees.
- Insert the GND and VCC pins into the GND and 5V sockets of the CN3 connector of the Carrier Board.
- Connect the OUT pin to socket 10 (DIO 1) on the CN2 connector of the Carrier Board.

The following image shows the Parallax PIR module fitted to the Carrier Board and Lighting/Sensor Expansion Board combination:





## Step 2 Program the combined sensor software

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

**0x11111111s\_DeviceSensorOccupancyIlluminance\_DR1175\_Router\_JN5168\_v0000.bin**  
**0x11111111s\_DeviceSensorOccupancyIlluminance\_DR1175\_EndDevice\_JN5168\_v0000.bin**

The Router build operates as a Router node in the network, extending the network for other devices to join and so must be permanently powered.

The End Device build operates as an End Device node in the network, End Devices spend the majority of their time asleep and thus are suitable for battery powered operation.

Either build may be used for this part of the demonstration, if you intend to experiment further by monitoring other occupancy sensors in the combined sensor it is recommended to use the Router build to allow it to receive the broadcast messages from the occupancy sensors.



*It may be necessary to remove the Parallax PIR module during programming, depending upon the power supply used, as it draws additional power.*

## Step 3 Add the combined sensor 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"](#).

## Step 4 Combined sensor feedback

The occupancy sensor does not provide any feedback when it joins a network. The JenNet-IP Browser interface provided by the gateway can be used to check if the occupancy sensor has joined the network.

---

### 4.1.9.2 Combined Sensor Monitoring from PC

To monitor the status of a combined occupancy/illuminance sensor, navigate to the OccupancyStatus MIB to monitor the occupancy state or the IlluminanceStatus MIB to monitor the illuminance state. These MIBs operate the same way as those for the individual sensor devices. The combined sensor's default name begins with an 'S' to allow identification in the Devices tab.

The status of the combined device is indicated by the LEDs on the Carrier Board:

- When unoccupied LEDs D3 and D6 are off
- When occupied and on target LEDs D3 and D6 are on
- When occupied and below target LED D3 is on and LED D6 is off
- When occupied and above target LED D3 is off and LED D6 is on

#### 4.1.9.3 Combined Sensor Control from PC

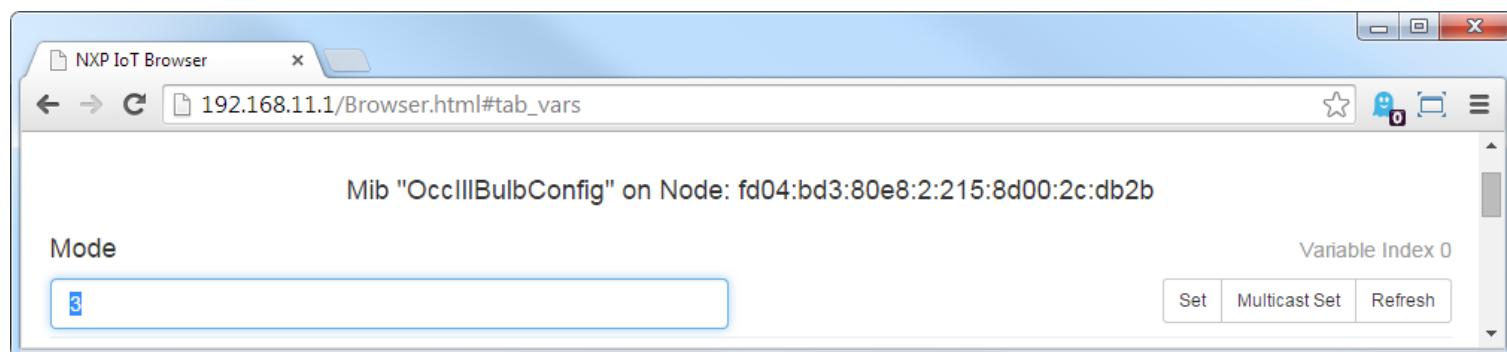
The target illuminance is altered in exactly the same way as in the illuminance sensor by changing the values of the **LuxTarget** and **LuxBand** variables in the IlluminanceControl MIB.

#### 4.1.9.4 Group Bulb Control from Combined Sensor

The lights on the nodes in the WPAN can be controlled from the combined sensor devices in the following way:

- When unoccupied the lights are turned off, (in the same way as the occupancy sensor device)
- When occupied the brightness of the lights is controlled to reach a specified target level, (in the same way as the illuminance sensor device)

Configuring bulbs for control from the combined sensor is identical to the configuration for the individual occupancy sensor and illuminance sensor. The **Mode** variable in the OccIllBulbConfig MIB needs to be set to 3 (the default value) to control bulbs based upon both the occupancy and illuminance readings, as shown in the following screenshot:



This **Mode** variable can also be altered to control the bulbs based upon the occupancy reading alone or the illuminance reading alone, in the same way as the individual sensor devices.

---

## 4.2 Standalone System Operation

The chapter describes how to use the contents of an evaluation kit to set up and run the JenNet-IP Smart Home demonstration in 'standalone' mode. This demonstration is based on an isolated WPAN with nodes containing lights, which can be controlled wirelessly from a remote control.

The standalone version of the JenNet-IP Smart Home demonstration does not provide IP connectivity. Therefore, unlike in the version of the demo described in [Section 4.1 "Gateway System Operation"](#), the standalone WPAN cannot be monitored and controlled from a remote PC via an IP connection. In practice, a JenNet-IP system may be developed as a standalone system that can be extended to a full system with IP connectivity by simply adding a gateway - for example, an entry-level lighting system may be sold as a standalone system consisting of lamps and a remote control until, with the potential to add IP connectivity by purchasing an optional border router.

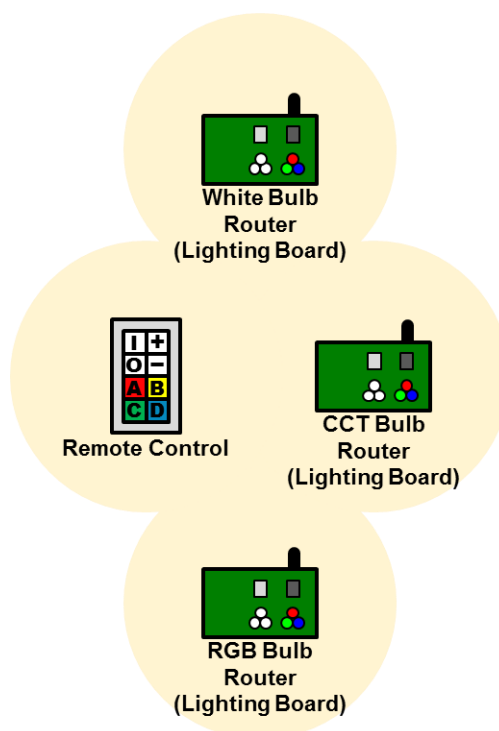
---

### 4.2.1 Standalone System Operation Overview

In the standalone version of the JenNet-IP Smart Home demonstration, lights on the nodes of a WPAN can be controlled from a remote control within the WPAN. The components of the evaluation kit used in the demonstration are as follows:

- **Carrier Boards with Lighting/Sensor Expansion Boards:** The four *Carrier Boards (DR1174)* supplied in the kit are pre-fitted with *Lighting/Sensor Expansion Boards (DR1175)* or *Generic Expansion Boards (DR1199)* 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. These boards are used to run the following devices:
  - **Bulb device:** The white LEDs or the RGB LED (on the Lighting/Sensor Expansion Boards) or LED2 (on the Generic Expansion Board) are the lights to be controlled.
- **Remote Control:** The remote control acts as a node of the WPAN. In the standalone version of the demo, described in this chapter, the unit can act as the Coordinator which creates the WPAN, but normally acts as a 'sleeping broadcaster'. In the latter mode, the device sleeps and only wakes when it is needed to broadcast control commands (and does not have the role of a conventional WPAN node). If sleeping, the unit must be activated using the Wake button below the keypad before any other keys are pressed.

The standalone version of the JenNet-IP Smart Home demo system is illustrated in the figure below:



During WPAN formation, the remote control is enabled as a Coordinator through which other nodes join the WPAN. However, once the network has been formed, the remote control is used in its default mode as a 'sleeping broadcaster'. Commands from the remote control will then be broadcast to nodes, so routing down the WPAN tree will not be adhered to.



Note: The remote control can be put into Coordinator mode for node commissioning (through the key sequence PRG ON OFF ON GRP). It will remain in this mode for 5 minutes, but the mode can be exited at any time by pressing the Wake button (below the keypad).

#### 4.2.1.1 Lighting Control from the Remote Control

In this demonstration, control commands will be entered into the remote control and wirelessly broadcast (in JenNet-IP packets) to the WPAN nodes. A command can be addressed to all nodes or to a pre-defined group of nodes.

A complete list of the operations that can be performed from the keypad is provided in [Section 4.1.4.4 "Remote Control Command Tables"](#). The table below provides a summary of the use of individual keys by the demonstration and the figure shows the keypad of the remote control.



Note: In this manual, operations are generally described as function sequences followed by the key sequences in square brackets - for example: PRG DOWN UP DOWN [# - + -]

Key	Function	Description
I	ON	Switch on light(s)
O	OFF	Switch off light(s)
+	UP	Increase brightness of light(s)
-	DOWN	Decrease brightness of light(s)
#	PRG	Programming mode
*	ALL	All groups
A	GRP	Group A
B		Group B
C		Group C
D		Group D
•	Wake	Wake Remote Control Unit from sleep



The remote control normally operates as a 'sleeping broadcaster'. Thus, the unit sleeps until it is needed. If sleeping, the unit can be activated using the Wake (circle) button below the keypad. Once woken, the unit remains active for 10 minutes following the last key press before going back to sleep. When the unit is active, pressing any key will cause the left LED to momentarily illuminate (if this does not happen, you must first activate the unit using the Wake button).

---

## 4.2.2 Setting Up the Standalone System

This section describes how to set up the standalone demo system using the evaluation kit remote control.

---

### 4.2.2.1 Setting Up the Remote Control

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

- Remote control programmed with appropriate firmware
- Batteries for the above board

To set up the remote control part of the system follow the instructions below:

### Step 1 Program the remote control software

The following pre-built binary is provided in the Application Note for use on the evaluation kit remote control.

**0x11111111s\_DeviceRemote\_RD6035\_JN5168\_v0000.bin**

This is the binary file for the remote control device running on the evaluation kit remote control.



*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 2 Setting up remote control hardware

Remove the battery compartment slide-cover on the rear of the remote control and insert two of the supplied AAA batteries (the required polarities are indicated inside the battery compartment). Then replace the cover.



On installing the batteries, the remote control will automatically power up. The unit will then attempt to join an existing WPAN. The left LED on the remote control will flash twice per second while the unit is trying to join the network.



Note 1: If the remote control has been previously used, it will remember the last network to which it belonged. To clear this information and return to the factory settings, either erase the EEPROM during programming or enter the following key sequence into the unit: PRG OFF DOWN OFF [# O - O].

Note 2: While the remote control will initially create the the WPAN as a Coordinator node, it will then discard its Coordinator functionality and become a 'sleeping broadcaster'. In order to wake the unit from sleep at any time, press the Wake (circle) button located below the keypad.

#### **Step 4 Create the standalone network using the remote control**

By default, the remote control is configured to join an existing WPAN in the full JenNet-IP Smart Home demonstration described in [Section 4.1 "Gateway System Operation"](#). To exit this mode and put the unit into standalone mode, enter the following key sequence into the unit (you may first need to activate the unit using the Wake button):

PRG DOWN UP DOWN [# - + -]

#### **Step 5 Remote control feedback**

The unit will then act as a WPAN Coordinator and create a WPAN. The left LED on the remote control will flash while the unit is creating the network and will stay illuminated for a few seconds once the network is created (with no other nodes yet). The remote control will then revert to 'sleeping broadcaster' mode.



---

## 4.2.3 Operating the Bulb Devices

This section describes how to setup and operate the bulb devices included in the Application Note using a remote control in a standalone network.

---

### 4.2.3.1 Setting Up the Bulb Devices

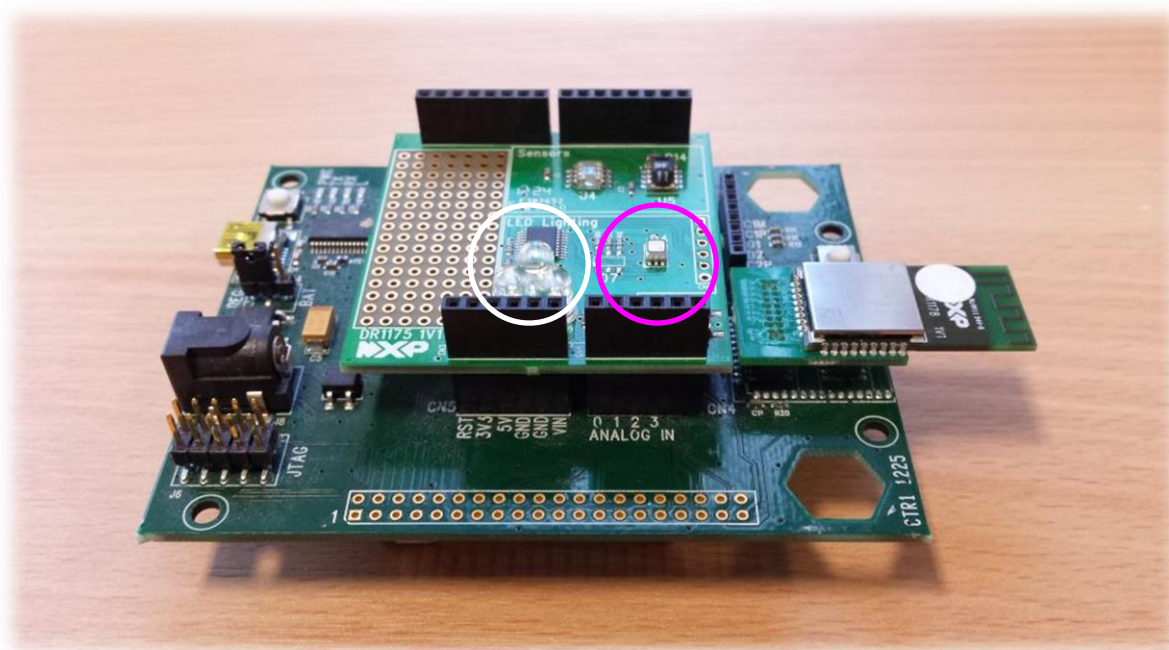
In setting up the bulb part of the demo system, you will need the following components:

- Remote control part of the system
- Three *Carrier Boards (DR1174)* fitted with JN516x modules and *Lighting/Sensor Expansion Boards (DR1175)*, antennae and batteries programmed with the required firmware.

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

#### **Step 1 Setup bulb hardware**

The bulb software runs on a Carrier Board fitted with the Lighting/Sensor Expansion Board as shown below:



The setup procedure is identical for the white, CCT and colour bulbs which differ only in the LEDs used to represent the bulb, (circled in the image above).

It is recommended that three boards are configured this way for this part of the demonstration.

## Step 2 Program the bulb software

The following pre-built binaries are provided in the Application Note for use on the evaluation kit boards. It is recommended that the three types of bulb software are used to create a white, CCT and colour bulb, (either of the white bulb builds may be used).

### **0x11111111s\_DeviceBulbWhite\_JN516X\_Router\_JN5168\_v0000.bin**

This binary file is for a white bulb driving the three white LEDs, shown with a white circle in the image above, on the evaluation kit Carrier Board and Lighting/Sensor Expansion Board combination.

### **0x11111111s\_DeviceBulbWhite\_DR1175\_Router\_JN5168\_v0000.bin**

This binary file is for a white bulb driving the RGB LED, shown with a magenta circle above, (displaying white only), on the evaluation kit Carrier Board and Lighting/Sensor Expansion Board combination.

### **0x11111111s\_DeviceBulbTemperature\_DR1175\_Router\_JN5168\_v0000.bin**

This binary file is for a colour controlled temperature bulb driving the RGB LED, shown with a magenta circle above, (displaying CCT colours only), on the evaluation kit Carrier Board and Lighting/Sensor Expansion Board combination.

### **0x11111111s\_DeviceBulbColour\_DR1175\_Router\_JN5168\_v0000.bin**

This binary file is for a colour bulb driving the RGB LED, shown with a magenta circle above, (displaying a full range of colours), on the evaluation kit Carrier Board and Lighting/Sensor Expansion Board combination.



*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 Commission bulb into the remote control's WPAN

To allow the node to join the WPAN of the remote control (note that at this stage, the node will be assigned to a group):

- a) Bring the remote control to within direct radio range of the node.
- b) Put the remote control into Coordinator mode and enable the node to join the WPAN by entering the following key sequence into the unit:

PRG ON OFF ON GRP [# I O I A/B/C/D/\*]



Note: In this key sequence, you must specify the group GRP to which the node will be assigned: A, B, C, D or \*. The star (\*) option refers to the All group. A node is automatically added to the All group (as well as to the specified group). However, you can specify the All group in the key sequence if you do not wish the node to be assigned to any other group.

- c) Following the above key sequence, the remote control will allow 5 minutes for nodes to join the network, during which the left LED on the unit will be illuminated. At the end of this period, the unit will revert from Coordinator mode to 'sleeping broadcaster' mode. You can escape from Coordinator mode at any time by pressing the Wake button (below the keypad).
- d) On power-up, the node will attempt to join the WPAN (for which the USB dongle is the Coordinator). While the node is trying to join the network, the LEDs on the expansion board will be fully illuminated (*they are very bright and, to avoid eye damage, you must not look directly into them for an extended period of time*).
- e) Wait for the node to successfully join the network, which is indicated by the LEDs on the node flashing twice and then remaining illuminated at full brightness. In the case of a successful join, the left LED on the remote control will also blink. If the node fails to join, power off the node (e.g. remove the batteries) and restart from **Step 3** (possibly bringing the remote control closer to the node).
- f) Once the node has joined the network, if the 5-minute timeout has not elapsed then press the Wake button (below the keypad) on the remote control to escape from Coordinator mode.

The three white LEDs or the RGB LED on the Lighting/Sensor Expansion Board are controlled by the application and act as the bulb's light source.

**Step 4 Bulb feedback**

Once the node has joined the network, the white LEDs will flash twice to indicate this and then remain fully illuminated.

**Step 5 Check that the node has joined the WPAN**

Use the remote control to check that the node has joined the WPAN. For example, use the unit to switch off the lights on the node by entering the following key sequence into the unit:

GRP OFF [A/B/C/D/\* O]

where GRP identifies the group to which the node was assigned.

Then switch all lights in the WPAN back on by entering the command GRP ON [\* I].

If the node fails to respond to commands from the Remote Control Unit, you should return to **Step 4** (and failing that, to **Step 3**).

**Step 6 Start and install the next node (if any)**

If there are still nodes to be started, start and install the next node as described from **Step 3**.

---

#### 4.2.4 Global Bulb Control from Remote Control

Global control of the bulbs in the standalone more network is identical to that in a gateway network as described in [Section 4.1.4.2 "Global Bulb Control from Remote Control"](#).

---

#### 4.2.5 Group Bulb Control from Remote Control

Global control of the bulbs in the standalone more network is identical to that in a Gateway network as described in [Section 4.1.4.3 "Group Bulb Control from Remote Control"](#).

---

## 5 MIB Variable Reference

This section provides reference information on the MIBs and variables created by the *JenNet-IP Smart Home (JN-AN-1162)*.

[Section 4 "System Operation"](#) describes how to perform common tasks using a limited set of MIB variables, this section provides comprehensive details of all the MIBs and their variables implemented in this Application Note including:

- Alternative ways to control bulb brightness and colour.
- Configuration of sensor device hardware.
- Configuration of sensor device control of bulbs.

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)*. These are the Node, JenNet, Groups, OND and Device ID MIBs.

There are a number of MIBs that are common to many different device types that are implemented in the application. These have been taken from *JenNet-IP Application Template (JN-AN-1190)* which contains full reference documentation for the common MIBs. These are the NodeStatus, NodeControl, NwkConfig, NwkProfile, NwkStatus, NwkSecurity, NwkTest and AdcStatus MIBs.

## 5.1 Bulb MIBs

The Bulb MIBs provide the main functionality for bulb devices, allowing them to be monitored, configured and controlled.

### 5.1.1 BulbConfig MIB (0xFFFFFE01)

The BulbConfig MIB contains variables that can be used to configure the settings used by the bulb.

#### 5.1.1.1 LumRate Variable

##### Description

The LumRate variable specifies the speed at which the bulb should change from one luminance level to another.

##### Storage

Permanent

##### Type

*Uint8*                      Unsigned Integer, 8 bits

##### Access

*Read, Write*

##### Values

*1 to 255*                      The amount to alter the current luminance by every 10ms while fading to the target luminance.

##### Default

*2*                                  Full range change takes 1.25 seconds.

##### Trap Notifications

On remote edit.

---

### 5.1.1.2 InitMode Variable

#### Description

The InitMode variable specifies the mode the bulb should enter at initialisation.

#### Storage

Permanent

#### Type

*UInt8*                      Unsigned Integer, 8 bits

#### Access

*Read, Write*

#### Values

<i>0 to 9</i>	Use a specific mode from those in the BulbControl MIB Mode variable.
<i>10 to 255</i>	Restore BulbControl MIB Mode variable value from flash.

#### Default

*1*                              Bulb turned on mode.

#### Trap Notifications

On remote edit.

---

### 5.1.1.3 InitLumTarget Variable

#### Description

The InitLumTarget variable specifies the luminance target that should be applied by the bulb at initialisation when the InitMode variable is applying a mode rather than restoring from flash.

#### Storage

Permanent

#### Type

*UInt8*                      Unsigned Integer, 8 bits

#### Access

*Read, Write*

#### Values

<i>0 to 255</i>	Initial luminosity of bulb.
-----------------	-----------------------------

#### Default

*255*                            Maximum luminosity.

#### Trap Notifications

On remote edit.

#### 5.1.1.4 DownUpCadFlags Variable

##### Description

The DownUpCadFlags variable specifies the conditions when the bulb should display a cadence effect if the network connection is down or up.

##### Storage

Permanent

##### Type

*UInt8*                      Unsigned Integer, 8 bits

##### Access

*Read, Write*

##### Values

<i>0x01</i>	The bulb should always display the down cadence when the network reset or is lost.
<i>0x02</i>	The bulb should display the down cadence when the network is down following a reset.
<i>0x04</i>	The bulb should display the down cadence when the network is down following a factory reset.
<i>0x10</i>	The bulb should always display the up cadence whenever the network is joined.
<i>0x20</i>	The bulb should display the up cadence the first time the network is joined following a reset.
<i>0x40</i>	The bulb should display the up cadence the first time the network is joined following a factory reset.

##### Default

<i>0x20</i>	The bulb should display the up cadence the first time the network is joined following a reset.
-------------	--

##### Trap Notifications

On remote edit.



---

### 5.1.1.5 DownCadence Variable

#### Description

The DownCadence variable specifies the cadence the bulb should display when the network is down as specified by the DownUpCadFlags variable.

#### Storage

Permanent

#### Type

*UInt32*                      Unsigned Integer, 32 bits

#### Access

*Read, Write*

#### Values

The DownCadence variable is actually composed of 4 different settings each being encoded into 1 byte of the specified value as follows:

<i>0x000000FF</i>	The minimum luminosity is encoded as the least significant byte of the value.
<i>0x0000FF00</i>	The maximum luminosity is encoded as the second least significant byte of the value.
<i>0x00FF0000</i>	The fade rate is encoded as the second most significant byte of the value. This is the number of luminosity steps that the bulb should fade by each 10ms.
<i>0xFF000000</i>	The switch time is encoded as the most significant byte of the value. This is the length of time in 10ms units to switch between the min and max luminosity values. If both a fade and switch values are specified the fade takes priority.

#### Default

*0x0002ff50*                      Fade between max and 30% luminosity, (though disabled by default DownUpCadFlags).

#### Trap Notifications

On remote edit.

### 5.1.1.6 DownCadTimer Variable

#### Description

The DownCadTimer variable specifies the length of time the bulb should display the down cadence as specified by the DownUpCadFlags variable.

#### Storage

Permanent

#### Type

*Uint16*                      Unsigned Integer, 16 bits

#### Access

*Read, Write*

#### Values

0	Disabled, the cadence effect is not enabled.
1 - 65534	The time to display the cadence effect in 10ms units.
65535	Continuously, the cadence effect is displayed until the LumCadTimer variable or the LumCadence variable is cleared.

#### Default

30000	5 minutes, (though disabled by default DownUpCadFlags).
-------	---

#### Trap Notifications

On remote edit.

---

### 5.1.1.7 UpCadence Variable

#### Description

The UpCadence variable specifies the cadence the bulb should display when the network is up (joined) as specified by the DownUpCadFlags variable.

#### Storage

Permanent

#### Type

*UInt32*                      Unsigned Integer, 32 bits

#### Access

*Read, Write*

#### Values

The UpCadence variable is actually composed of 4 different settings each being encoded into 1 byte of the specified value as follows:

<i>0x000000FF</i>	The minimum luminosity is encoded as the least significant byte of the value.
<i>0x0000FF00</i>	The maximum luminosity is encoded as the second least significant byte of the value.
<i>0x00FF0000</i>	The fade rate is encoded as the second most significant byte of the value. This is the number of luminosity steps that the bulb should fade by each 10ms.
<i>0xFF000000</i>	The switch time is encoded as the most significant byte of the value. This is the length of time in 10ms units to switch between the min and max luminosity values. If both a fade and switch values are specified the fade takes priority.

#### Default

*0x1000ff00*                      Switch quickly between max and min luminosity.

#### Trap Notifications

On remote edit.

### 5.1.1.8 UpCadTimer Variable

#### Description

The UpCadTimer variable specifies the length of time the bulb should display the down cadence as specified by the DownUpCadFlags variable.

#### Storage

Permanent

#### Type

*Uint16*                      Unsigned Integer, 16 bits

#### Access

*Read, Write*

#### Values

0	Disabled, the cadence effect is not enabled.
1 - 65534	The time to display the cadence effect in 10ms units.
65535	Continuously, the cadence effect is displayed until the LumCadTimer variable or the LumCadence variable is cleared.

#### Default

64                      640ms.

#### Trap Notifications

On remote edit.

---

## 5.1.2 BulbStatus MIB (0xFFFFFE00)

The BulbStatus MIB contains variables that indicate the status of the bulb. Each variable and its use are described in the following chapters:

---

### 5.1.2.1 OnCount Variable

#### Description

The variable specifies how many times the bulb has been turned on.

#### Storage

Permanent

#### Type

*Uint16*                      Unsigned Integer, 16 bits

#### Access

*Read*

#### Values

*0 to 65535*

#### Default

*0*

#### Trap Notifications

When bulb is turned on.

---

### 5.1.2.2 OnTime Variable

#### Description

The variable specifies how long the bulb has been turned on in seconds.

#### Storage

Permanent

#### Type

*Uint32*                      Unsigned Integer, 32 bits

#### Access

*Read*

#### Values

*0 to 4294967296*

#### Default

*0*

#### Trap Notifications

Hourly while the bulb is on.  
When the bulb is turned off.

---

### 5.1.2.3 OffTime Variable

#### Description

The variable specifies how long the bulb has been turned off in seconds.

#### Storage

Permanent.

#### Type

*UInt32*                      Unsigned Integer, 32 bits

#### Access

*Read*

#### Values

*0 to 4294967296*

#### Default

*0*

#### Trap Notifications

Hourly while the bulb is off.  
When the bulb is turned on.

---

### 5.1.2.4 ChipTemp Variable

#### Description

The chip temperature tenths of a degree Centigrade.

#### Type

*Int16*                      Signed Integer, 16 bits

#### Access

*Read*

#### Values

*-9996 to 4909*              Temperature read by the bulb.

#### Default

As read

#### Trap Notifications

None

---

### 5.1.2.5 BusVolts Variable

#### Description

The BusVolts variable contains the bus voltage in the bulb measured on ADC4 in volts.

#### Type

*Int16* Signed Integer, 16 bits

#### Access

*Read*

#### Values

*0 to 32767* Actual maximum is dependent upon bulb type.

#### Default

As read

#### Trap Notifications

None

### 5.1.3 BulbScene MIB (0xFFFFFE03)

The BulbScene MIB contains variables that can be used to configure the scenes used by the bulb.

The more flexible DeviceScene MIB should be used in preference to the BulbScene MIB, which adds the ability to configure scenes without changing the current settings. The BulbScene MIB is retained in the bulbs for backwards compatibility with older controlling devices.

#### 5.1.3.1 AddSceneld Variable

##### Description

When written to the AddSceneld variable adds or updates the specified scene ID with the current settings of the bulb. If the Scene ID is new but there is not an unused Sceneld variable then the write will fail.

##### Storage

Volatile

##### Type

*Uint16*                      Unsigned Integer, 16 bits

##### Access

*Read, Write*

##### Values

*1 to 65535*                      Scene ID to add.

##### Default

*0*                                  No scene added.

##### Trap Notifications

On remote edits.



---

### 5.1.3.2 DelScenId Variable

#### Description

When written to the DelScenId variable deletes the specified Scene ID if in use.

#### Storage

Volatile

#### Type

*Uint16*                      Unsigned Integer, 16 bits

#### Access

*Read, Write*

#### Values

*1 to 65535*                      Scene ID to delete.

#### Default

*0*                                  No scene deleted.

#### Trap Notifications

On remote edits.

### 5.1.3.3 Sceneld Table

#### Description

The Sceneld table specifies up to 8 Scene IDs that the bulb is currently participating in. The corresponding SceneLumTarget tables specify the bulb's settings for the scene.

#### Storage

Permanent

#### Type

*UInt16 [8]* Unsigned Integer, 16 bits, 8 entries

#### Access

*Read*

#### Values

*0* Unused scene.  
*1 to 65535* Sceneld to participate in.

#### Default

*0* Unused scene.

#### Trap Notifications

When altered by writing to the AddSceneld or DelSceneld variables.

### 5.1.3.4 SceneMode Table

#### Description

The SceneMode table specifies the bulb should be on or off for each scene.

#### Storage

Permanent

#### Type

*UInt8 [8]* Unsigned Integer, 8 bits, 8 entries

#### Access

*Read*

#### Values

*0* Off.  
*1* On.

#### Default

*0* Unused scene

#### Trap Notifications

When altered by writing to the AddSceneld or DelSceneld variables

---

### 5.1.3.5 SceneLumTarget Table

#### Description

The SceneLumTarget table specifies the luminance target for each scene.

#### Storage

Permanent

#### Type

*UInt8 [8]*                      Unsigned Integer, 8 bits, 8 entries

#### Access

*Read*

#### Values

*0 to 255*                      Luminance of bulb when scene is applied.

#### Default

*0*                                  Unused scene

#### Trap Notifications

When altered by writing to the AddSceneld or DelSceneld variables.

## 5.1.4 BulbControl MIB (0xFFFFFE04)

The BulbControl MIB contains variables that can be used to control the bulb.

### 5.1.4.1 Mode Variable

#### Description

The Mode variable specifies the operating mode of the bulb. At its most basic it can be used to turn the bulb on and off however it also provides access to additional modes for more advanced use.

#### Storage

Permanent

#### Type

*UInt8*                      Unsigned Integer, 8 bits

#### Access

*Read, Write*

#### Values

<i>0 (Off)</i>	In this mode the bulb is completely turned off. When entering this mode from another the bulb is instantly turned off, it does not fade to off.
<i>1 (On)</i>	In this mode the bulb is turned on with the luminance set by the Current and/or Target variables. When entering this mode from the Off mode the bulb is instantly turned on to the previous Target luminance level.
<i>2 (Toggle On/Off)</i>	Writing this mode toggles between the On and Off modes but only if the Mode is already set to Off or On. If the bulb is in On or Off mode when this value is written the request will be successful but the final value of the variable will be Off or On as appropriate. If the bulb is in any other mode when this value is written the remote write will return failed.
<i>3 (Test)</i>	In Test mode the bulb fades between minimum and maximum luminance while the node is trying to join a network. While in a network the bulb's luminance is determined by the strength of the signal to its parent node.

4 ( <i>Down</i> )	<p>When in Down mode the bulb fades from the current luminance towards the minimum luminance.</p> <p>If the bulb reaches the minimum luminance while in this mode the mode is updated to 1 (On).</p> <p>If the bulb is Off when entering this mode it will instantly be turned on to the previous Target luminance before starting to fade down.</p>
5 ( <i>Up</i> )	<p>When in Up mode the bulb fades from the current luminance towards the maximum luminance.</p> <p>If the bulb reaches the maximum luminance while in this mode the mode is updated to 1 (On).</p> <p>If the bulb is Off when entering this mode it will instantly be turned on to the previous Target luminance before starting to fade up.</p>
6 ( <i>Down if On</i> )	<p>This mode works in the same way as the Down mode but only if the bulb is already in On mode when this value is written. Otherwise a fail will be returned for the remote set request.</p>
7 ( <i>Up if On</i> )	<p>This mode works in the same way as the Up mode but only if the bulb is already in On mode when this value is written. Otherwise a fail will be returned for the remote set request.</p>
8 ( <i>On if Down/Up</i> )	<p>This mode turns the bulb on but only if it is fading up or down. Should be used to end fading when using the Down if On or Up if On modes.</p>
9 ( <i>Failed</i> )	<p>This mode indicates that the bulb hardware has failed and so can no longer be controlled.</p>

## **Default**

1 ( <i>On</i> )	<p>The default on power on can be altered using the BulbConfig MIB InitMode variable.</p>
-----------------	---

## **Trap Notifications**

On remote set.

When entering On mode from Down mode when the minimum luminance is reached.

When entering On mode from Up mode when maximum luminance is reached.

When the mode is changed as a result of applying a scene.

When the bulb hardware failure is detected.

### 5.1.4.2 Sceneld Variable

#### Description

The Sceneld variable when set switches the bulb to the settings for the specified Scene ID, if settings for the specified Scene ID are present in the BulbScene MIB.

#### Storage

Permanent

#### Type

*Uint16*                      Unsigned Integer, 16 bits

#### Access

*Read, Write*

#### Values

<i>0</i>	None.
<i>1 to 65535</i>	Scene ID to switch to.

#### Default

<i>0</i>	None.
----------	-------

#### Trap Notifications

On remote set.

---

### 5.1.4.3 LumTarget Variable

#### Description

The LumTarget variable specifies the required luminance of the bulb.

If the bulb is in the On mode when this is set it will fade from the current luminance to the target luminance.

If the bulb is in the Off mode the value will be retained and applied when the bulb is turned on.

#### Storage

Permanent

#### Type

*UInt8*                      Unsigned Integer, 8 bits

#### Access

*Read, Write*

#### Values

*0 to 255*                      Target luminance for the bulb.

#### Default

*255*                              The default on power on can be altered using the BulbConfig MIB InitMode and InitLumTarget variables.

#### Trap Notifications

On remote set.

When entering On mode from Down, Up, ID or Test modes.

When changed as a result of applying a scene.

When setting the LumCurrent level in On or Off modes if the LumTarget level is not already set to the new LumCurrent level.

When the LumTarget level is altered by setting the LumChange variable while in On or Off modes.

#### 5.1.4.4 LumCurrent Variable

##### Description

The LumCurrent variable specifies the current luminance of the bulb.

If the bulb is in the On mode when this is set it will instantly change to the set luminance.

If the bulb is in the Off mode the value will be retained and applied when the bulb is turned on.

When the LumCurrent variable is set the LumTarget variable is updated to the same value if it is currently different.

##### Storage

Volatile

##### Type

*UInt8*                      Unsigned Integer, 8 bits

##### Access

*Read, Write*

##### Values

*0 to 255*                      Current luminance for the bulb.

##### Default

*255*                              The default on power on can be altered using the BulbConfig MIB InitMode and InitLumTarget variables.

##### Trap Notifications

On remote set.

When the LumCurrent level fades to the LumTarget level when in On mode.

When entering On mode from Down, Up, ID or Test modes.

When entering On, Down or Up modes from Off mode and the LumCurrent level is not already set to the LumTarget.



---

### 5.1.4.5 LumChange Variable

#### Description

The LumChange variable when set changes the LumTarget variable by the set amount. This is most useful to jog the luminance level up and down by fixed amounts.

#### Storage

Volatile

#### Type

*Int16* Signed Integer, 16 bits

#### Access

*Read, Write*

#### Values

*-255 to 255* Target luminance change for the bulb.

#### Default

*0*

#### Trap Notifications

On remote set.

### 5.1.4.6 LumCadence Variable

#### Description

The LumCadence variable in conjunction with the LumCadTimer variable allows a wide range of luminosity level fading and switching effects to be implemented.

To take effect the LumCadence value must include a non-zero fade or switch setting and the LumCadTimer variable must be non-zero.

When a cadence effect is ended the bulb will restore the current mode of the lamp, if On the bulb will fade to the target luminosity from the final luminosity level of the cadence effect.

The LumCadence variable is used to specify the min and max luminosity of the effect together with timings for either fading or switching between those levels.

#### Storage

Volatile

#### Type

*Unt32*                      Unsigned Integer, 32 bits

#### Access

*Read, Write*

#### Values

The LumCadence variable is actually composed of 4 different settings each being encoded into 1 byte of the specified value as follows:

<i>0x000000FF</i>	The minimum luminosity is encoded as the least significant byte of the value.
<i>0x0000FF00</i>	The maximum luminosity is encoded as the second least significant byte of the value.
<i>0x00FF0000</i>	The fade rate is encoded as the second most significant byte of the value. This is the number of luminosity steps that the bulb should fade by each 10ms.
<i>0xFF000000</i>	The switch time is encoded as the most significant byte of the value. This is the length of time in 10ms units to switch between the min and max luminosity values. If both a fade and switch values are specified the fade takes priority.

#### Default

0

#### Trap Notifications

On remote set.

---

### 5.1.4.7 LumCadTimer Variable

#### Description

The LumCadTimer variable in conjunction with the LumCadence variable allows a wide range of luminosity level fading and switching effects to be implemented.

To take effect the LumCadence value must include a non-zero fade or switch setting and the LumCadTimer variable must be non-zero.

When a cadence effect is ended the bulb will restore the current mode of the lamp, if On the bulb will fade to the target luminosity from the final luminosity level of the cadence effect.

The LumCadTimer variable is used to control the duration of the cadence effect.

#### Storage

Volatile

#### Type

Unt16                      Unsigned Integer, 16 bits

#### Access

Read, Write

#### Values

0	Disabled, the cadence effect is not enabled.
1 - 65534	The time to display the cadence effect in 10ms units.
65535	Continuously, the cadence effect is displayed until the LumCadTimer variable or the LumCadence variable is cleared.

#### Default

0

#### Trap Notifications

On remote set.

## 5.2 Colour MIBs

The Colour MIBs provide the colour functionality for CCT and colour bulb devices, allowing them to be monitored, configured and controlled.

### 5.2.1 ColourConfig MIB (0xFFFFFE09)

The ColourConfig MIB contains variables that can be used to configure the settings used by colour bulbs.

#### 5.2.1.1 TransitionTime Variable

##### Description

The TransitionTime variable specifies the time to transition from the current colour to a target colour in 10ms intervals. Multiples of the transition time are also used to control the rate at which the colour components are altered when in the Up and Down modes set by the ColourControl MIB's Mode variable.

##### Storage

Permanent

##### Type

*Uint16*                      Unsigned Integer, 16 bits

##### Access

*Read, Write*

##### Values

*0 to 65535*                      Transition time in 10ms intervals

##### Default

*500*                              5 seconds.

##### Trap Notifications

On remote edit.

---

### 5.2.1.2 InitMode Variable

#### Description

The InitMode variable specifies the colour mode the bulb should enter at initialisation.

#### Storage

Permanent

#### Type

*UInt8*                      Unsigned Integer, 8 bits

#### Access

*Read, Write*

#### Values

<i>0 to 10</i>	Use a specific mode from those in the ColourControl MIB Mode variable.
<i>11 to 255</i>	Restore ColourControl MIB Mode variable value from flash.

#### Default

*0*                      Stop (unchanging) mode.

#### Trap Notifications

On remote edit.

### 5.2.1.3 InitXYTarget Variable

#### Description

The InitXYTarget variable specifies the colour target that should be applied by the bulb at initialisation when the InitMode variable is applying a mode rather than restoring from flash.

#### Storage

Permanent

#### Type

*UInt32*                      Unsigned Integer, 32 bits

#### Access

*Read, Write*

#### Values

*0x0 - 0xFFFFFFFF*    Initial colour of bulb.

#### Default

*Varies*                      White, (hardware dependent).

#### Trap Notifications

On remote edit.

### 5.2.1.4 XYPrimaryWhite Variable

#### Description

The XYPrimaryWhite variable specifies the white colour point of the bulb and is used to ensure that the colours are set correctly.

#### Storage

Permanent

#### Type

*UInt32*                      Unsigned Integer, 32 bits

#### Access

*Read, Write*

#### Values

*0x0 - 0xFFFFFFFF*    White point of bulb.

*Bits 31-16*                X Component

*Bits 15-0*                Y Component

#### Default

*Varies*                      Hardware dependent.

#### Trap Notifications

On remote edit.

---

### 5.2.1.5 XYPrimary1-6 Variables

#### Description

The XYPrimary1-6 variables specifies the primary colour points of the bulb and are used to ensure that the colours are set correctly.

XYPrimary1 is commonly used as the red point, XYPrimary2 as the green point and XYPrimary3 as the blue point. The other primaries may be used when the bulb hardware has emitters in additional colours, (the colour software however is written to operate on only red, green and blue primaries).

#### Storage

Permanent

#### Type

*Uint32*                      Unsigned Integer, 32 bits

#### Access

*Read, Write*

#### Values

<i>0x0 - 0xFFFFFFFF</i>	Colour points of bulb.
<i>Bits 31-16</i>	X Component
<i>Bits 15-0</i>	Y Component

#### Default

*Varies*                      Hardware dependent.

#### Trap Notifications

On remote edit.

### 5.2.1.6 CctMin and CctMax Variables

#### Description

These variables specify the colour temperature range that the bulb can display in mireds.

#### Storage

Permanent

#### Type

*Uint16*                      Unsigned Integer, 16 bits

#### Access

*Read, Write*

#### Values

*0 to 1000*                      In mireds.

#### Default

*Varies*                      Hardware dependent.

#### Trap Notifications

On remote edit.



---

## 5.2.2 ColourControl MIB (0xFFFFE0C)

The ColourControl MIB contains variables that can be used to control the colour of bulbs.

---

### 5.2.2.1 Mode Variable

#### Description

The Mode variable specifies the operating mode of the bulb.

#### Storage

Permanent

#### Type

*Uint8*                      Unsigned Integer, 8 bits

#### Access

*Read, Write*

#### Values

0 (Stop)	In this mode the colour of the bulb is static.
1 (Hue loop down)	In this mode the hue of the bulb is continually looped down.
2 (Hue loop up)	In this mode the hue of the bulb in continually looped up.
3 (Test Hue/Sat)	This test mode randomly alters the hue and saturation of the bulb.
4 (Test CCT)	This test mode randomly alters the luminance and colour temperature of the bulb in the orange area of the CCT spectrum.
5 (Hue Down)	When in Hue Down mode the bulb lowers the hue over time stopping when it reaches red. If the bulb reaches the minimum (red) while in this mode the mode is updated to 0 (Stop).
6 (Hue Up)	When in Hue Up mode the bulb increases the hue over time stopping when it reaches red. If the bulb reaches the maximum (red) while in this mode the mode is updated to 0 (Stop).
7 (Sat Down)	When in Sat Down mode the bulb lowers the saturation over time stopping when it reaches white. If the bulb reaches the minimum (white) while in this mode the mode is updated to 0 (Stop).
8 (Sat Up)	When in Sat Up mode the bulb increases the saturation over time stopping when fully saturated. If the bulb reaches the maximum while in this mode the mode is updated to 0 (Stop).

<i>7 (CCT Down)</i>	When in CCT Down mode the bulb lowers the colour temperature over time stopping when it reaches blue. If the bulb reaches the minimum (blue) while in this mode the mode is updated to 0 (Stop).
<i>8 (CCT Up)</i>	When in CCT Up mode the bulb increases the colour temperature over time stopping when it reaches red. If the bulb reaches the maximum (red) while in this mode the mode is updated to 0 (Stop).

### Default

<i>0 (Stop)</i>	The default on power on can be altered using the ColourConfig MIB InitMode variable.
-----------------	--

### Trap Notifications

On remote set.  
When entering Stop mode from the Down or Up modes when the limit is reached.  
When the mode is changed as a result of applying a scene.

---

## 5.2.2.2 Sceneld Variable

### Description

The Sceneld variable when set switches the colour to the settings for the specified Scene ID, if settings for the specified Scene ID are present in the DeviceScene MIB.

### Storage

Permanent

### Type

<i>Uint16</i>	Unsigned Integer, 16 bits
---------------	---------------------------

### Access

*Read, Write*

### Values

<i>0</i>	None.
<i>1 to 65535</i>	Scene ID to switch to.

### Default

<i>0</i>	None.
----------	-------

### Trap Notifications

On remote set.

---

### 5.2.2.3 XYTarget Variable

#### Description

The XYTarget variable specifies the required colour of the bulb in the XY space. The bulb will transition from the current colour to the target colour over time.

#### Storage

Permanent

#### Type

*UInt32*                      Unsigned Integer, 32 bits

#### Access

*Read, Write*

#### Values

*0x0 – 0xFFFFFFFF* Target colour for the bulb.  
*Bits 31-16*              X Component  
*Bits 15-0*                Y Component

#### Default

*White*                      The default value is the white point for the bulb.  
The default on power on can be altered using the  
BulbConfig MIB InitMode and InitLumTarget variables.

#### Trap Notifications

On remote set.  
When entering Stop mode from the Up or Down modes.  
When changed as a result of applying a scene.  
When altering the target colour via another variable.

## 5.2.2.4 XYCurrent Variable

### Description

The XYCurrent variable specifies the current colour of the bulb in the XY space. The bulb will change to the new colour instantly.

When the XYCurrent variable is set the XYTarget variable is updated to the same value if it is currently different.

### Storage

Volatile

### Type

*Uint32*                      Unsigned Integer, 32 bits

### Access

*Read, Write*

### Values

*0x0 to 0xFFFFFFFF* Current colour for the bulb.

*Bits 31-16*                      X Component

*Bits 15-0*                        Y Component

### Default

*White*                              The default value is the white point for the bulb.  
The default on power on can be altered using the  
BulbConfig MIB InitMode and InitLumTarget variables.

### Trap Notifications

On remote set.

When the XYCurrent level fades to the XYTarget level when in On mode.

When entering Stop mode from the Down or Up modes.

When altering the current colour via another variable.

---

### 5.2.2.5 XTarget and YTarget Variables

#### Description

The XTarget and YTarget variables specifies the required colour of the bulb in the XY space on a single axis. The bulb will transition from the current colour to the target colour over time.

#### Storage

Volatile

#### Type

*Uint16*                      Unsigned Integer, 16 bits

#### Access

*Read, Write*

#### Values

*0x0 – 0xFFFF*              Target colour for the bulb.

#### Default

*White*                      The default value is the white point for the bulb.  
The default on power on can be altered using the  
BulbConfig MIB InitMode and InitLumTarget variables.

#### Trap Notifications

On remote set.  
When entering Stop mode from the Up or Down modes.  
When changed as a result of applying a scene.  
When altering the target colour via another variable.

## 5.2.2.6 HueTarget Variable

### Description

The HueTarget variable specifies the required hue of the bulb. The bulb will transition from the current colour to the target colour over time.

### Storage

Volatile

### Type

*Uint16*                      Unsigned Integer, 16 bits

### Access

*Read, Write*

### Values

*0 – 3600*                      Target hue for the bulb in tenths of a degree.

### Default

*White*                      The default value is the white point for the bulb.

### Trap Notifications

On remote set.  
When entering Stop mode from the Up or Down modes.  
When changed as a result of applying a scene.  
When altering the target colour via another variable.

---

### 5.2.2.7 HueChange Variable

#### Description

The HueChange variable when set changes the HueTarget variable by the set amount. This is most useful to jog the hue level up and down by fixed amounts.

#### Storage

Volatile

#### Type

*Int16* Signed Integer, 16 bits

#### Access

*Read, Write*

#### Values

*-3600 to 3600* Target hue change for the bulb in tenths of a degree.

#### Default

*0*

#### Trap Notifications

On remote set.

### 5.2.2.8 SatTarget Variable

#### Description

The SatTarget variable specifies the required saturation of the bulb. The bulb will transition from the current colour to the target colour over time.

#### Storage

Volatile

#### Type

*UInt8*                      Unsigned Integer, 8 bits

#### Access

*Read, Write*

#### Values

*0 – 255*                      Target saturation for the bulb.

#### Default

*White*                      The default value is the white point for the bulb.

#### Trap Notifications

On remote set.  
When entering Stop mode from the Up or Down modes.  
When changed as a result of applying a scene.  
When altering the target colour via another variable.



---

### 5.2.2.9 SatChange Variable

#### Description

The SatChange variable when set changes the SatTarget variable by the set amount. This is most useful to jog the saturation level up and down by fixed amounts.

#### Storage

Volatile

#### Type

*Int8* Signed Integer, 8 bits

#### Access

*Read, Write*

#### Values

*-127 to 127* Target saturation change for the bulb in tenths of a degree.

#### Default

*0*

#### Trap Notifications

On remote set.

## 5.2.2.10 HueSatTarget Variable

### Description

The HueSatTarget variable specifies the required hue and saturation of the bulb in a single operation. The bulb will transition from the current colour to the target colour over time.

### Storage

Volatile

### Type

*UInt32*                      Unsigned Integer, 32 bits

### Access

*Read, Write*

### Values

*0x0 – 0xFFFFFFFF* Target hue and saturation for the bulb.

*Bits 23-8*                      Hue in tenths of a degree

*Bits 7-0*                      Saturation

### Default

*White*                      The default value is the white point for the bulb.

### Trap Notifications

On remote set.

When entering Stop mode from the Up or Down modes.

When changed as a result of applying a scene.

When altering the target colour via another variable.

---

### 5.2.2.11 HueSatCurrent Variable

#### Description

The HueSatCurrent variable specifies the required hue and saturation of the bulb in a single operation. The bulb will change to the new colour instantly.

#### Storage

Volatile

#### Type

*UInt32*                      Unsigned Integer, 32 bits

#### Access

*Read, Write*

#### Values

*0x0 – 0xFFFFFFFF* Target hue and saturation for the bulb.

*Bits 23-8*                      Hue in tenths of a degree

*Bits 7-0*                      Saturation

#### Default

*White*                      The default value is the white point for the bulb.

#### Trap Notifications

On remote set.

When entering Stop mode from the Up or Down modes.

When changed as a result of applying a scene.

When altering the target colour via another variable.

## 5.2.2.12 CctTarget Variable

### Description

The CctTarget variable specifies the required colour temperature of the bulb. The bulb will transition from the current colour to the target colour over time.

### Storage

Volatile

### Type

*Uint16*                      Unsigned Integer, 16 bits

### Access

*Read, Write*

### Values

*0 – 1000*                      Target colour temperature for the bulb in mireds.

### Default

*White*                      The default value is the white point for the bulb.

### Trap Notifications

On remote set.  
When entering Stop mode from the Up or Down modes.  
When changed as a result of applying a scene.  
When altering the target colour via another variable.

---

### 5.2.2.13 CctCurrent Variable

#### Description

The CctTarget variable specifies the required colour temperature of the bulb.  
The bulb will change to the new colour instantly.

#### Storage

Volatile

#### Type

*Uint16*                      Unsigned Integer, 16 bits

#### Access

*Read, Write*

#### Values

*0 – 1000*                      Target colour temperature for the bulb in mireds.

#### Default

*White*                      The default value is the white point for the bulb.

#### Trap Notifications

On remote set.  
When entering Stop mode from the Up or Down modes.  
When changed as a result of applying a scene.  
When altering the target colour via another variable.

### 5.2.2.14 CctChange Variable

#### Description

The CctChange variable when set changes the CctTarget variable by the set amount. This is most useful to jog the colour temperature level up and down by fixed amounts.

#### Storage

Volatile

#### Type

*Int16* Signed Integer, 16 bits

#### Access

*Read, Write*

#### Values

*-1000 to 1000* CCT change for the bulb in mireds.

#### Default

*0*

#### Trap Notifications

On remote set.

---

### 5.2.2.15 RedCurrent, GreenCurrent and BlueCurrent Variables

#### Description

These variables allow the current values for the red, green and blue channels to be read from the bulb.

#### Storage

Volatile

#### Type

*UInt8*                      Unsigned Integer, 8 bits

#### Access

*Read*

#### Values

*0 to 255*                      Target CCT change for the bulb in mireds.

#### Default

*White*                      The default value is the white point for the bulb.

#### Trap Notifications

When entering Stop mode from the Up or Down modes.  
When changed as a result of applying a scene.  
When altering the target colour via another variable.

## 5.3 Sensor MIBs

The Sensor MIBs provide the main functionality for sensor devices, allowing them to be monitored, configured and controlled.

### 5.3.1 OccupancyConfig MIB (0xFFFFFE31)

The OccupancyConfig MIB contains variables that can be used to configure the settings used by an occupancy sensor.

#### 5.3.1.1 Sensitivity Variable

##### Description

The Sensitivity variable specifies the sensitivity setting for devices that support such a feature.

##### Storage

Permanent

##### Type

*Uint8*                      Unsigned Integer, 8 bits

##### Access

*Read, Write*

##### Values

<i>0</i>	Disabled
<i>1 to 255</i>	Sensitivity where 1 is least sensitive and 255 most sensitive.

##### Default

<i>255</i>	Full sensitivity.
------------	-------------------

##### Trap Notifications

On remote edit.



---

### 5.3.1.2 UnoccupiedDelay Variable

#### Description

The UnoccupiedDelay variable specifies the time period that the occupancy sensor hardware must continuously indicate unoccupied for the device to enter the unoccupied state. The time period is specified in 10ms intervals.

#### Storage

Permanent

#### Type

*Uint32*                      Unsigned Integer, 32 bits

#### Access

*Read, Write*

#### Values

*0 to 4294967296*

#### Default

*3000*                      30 seconds.

#### Trap Notifications

On remote edit.

### 5.3.1.3 OccupiedDelay Variable

#### Description

When both OccupiedDelay and OccupiedEvents are set there must be OccupiedEvents within the OccupiedDelay period from the sensor hardware to register as occupied in the software.

When only OccupiedDelay is set the sensor hardware must indicate occupied continuously for the whole period to register as occupied in the software.

When OccupiedDelay and OccupiedEvents are both zero the sensor will enter the occupied state immediately upon an occupied transition from the sensor hardware.

#### Storage

Permanent

#### Type

*Uint16*                      Unsigned Integer, 16 bits

#### Access

*Read, Write*

#### Values

*0 to 65535*

#### Default

*0*

#### Trap Notifications

On remote edit.

---

### 5.3.1.4 OccupiedEvents Variable

#### Description

When both OccupiedDelay and OccupiedEvents are set there must be OccupiedEvents within the OccupiedDelay period from the sensor hardware to register as occupied in the software.

When only OccupiedEvents is set there must be OccupiedEvents from the sensor hardware to register as occupied in the software.

When OccupiedDelay and OccupiedEvents are both zero the sensor will enter the occupied state immediately upon an occupied transition from the sensor hardware.

#### Storage

Permanent

#### Type

*UInt8*                      Unsigned Integer, 8 bits

#### Access

*Read, Write*

#### Values

*0 to 255*

#### Default

*0*

#### Trap Notifications

On remote edit.

### 5.3.1.5 StateMibId Variable

#### Description

When the StateMibId variable is non-zero it enables the transmission of the occupancy state whenever the state changes and at regular intervals. The value of the StateMibId variable is the MIB ID written to in these transmissions.

#### Storage

Permanent

#### Type

*Uint32*                      Unsigned Integer, 32 bits

#### Access

*Read, Write*

#### Values

<i>0</i>	Disabled
<i>Others</i>	MIB ID to write to

#### Default

*0*

#### Trap Notifications

On remote edit.

---

### 5.3.1.6 StateVarIdx Variable

#### Description

When the StateMibId variable is non-zero it enables the transmission of the occupancy state whenever the state changes and at regular intervals. The value of the StateVarIdx variable is the variable index written to in these transmissions.

#### Storage

Permanent

#### Type

*UInt8*                      Unsigned Integer, 8 bits

#### Access

*Read, Write*

#### Values

0-255                      Variable index to write to

#### Default

255                      Essentially disabled (MIBs are unlikely to have 255 variables)

#### Trap Notifications

On remote edit.

### 5.3.1.7 StateAddress Variable

#### Description

When the StateMibId variable is non-zero it enables the transmission of the occupancy state whenever the state changes and at regular intervals. The value of the StateAddress variable is the address these transmissions are sent to. The default value is a group address derived from the sensor's MAC address and the OccupancyConfig MIB ID.

#### Storage

Permanent

#### Type

*Blob* Blob, 128 bits

#### Access

*Read, Write*

#### Values

*Any* IPv6 address

#### Default

*Device specific*

#### Trap Notifications

On remote edit.

---

### 5.3.1.8 StateRefresh Variable

#### Description

When the StateMibId variable is non-zero it enables the transmission of the occupancy state whenever the state changes and at regular intervals. The value of the StateRefresh variable specifies the period between transmissions if the state is unchanged in 10ms intervals. A value of 0 disables the refresh transmissions.

#### Storage

Permanent

#### Type

*Uint16*                      Unsigned integer, 16 bits

#### Access

*Read, Write*

#### Values

0	Disabled
1 to 65535	Refresh interval in 10ms intervals

#### Default

1000	10 seconds
------	------------

#### Trap Notifications

On remote edit.

## 5.3.2 OccupancyStatus MIB (0xFFFFFE30)

The OccupancyStatus MIB contains variables that can be used to monitor the status of an occupancy sensor.

### 5.3.2.1 Occupancy Variable

#### Description

The Occupancy variable indicates the occupancy state of the sensor.

#### Storage

Volatile

#### Type

*Uint8*                      Unsigned Integer, 8 bits

#### Access

*Read*

#### Values

<i>0</i>	Unoccupied
<i>1</i>	Occupied.
<i>Others</i>	Reserved

#### Default

*0*                      Unoccupied.

#### Trap Notifications

On state changes.



---

### 5.3.2.2 SensorType Variable

#### Description

The SensorType variable indicates the type of occupancy sensor.

#### Storage

Volatile (but usually a constant)

#### Type

*UInt8*                      Unsigned Integer, 8 bits

#### Access

*Read*

#### Values

<i>0</i>	Passive Infra-red
<i>1</i>	Ultrasound.
<i>2</i>	Light Dependant Resistor
<i>3</i>	Microwave
<i>Others</i>	Reserved
<i>255</i>	Unknown

#### Default

*Hardware dependant*

#### Trap Notifications

None.

### 5.3.3 OccupancyControl MIB (0xFFFFFE34)

The OccupancyControl MIB contains variables that can be used to control an occupancy sensor.

#### 5.3.3.1 Mode Variable

##### Description

The Mode variable is used to set the operating mode of the sensor.

##### Storage

Permanent

##### Type

*Uint8*                      Unsigned Integer, 8 bits

##### Access

*Read, Write*

##### Values

<i>0</i>	Disabled
<i>1</i>	Enabled.
<i>Others</i>	Reserved

##### Default

<i>1</i>	Enabled.
----------	----------

##### Trap Notifications

On remote edits.

---

### 5.3.4 IlluminanceConfig MIB (0xFFFFFE39)

The OccupancyConfig MIB contains variables that can be used to configure the settings used by an occupancy sensor.

---

#### 5.3.4.1 StateMibId Variable

##### Description

When the StateMibId variable is non-zero it enables the transmission of the illuminance state whenever the state changes and at regular intervals. The value of the StateMibId variable is the MIB ID written to in these transmissions.

##### Storage

Permanent

##### Type

*Uint16*                      Unsigned Integer, 16 bits

##### Access

*Read, Write*

##### Values

*0*                              Disabled  
*0x0001 to 0xFFFF*      MIB ID to write to

##### Default

*0*

##### Trap Notifications

On remote edit.

### 5.3.4.2 StateVarIdx Variable

#### Description

When the StateMibId variable is non-zero it enables the transmission of the illuminance state whenever the state changes and at regular intervals. The value of the StateVarIdx variable is the variable index written to in these transmissions.

#### Storage

Permanent

#### Type

*UInt8*                      Unsigned Integer, 8 bits

#### Access

*Read, Write*

#### Values

0-255                      Variable index to write to

#### Default

255                      Essentially disabled (MIBs are unlikely to have 255 variables)

#### Trap Notifications

On remote edit.

---

### 5.3.4.3 StateAddress Variable

#### Description

When the StateMibId variable is non-zero it enables the transmission of the illuminance state whenever the state changes and at regular intervals. The value of the StateAddress variable is the address these transmissions are sent to. The default value is a group address derived from the sensor's MAC address and the IlluminanceConfig MIB ID.

#### Storage

Permanent

#### Type

*Blob* Blob, 128 bits

#### Access

*Read, Write*

#### Values

*Any* IPv6 address

#### Default

*Device specific*

#### Trap Notifications

On remote edit.

### 5.3.4.4 StateRefresh Variable

#### Description

When the StateMibId variable is non-zero it enables the transmission of the illuminance state whenever the state changes and at regular intervals. The value of the StateRefresh variable specifies the period between transmissions if the state is unchanged in 10ms intervals. A value of 0 disables the refresh transmissions.

#### Storage

Permanent

#### Type

*Uint16*                      Unsigned integer, 16 bits

#### Access

*Read, Write*

#### Values

<i>1</i>	Disabled
<i>1 to 65535</i>	Refresh interval in 10ms intervals

#### Default

<i>1000</i>	10 seconds
-------------	------------

#### Trap Notifications

On remote edit.

---

### 5.3.5 IlluminanceStatus MIB (0xFFFFFE38)

The IlluminanceStatus MIB contains variables that can be used to monitor the status of an illuminance sensor.

---

#### 5.3.5.1 LuxCurrent Variable

##### Description

The LuxCurrent variable indicates the current Lux measurement of the illuminance sensor.

##### Storage

Volatile

##### Type

*Uint16*                      Unsigned Integer, 16 bits

##### Access

*Read*

##### Values

*0 to 65535*                      Current illuminance in Lux

##### Default

*None*

##### Trap Notifications

On changes.

### 5.3.5.2 TargetStatus Variable

#### Description

The TargetStatus variable indicates the status of the measured light level compared to the target band set in the IlluminanceControl MIB.

#### Storage

Volatile

#### Type

*Uint8*                      Unsigned Integer, 8 bits

#### Access

*Read*

#### Values

<i>0</i>	Disabled
<i>1</i>	OK, (within target band).
<i>2</i>	Low
<i>3</i>	High
<i>Others</i>	Reserved

#### Default

*None*

#### Trap Notifications

On changes.



---

### 5.3.5.3 SensorType Variable

#### Description

The SensorType variable indicates the type of illuminance sensor.

#### Storage

Volatile (but usually a constant)

#### Type

*UInt8*                      Unsigned Integer, 8 bits

#### Access

*Read*

#### Values

<i>0</i>	Photodiode
<i>1</i>	CMOS
<i>255</i>	Unknown
<i>Others</i>	Reserved

#### Default

*Hardware dependant*

#### Trap Notifications

None.

---

### 5.3.5.4 LuxMin Variable

#### Description

The LuxMin variable indicates the minimum illuminance the sensor can measure in Lux.

#### Storage

Volatile (but usually a constant)

#### Type

*Uint16*                      Unsigned Integer, 16 bits

#### Access

*Read*

#### Values

*0 to 65535*                      Minimum Lux

#### Default

*Hardware dependant*

#### Trap Notifications

None.

---

### 5.3.5.5 LuxMax Variable

#### Description

The LuxMax variable indicates the maximum illuminance the sensor can measure in Lux.

#### Storage

Volatile (but usually a constant)

#### Type

*Uint16*                      Unsigned Integer, 16 bits

#### Access

*Read*

#### Values

*0 to 65535*                      Maximum Lux

#### Default

*Hardware dependant*

#### Trap Notifications

None.

---

### 5.3.5.6 LuxTolerance Variable

#### Description

The LuxTolerance variable indicates the possible error of the illuminance sensor in Lux where known.

#### Storage

Volatile (but usually a constant)

#### Type

*UInt8*                      Unsigned Integer, 8 bits

#### Access

*Read*

#### Values

*0 to 65535*                      Minimum Lux

#### Default

*Hardware dependant*

#### Trap Notifications

None.

## 5.3.6 IlluminanceScene MIB (0xFFFFFE3B)

The IlluminanceScene MIB contains variables that can be used to configure the scenes used by the illuminance sensor.

### 5.3.6.1 AddSceneld Variable

#### Description

When written to the AddSceneld variable adds or updates the specified Scene ID with the current settings of the illuminance sensor. If the Scene ID is new but there is not an unused Sceneld variable then the write will fail.

#### Storage

Volatile

#### Type

*Uint16* Unsigned Integer, 16 bits

#### Access

*Read, Write*

#### Values

*1 to 65535* Scene ID to add.

#### Default

*0* No scene added.

#### Trap Notifications

On remote edits.

---

### 5.3.6.2 DelScenId Variable

#### Description

When written to the DelScenId variable deletes the specified Scene ID if in use.

#### Storage

Volatile

#### Type

*Uint16*                      Unsigned Integer, 16 bits

#### Access

*Read, Write*

#### Values

*1 to 65535*                      Scene ID to delete.

#### Default

*0*                                  No scene deleted.

#### Trap Notifications

On remote edits.

### 5.3.6.3 SetScene Variable

#### Description

When written to the SetScene variable sets or updates the specified scene with all the settings for the scene. This allows a scene to be without having to alter the device's current settings.

#### Storage

Volatile

#### Type

*Blob* Blob with the following structure declared in **MibIlluminanceScene.h**:

```
typedef struct
{
    uint16 u16Id;
    uint16 u16LuxTarget;
    uint16 u16LuxBand;
} PACK tsIlluminanceScene;
```

#### Access

*Read, Write*

#### Values

*Any*

#### Default

*0* No scene set

#### Trap Notifications

On remote edits.

---

### 5.3.6.4 SceneTable Variable

#### Description

The SceneTable table variable contains the Scene IDs and settings for all scenes the device is taking part in.

#### Storage

Permanent

#### Type

*Table Blob*

Table of blob each entry has the following structure declared in **MibIlluminanceScene.h**:

```
typedef struct
{
    uint16 u16Id;
    uint16 u16LuxTarget;
    uint16 u16LuxBand;
} PACK tsIlluminanceScene;
```

#### Access

*Read*

#### Values

*Any*

#### Default

*0*

No scenes configured

#### Trap Notifications

On remote edits.

## 5.3.7 IlluminanceControl MIB (0xFFFFFE3C)

The IlluminanceControl MIB contains variables that can be used to control an illuminance sensor.

### 5.3.7.1 Mode Variable

#### Description

The Mode variable is used to set the operating mode of the sensor.

#### Storage

Permanent

#### Type

*Uint8*                      Unsigned Integer, 8 bits

#### Access

*Read, Write*

#### Values

<i>0</i>	Disabled
<i>1</i>	Enabled.
<i>Others</i>	Reserved

#### Default

<i>1</i>	Enabled.
----------	----------

#### Trap Notifications

On remote edits.



---

### 5.3.7.2 SceneId Variable

#### Description

The SceneId variable when set switches the illuminance sensor to the settings for the specified Scene ID, if settings for the specified Scene ID are present in the IlluminanceScene MIB.

#### Storage

Permanent

#### Type

*Uint16*                      Unsigned Integer, 16 bits

#### Access

*Read, Write*

#### Values

<i>0</i>	None.
<i>1 to 65535</i>	Scene ID to switch to.

#### Default

<i>0</i>	None.
----------	-------

#### Trap Notifications

On remote set.

### 5.3.7.3 LuxTarget Variable

#### Description

The LuxTarget variable specifies the desired illuminance in the area the illuminance sensor is monitoring in Lux.

#### Storage

Permanent

#### Type

*Uint16*                      Unsigned Integer, 16 bits

#### Access

*Read, Write*

#### Values

*0 to 65535*                      Target illuminance in Lux (may be limited by hardware)

#### Default

*750*                              750 Lux

#### Trap Notifications

On remote set.  
Following changes due to using the Adjust variable.  
On changes when using the LuxTargetChange variable

---

### 5.3.7.4 LuxBand Variable

#### Description

The LuxBandTarget variable specifies the width of the desired illuminance band in the area the illuminance sensor is monitoring in Lux. The band is centered on the LuxTarget variable.

#### Storage

Permanent

#### Type

*Uint16*                      Unsigned Integer, 16 bits

#### Access

*Read, Write*

#### Values

*10 to 150*                      Target illuminance in Lux (may be limited by hardware)

#### Default

*50*                                  50 Lux

#### Trap Notifications

On remote set.  
Following changes due to using the Adjust variable.  
On changes when using the LuxBandChange variable

### 5.3.7.5 Adjust Variable

#### Description

The Adjust variable allows the LuxTarget and LuxBand variables to be altered without having to set specific values for the variables. This operates as mode control altering the variables while the Adjust variable has a particular value. This is most useful when setting these values from remote controls.

#### Storage

Volatile

#### Type

*UInt8*                      Unsigned Integer, 8 bits

#### Access

*Read, Write*

#### Values

<i>0</i>	No adjustments
<i>1 (Down)</i>	Decreases LuxTarget by 5 Lux every 10ms
<i>2 (Up)</i>	Increases LuxTarget by 5 Lux every 10ms
<i>3 (Narrow)</i>	Decreases LuxBand by 1 Lux every 10ms
<i>4 (Widen)</i>	Increases LuxBand by 1 Lux every 10ms
<i>Others</i>	Reserved

#### Default

*0*                      No adjustments

#### Trap Notifications

On remote edits.

On reaching the minimum or maximum values for the LuxTarget or LuxBand when reverting back to the No Adjustments value.

---

### 5.3.7.6 LuxTargetChange Variable

#### Description

The LuxTargetChange variable when written to allows the LuxTarget variable to be changed by a fixed amount in Lux. This is most useful to move the LuxTarget up or down by a fixed amount without having to set an explicit value. This may be useful on controls like a jog dial.

#### Storage

Volatile

#### Type

*Int16* Signed Integer, 16 bits

#### Access

*Read, Write*

#### Values

*-32768 to 32767* Changes to the LuxTarget variable are limited to the variable's range

#### Default

*0* No change

#### Trap Notifications

On remote edits.

### 5.3.7.7 LuxBandChange Variable

#### Description

The LuxBandChange variable when written to allows the LuxBand variable to be changed by a fixed amount in Lux. This is most useful to change the LuxBand by a fixed amount without having to set an explicit value. This may be useful on controls like a jog dial.

#### Storage

Volatile

#### Type

*Int16* Signed Integer, 16 bits

#### Access

*Read, Write*

#### Values

*-32768 to 32767* Changes to the LuxBand variable are limited to the variable's range

#### Default

*0* No change

#### Trap Notifications

On remote edits.

---

### 5.3.8 OccIIBulbConfig MIB (0xFFFFFE3F)

The OccIIBulbConfig MIB contains variables that can be used to configure the control of Bulb devices based upon readings from occupancy and illuminance sensors.

---

#### 5.3.8.1 Mode Variable

##### Description

The Mode variable specifies the sensor types that should be taken into account when controlling bulbs.

##### Storage

Permanent

##### Type

*UInt8*                      Unsigned Integer, 8 bits

##### Access

*Read, Write*

##### Values

0	Disabled, no bulb control packets are transmitted.
1	Occupancy, when unoccupied bulbs are turned off, when occupied bulbs are turned on at full brightness.
2	Illuminance, bulb brightness is altered until the measured illuminance is within the target band (bulbs may be turned on or off to achieve this). Best used when light from bulbs falls on illuminance sensor.
3	Occupancy and illuminance (Auto), when unoccupied bulbs are turned off, when occupied bulb brightness is altered until the measured illuminance is within the target band (bulbs may be turned on or off to achieve this). Best used when light from bulbs falls on sensor.
4	Occupancy and Illuminance (Max), when occupied and light level is low bulbs are turned on at maximum brightness, when unoccupied or light level is high bulbs are turned off. Best used when light from bulbs does not fall on sensor.
5	Always Off, disregards sensors and keeps bulbs off.
6	Always On, used to disregard sensors and keep bulbs on at maximum brightness.
<i>Others</i>	Reserved.

##### Default

1	Occupancy for occupancy only sensor devices.
2	Illuminance for illuminance only sensor devices.
3	Occupancy and illuminance for combined sensor devices.

## **Trap Notifications**

On remote edit.



---

### 5.3.8.2 LuminanceDelta Variable

#### Description

Specifies the amount that the BulbControl MIB LumTarget variable should be altered by when the measured illuminance is too low or too high.

#### Storage

Permanent

#### Type

*UInt8*                      Unsigned Integer, 8 bits

#### Access

*Read, Write*

#### Values

*0 to 255*

#### Default

1

#### Trap Notifications

On remote edit.

---

### 5.3.8.3 AdjustInterval Variable

#### Description

Specifies the time period between adjustments in 10ms intervals to the BulbControl MIB LumTarget variable when being altered. Adjustments are also limited by the time taken to obtain an illuminance reading.

#### Storage

Permanent

#### Type

*UInt16*                      Unsigned Integer, 16 bits

#### Access

*Read, Write*

#### Values

*0 to 65535*

#### Default

32                              320ms

#### Trap Notifications

On remote edit.

### 5.3.8.4 RefreshInterval Variable

#### Description

Specifies the time period between transmissions when no adjustments need to be made in 10ms intervals. This is used to ensure that bulbs are brought under control of the sensor when they are powered on if conditions are not changing.

#### Storage

Permanent

#### Type

*Uint16* Unsigned Integer, 16 bits

#### Access

*Read, Write*

#### Values

*0 to 65535*

#### Default

*1000* 10s

#### Trap Notifications

On remote edit.

### 5.3.8.5 Address Variable

#### Description

This is the IPv6 address that bulb commands are transmitted to. The default value is a group address derived from the sensor's MAC address and the IlluminanceConfig MIB ID.

#### Storage

Permanent

#### Type

*Blob* Blob, 128 bits

#### Access

*Read, Write*

#### Values

*Any* IPv6 address

#### Default

*Device specific*

#### Trap Notifications

On remote edit.

---

### 5.3.9 OccupancyMonitor MIB (0xFFFFFE32)

The OccupancyMonitor MIB contains variables that can be used to monitor occupancy sensors in other devices.

The “real” occupancy sensors must be configured to write to the Occupancy variable using the variables in the OccupancyConfig MIB. Where the transmitting device is using a broadcast group address the receiving device must be in the group.

The OccIIBulbConfig MIB is able to control bulbs based upon the readings from many occupancy sensors writing their data into the OccupancyMonitor MIB. The area will be considered to be occupied if *any* of the monitored occupancy sensors report occupied, or if the local OccupancyStatus MIB Occupancy variable is set.

---

#### 5.3.9.1 Mode Variable

##### Description

The Mode variable is used to set the operating mode of the monitor.

##### Storage

Permanent

##### Type

*UInt8*                      Unsigned Integer, 8 bits

##### Access

*Read, Write*

##### Values

<i>0</i>	Disabled
<i>1</i>	Enabled.
<i>Others</i>	Reserved

##### Default

<i>0</i>	Disabled.
----------	-----------

##### Trap Notifications

On remote edits.

### 5.3.9.2 Timeout Variable

#### Description

The Timeout variable is used to invalidate older readings and is set in seconds. Each time a device updates its reading a timer is started with this value. If the timer expires before another write is received from the device it is timed-out and not used when calculating occupancy. The Enabled variable contains a read-only bitmask indicating which device readings are considered valid.

#### Storage

Permanent

#### Type

*Uint16*                      Unsigned Integer, 16 bits

#### Access

*Read, Write*

#### Values

<i>0</i>	Disabled
<i>1 to 65535</i>	Timeout period in 1s intervals

#### Default

<i>300</i>	5 minutes.
------------	------------

#### Trap Notifications

On remote edits.

---

### 5.3.9.3 Occupancy Variable

#### Description

The Occupancy variable is made available for occupancy sensors in other devices to write into. The source address of the write is used to hold a record for each device internally even though all devices write to the same variable.

#### Storage

Volatile

#### Type

*UInt8*                      Unsigned Integer, 8 bits

#### Access

*Read, Write*

#### Values

<i>0</i>	Unoccupied
<i>1</i>	Occupied
<i>Others</i>	Reserved

#### Default

<i>0</i>	Unoccupied
----------	------------

#### Trap Notifications

On updates

### 5.3.9.4 Enabled Variable

#### Description

The Enabled variable provides an overview of which entries in the device table are in use, (have recent readings written to them), in the form of a bitmask with each bit mapping to its corresponding device in the DeviceTable variable.

#### Storage

Volatile

#### Type

*UInt32*                      Unsigned Integer, 32 bits

#### Access

*Read*

#### Values

*0 to 0xFFFFFFFF*      Bitmask of enabled sensors

#### Default

*0*

#### Trap Notifications

On changes.

---

### 5.3.9.5 Occupied Variable

#### Description

The Occupied variable provides an overview of which monitored devices are occupied in the form of a bitmask with each bit mapping to a corresponding entry in the DeviceTable variable.

#### Storage

Volatile

#### Type

*UInt32*                      Unsigned Integer, 32 bits

#### Access

*Read*

#### Values

*0 to 0xFFFFFFFF*      Bitmask of sensors reporting occupied

#### Default

*0*

#### Trap Notifications

On changes.

---

### 5.3.9.6 Unoccupied Variable

#### Description

The Unoccupied variable provides an overview of which monitored devices are unoccupied in the form of a bitmask with each bit mapping to a corresponding entry in the DeviceTable variable.

#### Storage

Volatile

#### Type

*UInt32*                      Unsigned Integer, 32 bits

#### Access

*Read*

#### Values

*0 to 0xFFFFFFFF*      Bitmask of sensors reporting occupied

#### Default

*0*

#### Trap Notifications

On changes.

---

### 5.3.9.7 MaxDevices Variable

#### Description

The MaxDevices variable indicates how many occupancy sensor devices can be monitored.

#### Storage

Constant

#### Type

*UInt8*                      Unsigned Integer, 8 bits

#### Access

*Read*

#### Values

*0 to 32*                      Number of available monitors

#### Default

*Device specific*

#### Trap Notifications

None

---

### 5.3.9.8 DeviceTable Variable

#### Description

The DeviceTable table variable contains the IPv6 addresses of the occupancy sensor devices currently being monitored.

#### Storage

Volatile

#### Type

*Table Blob*                      Table of blobs each entry is the IPv6 address structure in6\_addr declared in 6LP.h.

#### Access

*Read*

#### Values

*Any*

#### Default

*0*                                  No scenes configured

#### Trap Notifications

On remote edits.



---

## 5.4 Remote MIBs

The Remote MIBs provide a common level of functionality for remote control devices.

---

### 5.4.1 RemoteConfigGroup MIB (0xFFFFFE25)

The RemoteConfigGroup MIB contains variables to configure the group addresses associated with the remote's group buttons. These are used when transmitting broadcast group commands from the remote.

---

#### 5.4.1.1 Count Variable

##### Description

The Count variable specifies the number of group addresses supported by the remote control.

##### Storage

Permanent

##### Type

*UInt8*                      Unsigned Integer, 8 bits

##### Access

*Read*

##### Values

*0 to 255*                      Number of group addresses supported by device.

##### Default

*?*                              Device dependant.

##### Trap Notifications

None.

### 5.4.1.2 Finish Variable

#### Description

The Finish variable is used to notify the remote control that configuration has finished when the remote joins a network.

#### Storage

Volatile

#### Type

*UInt8* Unsigned Integer, 8 bits

#### Access

*Read, Write*

#### Values

0 Configuration not finished.  
1 to 255 Configuration finished.

#### Default

? Device dependant.

#### Trap Notifications

None.

### 5.4.1.3 AddrX Variables

#### Description

The AddrX variables (where X is numeric starting from 0) provide access to the group address variables associated with each button.

#### Storage

Permanent

#### Type

*Blob* Blob, 128 bits

#### Access

*Read, Write*

#### Values

*Any* Group address.

#### Default

? Device dependant.

#### Trap Notifications

None.

---

## 5.5 Device MIBs

The Device MIBs provide a common level of functionality across many device types. This allows basic control of different devices that implement the Device MIBs.

The variables in these MIBs duplicate a subset of the variables in the device specific MIBs.

---

### 5.5.1 DeviceConfig MIB (0xFFFFFEA1)

The DeviceConfig MIB is reserved for future use.

---

### 5.5.2 DeviceStatus MIB (0xFFFFFEA0)

The DeviceStatus MIB is reserved for future use.

## 5.5.3 DeviceControl MIB (0xFFFFFEA2)

The DeviceControl MIB contains variables that can be used to control generic devices.

### 5.5.3.1 Mode Variable

#### Description

The Mode variable specifies the operating mode of the device. At its most basic it can be used to turn the device on and off however it also provides access to additional modes for more advanced use.

#### Storage

Permanent

#### Type

*Uint8*                      Unsigned Integer, 8 bits

#### Access

*Read, Write*

#### Values

<i>0 (Off)</i>	In this mode the device is turned off.
<i>1 (On)</i>	In this mode the device is turned on.
<i>2 (Toggle On/Off)</i>	Writing this mode toggles between the On and Off modes but only if the Mode is already set to Off or On. If the device is in On or Off mode when this value is written the request will be successful but the final value of the variable will be Off or On as appropriate. If the device is in any other mode when this value is written the remote write will return failed.

#### Default

?                      Device dependant.

#### Trap Notifications

On remote set.  
When the mode is changed as a result of applying a scene.

---

### 5.5.3.2 Sceneld Variable

#### Description

The Sceneld variable when set switches the device to the settings for the specified Scene ID, if settings for the specified Scene ID are present in the Device.

#### Storage

Permanent

#### Type

*Uint16*                      Unsigned Integer, 16 bits

#### Access

*Read, Write*

#### Values

<i>0</i>	None.
<i>1 to 65535</i>	Scene ID to switch to.

#### Default

<i>0</i>	None.
----------	-------

#### Trap Notifications

On remote set.

## 5.5.4 DeviceScene MIB (0xFFFFFEA3)

The DeviceScene MIB contains variables that can be used to configure scenes in generic devices.

As this MIB is intended for use in many different device types the data and implementation of the MIB varies depending upon the device. This MIB is currently only implemented in the bulb devices.

### 5.5.4.1 AddSceneld Variable

#### Description

When written to the AddSceneld variable adds or updates the specified scene ID with the current settings of the device. If the Scene ID is new but there is not an unused Sceneld variable then the write will fail.

#### Storage

Volatile

#### Type

*Uint16*                      Unsigned Integer, 16 bits

#### Access

*Read, Write*

#### Values

*1 to 65535*                      Scene ID to add.

#### Default

*0*                                  No scene added.

#### Trap Notifications

On remote edits.

---

### 5.5.4.2 DelSceneld Variable

#### Description

When written to the DelSceneld variable deletes the specified Scene ID if in use.

#### Storage

Volatile

#### Type

*Uint16*                      Unsigned Integer, 16 bits

#### Access

*Read, Write*

#### Values

*1 to 65535*                      Scene ID to delete.

#### Default

*0*                                  No scene deleted.

#### Trap Notifications

On remote edits.

### 5.5.4.3 AddScene Variable

#### Description

When written to the AddScene variable sets or updates the specified scene with all the settings for the scene. This allows a scene to be set without having to alter the device's current settings.

The data structure used will be different in different devices types and also incorporates the Device Type ID that the scene applies to, allowing devices with a non-matching Device Type ID to ignore the request.

#### Storage

Volatile

#### Type (White Bulb – Device Type ID 0x00E1)

*Blob* Blob with the following structure declared in **MibDeviceScene.h**:

```
typedef struct
{
    uint16 u16DeviceTypeId = 0x00E1;
    uint16 u16SceneId;
    uint8  u8BulbControlMode;
    uint16 u8BulbControlLumTarget;
} PACK tsDeviceScene;
```

#### Type (CCT Bulb – Device Type ID 0x00F2)

*Blob* Blob with the following structure declared in **MibDeviceScene.h**:

```
typedef struct
{
    uint16 u16DeviceTypeId = 0x00F2;
    uint16 u16SceneId;
    uint8  u8BulbControlMode;
    uint16 u8BulbControlLumTarget;
    uint8  u8ColourControlMode;
    uint16 u16ColourControlCctTarget;
} PACK tsDeviceScene;
```

#### Type (Colour Bulb – Device Type ID 0x00F3)

*Blob* Blob with the following structure declared in **MibDeviceScene.h**:

```
typedef struct
{
    uint16 u16DeviceTypeId = 0x00F3;
    uint16 u16SceneId;
    uint8  u8BulbControlMode;
    uint16 u8BulbControlLumTarget;
    uint8  u8ColourControlMode;
    uint32 u8ColourControlXYTarget;
} PACK tsDeviceScene;
```



**Access**

*Read, Write*

**Values**

*Any*

**Default**

*0*                      No scene set

**Trap Notifications**

On remote edits.

#### 5.5.4.4 SceneTable Variable

##### Description

The SceneTable table variable contains the Scene IDs and settings for all scenes the device is taking part in.

##### Storage

Permanent

##### Type (White Bulb – Device Type ID 0x00E1)

*Table Blob*

Each blob entry in the table has the following structure declared in **MibDeviceScene.h**:

```
typedef struct
{
    uint16 u16DeviceTypeId = 0x00E1;
    uint16 u16SceneId;
    uint8  u8BulbControlMode;
    uint16 u8BulbControlLumTarget;
} PACK tsDeviceScene;
```

##### Type (CCT Bulb – Device Type ID 0x00F2)

*Table Blob*

Each blob entry in the table has the following structure declared in **MibDeviceScene.h**:

```
typedef struct
{
    uint16 u16DeviceTypeId = 0x00F2;
    uint16 u16SceneId;
    uint8  u8BulbControlMode;
    uint16 u8BulbControlLumTarget;
    uint8  u8ColourControlMode;
    uint16 u16ColourControlCctTarget;
} PACK tsDeviceScene;
```

##### Type (Colour Bulb – Device Type ID 0x00F3)

*Table Blob*

Each blob entry in the table has the following structure declared in **MibDeviceScene.h**:

```
typedef struct
{
    uint16 u16DeviceTypeId = 0x00F3;
    uint16 u16SceneId;
    uint8  u8BulbControlMode;
    uint16 u8BulbControlLumTarget;
    uint8  u8ColourControlMode;
    uint32 u8ColourControlXYTarget;
} PACK tsDeviceScene;
```

**Access**

*Read*

**Values**

*Any*

**Default**

*0*

No scenes configured

**Trap Notifications**

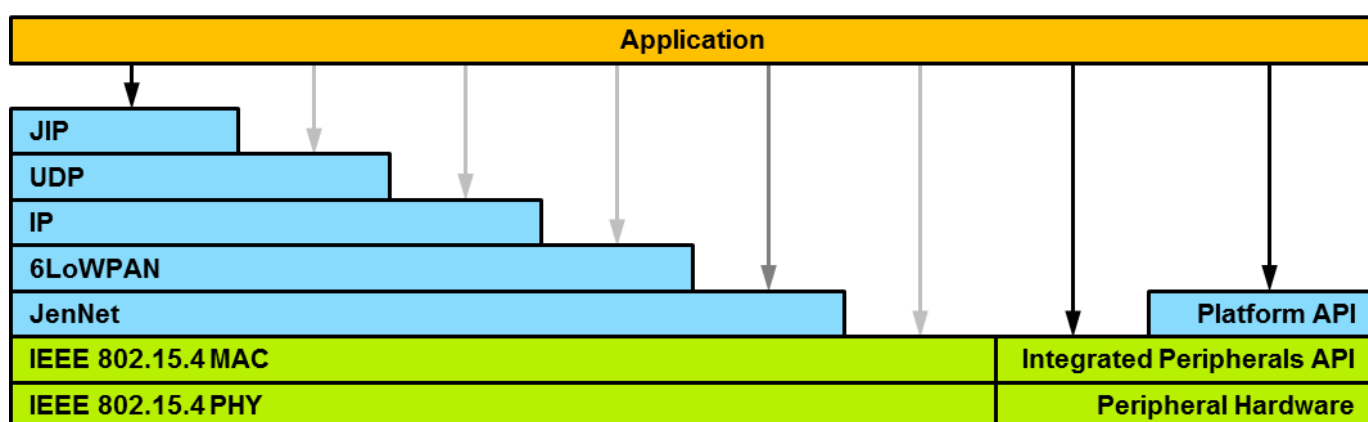
On remote edits.

## 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 **DeviceType**, 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 **DeviceType.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.

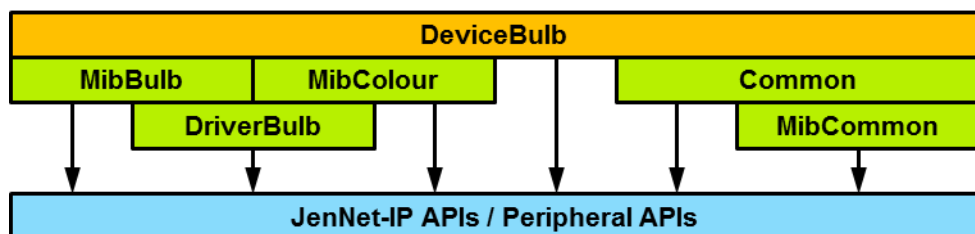
The core application functionality of a JenNet-IP device is provided by the code for the MIBs that the device supports. The code for the MIBs is grouped together where a set of MIBs provide related functionality. The code for the MIBs can be found in folders named **MibGroup** where *Group* is the name 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 **MibGroup** folders also exist to provide MIBs that are specific to certain device types. For example, the **MibBulb** folder contains the code for the bulb MIBs used by the bulb device. The functions that implement these MIBs are called directly from the **DeviceType.c** module.

Some device types will also include a set of hardware driver source files in a folder named **DriverType** 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}\_DeviceType\_  
{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 **DeviceType.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\_eJiplInit()** 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\_v6lpDataEvent()** function.

As this application is written to operate at the JIP level (reading and writing to MIB variables), any packets received from this level of the stack are simply discarded by **Node\_v6lpDataEvent()**.

---

```
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 **DeviceType.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 **MibGroup**, 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 **MibName** below, where **Name** should be replaced with the actual MIB's name.

---

#### 6.1.3.1 MibGroup.h

**MibGroup.h** contains defines used throughout the **MibGroup** 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 MibNameDef.h

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

---

#### 6.1.3.3 MibNameDec.c

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

Each MIB has a header file named **MibName.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 **tsMibNamePerm**, 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 **tsMibNameTemp**. The members of this structure map onto the temporary variables of the MIB.

---

```
typedef struct tsMibName;
```

Each MIB has a structure named **tsMibName** 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 MibName.c

Each MIB has a source file named **MibName.c**. These source files implement the functions required of each MIB. Many MIBs contain similar functions that carry out a common task in each MIB (though the effects in each MIB differ). For example, all MIBs contain an 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 **MibName\_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 DeviceBulb Folder

The **DeviceBulb** folder of the Application Note contains source code that is specific to the bulb devices in *JenNet-IP Smart Home (JN-AN-1162)*. The bulb device operates as a Router node extending the network for other devices to join. It is able to join and maintain its place within the network in addition to providing bulb functionality.

The bulb application is based upon the template device implemented in *JenNet-IP Application Template (JN-AN-1190)*. This section details the additions to template to create a bulb device. The reader should refer to *JenNet-IP Application Template (JN-AN-1190)* for details of how the template software works, including the implementation of the Common MIBs.

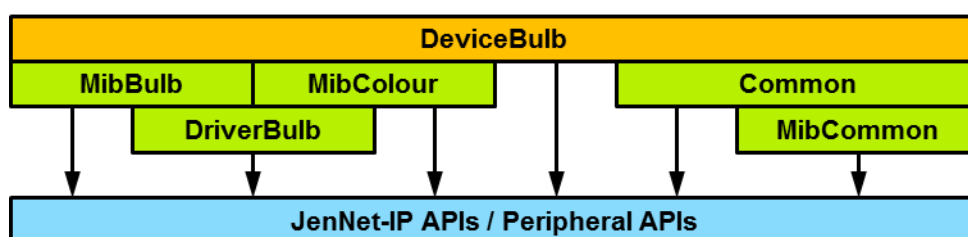
The common source **Common\Node.c** actually provides the majority of the network joining and maintenance code. **Node.c** in turn makes use of a number of MIBs that can be used to monitor, configure and control an individual node and its operation within the network. These MIBs are implemented in the **MibCommon** folder. This common code is described in *JenNet-IP Application Template (JN-AN-1190)*.

The source code in the **DeviceBulb** folder contains the main module implementing the standard JIP callback functions, these callback functions then make calls into the stack and MIB libraries as required.

The source code for the MIBs and their variables used to monitor and control the bulbs are contained in the **MibBulb** folder that is described in section [6.3 MibBulb Folder](#). The CCT and colour bulb software also makes use of a set of colour MIBs, the source code for these is in the **MibColour** folder.

The other source files are then hardware level drivers that are mainly called by code in the **MibBulb** library to control the operation of the bulb at the hardware layer.

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



---

## 6.2.1 DeviceBulb Makefile

The **makefile** for the bulb uses the variables listed in [Section 6.1.1.1 "Standard Device Type Makefile"](#) with the following differences:

---

### DEVICE\_NAME

The bulb device supports the following values:

- **JN516X** for native PWM output on the JN516x's timer pins. This can be compiled as a white bulb, which will drive the three white LEDs on the *Lighting/Sensor Expansion Board (DR1175)*, or as a colour bulb driving three PWM outputs for red, green and blue (this is included as an example only).
- **DR1175** to drive the RGB LED on the *Lighting/Sensor Expansion Board (DR1175)* using the PCA9634 driver chip. This can be compiled as a white, CCT or colour bulb all of which drive the RGB LED.
- **DR1173** to drive the NXP prototype RGB board (not included in the evaluation kit). This can only be compiled as a colour bulb.
- **DR1190** to drive the NXP SSL2108 asynchronous bulb reference design (not included in the evaluation kit). This can only be compiled as a white bulb.
- **DR1192** to drive the NXP SSL2108 synchronous bulb reference design (not included in the evaluation kit). This can only be compiled as a white bulb.
- **DR1221** to drive the NXP CCT bulb reference design (not included in the evaluation kit). This can only be compiled as a CCT bulb.

---

### NODE\_TYPE

The bulb device can only be compiled as a Router node type.

---

### DRIVER\_TYPE

This variable specifies which type of bulb to build and supports the following values:

- **White** to build a white bulb.
- **Temperature** to build a CCT bulb.
- **Colour** to build a colour bulb.

---

### DEVICE\_DOZE

When set to 1 the bulb software is placed in doze mode to preserve power when not processing, this is the default.

The DR1192 build is unable to accurately drive the bulb when dozing so doze mode is disabled by setting the value to 0.

---

#### **BLD\_MIB\_NAME Variables**

The bulb device makefile adds additional variables to control the building of the bulb and colour MIBs.

---

#### **REG\_MIB\_NAME Variables**

The bulb device makefile adds additional variables to control the registration of the Bulb and Colour MIBs.

---

#### **6.2.1.1 Binary File Naming**

The names of the bulb binary files incorporates the DRIVER\_TYPE variable to indicate if it is a white, CCT or colour bulb.

---

#### **6.2.2 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"](#) with the following differences:

The number of scenes available in the DeviceScene MIB is defined.

Default values for some bulb MIB variables are defined.

Additional debug flags for bulb and colour MIBs are included.



---

### 6.2.3 DeviceBulb.c

**DeviceBulb.c** contains the main source code for the bulb application. It follows the pattern described in [Section 6.1.1.3 "Standard DeviceType.c Features"](#) with the following changes:

---

#### 6.2.3.1 #includes

Additional #includes are used to provide access to the bulb and colour MIB modules used in **DeviceBulb.c**.

---

#### 6.2.3.2 External Variables

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

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

---

#### 6.2.3.3 Public Functions

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

---

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

The bulb driver software is initialised early on with a call to **DriverBulb\_vInit()** in order to turn the bulb on as soon as possible.

---

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

The bulb and colour MIBs are initialised here *following* the call to **Node\_vPdmInit()** once the PDM has been initialised.

---

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

The factory reset of the bulb and colour MIB's permanent data is performed here. **Device\_eJipInit()** Function

---

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

The bulb and colour MIBs are registered here.

---

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

Stack events are passed to the BulbControl and ColourControl MIBs here.

---

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

This function includes the same code as **DeviceTemplate.c** but adds:

- A call to **vDecimator()** when the thermal control loop is included in the application to allow compensation for temperature changes.
- A call to **MibBulbStatus\_vAnalyse()** to allow the bus voltage of the bulb to be monitored by the BulbStatus MIB.

---

```
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_vSecond ( uint32 u32TimerSeconds );
```

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

The timer is simply passed on to the bulb and colour MIBs that perform timing tasks based on the 1 second timer.

---

```
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 a production bulb build the bulb software is simply restarted.

In non-production bulb builds the bulb driver is used to directly flash the bulb bright and dim to indicate that an exception has taken place. The number of flashes indicates the exception type.

---

## 6.2.4 DeviceScene MIB

The **DeviceBulb** folder contains the code for the DeviceScene MIB with its specific implementation for bulb devices. This MIB follows the pattern described in [Section 6.1.3 "Standard MIB Module Features"](#) with the following changes.

---

### 6.2.4.1 MibDeviceScene.c

This source file contains the code that implements the DeviceScene MIB for bulb devices.

---

#### 6.2.4.1.1 External Variables

The external variables for the BulbScene, BulbControl and ColourControl MIBs data structures are declared to allow access by the DeviceScene MIB software.

---

#### 6.2.4.2 Public Functions

The following public functions are implemented in **MibDeviceScene.c**.

---

```
void MibDeviceScene_vInit ( void );
```

The scene data, read from the PDM, is transferred to the BulbScene MIB's data structures to keep the two MIBs synchronised for backwards compatibility.

---

```
uint8 MibDeviceScene_u8FindSceneId ( uint16 u16SceneId );
```

This function searches through the scene table looking for a specific Scene ID. If found the index of the scene in the table is returned.

---

```
teJIP_Status MibDeviceScene_eMakeScene ( uint16 u16SceneId );
```

This function creates or updates the specified scene using the current state of the device from the appropriate MIBs.

---

```
teJIP_Status MibDeviceScene_eAddScene (
                                     tsDeviceScene *psAddScene );
```

This function creates or updates the specified scene using the settings specified in the passed in scene structure. It only does this when the Device Type ID in the structure matches that of the device running the software.

---

```
teJIP_Status MibDeviceScene_eDelScene ( uint16 u16SceneId );
```

This function deletes the specified scene from the scene table.

---

```
teJIP_Status MibDeviceScene_eActivateScene (
                                           uint16 u16SceneId );
```

This function activates the specified scene, if stored, by calling functions in the appropriate MIBs.

---

```
void MibDeviceScene_vBulbScene ( uint8  u8Scene,
                                bool_t  bNotifyChange );
```

This function transfers a scene from the DeviceScene MIB to the BulbScene MIB to keep the two MIBs synchronised for backwards compatibility.

---

```
teJIP_Status MibDeviceScene_eSetAddSceneId ( uint16 u16Val,
                                              void    *pvCbData );
```

This is the callback function for a write to the AddSceneId variable. It creates or updates the specified scene using the device's current settings.

---

```
teJIP_Status MibDeviceScene_eSetDelSceneId ( uint16 u16Val,
                                              void    *pvCbData );
```

This is the callback function for a write to the DelSceneId variable. It deletes the specified scene.

---

```
teJIP_Status MibDeviceScene_eSetAddScene (
                                          const uint8 *pu8Val,
                                          uint8       u8Len,
                                          void        *pvCbData );
```

This is the callback function for a write to the AddScene variable. It creates or updates the specified scene using the settings included in the data structure.

---

```
void MibDeviceScene_vGetAddScene ( thJIP_Packet  hPacket,
                                   void          *pvCbData );
```

This is the callback function for reading the AddScene variable. It adds the variable data to the specified packet.

---

## 6.2.5 BulbScene MIB

The **DeviceBulb** folder contains the code for the BulbScene MIB with its specific implementation for bulb devices.

It is recommended that the generic DeviceScene MIB be used to configure scenes. The BulbScene MIB is provided for backwards compatibility with older software that expects to use this MIB to configure scenes in bulbs. The main code to control scenes is included in the DeviceScene MIB, the majority of the BulbScene MIB code therefore simply accesses the data and functions in the DeviceScene MIB.

This MIB follows the pattern described in [Section 6.1.3 "Standard MIB Module Features"](#) with the following changes.

---

### 6.2.5.1 MibBulbScene.c

This source file contains the code that implements the BulbScene MIB.

---

#### 6.2.5.1.1 External Variables

The external variable for the DeviceScene MIB's data structure is declared to allow access by the BulbScene MIB software.

---

#### 6.2.5.2 Public Functions

The following public functions are implemented in **MibDeviceScene.c**.

---

```
void MibBulbScene_vInit ( void );
```

This function is empty. The scene data is transferred into the BulbScene data tables by the DeviceScene MIB. The BulbScene MIB does not store any data of its own.

---

```
teJIP_Status MibBulbScene_eSetAddSceneId ( uint16 u16Val,  
                                           void    *pvCbData );
```

This is the callback function for a write to the AddSceneId variable. It creates or updates the specified scene using the device's current settings by calling into the DeviceScene MIB.

---

```
teJIP_Status MibBulbScene_eSetDelSceneId ( uint16 u16Val,  
                                           void    *pvCbData );
```

This is the callback function for a write to the DelSceneId variable. It deletes the specified scene by calling into the DeviceScene MIB.

---

## 6.3 MibBulb Folder

The **MibBulb** folder contains modules that implement MIBs for configuring, monitoring and controlling dimmable bulbs.

---

### 6.3.1 BulbConfig MIB

The BulbConfig MIB allows the operation of the bulb to be configured. The rate at which the brightness changes can be configured. The initial mode and brightness of the bulb when power is applied can be set. Cadence effects can be configured to provide feedback when the bulb joins and leaves the network.

This MIB follows the pattern described in [Section 6.1.3 "Standard MIB Module Features"](#) with the following alterations.

---

#### 6.3.1.1 MibBulbConfig.c

This source file contains the code that implements the BulbConfig MIB.

---

##### Public Functions

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

---

```
teJIP_Status MibBulbConfig_eSetUint8 ( uint8  u8Val,
                                       void    *pvCbData );
teJIP_Status MibBulbConfig_eSetUint16 ( uint16 u16Val,
                                       void    *pvCbData );
teJIP_Status MibBulbConfig_eSetUint32 ( uint32 u32Val,
                                       void    *pvCbData );
```

These generic functions are called by the stack to set the value of variables in the BulbConfig MIB and are specified in the MIB declaration in **MibBulbConfigDec.c**. When these functions are called the new values are saved by the PDM.

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

---

```
teJIP_Status MibBulbConfig_eSetLumRate ( uint8  u8Val,
                                       void    *pvCbData );
```

This function is called by the stack to set the value of the LumRate variable in the BulbConfig MIB and is specified in the MIB declaration in **MibBulbConfigDec.c**. This function validates the new value then saves it using the PDM.

---

### 6.3.2 BulbStatus MIB

The BulbStatus MIB provides information on the status of the bulb, including counts for how many times the bulb has been illuminated, length of time illuminated, temperature and bus voltage.

This MIB follows the pattern described in [Section 6.1.3 "Standard MIB Module Features"](#) with the following alterations.

---

#### 6.3.2.1 MibBulbStatus.c

This source file contains the code that implements the BulbStatus MIB.

---

##### Public Functions

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

---

```
void MibBulbStatus_vSecond ( void );
```

Timers recording how long the bulb has been illuminated are updated here.

---

```
void MibBulbStatus_vAnalogue ( uint8 u8Adc );
```

This function should be called when an analogue reading has completed.

Readings from the bus voltage ADC are passed into the bulb driver via the **DriverBulb\_i16Analogue()** function and the returned value stored in BusVolts MIB variable.

Readings from the on-chip temperature sensor are converted into 10<sup>th</sup> of degree Centigrade and stored in the ChipTemp MIB variable.

---

```
void MibBulbStatus_vOn ( void );
```

This function should be called each time the bulb is illuminated to allow the status of the bulb to be maintained.

---

```
void MibBulbStatus_vOff ( void );
```

This function should be called each time the bulb is extinguished to allow the status of the bulb to be maintained.

### 6.3.3 BulbControl and Device Control MIBs

The BulbControl MIB allows the bulb to be turned on or off and provides control of the brightness level. Scenes can be activated and cadence effects displayed for user feedback.

The source code for this MIB also implements the DeviceControl MIB which contains a subset of the BulbControl variables. These variables allow generic devices to be controlled in a standard way. The DeviceControl MIB allows devices to be turned on or off and scenes may be activated.

This MIB follows the pattern described in [Section 6.1.3 "Standard MIB Module Features"](#) with the following alterations.

#### 6.3.3.1 MibBulbControl.c

This source file contains the code that implements the BulbControl MIB.

#### Public Functions

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

```
void MibBulbControl_vInit (
    thJIP_Mib          hMibBulbControlInit,
    thJIP_Mib          hMibDeviceControlInit,
    tsMibBulbControl *psMibBulbControlInit,
    void               *psMibBulbStatusInit,
    void               *psMibBulbConfigInit,
    void               *psMibBulbSceneInit,
    void               *psMibCOlourControlInit );
```

If the data for the Groups MIB was not loaded from the PDM the bulb is placed into the "All Bulbs" group.

If the BulbConfig MIB specifies that the bulb should be set to a specific mode and luminance level at initialisation the settings are transferred from the BulbConfig MIB.

The initial mode of the bulb is validated to ensure it does not initialise to a state that is altering the bulb's luminance.

The bulb driver is initialized.

The network down cadence effect is applied if required.

```
void MibBulbControl_vDeviceControlRegister ( void );
```

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



---

```
void MibBulbControl_vSecond ( void );
```

The driver is checked to see if the hardware has failed. In the event of a failure the bulb is placed into failed mode and further control of the bulb is prevented.

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

---

```
void MibBulbControl_vTick ( void );
```

This function should be called every 10ms driven by the tick timer. It is used for timing small intervals, in particular controlling the fading of the bulb.

The tick is passed down to the bulb driver via the **DriverBulb\_vTick()** function to allow the timing of small intervals in the driver.

The tick is also passed into the **MibBulbControl\_vTickDriverReady()** function that monitors and reacts to the ready state of the bulb driver.

Next **MibBulbControl\_bTickLumCadence()** is called to update the bulb's brightness if cadence effects are being applied.

If no cadence effects are being applied a helper function is called depending upon the current mode of the bulb:

- **MibBulbControl\_vTickModeOff()**, when the bulb is in the OFF Mode.
- **MibBulbControl\_vTickModeOn()**, when the bulb is in the ON Mode.
- **MibBulbControl\_vTickModeDownUp()**, when the bulb is in the DOWN, UP, DOWN\_IF\_ON or UP\_IF\_ON Modes.
- **MibBulbControl\_vTickModeTest()**, when the bulb is in the TEST Mode.

---

```
void MibBulbControl_vTickDriverReady ( void );
```

This function is called every 10ms.

The bulb driver is checked to see if the bulb is ready to be driven at the hardware level, using the **DriverBulb\_bReady()** function.

If the driver becomes ready to control the latest user mode and brightness are applied to the bulb.

---

```
bool_t MibBulbControl_bTickLumCadence ( void );
```

This function applies cadence effects if required.

The cadence timer is updated, once expired the cadence display is ended.

If the cadence timer is still running the cadence effect is applied by fading or switching the brightness of the bulb as required.

---

```
void MibBulbControl_vTickModeTest ( void );
```

This function updates the brightness of the bulb while in test mode using the **MibBulbControl\_bFadeLumCurrent()** function.

While the node is a member of a network the brightness is set to the level of the signal strength to the parent node.

While the node is not a member of a network the brightness is faded between the maximum and minimum.

---

```
void MibBulbControl_vTickModeOff ( void );
```

This empty function is a placeholder to be used if timings need to be made while the bulb is off.

---

```
void MibBulbControl_vTickModeOn ( void );
```

This function controls the fading of the bulb's brightness while it is in the ON Mode using the **MibBulbControl\_bFadeLumCurrent()** function. This is done in response to the TargetLuminance variable being set.

---

```
void MibBulbControl_vTickModeDownUp ( void );
```

This function controls the fading of the bulb's brightness while it is in an UP, DOWN, UP\_IF\_ON or DOWN\_IF\_ON mode. The bulb's brightness is altered every 10ms until the Mode is changed or the maximum or minimum brightness is reached.

---

```
bool_t MibBulbControl_bFadeLumCurrent ( uint8 u8LumTarget,
                                         uint8 u8LumRate );
```

This function controls the current fade level of the bulb moving the current fade level towards the target fade level.

---

```
void MibBulbControlPatch_vStackEvent (
                                         te6LP_StackEvent eEvent );
```

Stack events should be passed into this function so the bulb can react to joining and leaving the network.

When the network is joined the state of the device within the network is updated and flagged to be written to the PDM. The event is passed into the **MibBulbControl\_vLumCadenceStackEvent()** function to apply a cadence effect if configured to do so.

When the network is lost the state of the device within the network is updated and flagged to be written to the PDM. The event is passed into the **MibBulbControl\_vLumCadenceStackEvent()** function to apply a cadence effect if configured to do so.

---

```
uint8 MibBulbControl_u8ParentLqi ( void );
```

This function reads the parent node's LQI and is used when in TEST mode.

---

```
void MibBulbControl_vLumCadence ( uint32 u32LumCadence,  
                                  int16  u16LumCadTimer );
```

This function is used to start or end displaying a cadence effect.

---

```
void MibBulbControl_vLumCadenceStop (void );
```

This function is used to stop displaying a cadence effect.

---

```
bool_t MibBulbControl_bLumCadence ( void );
```

This function is used to test if a cadence effect is currently being displayed.

---

```
void MibBulbControl_vLumCadenceStackEvent (  
                                             te6LP_StackEvent eEvent );
```

This function applies cadence effects, if configured to do so, when joining or leaving a network.

---

```
teJIP_Status MibBulbControl_eSetMode (uint8  u8Val,  
                                       void  *pvCbData );
```

This callback function is called by to handle writes to the Mode variable.

---

```
teJIP_Status MibDeviceControl_eSetMode ( uint8  u8Val,  
                                          void  *pvCbData );
```

This function is called by the stack to set the value of the Mode variable in the DeviceControl MIB.

It implements the handling for the OFF, ON and TOGGLE Mode variable values.

---

```
teJIP_Status MibBulbControl_eSetSceneId ( uint16 u16Val,  
                                           void   *pvCbData );
```

This function is called by the stack to set the value of the SceneId variable in the BulbControl MIB as specified by the declaration in **MibBulbControlDec.c**.

This function simply calls the **MibBulbDeviceControl\_eSetSceneId()** helper function.

---

```
teJIP_Status MibDeviceControl_eSetSceneId ( uint16 u16Val,  
                                             void   *pvCbData );
```

This function is called by the stack to set the value of the SceneId variable in the DeviceControl MIB as specified by the declaration in **MibBulbControlDec.c**.

This function simply calls the **MibBulbDeviceControl\_eSetSceneId()** helper function.

---

```
teJIP_Status MibBulbDeviceControl_eSetSceneId (
    uint16 u16Val,
    void    *pvCbData,
    bool_t   bBulbControl );
```

This helper function is called to activate a scene if the bulb is participating in the scene with the specified ID by setting the bulb's mode and brightness as specified for the scene.

If the bulb is not taking part in the scene no action is taken.

---

```
teJIP_Status MibBulbControl_eSetLumTarget ( uint8  u8Val,
    void    *pvCbData );
```

This function is called by the stack to set the value of the LumTarget variable in the BulbControl MIB as specified by the declaration in **MibBulbControlDec.c**.

When the bulb is illuminated it will begin to fade the brightness towards this value.

---

```
teJIP_Status MibBulbControl_eSetLumCurrent ( uint8  u8Val,
    void    *pvCbData );
```

This function is called by the stack to set the value of the LumCurrent variable in the BulbControl MIB as specified by the declaration in **MibBulbControlDec.c**.

When the bulb is illuminated it will immediately change the brightness to this value.

---

```
teJIP_Status MibBulbControl_eSetLumCurrent ( int16 i16Val,
    void    *pvCbData );
```

This function is called by the stack to set the value of the LumChange variable in the BulbControl MIB as specified by the declaration in **MibBulbControlDec.c**.

When the bulb is illuminated it will fade up or down by the set amount specified by the new variable value.

---

```
teJIP_Status MibBulbControl_eSetLumCadence (
    uint32  u32Val,
    void    *pvCbData );
```

This function is called by the stack to set the value of the LumCadence variable in the BulbControl MIB as specified by the declaration in **MibBulbControlDec.c**.

This value when used in conjunction with the LumCadTimer variable causes the bulb to display brightness fading or switch effects to provide user feedback. This variable sets the effect type, timings and brightness levels for the cadence effect.

This function simply calls the **MibBulbControl\_vLumCadence()** helper function.

---

```
teJIP_Status MibBulbControl_eSetLumCadTimer (
                                                    uint16  u16Val,
                                                    void    *pvCbData );
```

This patch function is called by the stack to set the value of the LumCadTimer variable in the BulbControl MIB as specified by the declaration in **MibBulbControlDec.c**.

This value when used in conjunction with the LumCadence variable causes the bulb to display brightness fading or switch effects to provide user feedback. This variable sets the duration of cadence effect.

This function simply calls the **MibBulbControl\_vLumCadence()** helper function.

---

```
teJIP_Status MibBulbControl_eSetModeOff ( uint8 *pu8Mode );
```

This helper function handles the Mode variable being set to OFF by extinguishing the bulb if a cadence effect is not being displayed.

---

```
teJIP_Status MibBulbControl_eSetModeOn ( uint8 *pu8Mode );
```

This helper function handles the Mode variable being set to ON by illuminating the bulb if a cadence effect is not being displayed. The brightness is also set to the required level if necessary.

---

```
teJIP_Status MibBulbControl_eSetModeDownUp ( uint8 *pu8Mode );
```

This helper function handles the Mode variable being set to UP, DOWN, UP\_IF\_ON or DOWN\_IF\_ON when a cadence effect is not being displayed. The bulb is illuminated if required.

---

```
teJIP_Status MibBulbControl_eSetModeToggle ( uint8 *pu8Mode );
```

This helper function handles the Mode variable being set to TOGGLE when a cadence effect is not being displayed. This toggles the bulb mode between ON and OFF.

---

```
teJIP_Status MibBulbControl_eSetModeTest ( uint8 *pu8Mode );
```

This helper function handles the Mode variable being set to TEST by illuminating the bulb if a cadence effect is not being displayed.

---

```
uint8 MibBulbControl_u8FindSceneId ( uint16 u16SceneId );
```

This function is only compiled if the DeviceScene MIB is not included in the application and is used to find a specific scene in the BulbScene MIB's scene table.

## 6.4 MibColour Folder

The **MibColour** folder contains modules that implement MIBs for configuring, monitoring and controlling the colour of CCT and colour bulbs.

This folder also contains modules for converting between different colour spaces, transitioning from one colour to another over time and a generic interpolation engine that is used during transitions.

### 6.4.1 ColourConfig MIB

The ColourConfig MIB allows the colour operation of the bulb to be configured.

The rate at which the colour changes can be configured. The initial mode and colour of the bulb when power is applied can be set. The colour primary points used to generate accurate colours and the range of CCT values are also accessible.

This MIB follows the pattern described in [Section 6.1.3 "Standard MIB Module Features"](#) with the following alterations.

#### 6.4.1.1 MibColourConfig.c

This source file contains the code that implements the ColourConfig MIB.

#### Public Functions

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

---

```
void MibColourConfig_vInit ( void );
```

If configuration data is not read from the PDM the variables are initialised to a set of defaults. In this case the InitXYTarget variable is set to white and the primaries are set to generic default values.

The matrices used for converted between colour spaces are then calculated from the primaries.

---

```
teJIP_Status MibColourConfig_eSetTransitionTime (
    uint16 u16Val,
    void *pvCbData );
```

This callback function is called when the TransitionTime variable is set remotely. It validates the new value and writes the data to the PDM.

---

```
teJIP_Status MibColourConfig_eSetInitMode ( uint8 u8Val,
    void *pvCbData );
```

This callback function is called when the InitMode variable is set remotely.

---

```
teJIP_Status MibColourConfig_eSetInitXYTarget (
    uint32 u32Val,
    void *pvCbData );
```

This callback function is called when the InitXYTarget variable is set remotely.

---

```
teJIP_Status MibColourConfig_eSetPrimary ( uint32 u32Val,  
                                           void    *pvCbData );
```

This callback function is called when the colour primary values are set remotely.  
The colour conversion matrices are recalculated.

---

```
teJIP_Status MibColourConfig_eSetCct ( uint16 u16Val,  
                                       void    *pvCbData );
```

This callback function is called when the CCT limit variables are set remotely.

---

## 6.4.2 ColourControl MIB

The ColourControl MIB allows the colour operation of the bulb to be controlled.

The colour of the bulb can be set using different colour spaces with changes being made immediately or over time.

This MIB follows the pattern described in [Section 6.1.3 "Standard MIB Module Features"](#) with the following alterations.

---

### 6.4.2.1 MibColourControl.c

This source file contains the code that implements the ColourControl MIB.

---

#### Public Functions

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

---

```
void MibColourControl_vInit ( void );
```

If data is not read from the PDM by the Groups MIB the bulb is placed into the "All Colour Bulbs" group

If the ColourControl MIB's data is not read from the PDM the target colour is initialised to white.

When the ColourConfig MIB specifies an initialisation mode and colour the values are transferred into the ColourControl MIB.

The data structures used to transition from one colour to another over time are initialised.

The representations of the colour that are not stored in the PDM are derived from the starting colour.

The bulb driver is used to set the initial colour of the bulb.

---

```
void MibColourControl_vSecond ( void );
```

If data stored in the PDM has changed the data is written to the PDM. This ensures that there is no more than one write per second of data that might change frequently to the EEPROM.

---

```
void MibColourControl_vTick ( void );
```

This function should be called every 10ms.

If the bulb is transitioning between two colours the next step of the transition is calculated and the bulb driver used to update the colour of the bulb.

When the final step of a transition has been calculated the variables that contain the current colour of the bulb are updated.

If the bulb completes a transition for one of the UP or DOWN modes due to reaching the maximum or minimum value, the target variables are updated with the final colour.

If the bulb completes a transition for one of the LOOP or TEST modes the next transition for the mode is started.

---

```
bool_t MibColourControl_bModeStop ( bool_t bInternal );
```

This function is called when the Mode variable is set to STOP.

If a colour transition is running it is stopped and the colour variables updated with the current colour.

When in a test mode the colour that was displayed before entering test mode is restored.

---

```
teJIP_Status MibColourControl_eSetMode ( uint8  u8Val,  
                                         void  *pvCbData );
```

This callback function is called when the Mode variable is set remotely. Internal functions are used to put the new mode into effect.

---

```
teJIP_Status MibColourControl_eSetSceneId ( uint16  u16Val,  
                                           void  *pvCbData );
```

This callback function is called when the SceneId variable is set remotely. It calls the **MibDeviceScene\_eActivateScene()** function to activate the specified scene.



---

```
teJIP_Status MibColourControl_eSetXYTarget ( uint32 u32Val,  
                                              void    *pvCbData );  
teJIP_Status MibColourControl_eSetXTarget  ( uint16 u16Val,  
                                              void    *pvCbData );  
teJIP_Status MibColourControl_eSetYTarget  ( uint16 u16Val,  
                                              void    *pvCbData );  
teJIP_Status MibColourControl_eSetHueTarget (          uint16 u16Val,  
                                              void    *pvCbData );  
teJIP_Status MibColourControl_eSetSatTarget (          uint8  u8Val,  
                                              void    *pvCbData );  
teJIP_Status MibColourControl_eSetHueSatTarget ( uint32 u32Val,  
                                              void    *pvCbData );  
teJIP_Status MibColourControl_eSetCctTarget (          uint16 u16Val,  
                                              void    *pvCbData );
```

These callback functions handle the target colour variables being set remotely, beginning the transition to the new colour.

---

```
teJIP_Status MibColourControl_eSetXYCurrent (          uint32 u32Val,  
                                              void    *pvCbData );  
teJIP_Status MibColourControl_eSetHueSatCurrent (          uint32 u32Val,  
                                              void    *pvCbData );  
teJIP_Status MibColourControl_eSetCctCurrent (          uint16 u16Val,  
                                              void    *pvCbData );
```

These callback functions handle the current colour variables being set remotely, changing to the new colour and updating the target variables to match.

---

```
teJIP_Status MibColourControl_eSetHueChange (          int16 u16Val,  
                                              void    *pvCbData );  
teJIP_Status MibColourControl_eSetSatChange (          int8   u8Val,  
                                              void    *pvCbData );  
teJIP_Status MibColourControl_eSetCctChange (          int16 u16Val,  
                                              void    *pvCbData );
```

These callback functions handle the change colour variables being set remotely, beginning the transition to the new colour.

---

## Private Functions

The following private functions are implemented and used internally in **MibColourControl.c**:

---

```
void MibColourControl_vInitTransition (
                                uint16 u16TransitionTime );
```

This function initialises a transition from one colour to another setting up the transition interpolation structures using the MIB variable values.

---

```
bool_t MibColourControl_bNewXYTarget (
                                bool_t   bNotifyChanged,
                                uint16 u16TransitionTime );
```

This function is used whenever any of the XY space target colour variables are changed. The XY target colour is converted into the other colour spaces and the target variables updated accordingly.

The transition to the new colour is initialised and started.

---

```
bool_t MibColourControl_bNewHueChange (
                                bool_t   bNotifyChanged,
                                uint16 u16TransitionTime );
```

This function is used when the HueChange variable is changed. The HueTarget variable is updated by the specified amount and the **MibColourControl\_bNewHueSatTarget()** function called to process the new target colour.

---

```
bool_t MibColourControl_bNewHueSatTarget (
                                bool_t   bNotifyChanged,
                                uint16 u16TransitionTime,
                                bool_t   bOptimise );
```

This function is used whenever any of the hue/saturation space target colour variables are changed. The hue/saturation target colour is converted into the other colour spaces and the target variables updated accordingly.

The transition to the new colour is initialised and started.

---

```
bool_t MibColourControl_bNewCctChange (
                                bool_t   bNotifyChanged,
                                uint16 u16TransitionTime );
```

This function is used when the CctChange variable is changed. The CctTarget variable is updated by the specified amount and the **MibColourControl\_bNewCctTarget()** function called to process the new target colour.

---

```
bool_t MibColourControl_bNewCctTarget (
                                         bool_t    bNotifyChanged,
                                         uint16    ul6TransitionTime );
```

This function is used when the CCT space target colour variable is changed. The CCT target colour is converted into the other colour spaces and the target variables updated accordingly.

The transition to the new colour is initialised and started.

---

```
bool_t MibColourControl_bModeTestHueSatNext ( void );
bool_t MibColourControl_bModeTestCctNext    ( void );
```

These functions are used when in the test modes to start a new transition when the current transition ends.

---

### 6.4.3 Colour Modules

The **MibColour** folder also contains modules to handle the conversion of colours between colour spaces, to transition from one colour to another over time and a generic interpolation engine used during transitions.

The ColourControl MIB makes extensive use of these modules.

---

#### 6.4.3.1 ColourConversion.c

This source file contains the code that converts between the different colour spaces.

All of the main processing on colours goes via the xyY colour space. When converting to or from the xyY colour space the colour conversion matrices derived from the colour primaries of the bulb are incorporated to ensure that different bulbs set to the same colours display the same actual colour (so long as the colour primaries are set accurately).

This module is an adapted version of the colour conversion software taken from the ZigBee lighting source code. As such it contains some conditional compilation options depending upon whether the code is being compiled for a ZigBee or JenNet-IP application. The header file for this module also includes some structures, types and #defines as used in the ZigBee applications.

---

#### Public Functions

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

---

```
teZCL_Status eCLD_ColourControl_GetRGB (
                                         tsCLD_ColourControlCustomDataStructure *psCommon,
                                         uint16                                ul6X,
                                         uint16                                ul6Y,
                                         uint8                                 *pu8Red,
                                         uint8                                 *pu8Green,
                                         uint8                                 *pu8Blue );
```

This function converts the passed in colour in the xyY colour space to the RGB colour space.

---

```
teZCL_Status eCLD_ColourControl_RGB2xyY (
    tsCLD_ColourControlCustomDataStructure *psCustomDataStructPtr,
    uint8 u8Red,
    uint8 u8Green,
    uint8 u8Blue,
    uint16 *pu16x,
    uint16 *pu16y,
    uint8 *pu8Y );
```

This function converts the passed in colour in the RGB colour space to the xyY colour space.

---

```
teZCL_Status eCLD_ColourControl_HSV2xyY (
    tsCLD_ColourControlCustomDataStructure *psCustomDataStructPtr,
    uint16 u16Hue,
    uint8 u8Saturation,
    uint8 u8Value,
    uint16 *pu16x,
    uint16 *pu16y,
    uint8 *pu8Y );
```

This function converts the passed in colour in the HSV colour space to the xyY colour space.

---

```
void vCLD_ColourControl_CCT2xyY( uint16 u16ColourTemperature,
    uint16 *pu16x,
    uint16 *pu16y,
    uint8 *pu8Y );
```

This function converts the passed in colour in the CCT colour space to the xyY colour space.

---

```
teZCL_Status eCLD_ColourControl_xyY2HSV (
    tsCLD_ColourControlCustomDataStructure *psCustomDataStructPtr,
    uint16 u16x,
    uint16 u16y,
    uint8 u8Y,
    uint16 *pu16Hue,
    uint8 *pu8Saturation,
    uint8 *pu8Value );
```

This function converts the passed in colour in the xyY colour space to the HSV colour space.

---

```
void vCLD_ColourControl_xyY2CCT (
    uint16 u16x,
    uint16 u16y,
    uint8 u8Y,
    uint16 *pu16ColourTemperature );
```

This function converts the passed in colour in the xyY colour space to the CCT colour space.

---

```
teZCL_Status eCLD_ColourControlCalculateConversionMatrices(  
    tsCLD_ColourControlCustomDataStructure *psCustomDataStructure,  
    float                                     fRedX,  
    float                                     fRedY,  
    float                                     fGreenX,  
    float                                     fGreenY,  
    float                                     fBlueX,  
    float                                     fBlueY,  
    float                                     fWhiteX,  
    float                                     fWhiteY );
```

This function calculates the colour conversion matrices from the passed in primary colours in the xyY colour space.

---

### Private Functions

**ColourConversion.c** implements the following internal functions.

---

```
void vCLD_ColourControl_HSV2RGB( float    fHue,  
                                float    fSaturation,  
                                float    fValue,  
                                float *pfRed,  
                                float *pfGreen,  
                                float *pfBlue );
```

This function converts from the HSV colour space to the RGB colour space.

---

```
void vCLD_ColourControl_RGB2HSV( float    fRed,  
                                float    fGreen,  
                                float    fBlue,  
                                float *pfHue,  
                                float *pfSaturation,  
                                float *pfValue );
```

This function converts from the RGB colour space to the HSV colour space.

---

```
void vCLD_ColourControl_RGB2XYZ( float    afMatrix[3][3],  
                                float    fRed,  
                                float    fGreen,  
                                float    fBlue,  
                                float *pfX,  
                                float *pfY,  
                                float *pfZ );
```

This function converts from the RGB colour space to the XYZ colour space.

---

```
void vCLD_ColourControl_XYZ2RGB ( float  afMatrix[3][3],
                                float  fX,
                                float  fY,
                                float  fZ,
                                float  *pfRed,
                                float  *pfGreen,
                                float  *pfBlue );
```

---

This function converts from the XYZ colour space to the RGB colour space.

---

```
teZCL_Status eCLD_ColourControl_XYZ2xyY ( float  fX,
                                           float  fY,
                                           float  fZ,
                                           float  *pfx,
                                           float  *pfy,
                                           float  *pfY );
```

---

This function converts from the XYZ colour space to the xyY colour space.

---

```
teZCL_Status eCLD_ColourControl_xyY2XYZ ( float  fx,
                                           float  fy,
                                           float  fY,
                                           float  *pfX,
                                           float  *pfY,
                                           float  *pfZ );
```

---

This functions converts from the xyY colour space to the XYZ colour space.

---

```
bool bCLD_ColourControl_NumberIsValid ( float fValue );
```

---

This function checks if the floating point value is in a valid range.

---

#### 6.4.3.2 ColourTransition.h, ColourTransition.c

---

These source files contain the code that controls the transition over time from one colour to another.

The module can transition the colour across the XY, Hue/Saturation or CCT colour spaces. However each step across these colour spaces involves significant floating point maths which takes time. In order to provide a smooth transition the movement through these colour spaces takes a series of coarse steps.

Within each of the coarse steps a number of fine steps across the RGB colour space is made. This results in a smooth transition through the appropriate colour space while losing some accuracy while the fine steps are made across the RGB colour space.

The transition code is written so that the timings are controlled externally. The ColourControl MIB uses this code to make a fine (RGB) step every 10ms with a coarse step every 80ms.

The colour transition code makes use of the generic interpolation engine, implemented in **Interpolation.c**, to calculate the coarse and fine steps.

---

## Structures

The following structures are defined in **ColourTransition.h**.

---

### tsColourTransition

```
typedef struct
{
    tsCLD_ColourControlCustomDataStructure *psColourData;
    tsInterpolation    sXYInterpolation;
    tsInterpolation    sHSInterpolation;
    tsInterpolation    sCctInterpolation;
    tsInterpolation    sRGBInterpolation;
    uint8              u8Mode;
} tsColourTransition;
```

This structure is used to contain all the details for a transition. This includes a pointer to the structure containing the colour conversion matrices, an interpolation structure for each colour space and a transition mode used to indicate the primary colour space the transition is moving through.

The module managing the transition is responsible for creating a transition structure which is passed into the transition functions. This allows the transition code to operate more than one transition at a time (if required).

---

## Public Functions

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

---

```
bool_t ColourTransition_bXYStart (
    tsColourTransition *psColourTransition );
bool_t ColourTransition_bHSStart (
    tsColourTransition *psColourTransition,
    bool_t             bOptimise           );
bool_t ColourTransition_bCctStart (
    tsColourTransition *psColourTransition );
```

These functions start a transition through the colour space in the function name. The main members of the transition structure parameter should be set before calling the function. These functions calculate the first step of the coarse colour space interpolation and initialise the fine interpolation through the RGB colour space.

The function to start a transition through the Hue/Saturation colour space includes an extra *bOptimise* parameter which has the follow effects:

- TRUE, transitions using the shortest path from the source to the target colour, passing through red when this is most efficient. This is used when transitioning as a result of the target colour being.
- FALSE, transitions without passing through red even if that requires taking a longer path to the target colour. This is used when transitioning due to being in the UP, DOWN or LOOP modes.

---

```
bool_t ColourTransition_bXYNext (
    tsColourTransition *psColourTransition);
bool_t ColourTransition_bHSNext (
    tsColourTransition *psColourTransition);
bool_t ColourTransition_bCctNext (
    tsColourTransition *psColourTransition);
```

These functions calculate the next step of the transition in the appropriate colour space. This includes calculating the next coarse step through the colour space when the fine steps through the RGB colour space have been completed.

---

```
bool_t ColourTransition_bEnd (
    tsColourTransition *psColourTransition );
```

This function ends a transition that is currently running. It is used when entering the STOP mode from the modes that alter the colour of the bulb.

---

## Private Functions

**ColourTransition.c** implements the following internal functions.

---

```
bool_t ColourTransition_bRGBInit (
    tsColourTransition *psColourTransition );
```

This function initialises the transition through the RGB colour space from the first coarse step in the transition through the primary colour space.

---

```
bool_t ColourTransition_bRGBJump (
    tsColourTransition *psColourTransition );
```

This function sets up the transition through the RGB colour space for the coarse steps through the primary colour space after the first.



---

### 6.4.3.3 Interpolation.h, Interpolation.c

These source files implement a generic interpolation engine that can operate on values in multiple dimensions. A source and target co-ordinate are specified along with the required number of steps and the engine will calculate the co-ordinates for each step, one at a time.

This engine is used by **ColourTransition.c** to calculate the coarse and fine steps during a colour transition.

---

#### Structures

The following structures are defined in **Interpolation.h**.

---

#### tsInterpolation

```
typedef struct
{
    /* Diagnostic data */
    char      acName[16];

    /* Input data */
    uint32    au32Source[DIMENSIONS_MAX];
    uint32    au32Target[DIMENSIONS_MAX];
    uint8     u8StepsPowerOfTwo;
    uint32    u32Steps;
    uint8     u8Dimensions;
    uint8     u8Bits;
    bool_t    bForced;

    /* Output data */
    uint32    au32SourceStep[DIMENSIONS_MAX];
    uint32    au32TargetStep[DIMENSIONS_MAX];
    uint32    u32RemainingSteps;

    /* Internal data */
    uint32    au32ScaledCumulativeDiff[DIMENSIONS_MAX];
    uint32    au32ScaledStepDiff[DIMENSIONS_MAX];
    uint8     u8Scale;
} tsInterpolation;
```

This structure is used to contain all the details for an interpolation calculation.

The diagnostic data members are used during debugging and can be used to name and identify an interpolation structure.

The input data section must be filled in prior to beginning an interpolation.

The output data is calculated by the engine and contains the results for the current step and the number of remaining steps.

The internal data is used when calculating each step.

The public functions all expect a pointer to an interpolation structure to be passed in as a parameter. This allows the engine to run multiple interpolations at the same time.

---

## Public Functions

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

---

```
bool_t Interpolation_bStart (
                                tsInterpolation *psInterpolation );
```

This function starts a new interpolation. It uses the data in the passed in interpolation structure to calculate the size of each step of the interpolation.

---

```
bool_t Interpolation_bInterpolate (
                                tsInterpolation *psInterpolation );
```

This function calculate the next step of an interpolation.

---

```
bool_t Interpolation_bEnd ( tsInterpolation *psInterpolation );
```

This function ends a running interpolation.

---

## 6.5 DriverBulb Folder

The **DriverBulb** folder contains the hardware drivers for bulb devices. These all share a common interface defined in **DriverBulb.h**. There are separate driver C files for each hardware platform, these have a suffix matching the value of the `DEVICE_NAME` variable in the makefile.

---

### 6.5.1 DriverBulb.h, DriverBulb\_Type.c

Software in the Bulb MIBs make calls into the lighting driver software which must implement a common set of functions as defined in **DriverBulb.h**. This set of functions is responsible for operating the bulb at the hardware level in response to commands from the higher Bulb MIBs that provide the JIP interface to the bulb.

The **DriverBulb\_Type.c** files may also contain additional helper functions specific to that particular bulb type.

As there are a number of different **DriverBulb\_Type.c** files the later sections of this chapter describe the task that should be implemented within each of the common functions rather than the specifics of a single driver.

---

#### Public Functions

The following public functions are implemented in **DriverBulb\_Type.c**.

---

```
void DriverBulb_vInit ( void );
```

This function is used to initialize the bulb driver. It is called directly from **DeviceBulb.c** early on in the initialization process in order to setup the hardware as soon as possible. There is also a later call from within the BulbControl MIB's initialization so this function must be tolerant of being called multiple times.

It is usual to set up any DIO lines that are used to control or monitor the bulb's operation and to also configure the PWM timer used to operate the bulb. The bulb is usually illuminated at this stage in order to provide immediate feedback to the user that the bulb is operating when power is applied.

---

```
void DriverBulb_vOn ( void );
```

This function is called from the Bulb MIBs to illuminate the bulb.

Some bulb drivers may simply toggle output lines while others may need to step through a number of states to achieve this.

---

```
void DriverBulb_vOff ( void );
```

This function is called from the MIBBulb modules to extinguish the bulb.

Some bulb drivers may simply toggle output lines while others may need to step through a number of states to achieve this.

---

```
void DriverBulb_vSetOnOff ( bool_t bOn );
```

This function can be used to turn the bulb on or off depending upon the value of the parameter.

---

```
void DriverBulb_vSetLevel ( uint8 u8Level );
```

This function is called from the Bulb MIBs to set the brightness level of the bulb. The *u8Level* parameter sets the brightness with 255 being the brightest setting and 0 being the dimmest setting.

Usually if the bulb is not illuminated it remains that way but the PWM output is updated to an appropriate value.

---

```
void DriverBulb_vSetColour ( uint32 u32Red,  
                             uint32 u32Green,  
                             uint32 u32Blue );
```

This function is used in colour bulbs using RGB emitters to set the colour of the bulb. Despite the use of *uint32* parameters the valid ranges are 0 to 255.

---

```
bool_t DriverBulb_bOn ( void );
```

This function is called from the Bulb MIBs to check if the bulb is illuminated.

---

```
bool_t DriverBulb_bReady ( void );
```

This function is called from the Bulb MIBs to check if the bulb is ready to be controlled.

Where bulbs require a minimum bus voltage to operate, the most recent voltage is checked and used to set the return value.

For bulbs that do not need to meet any conditions to be controlled this function may simply return TRUE.

---

```
bool_t DriverBulb_bFailed ( void );
```

This function may be used to report hardware failures in the bulb to the higher MibBulb layers. These higher layers can be configured to lock a failed bulb in order to prevent any further attempts to illuminate or control it.

Returning TRUE indicates a failed bulb. Bulbs that do not detect such failures may simply return FALSE from this function.

---

```
void DriverBulb_vTick ( void );
```

This function is called approximately every 10ms by the MibBulb code.

This may be used within the bulb driver for timing purposes.

---

```
int16 DriverBulb_i16Analogue (uint8  u8Adc,  
                             uint16 u16AdcRead );
```

This function is called by the MibBulb code to pass completed ADC readings to the bulb driver.

This is used in many drivers to monitor the bus voltage in order to determine if it is possible to drive the bulb.

---

```
void DriverBulb_vSetTunableWhiteColourTemperature (  
                                                int32 i32ColourTemperature );
```

This function is used in CCT bulbs to set the colour temperature of the bulb. The temperature should be passed in in Kelvins.

---

## 6.6 DeviceSensor Folder

The **DeviceSensor** folder of the Application Note contain source code that is specific to the sensor devices in *JenNet-IP Smart Home (JN-AN-1162)*. The following sensors types can be built:

- **Occupancy sensor**, monitors the occupancy of an area and optionally controls lights based upon the occupancy state.
- **Illuminance sensor**, monitors the lighting levels of an area and optionally controls lights based upon the illuminance state.
- **Combined occupancy/illuminance sensor**, a combination of the above two devices in a single device.

Each of the sensor type operates in a similar way with each differing only in the calls they make to the Sensor MIB modules in order to provide appropriate functionality for the sensor type.

The sensor devices can operate as the following node types:

- **Routers**, extending the network for other devices to join. They are able to join and maintain their place within the network in addition to providing sensor functionality.
- **End Devices**, sleeping to preserve power allowing them to be battery powered.

The sensor applications are based upon the template device implemented in *JenNet-IP Application Template (JN-AN-1190)*. This section details the additions to the template to create a sensor device. The reader should refer to *JenNet-IP Application Template (JN-AN-1190)* for details of how the template software works, including the implementation of the Common MIBs.

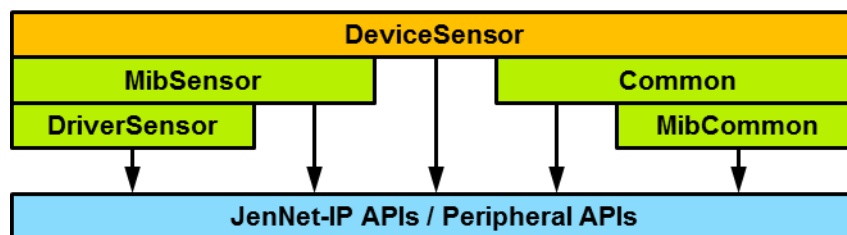
The common source **CommonNode.c** file actually provides the majority of the network joining and maintenance code. **Node.c** in turn makes use of a number of MIBs that can be used to monitor, configure and control the individual node and its operation within the network, these MIBs are implemented in the **MibCommon** folder. This common code is described in *JenNet-IP Application Template (JN-AN-1190)*.

The source code in the **DeviceSensor** folder contains the main module for a sensor device implementing the standard JIP callback functions, which then make calls into the stack and MIB code as required. The type of sensor built is determined by a set of options specified in the makefile.

The source code for the MIBs and their variables used to monitor and control the sensors are contained in the **MibSensor** folder.

The other source files are then hardware level drivers that are mainly called by code in the MibSensor modules to control the operation of the sensor at the hardware layer.

The following diagram shows the layers that form the DeviceSensor applications on top of the JenNet-IP WPAN stack:



---

### 6.6.1 DeviceSensor Makefile

The **makefile** for the sensors uses the variables listed in [Section 6.1.1.1 "Standard DeviceType Makefile"](#) with the following differences:

---

#### DEVICE\_NAME

The sensor device supports the following values:

- **DR1174** to use the basic *Carrier Board (DR1174)*. This option should be used when compiling the occupancy sensor.
- **DR1175** to use the ambient light sensor on the *Lighting/Sensor Expansion Board (DR1175)*. This option should be used when compiling the illuminance sensor or combined sensor.

---

#### NODE\_TYPE

The sensor devices can be compiled as either Router or End Device node types.

---

#### DRIVER\_TYPE

This variable specifies which type of sensor to build and supports the following values:

- **Occupancy** to build an occupancy sensor.
- **Illuminance** to build an illuminance sensor.
- **OccupancyIlluminance** to build a combined sensor.

---

#### BLD\_MIB\_NAME Variables

The sensor device makefile adds additional variables to control the building of the sensor MIBs.

---

#### REG\_MIB\_NAME Variables

The sensor device makefile adds additional variables to control the registration of the sensor MIBs.

---

### 6.6.1.1 Binary File Naming

The names of the sensor binary files incorporates the DRIVER\_TYPE variable to indicate the sensor type.

---

### 6.6.2 DeviceDefs.h

This header file contains a few #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 additions:

The digital I/O lines used to read the occupancy state and output the sensor status are defined..

Default values for some sensor MIB variables are defined.

Additional debug flags for sensor MIBs and drivers are included.



---

### 6.6.3 DeviceSensor.c

**DeviceSensor.c** contains the main source code for the sensor application. It follows the pattern described in [Section 6.1.1.3 "Standard DeviceType.c Features"](#) with the following changes.

---

#### 6.6.3.1 #includes

Additional #includes are used to provide access to the sensor MIB modules used in **DeviceSensor.c**.

---

#### 6.6.3.2 External Variables

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

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

---

#### 6.6.3.3 Public Functions

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

---

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

When waking from sleep operating as an End Device using the illuminance sensor the illuminance sensor is restarted using a call to **MibIlluminanceStatus\_vResume()**.

---

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

The sensor MIBs are initialised here *following* the call to **Node\_vPdmInit()** once the PDM has been initialised.

---

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

The factory reset of the sensor MIB's permanent data is performed here.

---

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

The sensor MIBs are registered with the stack here.

---

```
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 sensor MIBs are called to allow the MIBs to interact with the stack.

---

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

This function is called every 100ms to allow tasks to be timed.

Timer functions in the sensor MIBs are called to allow the MIBs to run timed tasks.

---

```
void Device_vSecond ( uint32 u32TimerSeconds );
```

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

The timer is simply passed on to the sensor MIBs that perform timing tasks based on the 1 second timer.

---

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

This function is used to place End Devices into sleep mode when the stack is not running.

The **MibIlluminanceStatus\_vSleep()** function is called to suspend running the illuminance sensor.

---

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

This callback function is called by the stack prior to entering sleep mode when the stack is running.

The **MibIlluminanceStatus\_vSleep()** function is called to suspend running the illuminance sensor.

---

## 6.7 MibSensor Folder

The **MibBulb** folder contains modules that implement MIBs for configuring, monitoring and controlling sensors.

---

### 6.7.1 OccupancyConfig MIB

The OccupancyConfig MIB provides variables to configure the operation of the occupancy sensor.

This MIB follows the pattern described in [Section 6.1.3 "Standard MIB Module Features"](#) with the following alterations.

---

#### 6.7.1.1 MibOccupancyConfig.c

This source file contains the code that implements the OccupancyConfig MIB.

---

##### Public Functions

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

---

```
void MibOccupancyConfig_vRegister ( void )
```

This function builds the default address, if required, for transmitting the occupancy state to in addition to registering the MIB with the stack. The default address is built from the MAC address and OccupancyConfig MIB's ID to form an address unique to each occupancy sensor.

---

### 6.7.2 OccupancyStatus MIB

The OccupancyStatus MIB provides variables to monitor the status of the occupancy sensor.

This MIB follows the pattern described in [Section 6.1.3 "Standard MIB Module Features"](#) with the following alterations.

---

#### 6.7.2.1 MibOccupancyStatus.c

This source file contains the code that implements the OccupancyStatus MIB.

---

##### Public Functions

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

---

```
void MibOccupancyStatus_vInit (
    thJIP_Mib                hMibOccupancyStatusInit,
    tsMibOccupancyStatus *psMibOccupancyStatusInit,
    void                    *pvMibOccupancyConfigInit,
    void                    *pvMibOccupancyControlInit )
```

---

In addition to the standard initialisation this function performs the following tasks:

- Initialises the occupancy driver using a call to **DriverOccupancy\_vInit()**.
- Initialises LEDs to display the occupancy state, if configured.

---

```
void MibOccupancyStatus_vTick ( void )
```

---

This function is called every 10ms when the stack is running.

This function transmits occupancy status updates when configured to do so.

---

```
void MibOccupancyStatus_vAppTimer100ms ( void );
```

---

This function reads the occupancy status from the occupancy driver using a call to **DriverOccupancy\_bRead()**.

The occupancy state and timers are updated and checked, if the occupancy state at the MIB level (which takes the timers into account) has changed the Occupancy MIB variable is updated.

When the occupancy state changes the **MibOccupancyStatus\_vQueueTxState()** function is called to transmit the new Occupancy variable value when the occupancy sensor is configured to do so.

If the Occupancy variable is unchanged the internal refresh timer is updated and if required the existing Occupancy variable value is retransmitted with a call to **MibOccupancyStatus\_vQueueTxState()** (if configured).

---

```
void MibOccupancyStatus_vQueueTxState ( bool_t bImmediate );
```

---

This function queues the transmission of the occupancy state to allow other devices to monitor the occupancy sensor.

In End Devices the *bImmediate* flag is used to indicate if the transmission should be sent immediately (resuming the stack if necessary) or if it can wait until the stack is scheduled to run (in order to poll the parent node for messages).

---

### 6.7.3 OccupancyControl MIB

The OccupancyControl MIB provides variables to control the operation of the Occupancy Sensor.

This MIB follows the pattern described in [Section 6.1.3 "Standard MIB Module Features"](#) with the following alterations.

---

#### 6.7.3.1 MibOccupancyControl.c

This source file contains the code that implements the OccupancyControl MIB.

---

#### Public Functions

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

---

```
void MibOccupancyControl_vInit (
    thJIP_Mib                hMibOccupancyControlInit,
    tsMibOccupancyControl *psMibOccupancyControlInit );
```

This function places the occupancy sensor into the "All Occupancy Sensors" group the first time it starts from the factory default state.

---

## 6.7.4 IlluminanceConfig MIB

The IlluminanceConfig MIB provides variables to configure the operation of the illuminance sensor.

This MIB follows the pattern described in [Section 6.1.3 "Standard MIB Module Features"](#) with the following alterations.

---

### 6.7.4.1 MibIlluminanceConfig.c

This source file contains the code that implements the IlluminanceConfig MIB.

---

#### Public Functions

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

---

```
void MibIlluminanceConfig_vRegister ( void )
```

This function builds the default address, if required, for transmitting the illuminance state to in addition to registering the MIB with the stack. The default address is built from the MAC address and IlluminanceConfig MIB's ID to form an address unique to each illuminance sensor.

---

## 6.7.5 IlluminanceStatus MIB

The IlluminanceStatus MIB provides variables to monitor the status of the illuminance sensor.

This MIB follows the pattern described in [Section 6.1.3 "Standard MIB Module Features"](#) with the following alterations.

---

### 6.7.5.1 MibIlluminanceStatus.c

This source file contains the code that implements the IlluminanceStatus MIB.

---

#### Public Functions

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

---

```
void MibIlluminanceStatus_vInit (  

    thJIP_Mib                hMibIlluminanceStatusInit,  

tsMibIlluminanceStatus *psMibIlluminanceStatusInit,  

void                     *pvMibIlluminanceConfigInit,  

void                     *pvMibIlluminanceControlInit )
```

In addition to the standard initialisation this function performs the following tasks:

- Initialises the illuminance driver using a call to **DriverIlluminance\_vInit()** and starts to take the initial reading using a call to **DriverIlluminance\_vStart()**.
- Initialises LEDs to display the illuminance state, if configured.

---

```
void MibIlluminanceStatus_vTick ( void )
```

This function is called every 10ms when the stack is running.

This function transmits illuminance status updates when configured to do so.

---

```
void MibOccupancyStatus_vAppTimer100ms ( void );
```

This function handles changes to the Mode variable turning the sensor on or off as required.

If a new reading has been calculated by the driver this function reads the illuminance from the illuminance driver using a call to **DriverIlluminance\_u16Lux()**, then starts the next reading. The reading is compared to the target illuminance band and the illuminance status updated as necessary.

The illuminance state and timers are updated and checked, if the illuminance state has changed the TargetStatus MIB variable is updated.

When the illuminance state changes the **MibIlluminanceStatus\_vQueueTxState()** function is called to transmit the new TargetStatus variable value when the illuminance sensor is configured to do so.

If the TargetStatus variable is unchanged the internal refresh timer is updated and if required the existing TargetStatus variable value is retransmitted with a call to **MibIlluminanceStatus\_vQueueTxState()** (if configured).

---

```
void MibIlluminanceStatus_vQueueTxState ( bool_t bImmediate );
```

This function queues the transmission of the illuminance state to allow other devices to monitor the illuminance sensor.

In End Devices the *bImmediate* flag is used to indicate if the transmission should be sent immediately (resuming the stack if necessary) or if it can wait until the stack is scheduled to run (in order to poll the parent node for messages).

---

```
void MibIlluminanceStatus_vResume ( void );
```

This function is used to resume running the illuminance sensor at the hardware level when waking from sleep in an End Device.

---

## 6.7.6 IlluminanceControl MIB

The IlluminanceControl MIB provides variables to control the operation of the illuminance sensor.

This MIB follows the pattern described in [Section 6.1.3 "Standard MIB Module Features"](#) with the following alterations.

---

### 6.7.6.1 MibIlluminanceControl.c

This source file contains the code that implements the IlluminanceControl MIB.

---

#### Public Functions

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

---

```
void MibIlluminanceControl_vInit(
    thJIP_Mib          hMibIlluminanceControlInit,
    tsMibIlluminanceControl *psMibIlluminanceControlInit,
    void               *pvMibIlluminanceStatusInit,
    void               *pvMibIlluminanceSceneInit );
```

This function places the illuminance sensor into the "All Illuminance Sensors" group the first time it starts from the factory default state.

---

```
teJIP_Status MibIlluminanceControl_eSetSceneId (
    uint16 u16Val,
    void    *pvCbData );
```

This callback function is called when the SceneId variable is set remotely.

It looks up the scene settings in the IlluminanceScene MIB and applies them to the device.

---

```
void MibIlluminanceControl_vTick ( void );
```

This function is called every 10ms.

In Router devices if the LuxAdjust variable is set appropriately it adjusts the LuxTarget or LuxBand variables over time. This allows remote controls to be used to alter the target illuminance.



---

## 6.7.7 IlluminanceScene MIB

The IlluminanceScene MIB provides variables to configure scenes in the illuminance sensor.

This MIB follows the pattern described in [Section 6.1.3 "Standard MIB Module Features"](#) with the following alterations.

---

### 6.7.7.1 MibIlluminanceScene.c

This source file contains the code that implements the IlluminanceScene MIB.

---

#### Public Functions

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

---

```
uint8 MibIlluminanceScene_u8FindSceneId ( uint16 u16SceneId );
```

This helper function is used to find a scene with the passed in ID in the scene table.

---

```
bool_t MibIlluminanceScene_bSetScene ( uint16 u16Id,  
                                         uint16 u16LuxTarget,  
                                         uint16 u16LuxBand );
```

This helper function is used to configure the settings for a scene.

It is called by the **MibIlluminanceScene\_eSetAddSceneId()** function when the AddSceneId variable is written to, passing the current illuminance settings to configure a scene.

It is called by the **MibIlluminanceScene\_eSetSetScene()** function when the SetScene variable is written to, passing in the illuminance settings contained in the value written to the variable.

---

## 6.7.8 OccIllBulbConfig MIB

The OccIllBulbConfig MIB provides variables to configure the control of bulb devices based upon readings from occupancy and/or illuminance sensors.

This MIB follows the pattern described in [Section 6.1.3 "Standard MIB Module Features"](#) with the following alterations.

---

### 6.7.8.1 MibOccIllBulbConfig.c

This source file contains the code that implements the OccIllBulbConfig MIB.

---

#### Public Functions

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

---

```
void MibOccIllBulbConfig_vInit (
    thJIP_Mib          hMibOccIllBulbConfigInit,
    tsMibOccIllBulbConfig *psMibOccIllBulbConfigInit,
    void               *pvMibOccupancyStatusInit,
    void               *pvMibOccupancyMonitorInit,
    void               *pvMibOccupancyControlInit,
    void               *pvMibIlluminanceStatusInit,
    void               *pvMibIlluminanceControlInit );
```

In addition to the standard handling this function initialises the LEDs to display the sensor state when configured to do so.

---

```
void MibOccIllBulbConfig_vRegister ( void );
```

This function builds the default address, if required, for transmitting the bulb control commands to in addition to registering the MIB with the stack. The default address is built from the MAC address and OccIllBulbConfig MIB's ID to form an address unique to each sensor.

---

```
void MibOccIllBulbConfig_vTick ( void );
```

This function is called every 10ms when the stack is running.

This function transmits bulb control commands when configured to do so.

---

```
void MibOccIllBulbConfig_vAppTimer100ms ( void );
```

This function checks the status of the occupancy and illuminance sensors.

If action needs to be taken to control bulbs the bulb control commands are queued with a call to **MibOccIllBulbConfig\_vQueueTxBulb()**.

The timers that are used to control the regular refresh bulb control commands are checked and if a refresh needs to be sent a call if made to **MibOccIllBulbConfig\_vQueueTxBulb()**.

---

```
bool_t MibOccIllBulbConfig_bOccupancy ( uint8 u8Occupancy );
```

This helper function is called when the bulb control commands need to be transmitted based upon the state of the occupancy sensor. The function calculates the appropriate values to write to the BulbControl MIB variables dependent on the occupancy state.

---

```
bool_t MibOccIllBulbConfig_bIlluminanceAuto (
                                         uint8 u8TargetStatus );
```

This helper function is called when the bulb control commands need to be transmitted based upon the state of the illuminance sensor. The function calculates the appropriate values to write to the BulbControl MIB variables dependent on the illuminance state to attempt to bring the illuminance level into the target band.

---

```
bool_t MibOccIllBulbConfig_bIlluminanceMax (
                                         uint8 u8TargetStatus );
```

This helper function is called when the bulb control commands need to be transmitted based upon the state of the illuminance sensor. The function calculates the appropriate values to write to the BulbControl MIB variables dependent on the illuminance state to attempt to turn the bulbs on fully when the light is low and off when it is high.

---

```
void MibOccIllBulbConfig_vRefreshIntervalUpdate ( void );
```

This helper function is used to update the refresh interval timer used to regularly re-transmit the latest bulb control commands when sensor readings are unchanged.

---

```
void MibOccIllBulbConfig_vQueueTxState ( bool_t bImmediate );
```

This function queues the transmission of the bulb control commands.

In End Devices the *bImmediate* flag is used to indicate if the transmission should be sent immediately (resuming the stack if necessary) or if it can wait until the stack is scheduled to run (in order to poll the parent node for messages).

---

## 6.7.9 OccupancyMonitor MIB

The OccupancyMonitor MIB provides variables to allow the monitoring of multiple external occupancy sensors from another sensor. The OccupancyMonitor MIB is used to receive the occupancy state from the external occupancy sensors and can be read to monitor the external occupancy sensors.

This MIB follows the pattern described in [Section 6.1.3 "Standard MIB Module Features"](#) with the following alterations.

---

### 6.7.9.1 MibOccupancyMonitor.c

This source file contains the code that implements the OccupancyMonitor MIB.

---

#### Public Functions

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

---

```
void MibOccupancyMonitor_vSecond ( void );
```

This function is called every second.

The timers used to detect whether monitored occupancy sensors have stopped transmitting data are updated. When the timeout value is reached for a monitored sensor the sensor is removed from the table and trap updates issued.

---

```
uint8 MibOccupancyMonitor_u8FindDevice (  

in6_addr *psIn6Address );
```

This helper function is used to find a device in the monitored device table.

---

```
teJIP_Status MibOccupancyMonitor_eSetOccupancy (  

uint8 u8Val,  

void *pvCbData );
```

This callback function is called when an occupancy sensor writes its occupancy state to the Occupancy variable.

All occupancy sensors write their occupancy status to this same variable.

If the sending occupancy sensor is not already in the device table it is added (if there is room for it).

The code in this function extracts the source address of the sending occupancy sensor and updates its occupancy status in the device table.

The bitmask variables used to indicate which device table entries are enabled and occupied or unoccupied are updated.

---

## 6.8 DriverSensor Folder

The **DriverSensor** folder contains the hardware drivers for sensor devices. The occupancy drivers share a common interface defined in **DriverOccupancy.h** and the illuminance drivers share a common interface defined in **DriverIlluminance.h**. There are separate driver C files for each hardware platform, these have a suffix indicating the hardware sensor they can drive.

---

### 6.8.1 DriverOccupancy.h, DriverOccupancy\_Type.c

The occupancy sensors allow for easy replacement of the hardware driver in order to support different sensor hardware. These drivers should all share a common interface as defined by **DriverOccupancy.h** but different implementations in **DriverOccupancy\_Type.c** files.

A single **DriverOccupancy\_DIO.c** file is provided which uses a digital input line to monitor the occupancy status. When the input line is high the area is considered occupied and when low unoccupied.

Software in the MibSensor modules makes calls into the occupancy driver software which must implement a common set of functions as defined in **DriverOccupancy.h**.

---

#### Public Functions

The following public functions are implemented in **DriverOccupancy\_Type.c**.

---

```
void DriverOccupancy_vInit ( void );
```

This function is used to initialize the occupancy driver. It sets up the specified DIO line as an input.

---

```
bool_t DriverOccupancy_bRead ( void );
```

This function is called to initiate a read of the occupancy sensor hardware. It reads the DIO input line status.

---

```
bool_t DriverOccupancy_bOccupancy ( void );
```

This function returns the most recent occupancy state read from the hardware.

---

```
uint8 DriverOccupancy_u8Type ( void );
```

This function returns the hardware type of the occupancy sensor.

---

## 6.8.2 DriverIlluminance.h, DriverIlluminance\_Type.c

The illuminance sensors allow for easy replacement of the hardware driver in order to support different sensor hardware. These drivers should all share a common interface as defined by **DriverIlluminance.h** but different implementations in **DriverIlluminance\_Type.c** files.

A single **DriverIlluminance\_TSL2550.c** file is provided which interfaces with the TAOS TSL2550 ambient light sensor on the *Lighting/Sensor Expansion Board (DR1175)* to monitor the illuminance status.

Software in the MibSensor modules makes calls into the illuminance driver software which must implement a common set of functions as defined in **DriverIlluminance.h**.

---

### #defines

Various #defines are used for the addresses and commands to control the TSL2550 via its serial bus.

Other #defines are used to indicate the timings of taking a measurement.

---

### Public Functions

The following public functions are implemented in **DriverOccupancy\_Type.c**.

---

```
bool_t DriverIlluminance_bInit ( void );
```

This function is used to initialize the illuminance sensor by issuing commands on the serial bus.

The return value indicates if the serial bus requests were successful.

---

```
bool_t DriverIlluminance_bPowerDown ( void );
```

This function is used to power down the illuminance sensor hardware.

---

```
bool_t DriverIlluminance_bOpen ( void );
```

This function is used resume running the sensor after waking from sleep.

The return value indicates if the request was successful.

---

```
bool_t DriverIlluminance_bClose ( void );
```

This function is used to place the illuminance sensor into sleep.

The return value indicates if request was successful.

---

```
bool_t DriverIlluminance_bStart ( void );
```

This function is starts the process of taking a reading from the sensor hardware.

The return value indicates if request was successful.

---

```
bool_t DriverIlluminance_bTick ( void );
```

This function should be called regularly is used to check for completed illuminance measurements.

The return value indicates if new measurement is available.

---

```
bool_t DriverIlluminance_bReady ( void );
```

This function can be called to check if a new measurement is available.

The return value indicates if new measurement is available.

---

```
uint16 DriverIlluminance_u16Lux ( void );
```

This function returns the most recent illuminance measurement in Lux.

---

```
uint8 DriverIlluminance_u8Type ( void );
```

This function returns the hardware type of the illuminance sensor.

---

```
uint16 DriverIlluminance_u16LuxMin ( void );
```

This function returns the minimum illuminance the sensor can measure in Lux.

---

```
uint16 DriverIlluminance_u16LuxMax ( void );
```

This function returns the maximum illuminance the sensor can measure in Lux.

---

```
uint16 DriverIlluminance_u16LuxTolerance ( void );
```

This function returns the possible errors in in the measurements provided by the sensor in Lux.

## 6.9 DeviceRemote Folder

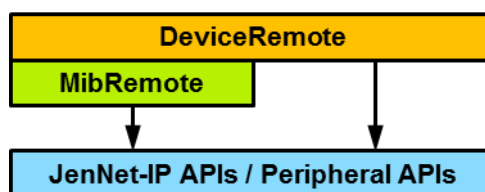
The **DeviceRemote** folder of the Application Note contains source code that is specific to the remote control devices in *JenNet-IP Smart Home (JN-AN-1162)*.

The remote control does not have many MIBs as it is largely a controlling device that writes to variables in the MIBs of devices being controlled. The MIBs that are available in the remote control are contained in the **MibRemote** folder.

The source code in the **DeviceRemote** folder contains the main module implementing the standard JIP callback functions, which then makes calls into the stack and MIB modules as required. Other modules in this folder encapsulate certain functionality of the remote control software, including hardware level drivers.

Due to its nature a sleeping broadcaster device it does not share any of the code from the **Common** folder or include the MIBs from the **MibCommon** folder.

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



### 6.9.1 DeviceRemote 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.

Many of the variables used in the makefile match those described in [Section 6.1.1.1 "Standard DeviceType Makefile"](#) the other variables are described below.

#### DEVICE\_NAME

The remote control device supports the following values:

- **RD6035** for the *Touch Capacitance Remote Control (DR1159-5V2)*.
- **DR1199** for the *Carrier Board (DR1174)* fitted with the *Generic Expansion Board (DR1199)*.



---

### 6.9.2 RemoteDefault.h

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

---

### 6.9.3 DeviceRemote.c

**DeviceRemote.c** contains the main source code for the remote control device application.

The standard JIP callback functions are implemented in this source file along with code to operate the application at the highest level. However various helper modules and functions are used to read the hardware at the lowest level and abstract key presses and sequences for handling by the upper layers.

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

---

#### Defines

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

---

#### DIO\_WAKEMASK

The RD6035 build enters a sleep mode after 15 minutes of inactivity. The remote will only exit this mode when a digital input changes state. The input to be used for this is configured by the #define DIO\_WAKEMASK. This mode of operation extends the battery life.

Other hardware designs that need to implement this feature can use the #define to configure one or more input lines to wake from sleep.

For hardware designs that do not feature a hardware button this value should be set to 0 to disable this feature.

---

#### Local Data

There are a few local variables in the application.

---

#### sRemotePdm

This structure contains data that is written to the PDM.

---

## Public Functions

The following public functions are implemented in **DeviceRemote.c**.

---

### **void AppColdStart ( void );**

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

This function first initialises the exception handler and hardware peripherals.

Then the PDM is initialised and the data record read if available.

The RemoteConfigGroup MIB is initialised. The stack is started with a call to **vStartStack()**. Then the RemoteConfigGroup MIB is registered with the stack.

The touch capacitance handler is initialised and the background capacitance is read by calling **eTouchProcess()**.

The remote control then remains in a loop until the network is formed, joined or re-joined. During this time the remote control enters the doze mode only but does not sleep to preserve power. Battery life while trying to join a network is reduced compared to normal operation when in a network. Once the remote is operating in the network it enters sleep mode with RAM held, when it wakes up via the **AppWarmStart()** function it is able to operate as a sleeping remote control and broadcast commands to other devices.

---

### **void AppWarmStart ( void );**

This function is the entry point to the application following a wake from sleep with memory held.

First hardware is re-initialised, including the touch capacitance driver.

The remote software then loops checking the touch pads for activity and taking appropriate actions until it is time for the remote to sleep once again.

---

### **void vInitPeripherals ( void );**

This function initializes the peripherals used by the remote control, including the touch capacitance drivers.

---

### **void vStartStack ( void );**

This function is called during a cold start to initialize the JenNet-IP stack and start it running to form or join a network.

---

```
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.

The **Remote\_vSetUserData()** function is called to set the correct Network ID and Device Type ID used while joining a network.

Network security is applied and if the device was previously a member of a network the operation of that network type is resumed.

When joining a network for the first time an appropriate joining profile is set and network keys chosen.

Finally the appropriate network key is set.

---

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

This function is used during network configuration to resume operation in a gateway network.

---

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

This function is used during network configuration to resume operation in a standalone network.

---

```
void Remote_vSetSecurityKey ( uint8 u8Key );
```

This function is used to set the security key being used by the stack.

---

```
void Remote_vSetProfile ( bool_t bStandalone );
```

This function set an appropriate stack run profile depending upon the stack mode.

---

```
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. 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 the application in this function.

---

```
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 peripherals are handled in this function:

E\_AHI\_DEVICE\_TICK\_TIMER

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

The remote control uses the tick timer event to time various activities in the application.

---

```
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 remote in the network. The following events are handled:

E\_STACK\_STARTED  
E\_STACK\_JOINED

Indicates the bulb has successfully formed, joined or re-joined a network.

Where a network is formed the LEDs and state of the device are updated.

When a gateway network is joined the remote control enters a learning state for 40 seconds. During this time it remains a full member of the network operating as a router. This time period allows the gateway to read or write to the MIB variables of a remote control that has joined its network before it becomes unavailable due to running as a sleeping broadcaster.

When a standalone network is joined the remote control enters the learning state for 10 seconds to allow the commissioning device to read or configure the newly commissioned remote control.

The user data containing the Network ID and Device Type IDs is refreshed, and the stack mode updated if required.

Finally PDM data is updated and written to flash to allow the network to be resumed in the event of a cold start.

E\_STACK\_RESET

Indicates the remote control has lost contact with its network. This is expected in the remote control software once it starts running in sleeping broadcaster mode which does not need to maintain full contact with the network.

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

#### `E_STACK_NODE_JOINED`

This event is raised whenever the remote control accepts a new device as a child, this should only happen in standalone networks when the remote control is being used to commission new devices into the network.

The software checks that the commissioned device type is correct if in commissioning mode. It will also initiate the setting of groups in bulbs and additional remote controls as they are accepted into the remote control's network.

#### `E_STACK_NODE_AUTHORISE`

This event is raised whenever a node attempts to communicate with the remote control using an unknown encryption key. This is usually during the process of a device trying to join the remote control's standalone network.

If the remote is in commissioning mode the joining node's MAC address is noted and the authorisation state machine is moved on, (this will result in the joining node's commissioning key being applied).

---

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

This function is called by the stack to report the outcome of a data transmission request.

In the remote control this sets the ready to sleep flag following a transmission attempt.

---

```
void vAppSleep ( bool_t bSleepType );
```

This function places the remote control into sleep mode after appropriately configuring various hardware peripherals.

---

```
void vAppSave ( void );
```

This function saves the PDM data to flash if required.

---

```
void vCbTouchEventButton ( eTouchEvent eEvent,  
                           uint8       u8ButtonNumber );
```

This callback function is called by the touch driver to pass key presses to the application.

---

```
void vTouchChecker ( void );
```

This function is regularly called by the application to check for new button events.

Events are passed into the **eKeyPressTracker()** function in **Key.c** which handles single key presses and also checks for key press sequences. If a key press sequence is detected appropriate actions are taken to initiate the task associated with the sequence.

---

```
void vJIP_Remote_SetResponse (
                                ts6LP_SockAddr *psAddr,
                                uint8           u8Handle,
                                uint8           u8ModuleIndex,
                                uint8           u8VarIndex,
                                teJIP_Status    eStatus );
```

This function is called by the stack when a set MIB variable response is received from a set MIB variable request that has been unicast to another device.

This is used during the commissioning of a device when its group variables are being configured. When success is returned the commissioning state machine is moved on or completed as required.

---

```
void Remote_vSetUserData ( void );
```

This function sets the Network ID and Device Type IDs contained in the beacon response and establish route messages transmitted by the node.

This function also sets the callback handlers for beacon responses and network authorization.

---

```
bool_t Remote_bBeaconNotifyCallback (
                                tsScanElement *psBeaconInfo,
                                uint16         ul6ProtocolVersion );
```

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

The beacon response is first checked to ensure it has come from a network the device may be interested in joining.

If the beacon response is acceptable but is from a node in standalone commissioning mode while the remote is trying to join a gateway network the process is begun to switch over to joining a standalone network.

---

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

This function is called by the stack each time a device is attempting to establish a route while joining the network.

The remote control only allows devices to join while in a commissioning mode.

The Network ID from the node attempting to join is compared to the Network ID of the remote control with the node only allowed to join if the two match. This keeps unwanted nodes out of a network.

The Device Type IDs of nodes requesting membership are also checked and only accepted if the remote control is in a commissioning mode for that device type.

---

```
void PTS_UartInit ( void );
```

This function is used to initialize the UART in non-debug builds to allow production testing of the touch keyboard.

---

## 6.9.4 DriverCapTouch.h, DriverCapTouch.c, DriverCapTouch\_DIO.c

These modules provide the low level driver software that detects presses and releases of the keys on the remote control.

There are two implementations of these drivers, both implement the same set of public functions as defined in the **DriverCapTouch.h** header file, though they both use different internal functions appropriate to the hardware used:

1. **DriverCapTouch.c** provides the true capacitance touch driver, reading the capacitance changes on the DIO lines converting them to key press and release events. This driver is used with all the touch capacitance remote control hardware designs.
2. **DriverCapTouch\_DIO.c** provides a driver that operates upon actual button inputs, (instead of capacitance touch pads). This driver is used to allow the *Generic Expansion Boards (DR1199)* to be used as a remote control.

---

### Public Functions

The following public functions are implemented in **DeviceRemote.c**.

---

```
teTouchStatus eTouchInit ( void );
```

This function is used to initialize the driver hardware.

---

```
teTouchStatus eTouchSleep ( void );
```

This function is used to prepare the driver for the device entering into a sleep mode.

---

```
teTouchStatus eTouchWake ( void );
```

This function is used to prepare the driver for use after waking from a sleep mode.

---

```
teTouchStatus eTouchProcess ( void );
```

This function must be regularly called in order to read and process the button or pad inputs.

---

### 6.9.5 DriverLed.h, DriverLed.c

This module provides a driver for the LEDs used on the remote control boards.

This module provides functions to turn LEDs on, off and also to flash at defined intervals.

---

### 6.9.6 Key.h, Key.c

This module tracks key presses passed in from the hardware driver, reacting to simple commands and detecting key sequences.

---

#### Public Functions

The following public functions are implemented in **Key.c**.

---

```
teKeyStatusCode eKeyPressTracker ( teTouchKeys eTouchKeys,
                                   bool_t       bNormal  );
```

This function should be called each time a key press is detected from the touch driver.

It maintains a history of key presses and returns status code values when a sequence is detected that requires further actions to be taken.

Simple one key commands are passed into the **vSetModeMibVar()** function for handling.

---

```
uint8 u8GetLastGroup ( void );
```

This function returns the index of the last pressed group key.

---

```
void vKeyTick ( void );
```

This function is called regularly to allow the key handler to time events.

---

```
teKeyType eGetKeyType ( teTouchKeys eTouchKeys );
```

This function returns the type of an individual key press.

The last pressed group selection key is retained for future use.



---

### 6.9.7 Mib.h, Mib.c

This module provides a set of functions that are used to write to MIB variables in other devices in order to control them.

---

#### Public Functions

The following public functions are implemented in **Mib.c**.

---

```
void vSetModeMibVar ( teTouchKeys eTouchKeys );
```

This function is used to set the BulbControl MIB Mode variable to the value specified by the touch key passed in. The command is broadcast to the address of the last pressed group key.

---

```
void vSetGroupMibVar ( MAC_ExtAddr_s *psMacAddr,
                      uint32          u32MibId,
                      uint8           u8VarIdx,
                      uint8           u8Group );
```

This function is used to write to the Groups MIB AddGroup and RemoveGroup variables to an individual device using a unicast. It used during commissioning of new devices.

---

```
teJIP_Status eBcastGroupMibVar ( uint16 u16GroupAddr,
                                uint32  u32MibId,
                                uint8   u8VarIdx,
                                uint8   u8Group );
```

This function is used to write to the Groups MIB AddGroup and RemoveGroup variables to multiple devices using a broadcast. The command is transmitted with a lower power than other commands as it is used to configure groups at short range for devices already in the network. The command is also prevented from being rebroadcast though the network so only devices in range of the remote are updated.

---

```
void vSetNodeControlMibVar ( uint8 u8CountDown );
```

This function is used to broadcast a write to the NodeControl MIB FactoryReset variable to schedule a factory reset. This is used for the decommission bulb command.

---

```
void vSetMibVarUint16 ( MAC_ExtAddr_s *psMacAddr,
                       uint32          u32MibId,
                       uint8           u8VarIdx,
                       uint16          u16Val );
```

This function is used to transmit a MIB variable write for variables of the type *uint16*.

---

```
void vSetMibVarUint8 ( MAC_ExtAddr_s *psMacAddr,  
                      uint32         u32MibId,  
                      uint8          u8VarIdx,  
                      uint8          u8Val    );
```

This function is used to transmit a MIB variable write for variables of the type *uint8*.

---

```
void vSetSafetoSleep ( void );
```

This function sets the “safe to sleep” flag.

---

### 6.9.8 ModeCommission.h, ModeCommission.c

This module handles the commissioning of new devices into the network.

It controls the acceptance of new devices and also once accepted applies any grouping configuration appropriate to the commissioning mode.

---

#### Public Functions

The following public functions are implemented in **ModeCommission.c**.

---

```
void vCommissionInit ( tsDevice      *psDevice ,  
                      tsAuthorise *psAuthorise );
```

This function begins the commissioning process placing the stack into standalone commissioning mode and starting the commissioning timer.

---

```
void vCommissionMode ( tsDevice      *psDevice ,  
                      tsAuthorise *psAuthorise,  
                      teSysState  *peSysState );
```

This function manages the commissioning process for a device maintaining a state machine to step through the various stages of commissioning.

This includes applying the commissioning key for new devices and placing them into groups by writing to the Groups MIB variables.

---

```
void vDecommissionInit ( void );
```

This function begins the decommissioning process. It prevents the rebroadcast of decommission commands, sets the number of decommission transmissions to be made and starts the decommissioning timer.

---

```
void vDecommissionEnd ( void );
```

This function ends the decommissioning process allowing future broadcasts to be rebroadcast through the network.

---

```
void vTtlOverride ( uint8 u8MaxBcastTtl );
```

This function overrides the “time to live” for broadcast packets. This is usually used to prevent rebroadcasting of transmissions by setting it to 0.

---

```
void vTtlRestore ( void );
```

This function restores the “time to live” for broadcast packets, allowing them to be rebroadcast to the whole network once again.

---

### 6.9.9 JipCallbacks.c

This module contains the JenNet-IP callback functions that are not used by the application as such they are all empty.

---

## 6.10 MibRemote Folder

The **MibRemote** folder implements MIBs that can be reused in many different remote types.

At this time only the RemoteConfigGroup MIB is included. This MIB allows the addresses associated with the remote's group buttons to be read or written. These are the group addresses that the remote control transmits commands to.

---

### 6.10.1 RemoteConfigGroup MIB

The RemoteConfigGroup MIB allows the addresses associated with the remote's group buttons to be configured. These are the addresses that bulb commands are transmitted to.

It can be useful to read or write to these variables when the remote first joins a gateway so the groups that can be controlled by the remote are known or configured as required. Once a remote has started sleeping it will be necessary to force it to re-join the network to make it available for data access.

This MIB follows the pattern described in [Section 6.1.3 "Standard MIB Module Features"](#) with the following alterations.

---

#### 6.10.1.1 MibRemoteConfigGroup.c

This source file contains the code that implements the RemoteConfigGroup MIB.

---

#### Public Functions

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

---

```
void MibRemoteConfigGroup_vRegister ( void );
```

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

If groups have not been read from the PDM then they are initialized with default values.

The address for the "All" group is set to the "All Bulbs" group address, this means that every remote control's "All" group will control every bulb in a network.

The remaining group addresses are set to values that incorporate the remote control's MAC address. This makes these group addresses unique to each remote control, group A on one remote will always be different to group A on another remote.

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

---

```
void MibRemoteConfigGroup_vBuildAddr (
                                     in6_addr      *psAddr,
                                     MAC_ExtAddr_s *psMacAddr,
                                     uint16        u16Group );
```

This function builds a group address from a combination of MAC address and/or a 16-bit group number.

---

```
teJIP_Status MibRemoteConfigGroup_eSetAddr ( uint8 *pu8Val,
                                              uint8  u8Len,
                                              void   *pvCbData );
```

This function is called by the stack to set the value of the Addr variables in the RemoteConfigGroup MIB and is specified in the MIB declaration in **MibRemoteConfigGroupDec.c**. When this function is called the new values are saved to the PDM.

---

```
void MibRemoteConfigGroup_vGetAddr ( thJIP_Packet hPacket,
                                     void          *pvCbData );
```

This function is called by the stack to get the value of the Addr variables in the RemoteConfigGroup MIB and is specified in the MIB declaration in **MibRemoteConfigGroupDec.c**.

---

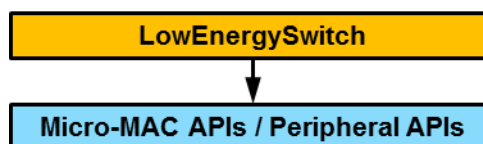
## 6.11 LowEnergySwitch Folder

The **LowEnergySwitch** folder of the application note contains source code that is specific to the low energy switches in *JenNet-IP Smart Home (JN-AN-1162)*

The low energy switch does not have any MIBs as it only transmits writes to variables in the MIBs of devices being controlled. At all other times it is either in deep sleep mode, (when battery powered), or completely powered down, (when powered by an energy harvester).

The source code in the **LowEnergySwitch** folder contains the only module implementing low energy switch. This application is built upon the limited IEEE 802.15.4 Micro-MAC. This version of the IEEE 802.15.4 MAC contains a minimal set of APIs supporting only the most basic radio operations making it suitable for use in devices that require very low energy use.

The following diagram shows the layers of that form the low energy switch application on top of the Micro-MAC stack:



---

### 6.11.1 LowEnergySwitch 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 radio parameters and settings.

Many of the variables used in the makefile match those described in [Section 6.1.1.1 "Standard Device Type Makefile"](#) the other variables are described below.

---

#### DEVICE\_NAME

The remote device supports the following values:

- **DR1174** for the *Carrier Board (DR1174)*.
- **DR1197** for the *ZF Energy Harvester Reference Design (DR1197)*.

---

#### CHANNEL

This variable specifies the channel that the device may operate on. The low energy switch can only operate on a single channel. The default value of 21 produces a binary that only operates on channel 21. To operate on a different channel this variable must be changed. The value is surfaced to the source code via the `#define MK_CHANNEL`.

---

### 6.11.2 LowEnergySwitch.c

**LowEnergySwitch.c** contains the source code for the low energy switch application.

The standard JIP callback functions are not present as this application makes use of the Micro-MAC APIs providing access to the minimal set of functions required to perform the most common radio tasks.

The following sections briefly describe the features of the **LowEnergySwitch.c** source code. The functions are presented in this documentation in the order they are called, though they appear in a different order in the source code file.

---

#### Defines

There are a number of local `#define` values in **LowEnergySwitch.c** that control the operation of the low energy switch. The most notable are described below:

---

#### DIO\_BUTTON\_MASK

This define is used when the hardware platform includes buttons that must be read to control the operation of the low energy switch. This value is taken from `MK_DIO_BUTTON` defined in the **makefile**.

The DR1197 build for the ZF Energy Harvesting Reference Design uses DIO8 as an input to ensure that commands packets are only transmitted when the switch is pressed and nothing is transmitted when the switch is released.

---

## **TAG\_CHANNEL**

This define is used to set the single radio channel the switch transmits upon. The value is taken from the MK\_CHANNEL defined in the **makefile**.

---

## **Public Functions**

The following public functions are implemented in **LowEnergySwitch.c**.

---

### **void AppColdStart ( void );**

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

This function simply calls the **AppWarmStart()** function.

---

### **void AppWarmStart ( void );**

This function is the entry point to the application following a wake from sleep with memory held. This application does not use such a sleep mode and so is never called directly in this way, though it is called from the **AppColdStart()** function.

This function first performs system initialisation, including the radio and hardware peripherals.

If a button input is specified the button is read and the results check to determine if a packet should be transmitted. Where a button is not specified the command is always transmitted.

If a command is to be transmitted the current frame counter for the command is read from EEPROM, incremented and written back to EEPROM. The command value is derived from the least significant bit of the frame counter which results in alternating on and off commands each time the application runs.

The command is then transmitted using a call to **vSendFrame()**.

Finally the application is ended in one of two ways depending upon the power source being used:

1. Energy Harvester: When building for an energy harvester device the software is put into a while loop until the harvested energy runs out.
2. Battery: When building for battery power the device is placed into deep sleep mode to preserve power until the software is reset.

---

### **void vSendFrame ( uint8 u8Command );**

This function builds the command packet and encrypts it using the fixed security key. The packet is then transmitted three times. The **vInterruptFired()** callback function is used to flag when the transmission has completed and the **vHwDeviceIntCallback()** callback function is used to insert a small delay between each transmission using the a wake timer.

---

### **void vInterruptFired ( uint32 u32InterruptBitmap );**

The **vInterruptFired()** callback function is used to flag when the transmission has completed.



---

```
void vInterruptFired ( uint32 u32DeviceId  
                      uint32 u32ItemBitmap );
```

The **vInterruptFired()** callback function is used to flag when the wake timer delay is over.

## 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 28/01/2015: Public v2004

##### Changes

Public release on Beyond Studio for NXP Toolchain.

##### Lpap440: Remote – fast standalone commissioning needs to be re-enabled

Done, to work with lpap443.

Also uses a standalone profile when in standalone mode so it gets passed to the joining devices allowing them to check the configured ping interval upon joining.

##### 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 21/10/2014: Internal v2003

### Changes

This release includes a draft version of the manual for review and the following changes:

#### **Lpap076: Bulb – Update to current bulb drivers**

Updated to use latest bulb drivers.

#### **Lpap297: Bulbs – Re-introduce BulbConfig MIB to white bulbs**

Device IDs for existing bulbs also updated.

#### **Lpap298: Bulbs – Add generic DeviceScene MIB**

Provides a way to configure scenes in all device types using the same mechanism.

#### **Lpap363: Documentation – Update with new features and code structure**

Draft copy created for review.

#### **Lpap365: Devices – Choose best settings for u8JNT\_IndirectTxBuffers**

To minimise packet loss when sending to End Devices.

#### **Lpap400: Devices – Protect against EEPROM corruption**

If power is lost during a write of the factory reset data.

#### **Lpap406: Sensors – Combine separate sensor folders into a single folder**

Use compilation options to select what type of sensor to build.

#### **Lpap437: Bulb – Add CctMax and CctMin variables to ColourConfig MIB**

Placeholder values added.

#### **Lpap438: Remote – Add commissioning of CCT and colour bulbs**

To allow their addition to standalone mode networks.

#### **Lpap439: Devices – Set appropriate profile when entering and leaving standalone mode**

Allowing devices to correctly operate in a standalone mode network.

### Compatibility

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

## A.3 21/10/2014: Internal v2002

### Changes

This release tidies up the compilation warnings from the previous release and adds a preview of the colour bulb device.

### Lpap303: Colour Bulb – Add Colour Bulb MIBs

The Colour Bulb MIBs are now fully functional.

### Compatibility

Product Type	Part Number	Public Version	Internal Version	Supported Chips
SDK Toolchain	JN-SW-4141	v?.?	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 a preview of the colour bulb device.

#### Lpap303: Colour Bulb – Add Colour Bulb MIBs [Not yet complete]

The majority of the ColourControl MIB variables for setting colour in the XY and Hue, Saturation colour spaces are implemented including transitions across those spaces. The Colour Wheel in the Gateway's Smart Devices interface can be used to set colour. The MIB Browser can be used to explore the other implemented MIB variables.

ColourControl MIB variables that have not been implemented are read only.

The ColourConfig MIB transition time variable is used when making transitions.

Colour scenes are not implemented.

#### 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.

### Compatibility

Product Type	Part Number	Public Version	Internal Version	Supported Chips
SDK Toolchain	JN-SW-4141	v?..?	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 lots 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	v?..?	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 mess up 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.1 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.2 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.

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.3 08/05/2014: Internal v1065

### Changes

This main purpose of this release is to complete 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.4 16/04/2014: Internal v1064

### Changes

This main purpose of this release is to port the applications on to the new Mini-MAC to free up additional code-space for application use.

The following changes were made in this release:

#### **Lpap339: End Device Sensors - Allow broadcasts while trying to re-join**

Applications updated to allow this along with a stack fix.

#### **Lpap366: End Devices – remove software reset following network loss**

Done, (this was a workaround for lpsw4766).

#### **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.5 27/03/2014: Internal v1063

### Changes

This main purpose of this release is to port the applications on to the new Mini-MAC to free up additional code-space for application use.

The following changes were made in this release:

#### Lpap332: MibBulb - Remove MIB library

Flatten structure, remove patch functions, adapt timings for End Device use, drop 4x support.

#### Lpap357: Remote - Cloning remotes into a standalone network is broken

Fixed

#### 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.

#### Lpap339: End Device Sensors - Allow broadcasts while trying to re-join

Stack should allow this, need to adapt application and test, requires fix for Lpsw4766.

### 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.6 03/02/2014: Internal v1062

### Changes

The main purpose of this release is to optimise the battery life of the End Device Occupancy Sensor. The End Device Illuminance Sensor has also been added to the package. Only source code for the Occupancy Sensor and Illuminance Sensor is included in the release package.

The following changes were made in this release:

#### **Lpap349: DeviceOccupancyIlluminance – Adapt for new template**

Allowing use as Router or End Device.

#### **Lpap356: Devices – Initialise DIO lines for lowest power consumption**

For best battery life on End Devices.

### 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.7 21/01/2014: Internal v1061

### Changes

This main purpose of this release is to optimise the battery life of the End Device Occupancy Sensor. The End Device Illuminance Sensor has also been added to the package. Only source code for the Occupancy Sensor and Illuminance Sensor is included in the release package.

The following changes were made in this release:

#### Lpap337: DeviceIlluminance – adapt for End Device operation

Techniques used for the Occupancy Sensor have been applied to the Illuminance Sensor.

#### Lpap340: End Device Sensors – 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.

#### Lpap341: End Devices – Optimise startup time

Wait until the stack or UART needs to be used before waiting for the 32MHz clock to stabilise.

#### Lpap342: End Devices – Use network key for rejoin following a power cycle

Instead of the commissioning key.

#### Lpap351: Sensors – 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, 2s parent poll, 5m OND query, 100ms between OND query and data poll, 30s for sensor refresh transmissions.

### 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 27/11/2013: Internal v1060

### Changes

This main purpose of this release is to allow the Occupancy Sensor Device to run as an EndDevice. Only source code for the Occupancy Sensor is included in the release package.

The following changes were made in this release:

#### **Lpap268: Sensors - add new modes to OccIllBulbConfig MIB Mode variable**

Renamed the existing mode OCC\_ILL (3) to OCC\_ILL\_AUTO (3). This mode automatically adjusts the bulb brightness to get the illuminance into the target band. This is therefore best used where the light of the bulbs falls upon the sensor.

Added OCC\_ILL\_MAX (4) mode. This mode:

- Turns the bulbs on at maximum brightness when occupied and the illuminance is low
- Turn off the bulbs when the illuminance is high.

This is therefore best used where the light of the bulbs does not fall upon the sensor.

Added ALWAYS\_OFF and ALWAYS\_ON modes, that disregard the sensor readings and turns the bulbs off or on respectively.

#### **Lpap269: Remote - Add activate scene support**

Buttons 1-4 activate scenes 0xA00A-0xD00D.

#### **Lpap300: MibCommon – remove MIB libraries and refactor 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 for only JN516x chips.

#### **Lpap301: DeviceOccupancy - allow operation as sleeping End Device**

In this mode the sensor wakes every 100ms to read the input sensor, only starting the stack when it has data to transmit.

When running as an End Device it cannot switch into standalone mode and so cannot control bulbs when the network is lost.

End Devices cannot receive broadcasts messages if the device is being used to receive occupancy updates from other sensors they will need to be unicast to the End Device sensor.

### Lpap334: MibSensor – adapt for End Device use

Done for MIBs used by Device Occupancy

### 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

---

## B.9 04/09/2013: Public v1059

### Changes

The following changes were made in this release:

#### Low Energy Switches

Low Energy Switch application has been added suitable for use with coin cell or energy harvesting power sources.

#### Lpap217: Remove commented out include paths from makefiles

Done

#### Lpap229: Place Sensors into default groups

All Sensors are placed into the “All Devices” group.

Occupancy and Occupancy/Illuminance Sensors are placed into the “All Occupancy Sensors” group.

Illuminance and Occupancy/Illuminance Sensors are placed into the “All Illuminance Sensors” group.

#### Lpap231: Remove JN514x build targets from Eclipse

Removed due to diverging stack APIs.

#### Lpap233: Add DeviceControl MIB to Sensors

Done.

#### Lpap252: Move Occupancy Sensor DIO to DIO 1

Allowing use with either Parallax PIR module or Generic Expansion Board.

### 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

---

## B.10 23/07/2013: Internal v1058

### Changes

The following changes were made in this release:

#### Minor Changes

Fixed Remote Controls not joining networks with addition of fast commissioning code.

### Compatibility

Product Type	Part Number	Public Version	Internal Version	Supported Chips
SDK Toolchain	JN-SW-4041	v1.1	-	JN5148-J01 JN5142-J01
JenNet-IP JN514x SDK Libraries	JN-SW-4051	v1.2	v695	JN5148-J01 JN5142-J01
JenNet-IP JN516x SDK Libraries	JN-SW-4065	-	v932	JN5168 JN5164

## B.11 18/07/2013: Internal v1057

### Changes

The following changes were made in this release:

#### OccupancyMonitor MIB

A single Occupancy variable can now be written to by all devices instead of each device having to be configured for a different variable.

Added Unoccupied variable for monitoring lack of occupancy.

Added DeviceTable variable to allow the devices being monitored to be read.

Renamed Devices variables to MaxDevices.

#### Lpsw3905: Fast standalone commissioning

Now uses fast commissioning for standalone mode. Joining nodes are all configured with the fast commissioning parameters. Remote controls transmit double fast commissioning packets when starting commissioning and every 30 seconds whilst commissioning.

#### Lpsw4023: Implement use of stay awake requests from JIP on End Devices

End Devices will stay awake for an extra 500ms following each stay awake request.

### Minor Changes

Re-instated mimic of PWM output to DIO17 in DriverBulb\_PWM.c.

Bulb software built using the SSL2108SYNC driver no longer doze.

### Compatibility

Product Type	Part Number	Public Version	Internal Version	Supported Chips
SDK Toolchain	JN-SW-4041	v1.1	-	JN5148-J01 JN5142-J01
JenNet-IP JN514x SDK Libraries	JN-SW-4051	v1.2	v695	JN5148-J01 JN5142-J01
JenNet-IP JN516x SDK Libraries	JN-SW-4065	-	v932	JN5168 JN5164



---

## B.12 12/07/2013: Internal v1056

### Changes

Updated sensor ALPHA which implements a bulb controlling occupancy and/or illuminance sensor device. See Sensor MIBs and Appendix B for details. Note that we intend to further alter the OccupancyMonitor MIB to remove the OccupancyX variables and replace with a single variable all devices can write to. This is to simplify the configuration of multiple occupancy sensor devices where each needs to be configured with a different variable to write their state into. This will use the source address of the write once the stack has been updated.

- Split DeviceSensor into three different devices DeviceOccupancy, DeviceIlluminance and DeviceOccupancyIlluminance.
- Setting OccIIBulbConfig MIB RefreshInterval variable to zero disables refresh transmissions.
- Setting OccIIBulbConfig MIB AdjustInterval variable to zero disables adjustment transmissions.
- OccupancyConfig MIB OccupiedDelay and OccupiedEvents variables can now be set individually.
- Added variables to OccupancyConfig MIB to allow transmission of occupied state to other devices, (most usefully the OccupancyMonitor MIB).
- Added IlluminanceConfig MIB which includes variables to allow transmission of illuminance state to other devices.
- Added OccupancyMonitor MIB to receive and combine occupancy states from other occupancy sensors.
- Added variables to IlluminanceControl MIB to provide additional methods for altering the target band from remote controls.
- Added checks in IlluminanceControl MIB to ensure the LuxTarget and LuxBand variables cannot be set out of range.
- Added full documentation for sensor MIBs and explained the use of OccupancyMonitor MIB in Appendix B.

### Compatibility

Product Type	Part Number	Public Version	Internal Version	Supported Chips
SDK Toolchain	JN-SW-4041	v1.1	-	JN5148-J01 JN5142-J01
JenNet-IP JN514x SDK Libraries	JN-SW-4051	v1.2	v695	JN5148-J01 JN5142-J01
JenNet-IP JN516x SDK Libraries	JN-SW-4065	-	v910	JN5168 JN5164

## B.13 27/06/2013: Internal v1055

### Changes

Added DeviceSensor ALPHA which implements a bulb controlling occupancy and illuminance sensor device.

### Compatibility

Product Type	Part Number	Public Version	Internal Version	Supported Chips
SDK Toolchain	JN-SW-4041	v1.1	-	JN5148-J01 JN5142-J01
JenNet-IP JN514x SDK Libraries	JN-SW-4051	v1.2	v695	JN5148-J01 JN5142-J01
JenNet-IP JN516x SDK Libraries	JN-SW-4065	-	v910	JN5168 JN5164

---

## B.14 13/06/2013: Internal v1054

### Changes

Built on new stack release v910.

#### Lpap190: Bulb – Scene levels not being properly applied on DR1175 bulb builds

Fixed – driver was not retaining light level whilst off, the replacement of the DR1175 driver with the generic PWM driver has fixed this.

#### Lpap214: End Devices – End Devices going into standalone mode

Fixed.

#### Lpsw3928: Device – Implement I/O Device Type

DeviceDio in Application Template includes this functionality.

#### Lpsw3931: Remote – Implement battery powered switch

Done.

#### Lpsw3941: Devices – Implement code simplification

Core network code now in common modules

#### Lpsw3994: Devices – Exception in PDM after binary update

Fixed.

### Compatibility

Product Type	Part Number	Public Version	Internal Version	Supported Chips
SDK Toolchain	JN-SW-4041	v1.1	-	JN5148-J01 JN5142-J01
JenNet-IP JN514x SDK Libraries	JN-SW-4051	v1.2	v695	JN5148-J01 JN5142-J01
JenNet-IP JN516x SDK Libraries	JN-SW-4065	-	<b>v910</b>	JN5168 JN5164

---

## B.15 28/05/2013: Internal v1053

### Changes

Built on new stack release, includes the following new notable features:

New DeviceRemote builds for use in switches with push buttons.

DemoSensor device intended as a starting point for discussion and evaluation of Occupancy and Light Level sensor with automatic bulb control. This software is not production quality in particular the MIBs will need to be significantly re-organised. See DemoSensor\Doc\Readme.txt for brief instructions.

### Lpap173: Bulb – Add PWM only driver

Added.

**Note:** the new **DriverBulb\_PWM.c** is now used in the Bulb DR1175 build instead of DriverBulb\_DR1175.c (which will be removed in a future release).

### Lpap195: Bulb – Enable network state colours in PCA9634Z End Device Bulb

Fixed.

### Lpap196: Remote – Remote DR1047 build not working

Fixed.

Note this build target has been removed from Eclipse and replaced by the more suitable DR1199 and DR1047A Remote builds. This DR1047 build is still available via the command line or adding a build target.

### Lpap211: Devices – Improve default Node MIB Name Variable value

The default value for the Node MIB Name variable value will be:

TPPPNCC MMMMMM

where:

T indicates device type

B=Bulb, R=Remote, S=Sensor

PPPP is the Product ID in hex digits

N indicates node type

c=Coordinator, r=Router, e=End Device

CC indicates chip type

2J = JN5142J01, 48 = JN4148, 8J = JN5148J01, 64 = JN5164, 68 = JN5168

MMMMMM is the least significant 6 digits of the MAC Address in hex digits

Alternatively the TTPPPNCC part can be overridden by specifying a value for the JIP\_NODE\_NAME makefile variable on the command line

### Lpap212: Devices – Allow factory reset magic number to be overridden from make command line

Add makefile variable `FACTORY_RESET_MAGIC` to allow the factory reset magic number to be overridden.

Intended for testing use.

### Lpap213: Remote - Add button driven remote control (switch)

Added `DeviceRemote_DR1199_JN5148J01` and `DeviceRemote_DR1199_JN5168` builds running on Evaluation Kit Generic Shield.

Added `DeviceRemote_DR1047A_JN5148J01` running on Evaluation Kit Controller Board.

## Compatibility

Product Type	Part Number	Public Version	Internal Version	Supported Chips
SDK Toolchain	JN-SW-4041	v1.1	-	JN5148-J01 JN5142-J01
JenNet-IP JN514x SDK Libraries	JN-SW-4051	v1.2	v695	JN5148-J01 JN5142-J01
JenNet-IP JN516x SDK Libraries	JN-SW-4065	-	<b>V905</b>	JN5168 JN5164

---

## B.16 08/05/2013: Internal v1052

### Changes

Built on new but patched stack release, mainly as a marker for first attempts at End Devices and shared common code.

### Lpap174: Devices – Move common networking functionality into common source files

Common code for many device types moved out of `DeviceBulb\DeviceBulb.c` and into `Common\Node.c` to allow reuse in other projects. `Node.c` supports both Routers and End Devices.

### Lpap194: Devices – Add basic End Device support

PCA9634 bulbs may be built as End Devices for experimentation.

## Compatibility

Product Type	Part Number	Public Version	Internal Version	Supported Chips
SDK Toolchain	JN-SW-4041	v1.1	-	JN5148-J01 JN5142-J01
JenNet-IP JN514x SDK Libraries	JN-SW-4051	v1.2	v695	JN5148-J01 JN5142-J01
JenNet-IP JN516x SDK Libraries	JN-SW-4065	-	<b>v857 + PATCH</b>	JN5168 JN5164

## B.17 24/01/2013: Public 1v5 (Internal v1050)

### Changes

Built on new JenNet-IP stack 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 JN514x SDK Libraries	JN-SW-4051	v1.2	v695	JN5148-J01 JN5142-J01
JenNet-IP JN516x SDK Libraries	JN-SW-4065	-	v857	JN5168 JN5164

## B.18 21/01/2013: Internal v1049

### Changes

#### Lpap167: Bulb JN516x – Check validity of EEPROM data on start-up

Validates EEPROM data as the number of sectors allocated to the PDM has changed which will confuse the PDM if an upgrade takes place without erasing the EEPROM (as happens with OND).

On start-up the data in the EEPROM sector used for factory reset detection should be validated against a new magic number and also the Device ID (to protect against changes in devices) and the EEPROM wiped if either do not match.

### Compatibility

Product Type	Part Number	Public Version	Internal Version	Supported Chips
SDK Toolchain	JN-SW-4041	v1.1	-	JN5148-J01 JN5142-J01
JenNet-IP JN514x SDK Libraries	JN-SW-4051	v1.2	v695	JN5148-J01 JN5142-J01
JenNet-IP JN516x SDK Libraries	JN-SW-4065	-	v811	JN5168 JN5164

## B.19 18/01/2013: Internal v1048

Withdrawn

---

## B.20 11/01/2013: Internal v1047

### Changes

#### Lpap163: Bulb JN516x – Moved factory reset flags into EEPROM

Now uses an EEPROM sector to store factory reset detection flags.

#### Bulb – Make thermal control loop optional

The thermal control loop is now optional and disabled by default in public bulb releases.

#### Bulb JN516x – Enables high temperature operation

Calls vAHI\_ExtendedTemperatureOperation(TRUE).

#### Remote – Replace bAHI\_PhyRadioSetPower()

eAppApiPlmeSet(PHY\_PIB\_ATTR\_TX\_POWER, 34+10\*x) should be used instead.

### Compatibility

Product Type	Part Number	Public Version	Internal Version	Supported Chips
SDK Toolchain	JN-SW-4041	v1.1	-	JN5148-J01 JN5142-J01
JenNet-IP JN514x SDK Libraries	JN-SW-4051	v1.2	v695	JN5148-J01 JN5142-J01
JenNet-IP JN516x SDK Libraries	JN-SW-4065	-	v811	JN5168 JN5164

## B.21 20/12/2012: Public 1v4 (Internal v1046)

This release adds support for the JN516x chip family.

### Known Issues

#### Lpap163: Bulb JN516x – Move factory reset flags into EEPROM

The wake timer register method currently used is very dependent upon the hardware power supply, though it should be sufficient with Evaluation Kit boards. For production bulbs this data ought to be stored in EEPROM either directly or via the PDM.

### Changes

#### Lpap160: Bulb SSL2108 SYNC – Reduces flicker by using constant current

Avoids flicker on some hardware platforms that are sensitive to current draw.

#### Lpap162: Bulb – Allows deletion of “All Devices” and “All Bulbs” groups

Previously these groups would be restored upon joining or re-joining a network. A User may have valid reasons to want to remove a bulb from these groups.

#### Lpap164: Devices JN516x – Validate PDM data matches that used by running software

Where data in the PDM indicates the data was saved for a device with a different JIP Device ID to the running software the PDM data is deleted and the device restarted. Most useful during development on JN516x Evaluation Kits as the software in a module may change often without necessarily clearing out the EEPROM when re-programmed.

### Compatibility

Product Type	Part Number	Public Version	Internal Version	Supported Chips
SDK Toolchain	JN-SW-4041	v1.1	-	JN5148-J01 JN5142-J01
JenNet-IP JN514x SDK Libraries	JN-SW-4051	v1.2	v695	JN5148-J01 JN5142-J01
JenNet-IP JN516x SDK Libraries	JN-SW-4065	v1.0	v783	JN5168 JN5164



---

## B.22 26/11/2012: Internal v1044

### Changes

#### Other: Bulb – Added SSL2108 Synchronised driver

This driver removes flicker on some SSL2108 bulb driver hardware platforms.

### Compatibility

Product Type	Part Number	Public Version	Internal Version	Supported Chips
SDK Toolchain	JN-SW-4041	v1.1	-	JN5148-J01 JN5142-J01
JenNet-IP SDK Libraries	JN-SW-4051	v1.2	v695	JN5148-J01 JN5142-J01

---

## B.23 09/11/2012: Public 1v3

### Changes

#### Lpap161: Bulb – Setting scenes is not activating scenes

Fixed.

### Compatibility

Product Type	Part Number	Public Version	Internal Version	Supported Chips
SDK Toolchain	JN-SW-4041	v1.1	-	JN5148-J01 JN5142-J01
JenNet-IP SDK Libraries	JN-SW-4051	v1.2	v695	JN5148-J01 JN5142-J01

## B.24 05/11/2012: Internal 1v2

### Changes

#### Lpap115: Bulb – ThermalControl.c Warning “will never be executed”

Fixed.

#### Lpap133: Bulb – Correct reduction of routing table size for debug builds

Now uses correct debug #define.

#### Lpap157: Remote – Limit abort for commissioning to hardware button only

Touch pads no longer abort commissioning, sometimes radio communications could trigger a pad and hence abort commissioning.

### Compatibility

Product Type	Part Number	Public Version	Internal Version	Supported Chips
SDK Toolchain	JN-SW-4041	v1.1	-	JN5148-J01 JN5142-J01
JenNet-IP SDK Libraries	JN-SW-4051	v1.2	v695	JN5148-J01 JN5142-J01

## B.25 01/10/2012: Internal 1v1v5

Internal only development release.

## B.26 26/09/2012: Public 1v1v4

### Changes

#### Lpap147: Remote – incorrectly using bulb device Type

Fixed to use Remote Device Type again allowing commissioning of additional remotes into standalone systems.

### Compatibility

Product Type	Part Number	Public Version	Internal Version	Supported Chips
SDK Toolchain	JN-SW-4041	v1.1	-	JN5148-J01 JN5142-J01
JenNet-IP SDK Libraries	JN-SW-4051	v1.1	v607	JN5148-J01 JN5142-J01

---

## B.27 14/09/2012: Internal 1v1v3

### Changes

#### Lpap139: Bulb – User profile being applied when joining a network

Application source code for this release is unchanged from version 1v1v1 but it was built upon an updated JenNet-IP stack release v607.

### Compatibility

Product Type	Part Number	Public Version	Internal Version	Supported Chips
SDK Toolchain	JN-SW-4041	v1.1	-	JN5148-J01 JN5142-J01
JenNet-IP SDK Libraries	JN-SW-4051	v1.1	v607	JN5148-J01 JN5142-J01

---

## B.28 Internal 1v1v2

This was an unreleased internal version.

---

## B.29 10/09/2012: Internal 1v1v1

### Changes

#### Lpap139: Bulb – User profile being applied when joining a network

Removed so standard profile continues to be used.

### Compatibility

Product Type	Part Number	Public Version	Internal Version	Supported Chips
SDK Toolchain	JN-SW-4041	v1.1	-	JN5148-J01 JN5142-J01
JenNet-IP SDK Libraries	JN-SW-4051	-	v594	JN5148-J01 JN5142-J01

## B.30 06/09/2012: Internal 1v1

### Changes

#### Lpap135: Bulb – Use standard join profile 6 instead of user profile

Allowing the profile to be overridden as required by the network coordinator.

#### Lpap136: Bulb – Apply minimum LQIs in beacon response handler

To allow differentiation between joins and re-joins when using join profiles with the lowest LQI settings.

#### Lpap137: Bulb – delete child table when resetting stack on mode changes

To aid node recovery.

#### Lpap138: Bulb – revert to using join profile 6 on STACK\_RESETs when joining a network for the first time

To prevent networks running different profiles overriding the profile when the bulb is not allowed to join those networks.

### Compatibility

Product Type	Part Number	Public Version	Internal Version	Supported Chips
SDK Toolchain	JN-SW-4041	v1.1	-	JN5148-J01 JN5142-J01
JenNet-IP SDK Libraries	JN-SW-4051	-	v592	JN5148-J01 JN5142-J01

---

## B.31 10/08/2012: Internal 1v0v2

### Changes

#### Lpap53: Remote – Factory reset takes too long

**PDM\_vDelete()** was not returning and so the watchdog kicked in after 16 seconds. Workaround added to delete the flash sectors directly using AHI calls.

#### Lpap105: Bulb – Check that ending cadence effects return to the bulb to an off state

**DriverBulb\_DR1175.c** was not updating the on state of the bulb correctly causing the bulb to not be turned off after a cadence effect (if it was off before the cadence effect was started).

#### Lpap117: Bulb – Remove commented out code in **DriverBulb\_DR1175.c** **vCbTimer1()** function

Removed to avoid confusion.

### Compatibility

Product Type	Part Number	Public Version	Internal Version	Supported Chips
SDK Toolchain	JN-SW-4041	v1.1	-	JN5148-J01 JN5142-J01
JenNet-IP SDK Libraries	JN-SW-4051	v1.0	v533	JN5148-J01 JN5142-J01

## B.32 13/07/2012: Internal 1v0v1

### Changes

#### Lpap112: Bulb – Security frame counter rolling back on stack reset

Fixed, frame counter from PIB now passed into NwkSecurity Resume functions used when changing stack modes.

#### Lpap114: Eclipse – Warning "Error launching external scanner info generator"

Fixed, project files updated to correct this.

#### Lpap116: NwkTest MIB – expanded for bi-directional measurements.

Added MacRetries, TxLqiMin, TxLqiMax, TxLqiMean and RxLqi variables.

### Compatibility

Product Type	Part Number	Public Version	Internal Version	Supported Chips
SDK Toolchain	JN-SW-4041	v1.1	-	JN5148-J01 JN5142-J01
JenNet-IP SDK Libraries	JN-SW-4051	v1.0	v533	JN5148-J01 JN5142-J01

## B.33 09/07/2012: Public 1v0

### Compatibility

Product Type	Part Number	Public Version	Internal Version	Supported Chips
SDK Toolchain	JN-SW-4041	v1.1	-	JN5148-J01 JN5142-J01
JenNet-IP SDK Libraries	JN-SW-4051	v1.0	v533	JN5148-J01 JN5142-J01



## 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](http://www.nxp.com)