JenNet-IP LAN/WAN Stack User Guide

JN-UG-3086
Revision 1.3
15 August 2013

Contents

Part I: Concept Information

Part II: C JenNet-IP API

Part III: Java JenNet-IP API

Part IV: Appendices

About this Manual

This manual is the main reference resource in developing applications for devices on the LAN/WAN side of an NXP JenNet-IP system. It describes the LAN/WAN stack software over which applications run on these devices. The manual first introduces the basic principles of a JenNet-IP system and the internet protocols on which JenNet-IP (JIP) is built. It then describes the Application Programming Interfaces (APIs) that can be used to develop JenNet-IP applications that run on IP-connected devices such as PCs, tablets, mobile phones and IP routers. The API resources (functions, network parameters, enumerations, structures, etc) are fully detailed.

Note: JenNet-IP systems are fully introduced in the *JenNet-IP WPAN Stack User Guide (JN-UG-3080)*, which also describes application development for the WPAN (IEEE 802.15.4) side of a JenNet-IP system.

Organisation

The manual is divided into 4 parts:

- Part I: Concept Information comprises 2 chapters providing background information for JenNet-IP (JIP):
  - Chapter 1 introduces JenNet-IP systems.
  - Chapter 2 outlines the IP (Internet Protocol) concepts that you will need for an understanding of JenNet-IP systems.
- Part II: C JenNet-IP API comprises 3 chapters detailing the C JenNet-IP API (or C JIP API), which can be used to develop applications for an IP Host device in a JenNet-IP system:
  - Chapter 3 details the main tasks to implement in an application using the C JIP API functions.
  - Chapter 4 details the functions of the C JIP API.
  - Chapter 5 details the structures of the C JIP API.
- Part III: Java JenNet-IP API comprises 5 chapters detailing the Java JenNet-IP API (or Java JIP API), which can be used to develop applications for an IP Host device in a JenNet-IP system:
  - Chapter 6 details the main tasks to implement in an application using the Java JIP API functions.
  - Chapter 7 details the Java JIP API com.nxp.jip package.
  - Chapter 8 details the Java JIP API com.nxp.jip.variables package.
  - Chapter 9 details the Java JIP API com.nxp.jip.service package.
  - Chapter 10 details the Java JIP API com.nxp.jip.service.persist package.

- Part IV: Appendices describes the following miscellaneous topics:
  - JenNet-IP Browser
  - JenNet-IP Command Line Interface (CLI)
  - The key terminology used in JenNet-IP networks

Conventions

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

Function parameters are represented in *italics* type.

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

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

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

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

Acronyms and Abbreviations

API	Application Programming Interface
CLI	Command Line Interface
ICMP	Internet Control Message Protocol
IP	Internet Protocol
JenNet	Jennic Network
JIP	JenNet-IP
LAN	Local Area Network
MIB	Management Information Base
MLD	Multicast Listener Discovery
MTU	Maximum Transmission Unit

NVM	Non-Volatile Memory
OND	Over-Network Download
PAN	Personal Area Network
SDK	Software Developer’s Kit
SSBL	Second-Stage Bootloader
UDP	User Datagram Protocol
WAN	Wide Area Network
WPAN	Wireless Personal Area Network
6LoWPAN	IPv6 over Low power Wireless Personal Area Networks

Related Documents

JN-UG-3080	JenNet-IP WPAN Stack User Guide
JN-UG-3089	JenNet-IP EK040 Evaluation Kit User Guide
JN-UG-3093	JN516x-EK001 Evaluation Kit User Guide
JN-AN-1110	JenNet-IP Border Router Application Note
JN-AN-1162	JenNet-IP Smart Home Application Note

Support Resources

To access JN516x support resources such as SDKs, Application Notes and User Guides, visit the Wireless Connectivity TechZone:

www.nxp.com/techzones/wireless-connectivity

For JN514x resources, visit the NXP/Jennic web site: **www.jennic.com/support**

Trademarks

All trademarks are the property of their respective owners.

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

Part I:
Concept Information

1. JenNet-IP Overview

This chapter provides an overview of a JenNet-IP system, including the essential background information for developing applications to be run on devices in the LAN/WAN part of the system (for example, PCs, tablets, mobile phones and IP routers which may provide system access from the Internet).

Note: A more complete introduction to JenNet-IP is provided in the *JenNet-IP WPAN Stack User Guide (JN-UG-3080)*. The JenNet-IP user documentation is described below in Section 1.1.

1.1 JenNet-IP User Documentation

The full JenNet-IP user documentation set comprises the following:

Part Number	Document Title	Desciption
JN-UG-3080	JenNet-IP WPAN Stack User Guide	Provides a general introduction to JenNet-IP and details the software resources for developing applications that run on devices on the WPAN (IEEE 802.15.4) side of a JenNet-IP system
JN-UG-3086	JenNet-IP LAN/WAN Stack User Guide (this manual)	Details the software resources for developing applications that run on devices on the LAN/WAN side of a JenNet-IP system
JN-AN-1110	JenNet-IP Border-Router Application Note	Provides information and software for developing a custom Border-Router device which interfaces the WPAN and LAN/WAN sides of a JenNet-IP system

Table 1: JenNet-IP User Documentation

If you are new to JenNet-IP, you should first read the introductory chapters of the *JenNet-IP WPAN Stack User Guide (JN-UG-3080)* to familiarise yourself with the necessary concepts. The manual also introduces the NXP JenNet-IP Software Developer's Kit (SDK) which contains support software to facilitate application development for both the WPAN and LAN/WAN sides of a JenNet-IP system.

If you are developing applications for devices in the IP domain, you should then use this manual (JN-UG-3086) as follows:

- If you are unfamiliar with Internet Protocol concepts, you are advised to study Chapter 2
- Irrespective of your previous experience, you are advised to read the rest of this chapter and refer to one of the following during application development:
  - For C-based development, refer to Part II: C JenNet-IP API
  - For Java-based development, refer to Part III: Java JenNet-IP API

1.2 A JenNet-IP System

A JenNet-IP system is based on 6LoWPAN (IPv6 over Low power Wireless Personal Area Networks). It consists of two parts:

- **WPAN domain:** Contains one or more Wireless Personal Area Networks (WPANs), also referred to as wireless clusters, that operate using the NXP JenNet protocol built on top of the IEEE 802.15.4 standard
- **LAN/WAN domain:** Contains a Local Area Network (LAN), such as an Ethernet bus, that may be connected to a Wide Area Network (WAN), such as the Internet, allowing the WPAN(s) to be monitored and controlled via IP

A typical system is illustrated in Figure 1 below.

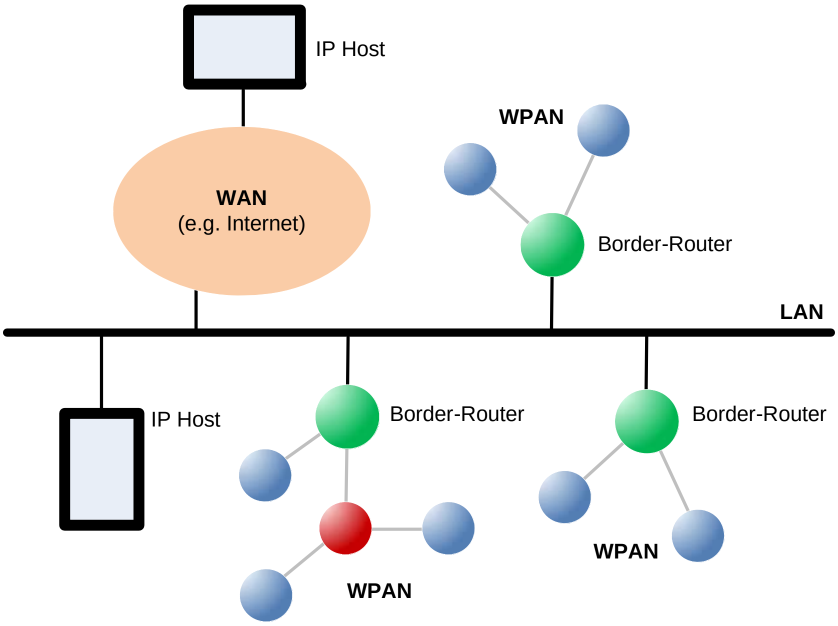

Figure 1: Typical JenNet-IP System

The main components of a JenNet-IP system (as illustrated in Figure 1) are as follows:

- **WPAN:** A wireless network operating over JenNet/IEEE 802.15.4 and containing nodes based around NXP JN51xx wireless microcontrollers
- **LAN:** A local IP-based bus (e.g. Ethernet) to which the WPANs are connected
- **Border-Router (WPAN-LAN Router):** A device used to connect a WPAN to the LAN
- **WAN:** A wide-range IP-based network (e.g. the Internet) connected to the LAN
- **IP Host:** A device on a WAN or the LAN with an IP connection to the system - for example, this may be a PC, tablet or mobile phone

The above components are described in more detail in the sub-sections below.

1.2.1 WPAN (Wireless Cluster)

A WPAN or ‘wireless cluster’ in a JenNet-IP system is a wireless network that operates using the NXP JenNet protocol which is built over the industry-standard IEEE 802.15.4 wireless network protocol. Each WPAN contains a single Co-ordinator node and a number of other nodes, called Routers and End Devices. The Co-ordinator is normally incorporated in the Border-Router device, described in Section 1.2.3. A wireless node is based on an NXP JN5168, JN5164, JN5148-J01 or JN5142-J01 microcontroller.

Messages are sent between the wireless nodes of a JenNet-IP system as IPv6 packets which are compressed and embedded in IEEE 802.15.4 frames. The delivery of a message uses the destination IPv6 address from the embedded IPv6 packet.

1.2.2 LAN

The LAN in a JenNet-IP system connects together the WPANs of the system. It is typically an Ethernet bus. The bus allows the WPANs to communicate with each other (send a message from a node in one network to another node in a different network) by means of IPv6 packets. The LAN may also provide a connection to a WAN, such as the Internet (and this WAN may provide connections to other JenNet-IP systems consisting of a LAN and associated WPANs).

1.2.3 Border-Router (WPAN-LAN Router)

The Border-Router is a device used to connect a WPAN to the LAN. It is also sometimes referred to as an Edge-Router. The Border-Router is usually incorporated in the same device as the network Co-ordinator.

Within a WPAN, messages are transported as IEEE 802.15.4 frames with compressed IPv6 packets embedded in their payloads. However, on the LAN they are transported as uncompressed IPv6 packets encapsulated in the LAN frames (e.g. Ethernet). The Border-Router must therefore:

- Take an IEEE 802.15.4 frame from its WPAN, extract the compressed IPv6 packet from the frame payload, uncompress the packet and insert it into a frame for transportation on the LAN.
- Take an encapsulated IPv6 packet from the LAN, extract the packet from the frame, compress the packet and then insert it into the payload of an IEEE 802.15.4 frame for transportation within the WPAN.

To receive messages destined for its own WPAN, a Border-Router must ‘listen’ on the LAN for messages addressed to members of its WPAN - for this, the Border-Router must analyse the destination IPv6 address in each IPv6 packet broadcast on the LAN.

1.2.4 WAN

The LAN may be connected to one or more WANs to allow remote access to the attached WPAN(s) through IP-based communication. A WAN is typically the Internet, allowing access from virtually anywhere in the world.

1.2.5 IP Hosts

An IP Host is a device with an IP connection to the system, from which the WPAN(s) can be accessed - for example, this may be a PC, tablet or mobile phone. It may also be an intermediary device which interacts with a WPAN and serves web pages to IP Hosts containing a user interface.

1.3 Software Architecture and Components

This section introduces the software that is used in the different parts of a JenNet-IP system, first taking a high-level view in Section 1.3.1, and then taking a more detailed view in Section 1.3.2 (IPv6 case) and Section 1.3.3 (IPv4 case).

1.3.1 Software Overview

The software in a JenNet-IP system runs in three distinct parts of the system:

- Nodes of the WPAN
- Border-Router between the WPAN and LAN/WAN domains
- Devices in the LAN/WAN domain

These divisions are illustrated in the figure below.

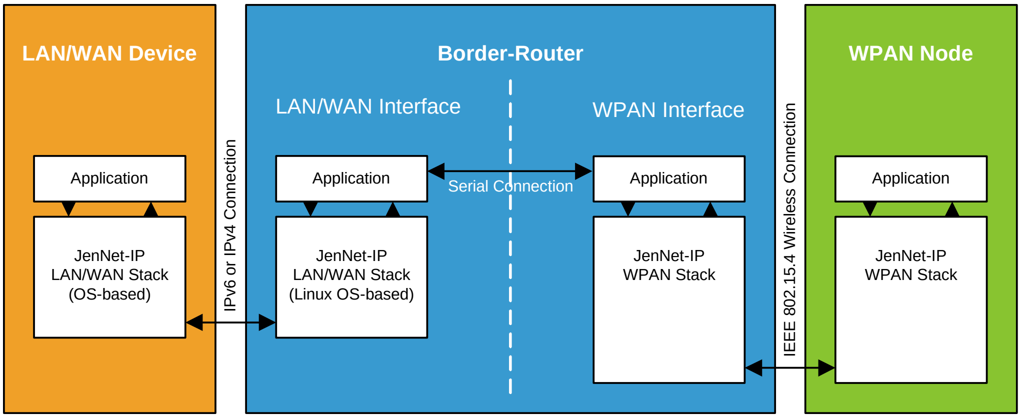

Figure 2: Software Divisions in JenNet-IP System

Working from right to left in the above diagram:

- **WPAN Node:** The user application operates over the JenNet-IP WPAN stack, which communicates with the Border-Router via an IEEE 802.15.4 radio link.
- **Border-Router:** This device has both LAN/WAN and WPAN interfaces:
  - **WPAN Interface:** This side of the Border-Router runs a JenNet-IP WPAN stack, which communicates with the equivalent stack on the WPAN nodes - this side of the Border-Router usually acts as the WPAN Co-ordinator node

- **LAN/WAN Interface:** This side of the Border-Router runs a JenNet-IP LAN/WAN stack, which communicates with the equivalent stack on the remote IP Host (LAN/WAN device) - this side of the Border-Router must be a Linux-based device

  The two sides of the Border-Router communicate via a serial link.

- **LAN/WAN Device:** The user application operates over a JenNet-IP LAN/WAN stack, which is connected to the Border-Router via an IP (IPv6 or IPv4) link.

The above architecture is described in more detail in Section 1.3.2 and Section 1.3.3.

1.3.2 Software Components (IPv6 Case)

This section provides more details of the JenNet-IP software components introduced in Section 1.3.1, in the case of an IPv6 connection to the IP domain.

> **Note:** The JenNet-IP software components that are required in the case of an IPv4 connection to the IP domain are outlined in Section 1.3.3.

The figure below is a more detailed version of Figure 2, showing the contents of the JenNet-IP stacks (below the applications) and other software components required in the Border-Router.

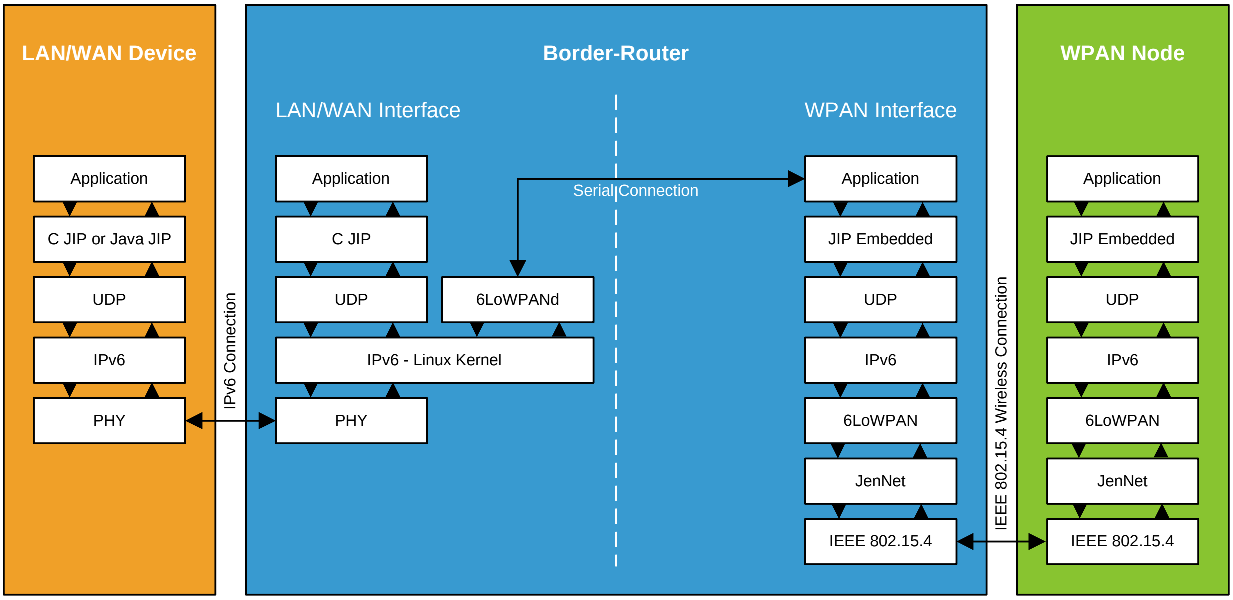

Figure 3: Software Components in JenNet-IP System (IPv6 Case)

Again, working from right to left in the above diagram:

WPAN Node

The following software runs on the NXP JN514x microcontroller in a node of a WPAN:

- **Application:** This software is developed using C APIs provided in the *JN516x JenNet-IP SDK (JN-SW-4065)* and *JN514x JenNet-IP SDK (JN-SW-4051)*. In particular, the JenNet-IP Embedded API is needed (described in the *JenNet-IP WPAN Stack User Guide (JN-UG-3080)*).
- **JenNet-IP WPAN Stack:** This software stack is also provided in the *JN516x JenNet-IP SDK (JN-SW-4065)* and *JN514x JenNet-IP SDK (JN-SW-4051)*. It consists of the stack layers indicated in Figure 3 and detailed in the *JenNet-IP WPAN Stack User Guide (JN-UG-3080)*.

Border-Router

The software that runs on the Border-Router provides the interface between the WPAN and LAN/WAN domains. The device has an interface to the WPAN and an interface to the LAN/WAN domain, with a dedicated software stack at each of these two interfaces:

- **Software at WPAN interface:** This is similar to the software that runs on a WPAN node (see above), comprising a user application over the JenNet-IP WPAN stack, with the addition of a serial protocol that allows internal communication with the software stack at the LAN/WAN interface (see below). The WPAN stack on the Border-Router normally provides the services of a Co-ordinator node for the WPAN.
- **Software at LAN/WAN interface:** This software allows a LAN/WAN device to interact with the Border-Router and, in turn, with the WPAN. It comprises:
  - **Application (optional):** This application is optional and, if implemented, allows the operator to interact with the system via web pages served to a web browser running on the LAN/WAN device. The application is developed using the C JenNet-IP API provided in the *JN516x JenNet-IP SDK (JN-SW-4065)* and *JN514x JenNet-IP SDK (JN-SW-4051)*, and described in Part II: C JenNet-IP API. This API allows the development of an application for any Linux-based platform.
  - **JenNet-IP LAN/WAN Stack:** This software stack includes both JenNet-IP components and standard Linux OS components, and is described in Section 1.4. The JenNet-IP application **6LoWPANd** implements the serial protocol which allows internal communication between the Linux kernel and the application at the WPAN interface. This application is supplied by NXP (see below) but a custom application can be used to implement this serial communication.

The JenNet-IP WPAN and LAN/WAN stacks can be implemented within the same device or in separate devices (connected via a serial link). For example, in the cases of the JenNet-IP EK040 and JN516x-EK001 Evaluation Kits, the LAN/WAN stack is implemented in a Linksys router and the WPAN stack is implemented on a JN51xx-based dongle which plugs into a USB port of the router (the dongle is referred to as the Border-Router node). The necessary JenNet-IP software components for the LAN/WAN stack are supplied in the firmware of the Linksys router. If you wish to design your own Border-Router, you will need to compile **6LoWPANd** for your target from the source code provided in the Application Note *JenNet-IP Border-Router (JN-AN-1110)* or develop your own **6LoWPANd** application to allow serial communication between the two interfaces.

Further software may also be required in the Border-Router, depending on the features implemented. For example, if the Over Network Download (OND) feature is to be used then the application **FWDISTRIBUTION** will be needed. Again, if you wish to design your own Border-Router, you will need to compile **FWDISTRIBUTION** for your target from the source code provided in the Application Note *JenNet-IP Border-Router (JN-AN-1110)* or develop your own **FWDISTRIBUTION** application.

LAN/WAN Device

The following software runs on a LAN/WAN device (an IP Host), such as a PC, tablet or mobile phone, to allow the WPAN to be monitored and controlled:

- **User Application (optional):** This software can be used to monitor/control the WPAN. It can be developed using the C JenNet-IP API, described in Part II: C JenNet-IP API, or the Java JenNet-IP API, described in Part III: Java JenNet-IP API. Alternatively, a standard test application known as the **JenNet-IP Browser** can be used which is supplied as a Java executable in the JenNet-IP SDK (and is introduced in Section 1.8.1). This application is not needed if a standard web browser is used as a user interface which receives web pages served by an application on the LAN/WAN side of the Border-Router (see above).
- **JenNet-IP LAN/WAN Stack:** This software stack includes both JenNet-IP components and standard OS components. The JenNet-IP components are provided in the JenNet-IP SDK. The stack consists of the layers indicated in Figure 3 and detailed in Section 1.4.

1.3.3 Software Components (IPv4 Case)

This section provides more details of the JenNet-IP software components introduced in Section 1.3.1, in the case of an IPv4 connection to the IP domain.

Note: The JenNet-IP software components that are required in the case of an IPv6 connection to the IP domain are described in Section 1.3.2.

The figure below is a more detailed version of Figure 2, showing the contents of the JenNet-IP stacks (below the applications) and other software components required in the Border-Router.

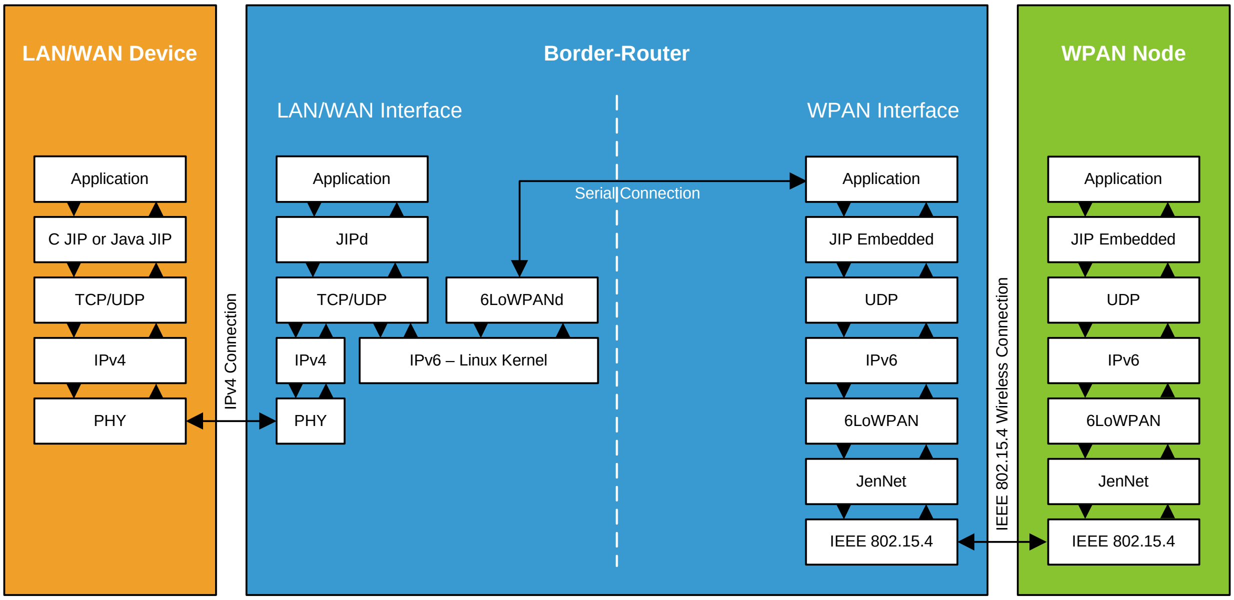

Figure 4: Software Components in JenNet-IP System (IPv4 Case)

The software depicted in Figure 4 is similar to that described for the IPv6 case in Section 1.3.2, with the following differences:

- **LAN/WAN Device:** In the JenNet-IP LAN/WAN stack on this device:
  - The UDP layer is replaced by a TCP/UDP layer
  - The IPv6 layer is replaced by an IPv4 layer
- **Border-Router:** In the JenNet-IP LAN/WAN stack at the LAN/WAN interface:
  - **JIPd** is a special application which is supplied in the JenNet-IP SDK and which implements the JIPv4 protocol over TCP/UDP (JIPv4 encapsulates JIP packets, including their IPv6 addressing, into either IPv4 UDP datagrams or an IPv4 TCP stream)
  - IPv4 and IPv6 co-exist side-by-side, IPv4 for the connection to the LAN/WAN domain and IPv6 for the communications with the WPAN (IPv6 packets are embedded in IEEE 802.15.4 frames)

1.4 JenNet-IP LAN/WAN Stack

This section provides details of the stack that runs on devices in the LAN/WAN domain of a JenNet-IP system, including the LAN/WAN devices (IP Hosts) and the LAN/WAN side of the Border-Router (see Section 1.3).

The diagram in Figure 5 below shows the levels and layers of the JenNet-IP LAN/WAN stack.

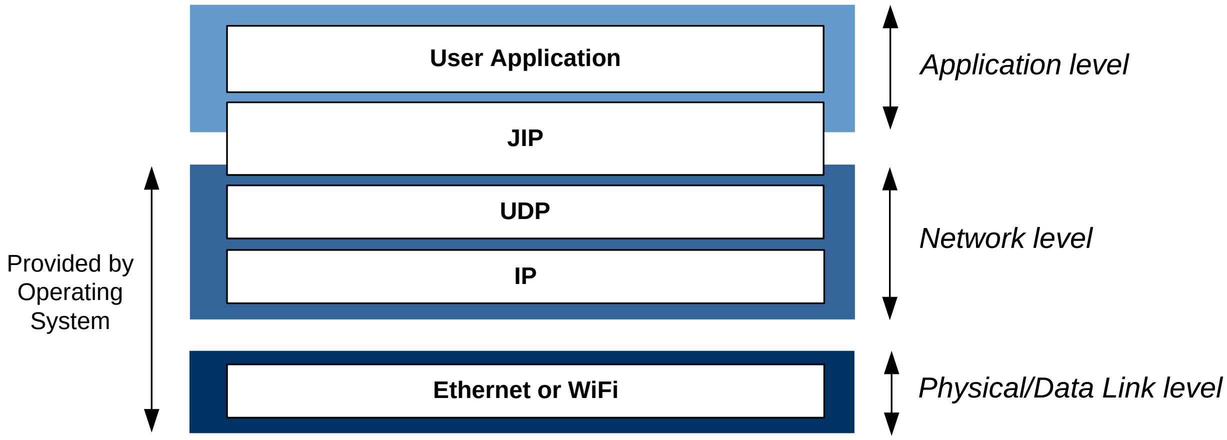

Figure 5: NXP JenNet-IP Stack - LAN/WAN Side

The three basic levels are now detailed in the sub-sections below.

1.4.1 Application Level

The Application level provides services for the application processes that wish to communicate with the devices/nodes in the wireless network. Within the Application level are the user application and JenNet-IP (or JIP) layer.

User Application

The user application is normally designed to monitor and control remote WPANs via an IP connection. This manual is concerned with the development of this software. The application interacts with the network through the JIP layer, which operates purely on a remote basis but also locally maintains information about a remote WPAN.

JIP

JenNet-IP or JIP is NXP's proprietary protocol which provides the user application with access to device functionality. JenNet-IP APIs are provided for this purpose. The JIP layer allows access to the nodes of a remote WPAN in order to set and retrieve values in MIBs (Management Information Bases) on the nodes.

The basic concepts which underlie the JIP layer are very similar to the industry-standard Simple Network Management Protocol (SNMP) in that configurable MIB variables and useful information can be accessed via a common protocol. Access to these variables may possibly result in additional actions - for example, setting the RF channel variable will not only set the value but also result in the channel being changed, while reading the current DIO pin levels will have no side effect.

The JIP layer also allows 'traps' to be associated with variables. A trap is a mechanism by which a notification event is generated if the associated variable changes. Traps can be configured/unconfigured for individual variables.

The JIP layer is described in more detail in Section 1.5.

Note: JIP is the default application-level protocol but developers can alternatively use their own custom UDP-based protocol, if desired.

1.4.2 Network Level

The Network level manages communications with the network. Its components are provided by the Operating System (Linux in the case of the Border-Router).

The following protocols are provided for assembling/disassembling IPv6 packets:

User Datagram Protocol (UDP)

The UDP layer is a simple message-based connectionless protocol. JenNet-IP packets are implemented as UDP packets embedded in the payloads of IP packets. Thus, this layer is concerned with constructing/deconstructing UDP packets. An IP-UDP socket interface is provided to allow packets to be passed to/from the IP layer (below). The NXP UDP socket interface follows the Berkeley Socket API.

Internet Protocol (IP)

The IP layer provides functionality for delivering packets over a network, using IPv6 or IPv4 (in both cases, an IPv6 destination address is included in the original JenNet-IP packet). The layer is responsible for assembling/disassembling IP packets by inserting/extracting UDP packets, and handling the IP packet headers.

Note: The JenNet-IP APIs (introduced in Section 1.5) allow the user application to interact with the UDP and IP layers. Most operations are performed through interactions with the UDP layer.

1.4.3 Physical/Data Link Level

The Physical/Data Link level implemented in the LAN/WAN domain depends on the particular IP transmission medium utilised by the IP Host device - for example, this medium could be Ethernet or WiFi.

The components of this level are provided by the Operating System (Linux in the case of the Border-Router).

1.5 Essential JenNet-IP Concepts

JenNet-IP or JIP is NXP's proprietary protocol which provides the user application with access to device functionality. On the LAN/WAN side of a JenNet-IP system, there are two associated APIs comprising functions (and associated resources) which facilitate this access:

- **C JenNet-IP API (C JIP API):** This API is used to develop applications that will run on an IP Host device (such as a PC, tablet or mobile phone) or on the LAN/WAN side of the Border-Router. Use of these functions is described in Part II: C JenNet-IP API.
- **Java JenNet-IP API (Java JIP API):** This API is used to develop applications that will run on an IP Host device (such as a PC, tablet or mobile phone). Use of these functions is described in Part III: Java JenNet-IP API.

> **Note:** In addition to the above APIs, the JenNet-IP SDK includes the JenNet-IP CLI (Command Line Interface) which allows access to JenNet-IP devices (such as WPAN nodes) from the command line on an IP Host. The JenNet-IP CLI is described in Appendix B.

In a JenNet-IP system, data is held on WPAN nodes in one or more Management Information Bases (MIBs). A MIB comprises a table of local variables and their values - for example, a MIB on an environment monitoring node may contain variables for temperature, humidity and wind speed. The functionality to interact with a MIB is incorporated in the JIP layer of the stack. MIBs are described further in Section 1.5.1.

> **Note:** JenNet-IP provides high-level functionality that allows the application to interact with MIB variables. For application developers who wish to work with JIP and MIBs at a lower level, the necessary JIP principles are outlined in an appendix of the *JenNet-IP WPAN Stack User Guide (JN-UG-3080)*.

1.5.1 MIBs and MIB Variables

A MIB (Management Information Base) is a database containing local variables and their values, held in memory on a wireless node. A MIB allows variables to be collected into a logical group. Up to 255 MIBs can exist on each node. The JenNet-IP WPAN stack creates five standard MIBs (described in the *JenNet-IP WPAN Stack User Guide (JN-UG-3080)*) and, therefore, the local application can create up to 250 MIBs.

The application on a wireless node can define one or more MIB types, each with a unique identifier, name and set of variables. A MIB of a particular type can then be declared and registered with JenNet-IP. Each MIB is given a unique name and handle.

A MIB type (and therefore MIB) can have up to 255 variables. Each variable is assigned the following:

- Handle
- Name
- Type
- Remote access rights (constant, read-only, read-write)
- 'Set' and 'Get' callback functions (on wireless node only)

The callback functions are user-defined and called by the stack whenever a request is received to set or get the value of the variable. A variable can be enabled or disabled - in the disabled state, it is not possible to set or get the variable's value.

Note that it is the local application on a WPAN node that defines a MIB type (and the variables within it) and creates a MIB. However, remote applications (e.g. on an IP Host device) can send requests to access a MIB and its variables.

A MIB variable can have an associated 'trap' to allow automated monitoring of the variable's value/state. Traps are described in Section 1.5.2 below.

1.5.2 Traps

Traps are provided by the JIP layer of the stack and are similar to the industry-standard SNMP traps. A trap is associated with a specific MIB variable on a remote node (see Section 1.5.1) and is used to monitor the state of the variable. If a trap has been set on a particular variable, any change in the variable will result in the generation of a trap notification event to inform the application which set the trap. This may result from a change in the value or in the enabled state of the MIB variable.

Traps can be globally suspended and resumed by the local application.

1.6 Network Data and Standard MIBs

Each node of a WPAN holds certain information about itself and the network to which it belongs. This data is stored in five standard MIBs that are created by the JenNet-IP WPAN stack on the node, which include the Node MIB and the JenNet MIB.

The Node MIB includes variables for:

- IEEE/MAC address
- Node name
- Application version
- Radio transmission power setting

The JenNet MIB includes variables for:

- Network device type of node
- Depth of node in tree
- Number of descendents of node in tree

- Neighbour table of node

The standard MIBs and their variables are described in the *JenNet-IP WPAN Stack User Guide (JN-UG-3080)*.

Information held in the standard MIBs on a node can be read by an application on an IP Host device as described in Section 3.8.1.

1.7 Application Development

The software resources described in this manual can be used to develop an application which provides user access to a WPAN from a LAN/WAN device (IP Host). The application runs on the JenNet-IP LAN/WAN stack (described in Section 1.4) on the LAN/WAN side of the Border-Router or on an LAN/WAN device:

- **Border-Router:** If implemented, this application would typically serve web pages to a normal web browser running on a LAN/WAN device, allowing the user to access the WPAN from the web browser. The application is developed using the C JIP API, which is provided in the *JN516x JenNet-IP SDK (JN-SW-4065)* and *JN514x JenNet-IP SDK (JN-SW-4051),* and which allows the development of an application for a Linux-based platform.
  - To develop an application of this type using the C JIP API, refer to Part II: C JenNet-IP API.
- **LAN/WAN device:** If implemented, this software provides a dedicated interface for accessing a WPAN. It can be developed using the C JIP API (on a Linux-based platform) or the Java JIP API provided in the *JN516x JenNet-IP SDK (JN-SW-4065)* and *JN514x JenNet-IP SDK (JN-SW-4051)*.
  - To develop an application of this type using the C JIP API, refer to Part II: C JenNet-IP API.
  - To develop an application of this type using the Java JIP API, refer to Part III: Java JenNet-IP API.

Note: Generally, to access a WPAN, an application is only needed for one of the above devices. However, it is possible to implement applications on both devices, providing the options of using a normal web browser or a dedicated interface on the LAN/WAN device.

Examples of the above applications are provided by NXP, as described below in Section 1.8.

1.8 JenNet-IP Browser (Examples)

Examples of the applications described in Section 1.7 are provided by NXP, as described below. The example applications are both called the JenNet-IP Browser and allow a user to interact with the WPAN nodes of a JenNet-IP system. Each application provides a generic engineering interface to the WPAN, allowing MIB variables on nodes to be inspected and/or edited.

1.8.1 Java Executable

A Java version of the JenNet-IP Browser is supplied with the JenNet-IP software as an executable that may be run on a LAN/WAN device with an IP connection to the Border-Router of a WPAN. This is the application that sits above the JenNet-IP LAN/WAN stack on the LAN/WAN device in Figure 2, Figure 3 and Figure 4. It represents an example of a test application that a developer may design using the Java JIP API (described in Part III: Java JenNet-IP API).

Note: The C JIP API (described in Chapter 3) can alternatively be used to develop a similar application for a Linux-based platform.

The JenNet-IP Browser functionality and the pre-requisites for using the application are detailed in Appendix A. Use of the application is fully descibed in an online manual which is provided within the application and is accessed from the Help menu of the interface.

1.8.2 Border-Router Firmware

A C-version of the JenNet-IP Browser is provided in the firmware of the Linksys and Buffalo routers used in JenNet-IP demonstration systems (and runs on the router). This is the optional application that sits above the JenNet-IP LAN/WAN stack in the Border-Router in Figure 2 and Figure 3, and was developed using the C JIP API (described in Part II: C JenNet-IP API). It is accessed from a normal web browser running on the LAN/WAN device. For further information on this application, see Appendix A. This application is also used as part of the set-up procedure of the JenNet-IP Smart Home demonstration which is described in the Application Note *JenNet-IP Smart Home (JN-AN-1162).*

Note: In addition to the above JenNet-IP Browser, the Border-Router firmware includes the JenNet-IP CLI (Command Line Interface) which allows access to JenNet-IP devices (such as WPAN nodes) from the command line on a LAN/WAN device. The JenNet-IP CLI is described in Appendix B.

2. Internet Protocol Concepts

This chapter introduces some of the key concepts related to the protocols used on the Internet and within some wired networks - the Internet protocols. It provides sufficient information to allow you to understand and configure the IP aspects of a JenNet-IP system.

Note: If you are already familiar with the Internet protocols IPv4 and IPv6, you may decide to skip this chapter.

Internet protocols are used by computers/servers on the worldwide web, and other communications networks, to transfer data between each other. The devices in an IP-based network are referred to as hosts. The transferred data can take many forms - pure data, documents/graphics, audio (VoIP), video, etc. The data is transported across an IP network from source to destination in a packet or datagram, which is a group of data bits comprising header information and payload data.

2.1 IP Data Packets

An IP data packet may need to pass through many IP hosts or networks to reach the final destination.

2.1.1 Connectionless Transport

IP is a connectionless protocol, which means that no circuit set-up is required before a packet is sent out - that is, there is no pre-determined path to reach the destination device. In contrast, the public telephone network is a circuit-switched network which requires a circuit to be established before a phone call can commence.

The destination device for an IP packet is represented by an IP address in the packet header. The network hosts have knowledge of the IP addresses of other devices and networks, and forward the packet to other hosts nearer to the destination device. Packets between particular source and destination devices may take different routes through the network(s) at different times, according to local conditions such as traffic loads and link failures.

2.1.2 Packet Delivery Reliability

IP is an unreliable service based on ‘best effort’ delivery. The network makes no guarantee about the proper arrival of packets, data corruption, out-of-order delivery, duplicate arrival, and lost or dropped/discarded packets. The packet header contains a checksum to ensure that the header is error-free. Received packets with corrupted headers are discarded immediately (but no notification is sent to the source node to indicate that a bad packet has been received).

2.2 IP Stack

The creation, transmission and reception of IP packets are handled by an IP software stack, which runs on each IP network device. The upper layers of the stack are closer to the user application, while the lower layers translate data into a form that can be handled by the physical transmission medium (for example, Ethernet).

The layers of the IP stack are illustrated in the figure below.

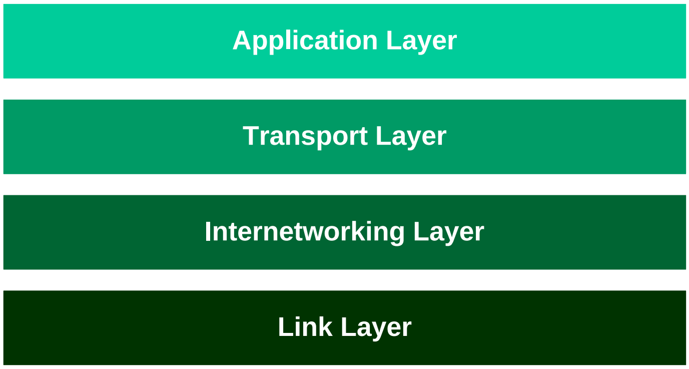

Figure 6: IP Software Stack

The above layers of the IP stack are outlined below.

Application Layer

The Application Layer contains user-defined programs as well as services commonly used by users, such as e-mail (SMTP) and terminal emulation (telnet). For example, in the case of a manufacturing plant, the user may develop an application for a device which monitors the state of a piece of equipment (e.g. vibration, temperature, etc). This application may send its sensor measurements to a central monitoring point using IP packets created in the Transport Layer of the IP stack (see below).

Transport Layer

The Transport Layer is responsible for delivering data and receiving data deliveries, as follows:

- Assembling IP packets that contain data to be sent to an application running on a remote device
- Disassembling IP packets received from a remote device and passing the extracted data to the relevant local application in the Application Layer

There may be multiple applications running on a device and an individual application is identified for IP communication purposes by means of a ‘socket’, which is a logical entity associated with the local IP address and port through which communications with the application will be conducted (also see Section 2.4). When sending an IP packet to a remote application, the Transport Layer inserts the source IP address, destination IP address and destination socket number into the packet header. When receiving an IP packet, the Transport Layer identifies the required destination application through the socket number extracted from the packet header.

Different Transport Layer protocols are available, including UDP and TCP:

- **UDP** (User Datagram Protocol) is a basic protocol offering connectionless delivery and the application multiplexing mechanism described above.
- **TCP** (Transmission Control Protocol) is a more sophisticated protocol which supports virtual circuits, allowing connection-oriented communication across a network that uses a packet-based transport mechanism.

TCP is used in many application areas, including web browsing and e-mail transfer. UDP typically gives higher throughput and shorter latency, and is therefore often used for real-time multi-media communication where occasional packet loss is acceptable (for example, IP-TV, IP-telephony, and online computer games). JenNet-IP uses UDP.

Internetworking Layer

The source and destination devices for the transfer of an IP packet may reside in geographically separate networks, which may also be of different media types. It is the role of the Internetworking Layer (or more commonly, Internet Layer) to support IP packet delivery to a different network.

To reach its destination, a packet may have to pass through several intermediate networks, being relayed along the way by routers. A router is a computer with software and hardware dedicated to the tasks of routing and forwarding information. A router can connect two or more networks with different physical interfaces (e.g. Ethernet, 802.11 WLAN, etc).

The Internet Protocol (IP) defines the addressing methods and structures for datagram encapsulation. The first major addressing structure was defined in Internet Protocol version 4 (IPv4), and this is still the dominant protocol of the Internet. A successor to IPv4, called IPv6, is rapidly being deployed and is likely to take over as the dominant packet transfer protocol. IPv6 is described in Section 2.3.

Link Layer

The Link Layer specifies how to send information between two points, the connection between the points being termed a link. A link can be considered as a single source-to-destination hop - that is, with no switching or routing in-between. The Link Layer consists of two sub-layers, the Media Access Control (MAC) sub-layer and the Physical (PHY) sub-layer:

- **MAC:** This is concerned with controlling access to the physical transmission medium to ensure that connected devices can transmit and receive information. The MAC enforces a set of rules which define when a particular device may transmit - this involve techniques such as token passing, as used on a token ring, or variations of Carrier Sense Multiple Access (CSMA), used in Ethernet and wireless systems.
- **PHY:** This describes the medium and modulation used to carry information (for example, Ethernet, IEEE 802.11 wireless or, in NXP's JenNet-IP system, IEEE 802.15.4 wireless).

One of the services that the Link Layer must perform is mapping between a device's IP address and its link layer address used by the PHY and MAC. The mapping function is dependent on the version of IP used (which defines the IP address format) and also the media type. For IPv4, the Address Resolution Protocol (ARP) is used, while for IPv6, the Neighbour Discovery Protocol (NDP) performs a similar function.

2.3 Internet Protocol version 6 (IPv6)

The Internet has so far predominantly used the Internet Protocol version 4 (IPv4). IPv4 uses 32-bit (4-byte) addresses, giving rise to an address space containing 2^{32} or nearly 4300 million unique IP addresses. Part of this address space is reserved for special purposes, such as private networks and multi-cast addresses, reducing the number of addresses available for public Internet use by approximately 34 million. However, the IPv4 address space is now effectively exhausted and therefore cannot support the future expansion of the Internet.

Internet Protocol version 6 (IPv6) has, so far, been introduced to a limited extent, and is destined to supersede IPv4 as a means of avoiding IP address exhaustion.

Some of the features of IPv6 are:

- Conservative extension of IPv4
- New packet format (IPv4 and IPv6 packet headers are significantly different and therefore not interoperable)
- 128-bit addresses, with an enormous increase in address space over that of 32-bit IPv4 addresses
- Little or no change needed to most Transport Layer and Application Layer protocols in order to operate over IPv6, the exceptions being protocols that embed network-layer addresses (such as FTP)
- Simplified address assignment and renumbering when changing Internet Service Providers (ISPs)

2.3.1 IPv6 Addresses

IPv6 uses 128-bit (16-byte) addresses, giving rise to an address space containing 2^{128} (or approximately 3.4 x 10^{38}) unique IP addresses - that is more than 3 hundred trillion trillion trillion addresses (using the American trillion of 10^{12}). It is unimaginable that this address space will ever be exhausted. The percentage utilisation of this address space is likely to remain extremely low, even with the rapid expansion of the Internet. This will allow a more systematic and hierarchical allocation of IP addresses, as well as more efficient routing.

IPv6 128-bit addresses are normally represented as eight groups of four hexadecimal digits, where each group is separated by a colon (:). For example:

```
2001:DB8F:756A:0000:0000:9B67:084C:6112
```

Any leading zeros in a group of four hex digits may be omitted. Continuing from the previous example:

```
2001:DB8F:756A:0:0:9B67:84C:6112
```

One or more consecutive groups of 0s can be replaced with a double-colon (::). For example:

```
2001:DB8F:756A::9B67:84C:6112
```

Substitution with a double-colon may be performed only once within an address, since multiple occurrences of the double-colon can be ambiguous. For example, writing the address 2001:DB84:0:385A:0:0:0:3A6D as 2001:DB84::385A::3A6D, the latter could represent any one of the following:

```
2001:DB84:0:0:0:385A:0:3A6D
2001:DB84:0:0:385A:0:0:3A6D
2001:DB84:0:385A:0:0:0:3A6D
```

2.3.2 IPv6 Address Components

A 128-bit IPv6 address consists of two 64-bit parts:

- **Address Prefix:** This comprises the 64 most significant bits of the address and identifies the network. This prefix will therefore be the same for all devices in a network. A number of special prefixes exist (e.g. the link-local prefix - see Section 2.3.4). The Address Prefix is itself subdivided into two parts:
    - **Site Prefix:** This comprises the 48 most significant bits of the Address Prefix and is allocated by an Internet Service Provider (ISP) or the Regional Internet Registry (RIR).
    - **Subnet ID:** This comprises the 16 least significant bits of the Address Prefix and, as the name suggests, identifies a particular subnet on the organisation’s site. It is assigned by the local IT administrator.
- **Host Interface ID:** This comprises the 64 least significant bits of the address and identifies a particular device in the network. It is normally taken to be the IEEE (MAC) address of the device (which is itself a universally unique identifier), with bit 57 inverted.

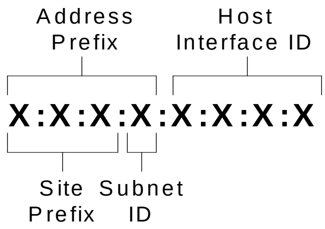

Figure 7: IPv6 Address Components

For full details of the IPv6 addressing scheme, refer to RFC 4291 available from the IETF (**www.ietf.org**).

2.3.3 IPv6 Address Blocks

IPv6 addresses are allocated in blocks, where a block contains a numerically contiguous set of addresses. A block of addresses will be allocated to an organisation’s network(s). Each block is aligned to a bit boundary of the IPv6 address space, so the size of a block must be a power of 2.

2.3.4 IPv6 Address Scopes

A block of IPv6 addresses has ‘scope’, which is a 'region' or 'span' in which the addresses are unique:

- **Link-Local Addresses:** These addresses are allocated to devices connected to the same logical link (for example, on the same Ethernet segment). For link-local addresses, the first ten bits of the Address Prefix are 1111111010. These addresses are not globally unique and should not be externally exposed.
- **Site Network Addresses:** These addresses are allocated to devices in a private network and are used for local unicast communications within the network. The Address Prefix is, in fact, globally unique and so the addresses can be exposed externally without the risk of address conflicts.
- **Global Network Addresses:** These addresses are allocated to devices in a network that can be accessed externally via the Internet. The Address Prefix is therefore globally unique.

A device may be allocated IPv6 addresses from more than one of the above scopes.

2.3.5 IPv6 Multicast Addresses

IPv6 supports multicast addresses which allow an IP packet to be targeted at multiple devices. An IPv6 multicast address is associated with a group of devices. Each device in a multicast group keeps a local record of the multicast address of the group (note that a device can be a member of more than one multicast group).

An IP packet containing a multicast address is actually broadcast by the source node on each of its links (to its neighbours). It is the responsibility of each receiving device to determine whether it is a member of the group corresponding to the multicast address in the packet (and therefore whether the packet should be accepted). If the receiving device is an IP Router, it will also need to pass on the packet, but is able to do this selectively. For each of its links, an IP Router maintains a list of the groups to which nodes on the link belong. It can therefore intelligently route a multicast packet down those links which contain nodes that belong to the relevant group. The extent of the broadcast is controlled using the ‘scope’ field in the multicast address (see below) - so, for example, the broadcast may be restricted to the devices within the local site.

All IPv6 multicast addresses contain the prefix FF00::/8. This leading byte is followed by a byte containing a 4-bit ‘Flags’ value and a 4-bit ‘Scope’ value (see Section 2.3.4 for an introduction to scope). The rest of the address identifies the multicast group.

8 bits	4 bits	4 bits	112 bits
Prefix (FF)	Flags	Scope	Group ID

Figure 8: IPv6 Multicast Address Components

For more information on the Flags and Scope values, refer to the description of the Groups module in the *JenNet-IP WPAN Stack User Guide (JN-UG-3080)*. Also refer to RFC 4291, available from the IETF (**www.ietf.org**).

Note: To avoid the propagation of a multicast IPv6 packet through the entire Internet, the scope in a multicast packet should normally be set to ‘site-local’ (0x5). Also, to send a multicast packet from a remote device on the Internet to a local site, the packet should normally be unicast to a ‘rendezvous point’ from where it can be broadcast through the site - the use of a rendezvous point can be enabled within the Flags value.

2.4 UDP Sockets

UDP ‘sockets’ streamline the transmission and reception of UDP packets, from the application’s viewpoint. A socket is a logical entity which is associated with a particular communications port on the local device. In fact, the socket is bound to a specific IPv6 address (the device may have more than one) and port.

Sockets are particularly useful when receiving UDP packets. The stack will automatically route a message arriving on a local communications port by passing the packet to the application via the relevant socket. Packets arriving on ports that are not bound to sockets are discarded. Thus, the application does not need to be concerned with the full set of local ports.

In JenNet-IP, the use of sockets is transparent to the application, as they are managed by the underlying software stack.

Part II:
C JenNet-IP API

3. IP Application Development (C Version)

The chapter outlines the coding of a C application that runs on a LAN/WAN device (IP Host) or on a Border-Router of a JenNet-IP system. This application may typically be used to remotely access a WPAN from a LAN/WAN device, such as a PC. References are made in this chapter to the C JenNet-IP API (or C JIP API) functions that you will need to use in your code.

Note: Details of all the API functions referenced in this chapter can be found in Chapter 4.

3.1 Overview

A LAN/WAN device (such as PC, tablet or mobile phone) can be used to interact with a WPAN of a JenNet-IP system. The device may be located either:

- remotely from the WPAN, as in the case of a lighting system controlled from a PC in an office in another town, or
- locally to the WPAN, as in the case of a lighting system controlled from a mobile phone within the same building

The application that facilitates this interaction may run on the LAN/WAN device or on the Border-Router connected to the WPAN:

- If implemented directly on the LAN/WAN device, the application provides a dedicated interface for accessing the WPAN
- If implemented on the Border-Router, the application runs on the LAN/WAN side - for example, the application may serve web pages to the LAN/WAN device, where they can be displayed in a normal web browser

The C JIP API can be used to develop both types of application, as described in this chapter.

Note: An application to be run on an IP Host device can alternatively be implemented using the Java JIP API (described in Part III: Java JenNet-IP API), if the device is not a Linux-based platform.

3.2 JIP Sessions

A communication link between the application and a WPAN is logically represented by a ‘JIP session’. If the application opens simultaneous communication links with multiple WPANs, multiple JIP sessions will exist concurrently. Within a JIP session, the application must create and maintain a ‘context data’ structure for the corresponding WPAN (see Section 5.1). This context data comprises information about the WPAN, details of the network nodes, and details of the MIBs and MIB variables that exist on the nodes.

When a JIP session is created, the corresponding context data structure is empty. The application must then connect to the Border-Router of the target WPAN and discover the details of the network (nodes, MIBs and MIB variables) in order to fill in the context data structure.

During the session, the application can then monitor and control the WPAN. Control is achieved by writing to MIB variables on the nodes. Monitoring can be achieved by reading the MIB variables or setting up JIP traps to provide automatic notifications when the variables change.

The function calls required to implement the above access, control and monitoring are indicated in the rest of this chapter:

- Initialising a JIP session is described in Section 3.3
- Connecting to a Border-Router is described in Section 3.4
- Discovering the WPAN attached to the Border-Router is described in Section 3.5
- Discovering the nodes of the WPAN, including their MIBs and MIB variables, is described in Section 3.6
- Monitoring the WPAN is described in Section 3.7
- Accessing MIB variables on wireless nodes is described in Section 3.8
- Protecting the context data structure for the WPAN is described in Section 3.9
- Persisting the context data for the WPAN is described in Section 3.10

3.3 Initialising a JIP Session

Application access to an individual WPAN is implemented in a JIP session for that network. A JIP session must first be initialised by calling the function **eJIP_Init()**. In this function call, a local `tsJIP_Context` structure (see Section 3.1) must be specified which will be used to store context data relating to the WPAN (the type of context data must also be specified to be for a client or server). This structure will also subsequently be used to identify the session and WPAN. The structure is initially empty and will be automatically populated during the discovery stages.

Note: A JIP session created using **eJIP_Init()** can be closed using **eJIP_Destroy()** when the session is no longer needed. This function will free the IP connection (see Section 3.4), remove any traps and free memory space associated with the session.

3.4 Connecting to a Border-Router (of a WPAN)

Once a JIP session has been initialised for a WPAN (see Section 3.3), a communication link must be established with the Border-Router of the network. This link can be an IPv6 connection or an IPv4 connection, depending on the IP version used by the intervening network. If the application runs on the LAN/WAN side of the Border-Router itself, an IPv6 connection with the WPAN side of the Border-Router must be set up (with the 'Border-Router node' of the WPAN, which may or may not be located in a physically separate device).

The functions used to establish these connections are as follows:

- **eJIP_Connect()** is used to set up an IPv6 connection (UDP)
- **eJIP_Connect4()** is used to set up an IPv4 connection (UDP or TCP)

Both of the above functions create a socket (associated with the specified IP address of the Border-Router and a local port) to be used for the communication link.

In both of the above functions, the IPv6 address of the WPAN side of the Border-Router (normally the Co-ordinator) must be specified since IPv6 addresses are always used to access nodes of the WPAN (see Note below). Even in the case of an IPv4 connection, the target node of an access is specified using its IPv6 address, which is inserted in the IPv4 packet along with the payload.

Note: For an IPv4 connection, if the IPv6 address of the WPAN side of the Border-Router is not known, an IPv6 address of zero can be specified which indicates that the true IPv6 address is to be auto-detected.

3.5 Discovering the WPAN

Once an IP connection has been established to the Border-Router of a WPAN (see Section 3.4), the application can initiate a 'discovery' of the network using the function **eJIPService_DiscoverNetwork()**. This function requests the details of the WPAN, including information on the constituent nodes, the MIBs that exist on each node and the variables of each MIB. On receiving this information, the function inserts it into the relevant context data structure for the WPAN, which will then contain a full description of the WPAN, its nodes and services.

Note: The contents of the context data structure for a WPAN can be output (via the standard output) using the function **eJIP_PrintNetworkContent()**.

3.6 Discovering Nodes and MIBs

Once a WPAN has been 'discovered' (see Section 3.5), information about the constituent nodes can be obtained locally from the populated context data structure for the network.

3.6.1 Node Information

The IPv6 addresses of all the nodes in the WPAN can be obtained from the context data structure using the function **eJIP_GetNodeAddressList()**. This function returns the number of nodes in the WPAN and a list of their IPv6 addresses. It is possible to use this function to produce a node list which is filtered according to Device ID. The function will allocate memory space (using **malloc()**) to receive the address list. Once the function has returned and the application has read the information from this list, the application should de-allocate this space (using **free()**) so that the space can be re-used.

The details of the node with a given IPv6 address can be obtained using the function **psJIP_LookupNode()**. This function returns a pointer to the relevant `tsNode` node structure (see Section 5.3) from the context data for the WPAN.

Note: The function **sJIP_LookupNode()** internally locks the node structure using the function **eJIP_LockNode()**. Once the application has finished with this node structure, it should unlock the structure using **eJIP_UnlockNode()**.

The returned information includes the Device ID of the node, the number of MIBs on the node and a pointer to a list of `tsMib` structures (see Section 5.4) containing details of the MIBs. Device ID is fully described in the *JenNet-IP WPAN Stack User Guide (JN-UG-3080)* and is also defined in the Glossary of Appendix C.

Searching the context data for specific MIBs and MIB variables on a node is described in Section 3.6.2.

3.6.2 MIB Information

Four functions are provided for investigating the MIBs on a node:

- **psJIP_LookupMib()** can be used to determine whether a MIB with a given name exists on a particular node. The function searches the relevant node information in the locally held context data for the WPAN. If a MIB with the required name is found, the function returns a pointer to the `tsMib` structure for the MIB. The function can be called again to resume the search from the next MIB in the list of MIBs on the node, in case there are multiple MIBs with the same name.
- **psJIP_LookupMibID()** can be used to determine whether a MIB with a given MIB ID exists on a particular node. The function searches the relevant node information in the locally held context data for the WPAN. If a MIB with the required ID is found, the function returns a pointer to the `tsMib` structure for the MIB. The function can be called again to resume the search from the next MIB in the list of MIBs on the node, in case there are multiple MIBs with the same ID.
- **psJIP_LookupVar()** can be used to determine whether a variable with a given name exists in a particular MIB on a node. The function searches the relevant node and MIB information in the locally held context data for the WPAN. If a MIB variable with the required name is found, the function returns a a pointer to the `tsVar` structure for the variable (see Section 5.5). The function can be called again to resume the search from the next variable in the list of MIB variables, in case there are multiple variables with the same name in the MIB.
- **psJIP_LookupVarIndex()** can be used to determine whether a variable with a given index value exists in a particular MIB on a node. The function searches the relevant node and MIB information in the locally held context data for the WPAN. If a MIB variable with the required index value is found, the function returns a a pointer to the `tsVar` structure for the variable (see Section 5.5).

Note: Functions are provided for remotely accessing MIBs and MIB variables on the nodes of a WPAN. This access is described in Section 3.8.

3.7 Monitoring the WPAN

A WPAN can be monitored for changes such as a node joining or leaving the network, or moving (to a new parent) within the network. This monitoring is initiated using the function **eJIPService_MonitorNetwork()** which launches a new 'network monitor' thread that will notify the application of changes in the network by invoking a user-defined callback function. The callback function prototype is detailed in the description of **eJIPService_MonitorNetwork()** in Section 4.2.

Note 1: The user-defined callback function is called within the context of the 'network monitor' thread.

Note 2: Once monitoring has been started as described above, it can subsequently be stopped using the function **eJIPService_MonitorNetworkStop()**.

3.8 Remotely Accessing MIBs

The MIB variables on a WPAN node may need to be remotely accessed from a LAN/WAN device in order to read from or write to the variables. In addition, JIP traps can be configured on a MIB variable in order to provide automatic notifications when the variable changes. These types of access are described in the sub-sections below.

3.8.1 Reading from MIB Variables

A request to read the current data from a particular MIB variable on a WPAN node can be submitted using the function **eJIP_GetVar()**. The target variable is specified using the `tsVar` structure (see Section 5.5) which corresponds to the MIB variable in the local context data structure. When a response is received containing the read data, the function will update this local `tsVar` structure with the new data.

3.8.2 Writing to MIB Variables

A request to write data to a particular MIB variable on a WPAN node can be submitted using the using the function **eJIP_SetVar()**. The target variable is specified using the `tsVar` structure (see Section 5.5) which corresponds to the MIB variable in the local context data structure. When a response is received, if the write was successful then the function will update the local `tsVar` structure with the new data.

Alternatively, the function **eJIP_MulticastSetVar()** can be used to send a write request to multiple nodes in order to update the same MIB variable on these nodes. The request will be received by all nodes in the WPAN but only implemented on nodes in the multicast group with the specified IPv6 multicast address. The target variable is identified in the function call by specifying the relevant `tsVar` structure for any node which belongs to the relevant multicast group. No responses are issued by the recipient nodes and the function returns immediately after sending the request. The locally held context data for the relevant WPAN is not updated with the new data for this variable and so this context data will become desynchronised with the data on the nodes (unless the application takes steps to maintain synchronisation).

Note: Neither of the above 'Set' functions can be used to write data to a MIB variable of the type 'table of blobs'.

3.8.3 Using JIP Traps on MIB Variables

The application on a LAN/WAN device or the Border-Router can set up a JIP trap on a MIB variable on a WPAN node. This results in the automatic generation of a notification to the application when the MIB variable on the node is changed in some way.

A JIP trap can be remotely configured on a MIB variable using the function **eJIP_TrapVar()**. The relevant variable is specified using the `tsVar` structure (see Section 5.5) which corresponds to the MIB variable in the local context data structure. A user-defined callback function must also be specified which will be invoked to handle a trap notification when it is generated. The callback function prototype is detailed in the description of **eJIP_TrapVar()** in Section 4.4. Each trap notification is handled in its own 'trap' thread, in which the callback function is called. During execution of the callback function, the relevant context data structure is automatically protected by a mutex (see Section 3.9).

Caution 1: *The application must be designed such that its own data structures are thread-safe within a 'trap' thread (see above).*

Caution 2: *A trap that has been configured on a MIB variable is not guaranteed to be generated when a change in the variable occurs.*

The trap configured on a MIB variable can be removed using the function **eJIP_UntrapVar()**.

Note: As an alternative to the above, the application on the device can use the function **eJIP_GroupJoin()** to enrol itself into a multicast group to which trap notifications from a remote MIB variable are sent (these traps are set up on the remote node). In this case, the received trap notifications are processed by the handler specified in the local `tsVar` structure for the variable.

3.9 Protecting Context Data

During a JIP session, there may be critical sections of code during which the context data for the WPAN must not be modified by any other thread. In order to protect the context data structure in these circumstances, a mutex can be applied to the structure using the function **eJIP_Lock()**. This will prevent the structure from being changed by other threads. The mutex can be removed using the function **eJIP_Unlock()**, allowing other threads to modify the structure.

The `tsNode` structure (within the context data structure) for a particular node can also be protected by a mutex to prevent other threads from modifying the structure. This mutex is applied using the function **eJIP_LockNode()** and removed using the function **eJIP_UnlockNode()**. Note that **eJIP_LockNode()** offers two possible outcomes when the structure cannot be locked (e.g. because it is already locked by another thread) - that is, either to return immediately without applying the mutex or to suspend the thread in which the function was called until the mutex can be applied.

3.10 Persisting Context Data

The context data which reflects the composition of a remote WPAN can be preserved in Non-Volatile Memory (NVM) so that it is still available following a break in execution of the application (e.g. due to a power outage or power cycle). For example, on a PC, the NVM used may be the hard disk.

For the purpose of persisting context data in this way, this data is treated in two parts:

- Network context data comprising basic information about the composition of the network, including the IPv6 addresses and Device IDs of the nodes in the network - see Section 3.10.1
- Node context data which defines the MIBs that reside on nodes with different Device IDs in the network - see Section 3.10.2

3.10.1 Network Context Data

Network context data can be saved to NVM at any time using the function **eJIPService_PersistXMLSaveNetwork()**. This function saves the IPv6 addresses and Device IDs of the nodes in the network (from the `tsJIP_Context` structure). The data is written to an XML file and the name (or full path) of this file must be specified.

The saved context data can be retrieved at any time using the function **eJIPService_PersistXMLLoadNetwork()**. This function reads the relevant XML file and inserts the read data into the context data structure held in RAM. Alternatively, in the future, the function may also be used to retrieve this network context data from an XML file held on a web server, for which a URL must be specified.

3.10.2 Node Context Data

Node context data can be saved to NVM at any time using the function **eJIPService_PersistXMLSaveDefinitions()**. This function saves information (from the specified `tsJIP_Context` structure) concerning the MIBs and associated variables that reside on nodes of each Device ID in the network. Note that the data stored in the MIB variables is not saved. The data is written to an XML file and the name (or full path) of this file must be specified.

The saved context data can be retrieved at any time using the function **eJIPService_PersistXMLLoadDefinitions()**. This function reads the relevant XML file and the read data is held internally to allow the subsequent rapid discovery of nodes. Alternatively, in the future, this function may also be used to retrieve this node context data from an XML file held on a web server, for which a URL must be specified.

Use of these functions to store and retrieve node context data avoids the need to rediscover the nodes of a network following a break in execution of the application.

4. C JIP API Functions

The C JIP API is used to develop applications which run on an IP host of a JenNet-IP system. This chapter details the functions of the API. The functions are defined in the header file **jip.h**.

The C JIP API functions are divided into the following categories:

- JIP management functions, detailed in Section 4.1
- Network discovery functions, detailed in Section 4.2
- Persistent data functions, detailed in Section 4.3
- MIB access functions, detailed in Section 4.4

4.1 JIP Management Functions

This section describes the JIP management functions that are used in setting up and managing a JIP session (in which a JenNet-IP WPAN will be ’discovered’ and interrogated).

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

Function	Page
eJIP_Init	54
eJIP_Connect	55
eJIP_Connect4	56
eJIP_Destroy	57
eJIP_Lock	58
eJIP_Unlock	59
eJIP_LockNode	60
eJIP_UnlockNode	61
eJIP_GroupJoin	62
eJIP_GroupLeave	63

eJIP_Init

```
teJIP_Status eJIP_Init(tsJIP_Context *psJipContext,
                       teJIP_ContextType eJIP_ContextType);
```

Description

This function initialises a JIP session and sets up a structure ready to receive network context data (during network discovery) - this structure is also used to identify the JIP session in other function calls. The type of context data must be specified to be for a client or server - normally, this should be set to ‘client’ (E_JIP_CONTEXT_CLIENT).

Parameters

psJipContext Pointer to structure to receive network context data (see Section 5.1)

eJIP_ContextType Type of context to initialise (client or server):
E_JIP_CONTEXT_CLIENT
E_JIP_CONTEXT_SERVER

Returns

E_JIP_OK

eJIP_Connect

```
teJIP_Status eJIP_Connect(
                        tsJIP_Context *psJipContext,
                        const char *pcAddress,
                        const int iPort);
```

Description

This function is used to establish an IPv6 connection to the JenNet-IP Border-Router. The IPv6 address of the WPAN side of the Border-Router must be specified. The function sets up a UDP socket (associated with the specified IPv6 address and port number) from which to issue requests to the WPAN connected to the Border-Router.

Note: This function must be used in an application which runs on a device with an IPv6 connection to the Border-Router or which runs within the Border-Router itself (on the LAN/WAN side).

Parameters

psJipContext* Pointer to network context data structure (set up using function **eJIP_Init())

*****_pcAddress_ Pointer to string representing IPv6 address of the WPAN side of the Border-Router to which connection will be made

iPort Port number for socket (usually JIP_DEFAULT_PORT)

Returns

E_JIP_OK

eJIP_Connect4

```
teJIP_Status eJIP_Connect4(
                            tsJIP_Context *psJipContext,
                            const char *pcIPv4Address,
                            const char *pcIPv6Address,
                            const int iPort,
                            const bool_t bTCP);
```

Description

This function is used to establish an IPv4 connection to the JenNet-IP Border-Router with the specified IPv4 address. The function sets up a socket (associated with the specified IPv4 address and port number) from which to issue requests to the WPAN connected to the Border-Router. The connection can be configured to use UDP or TCP packets. The IPv6 address of the WPAN side of the Border-Router must also be specified or auto-detected (the WPAN side normally acts as the network Co-ordinator).

> **Note:** In subsequent communications sent to the WPAN, the target node will be specified using its IPv6 address, which will be inserted in the IPv4 packet along with the payload.

Parameters

**psJipContext*	Pointer to network context data structure (set up using function **eJIP_Init()**)
**pcIPv4Address*	Pointer to string representing IPv4 address of Border-Router to which connection will be made
**pcIPv6Address*	Pointer to string representing IPv6 address of the WPAN side of the Border-Router (set to ::0 to auto-detect)
iPort	Port number for socket (usually JIP_DEFAULT_PORT)
bTCP	Protocol to be used for IPv4 connection (TCP or UDP): TRUE - TCP FALSE - UDP

Returns

E_JIP_OK

eJIP_Destroy

teJIP_Status eJIP_Destroy(tsJIP_Context ****psJipContext*);

Description

This function terminates a JIP session (initiated using **eJIP_Init()**), including closing the IP connection, untrapping MIB variables and freeing all memory related to this session.

Parameters

**sJipContext*	Pointer to structure containing network context data for the JIP session to be closed

Returns

E_JIP_OK

eJIP_Lock

```
teJIP_Status eJIP_Lock(tsJIP_Context *psJipContext);
```

Description

This function locks the JIP session context using a mutex which prevents other threads from changing the internal structures associated with the session.

If this function is called by the application (for example, to access the node list) then the context should be unlocked again when the mutex protection is no longer needed by the application. The function **eJIP_Unlock()** is used to unlock the context.

Parameters

psJipContext* Pointer to network context data structure for JIP session (set up using function **eJIP_Init())

Returns

E_JIP_OK

eJIP_Unlock

teJIP_Status eJIP_Unlock(tsJIP_Context ***psJipContext);**

Description

This function unlocks the JIP session context which has been previously locked for the current thread using the **eJIP_Lock()** function.

Refer to the description of **eJIP_Lock()** for more information on locking the JIP session context.

Parameters

psJipContext	Pointer to network context data structure for JIP session (set up using function **eJIP_Init()**)

Returns

E_JIP_OK

eJIP_LockNode

```
teJIP_Status eJIP_LockNode(tsNode *psNode,
                           bool_t bWait);
```

Description

This function locks the specified node structure (which is part of the JIP session context) using a mutex which prevents other threads from changing the structure.

If this function is called, the node structure should be unlocked again when the mutex protection is no longer needed. The function **eJIP_UnlockNode()** is used to unlock the structure.

The function provides the option to suspend the thread in which it is called until the lock can be acquired - in this case, the function will not return until it has the lock. Alternatively, the function can return immediately if the lock cannot be acquired.

Parameters

***psNode*	Pointer to node structure to be protected
bWait	Indicates whether thread should be suspended until the lock can be acquired: TRUE - Suspend thread until structure can be locked FALSE - Return immediately if structure cannot yet be locked

Returns

E_JIP_OK

E_JIP_ERROR_WOULD_BLOCK (if lock cannot be acquired and *bWait*=FALSE)

eJIP_UnlockNode

```
teJIP_Status eJIP_UnlockNode(tsNode *psNode);
```

Description

This function unlocks a node structure which has been previously locked for the current thread using the **eJIP_LockNode()** function.

Refer to the description of **eJIP_LockNode()** for more information on locking a node structure.

Parameters

psNode	Pointer to node structure to be unlocked

Returns

E_JIP_OK

eJIP_GroupJoin

```
teJIP_Status eJIP_GroupJoin(tsJIP_Context *psJIP_Context,
                            const char *pcAddress);
```

Description

This function allows the IP host to join the multicast group corresponding to the specified IPv6 multicast address, so that it can receive trap notifications from a node sending notifications to this group.

A pointer must also be provided to the JIP context structure that was set up in the call to **eJIP_Init()**, where the context type was set for a client.

The IP host can subsequently leave this group using **eJIP_GroupLeave()**.

Parameters

psJIP_Context Pointer to JIP context structure (context type must be E_JIP_CONTEXT_CLIENT)

pcAddress String representing IPv6 multicast address of the group to join

Returns

E_JIP_OK

eJIP_GroupLeave

teJIP_Status eJIP_GroupLeave(tsJIP_Context ***psJIP_Context*,**
const char ***pcAddress*);**

Description

This function allows the IP host to leave the multicast group corresponding to the specified IPv6 multicast address, so that it no longer receives trap notifications from a node sending notifications to this group.

A pointer must also be provided to the JIP context structure that was set up in the call to **eJIP_Init()**, where the context type was set for a client.

The IP host must have previously joined this group using **eJIP_GroupJoin()**.

Parameters

psJIP_Context	Pointer to JIP context structure (context type must be E_JIP_CONTEXT_CLIENT)
pcAddress	String representing IPv6 multicast address of the group to leave

Returns

E_JIP_OK

4.2 Network Discovery Functions

This section describes the network discovery functions that are used during a JIP session to find information about the JenNet-IP WPAN that is connected to a Border-Router (associated with the session).

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

eJIPService_DiscoverNetwork

teJIP_Status eJIPService_DiscoverNetwork(
tsJIP_Context ****psJipContext*);**

Description

This function is used to 'discover' the WPAN that is attached to the Border-Router associated with a JIP session (initiated using **eJIP_Init()** and identified by the specified structure). The function can be called only after an IP connection to the Border-Router has been established using either **eJIP_Connect()** or **eJIP_Connect4()**.

The function requests the details of the WPAN, including information on the constituent nodes, the MIBs that exist on each node and the variables of each MIB. On receiving this information, the function inserts it into the relevant `tsJIP_Context`, `tsNode`, `tsMib` and `tsVar` structures.

This is a blocking function and may take several seconds to return. Once it has completed, the `tsJIP_Context` structure contains a full description of the WPAN, its nodes and services.

Parameters

**psJipContext*	Pointer to structure to receive network context data for the discovered WPAN (see Section 5.1)

Returns

E_JIP_OK

eJIPService_MonitorNetwork

```
teJIP_Status eJIPService_MonitorNetwork(
                    tsJIP_Context *psJipContext,
                    tprCbNetworkChange prCbNetworkChange);
```

Description

This function is used to start monitoring the WPAN associated with a JIP session (initiated using **eJIP_Init()** and identified by the specified structure). Monitoring involves detecting and reporting a change such as a node joining or leaving the network, or moving (to a new parent) within the network.

The function will spawn a new thread, the 'network monitor' thread, which will notify the application of changes in the network by invoking the specified user-defined callback function (the callback function is called in this thread's context). The application must not perform blocking operations in this thread.

The prototype of the callback function is as follows:

```
typedef void(*tprCbNetworkChange)(teJIP_NetworkChangeEvent eEvent,
                                  struct _tsNode *psNode);
```

where *eEvent* indicates the nature of the change and *psNode* points to the structure (within the JIP context data) for the node on which the change has taken place. Note that this node structure is locked to the network context data structure for the WPAN.

Parameters

**psJipContext*	Pointer to structure containing network context data for WPAN to be monitored (see Section 5.1)
**prCbNetworkChange*	Pointer to a user-defined callback function to generate change notifications to the application

Returns

E_JIP_OK

eJIPService_MonitorNetworkStop

```
teJIP_Status eJIPService_MonitorNetworkStop(
                                tsJIP_Context *psJipContext);
```

Description

This function is used to stop monitoring of the specified WPAN (where the monitoring was previously started using **eJIPService_MonitorNetwork()**).

The function will destroy the associated ‘network monitor’ thread.

Parameters

**psJipContext*	Pointer to structure containing network context data for WPAN for which monitoring is to be stopped (see Section 5.1)

Returns

E_JIP_OK

eJIP_GetNodeAddressList

```
teJIP_Status eJIP_GetNodeAddressList(
                              tsJIP_Context *psJipContext,
                              const uint32_t u32DeviceIdFilter,
                              tsJIPAddress **ppsAddresses,
                              uint32_t *pu32NumAddresses);
```

Description

This function can be used to obtain a list of the IPv6 addresses of the nodes in the WPAN associated with a JIP session (identified by the specified context data structure). The function can only be called once the network has been discovered (using the function **eJIPService_DiscoverNetwork()**). The addresses are obtained from the context data structure.

The obtained node list can be filtered according to a specified Device ID - for example, so that it contains only nodes that are lamps. Filtering can be disabled (so that all nodes are listed) using the enumeration JIP_DEVICE_ID_ALL. Device ID is descibed in the *JenNet-IP WPAN Stack User Guide (JN-UG-3080)*.

A pointer must be provided to a location that will receive a pointer to the memory space that will receive the list. The function will 'malloc' the required memory space (according the the number of nodes in the network). Once the function has returned, the application should read the information from the list and then free the corresponding memory space (using **free()**) so that this space can be re-used.

The obtained node list is only a snapshot of the current state of the WPAN. The function may be called again after the network has been re-discovered or monitoring has indicated a change in the network.

Parameters

**psJipContext*	Pointer to structure network context data structure for the WPAN (see Section 5.1)
u32DeviceIdFilter	Device ID with which to filter list of nodes or JIP_DEVICE_ID_ALL to indicate no filtering
***ppsAddresses*	Pointer to a location to receive a pointer to the obtained list of the IPv6 addresses
***pu32NumAddresses*	Pointer to a location to receive number of IPv6 addresses in the list

Returns

E_JIP_OK

psJIP_LookupNode

```
tsNode* psJIP_LookupNode(
                        tsJIP_Context *psJipContext,
                        tsJIPAddress *psAddress);
```

Description

This function can be used to obtain a pointer to information about the node with the specified IPv6 address (if the node exists in the specified WPAN). The function searches the node list which is embedded in the specified network context data structure and returns a pointer to the relevant node structure.

The function must only be called after a network discovery has been performed using the function **eJIPService_DiscoverNetwork()**.

psJIP_LookupNode() internally locks the relevant node structure using the function **eJIP_LockNode()**. Once the application has finished with the returned node structure, it should unlock the structure using **eJIP_UnlockNode()**.

Parameters

**psJipContext*	Pointer to structure network context data structure for the WPAN (see Section 5.1)
******psAddress*	Pointer to structure containing the IPv6 address of the node of interest

Returns

Pointer to `tsNode` structure containing the requested node information (see Section 5.3) - if the specified node cannot be found, a null pointer is returned.

psJIP_LookupMib

```
tsMib* psJIP_LookupMib(tsNode *psNode,
                       tsMib *psStartMib,
                       const char *pcName);
```

Description

This function can be used to determine whether the specified node of a WPAN has a MIB with the specified name. The function searches the node's MIB information held locally in the network context data structure for the WPAN and returns a pointer to any matching MIB.

The search can be configured to start at any MIB in the MIB list. Once the function has returned a MIB, it can be called again to resume the search from the next MIB in the list. In this way, the function can be used to return multiple MIBs with the same name.

Before calling this function, the application thread must lock the relevant node structure (within the network context data structure) using the function **eJIP_LockNode()** or **psJIP_LookupNode()**. Once the application has finished with the node structure, it should unlock the structure using **eJIP_UnlockNode()**.

Parameters

******psNode* Pointer to `tsNode` structure for the node of interest

******psStartMib* Pointer to `tsMib` structure for the MIB at which the search is to start - a null pointer means to start at the first MIB listed

******pcName* Pointer to character string representing the name of the MIB to search for

Returns

Pointer to a MIB with the given name - if there is no MIB with the name, a null pointer is returned

psJIP_LookupMibId

```
tsMib *psJIP_LookupMibId(tsNode *psNode,
                         tsMib *psStartMib,
                         uint32_t u32MibId);
```

Description

This function can be used to determine whether the specified node of a WPAN has a MIB with the specified MIB ID. The function searches the node's MIB information held locally in the network context data structure for the WPAN and returns a pointer to any matching MIB.

The search can be configured to start at any MIB in the MIB list. Once the function has returned a MIB, it can be called again to resume the search from the next MIB in the list. In this way, the function can be used to return multiple MIBs with the same ID.

Before calling this function, the application thread must lock the relevant node structure (within the network context data structure) using the function **eJIP_LockNode()** or **psJIP_LookupNode()**. Once the application has finished with the node structure, it should unlock the structure using **eJIP_UnlockNode()**.

Parameters

psNode	Pointer to `tsNode` structure for the node of interest
psStartMib	Pointer to `tsMib` structure for the MIB at which the search is to start - a null pointer means to start at the first MIB listed
u32MibId	MIB ID to search for

Returns

Pointer to the MIB with the given ID - if there is no MIB with the ID, a null pointer is returned

psJIP_LookupVar

```
tsVar* psJIP_LookupVar(tsMib *psMib,
                       tsVar *psStartVar,
                       const char *pcName);
```

Description

This function can be used to determine whether the specified MIB has a variable with the specified name. The function searches the relevant MIB information held locally in the network context data structure for the WPAN and returns a pointer to any matching variable.

The search can be configured to start at any variable in the list of variables for the MIB. Once the function has returned a variable, it can be called again to resume the search from the next variable in the list. In this way, the function can be used to return multiple variables with the same name.

Before calling this function, the application thread must lock the relevant node structure (within the network context data structure) using the function **eJIP_LockNode()** or **psJIP_LookupNode()**. Once the application has finished with the node structure, it should unlock the structure using **eJIP_UnlockNode()**.

Parameters

psMib	Pointer to `tsMib` structure for the MIB of interest
psStartVar	Pointer to `tsVar` structure for the variable at which the search is to start - a null pointer means to start at the first variable listed for the MIB
pcName	Pointer to character string representing the name of the variable to search for

Returns

Pointer to a variable with the given name - if there is no variable with the name, a null pointer will be returned

psJIP_LookupVarIndex

```
tsVar *psJIP_LookupVarIndex(tsMib *psMib,
                            uint8_t u8Index);
```

Description

This function can be used to determine whether the specified MIB has a variable with the specified index value. The function searches the relevant MIB information held locally in the network context data structure for the WPAN and returns a pointer to any matching variable.

Before calling this function, the application thread must lock the relevant node structure (within the network context data structure) using the function **eJIP_LockNode()** or **psJIP_LookupNode()**. Once the application has finished with the node structure, it should unlock the structure using **eJIP_UnlockNode()**.

Parameters

***psMib*	Pointer to `tsMib` structure for the MIB of interest
u8Index	Index value to search for

Returns

Pointer to a variable with the given index value - if there is no variable with this index value, a null pointer will be returned

eJIP_PrintNetworkContent

```
teJIP_Status eJIP_PrintNetworkContent(
                        tsJIP_Context *psJipContext);
```

Description

This function can be used to print the network context data for a WPAN from the specified `tsJIP_Context` structure. The function dumps the list of nodes, their MIBs, variables and variable values through the standard output.

The function must only be called after a network discovery has been performed using the function **eJIPService_DiscoverNetwork()**.

Parameters

**psJipContext*	Pointer to structure network context data structure to print (see Section 5.1)

Returns

E_JIP_OK

4.3 Persistent Data Functions

This section describes the persistent data functions that are used during a JIP session to save network context data to Non-Volatile Memory (NVM) and retrieve this data from NVM (or, in the future, from a web server).

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

Function	Page
eJIPService_PersistXMLSaveNetwork	76
eJIPService_PersistXMLLoadNetwork	77
eJIPService_PersistXMLSaveDefinitions	78
eJIPService_PersistXMLLoadDefinitions	79

eJIPService_PersistXMLSaveNetwork

```
teJIP_Status eJIPService_PersistXMLSaveNetwork(
                        tsJIP_Context *psJipContext,
                        const char *pcFileName);
```

Description

This function can be used to save network context data to an XML file in local Non-Volatile Memory (NVM). This data describes the make-up of the network by including the IPv6 addresses and Device IDs of all the nodes in the network (other data relating to the nodes, MIBs and MIB variables can be saved using the function **eJIPService_PersistXMLSaveDefinitions()**).

eJIPService_PersistXMLSaveNetwork() can be called just before the end of a JIP session to preserve context data for future use.

The saved context data can later be retrieved from NVM using the function **eJIPService_PersistXMLLoadNetwork()**.

Parameters

psJipContext	Pointer to network context data structure (see Section 5.1)
pcFileName	Pointer to string containing name (or path) of XML file

Returns

E_JIP_OK

eJIPService_PersistXMLLoadNetwork

```
teJIP_Status eJIPService_PersistXMLLoadNetwork(
                        tsJIP_Context *psJipContext,
                        const char *pcFileName);
```

Description

This function can be used to load into memory the contents of a network context XML file, which may have been previously saved to local NVM using the function **eJIPService_PersistXMLSaveNetwork()**. Alternatively, in the future, this file may reside on a web server (for which a URL must be provided).

The retrieved context data is inserted into the `tsJIP_Context` structure.

The function can be called after a JIP session has been opened using **eJIP_Init()** if context data has been saved from a previous session for the relevant WPAN. In this case, the function should be called only after the node context data has been loaded using **eJIPService_PersistXMLLoadDefinitions()**.

Parameters

psJipContext	Pointer to network context data structure (see Section 5.1)
***pcFileName*	Pointer to string containing name (or path) of XML file

Returns

E_JIP_OK

eJIPService_PersistXMLSaveDefinitions

```
teJIP_Status eJIPService_PersistXMLSaveDefinitions(
                          tsJIP_Context *psJipContext,
                          const char *pcFileName);
```

Description

This function can be used to save node context data to an XML file in local Non-Volatile Memory (NVM). This data defines the Device IDs of the nodes of a network, including the MIBs (and MIB variables) that reside on the nodes with each Device ID.

Note that only the definitions are saved, the MIB variable values are not saved.

The function can be called just before the end of a JIP session to preserve context data for future use.

The saved context data can later be retrieved from NVM using the function **eJIPService_PersistXMLLoadDefinitions()**.

Parameters

psJipContext	Pointer to network context data structure (see Section 5.1)
***pcFileName*	Pointer to string containing name (or path) of XML file

Returns

E_JIP_OK

eJIPService_PersistXMLLoadDefinitions

```
teJIP_Status eJIPService_PersistXMLLoadDefinitions(
                        tsJIP_Context *psJipContext,
                        const char *pcFileName,
                        const int iPopulateNetwork);
```

Description

This function can used to load into memory the contents of a node context XML file, which may have been previously saved to NVM using the function **eJIPService_PersistXMLSaveDefinitions()**. Alternatively, in the future, this file may reside on a web server (for which a URL must be provided).

The retrieved context data is held internally to allow the rapid discovery of nodes.

The function can be called after a JIP session has been opened using **eJIP_Init()** if context data has been saved from a previous session for the relevant WPAN. In this case, the function should be called before the network context data is loaded using **eJIPService_PersistXMLLoadNetwork()** or the network is discovered using **eJIPService_DiscoverNetwork()**.

Parameters

psJipContext	Pointer to network context data structure (see Section 5.1)
pcFileName	Pointer to string containing name (or path) of XML file

Returns

E_JIP_OK

4.4 MIB Access Functions

This section describes the functions that are used to access MIBs and their variables, including the use of JIP traps to monitor MIB variables.

The MIB access functions are listed below, along with its page reference:

eJIP_GetVar

teJIP_Status eJIP_GetVar(tsJIP_Context *psJipContext*,
tsVar *psVar***);**

Description

This function can be used to send a request to read the specified MIB variable on a node of a WPAN. The variable is specified using a pointer to a local `tsVar` structure (embedded in the `tsJIP_Context` structure for the WPAN). The specified `tsVar` structure is unique to the relevant node and MIB on the node.

The supplied pointer to the relevant `tsVar` structure could have been obtained using the function **psJIP_LookupVar()**.

The function is blocking and will not return until a response to the request has been received or a timeout has occurred. If the request was successful and data is returned, the `pvData` pointer of the local `tsVar` structure will be automatically set to point to the obtained data.

Parameters

*psJipContext — Pointer to network context data structure (see Section 5.1)

*psVar — Pointer to local `tsVar` structure corresponding to MIB variable to be read (this pointer also identifies the node and the MIB) (see Section 5.5)

Returns

E_JIP_OK

E_JIP_ERROR_TIMEOUT

E_JIP_ERROR_BAD_MIB_INDEX

E_JIP_ERROR_BAD_VAR_INDEX

E_JIP_ERROR_NO_ACCESS

E_JIP_ERROR_BAD_BUFFER_SIZE

E_JIP_ERROR_WRONG_TYPE

E_JIP_ERROR_DISABLED

E_JIP_ERROR_FAILED

eJIP_SetVar

```
teJIP_Status eJIP_SetVar(tsJIP_Context *psJipContext,
                         tsVar *psVar,
                         void *pvNewData,
                         uint32_t u32Size);
```

Description

This function can be used to send a request to set the specified MIB variable on a node of a WPAN. The relevant variable is specified using a pointer to a local `tsVar` structure (embedded in the `tsJIP_Context` structure for the WPAN). The specified `tsVar` structure is unique to the relevant node and MIB on the node.

The supplied pointer to the relevant `tsVar` structure could have been obtained using the function **psJIP_LookupVar()**.

The function is blocking and will not return until a response to the request has been received or a timeout has occurred. If the request was successful and the variable was set, the `pvData` pointer of the local `tsVar` structure will be automatically set to point to the set data.

Note that it is not possible to use this function to set variables of the 'table of blobs' data type.

Parameters

**psJipContext*	Pointer to network context data structure (see Section 5.1)
***psVar*	Pointer to local `tsVar` structure corresponding to MIB variable to be set (this pointer also identifies the node and the MIB) (see Section 5.5)
***pvNewData*	Pointer to data to be assigned to the variable
u32Size	Data size for variable, in bytes

Returns

E_JIP_OK

E_JIP_ERROR_TIMEOUT

E_JIP_ERROR_BAD_VALUE

E_JIP_ERROR_BAD_MIB_INDEX

E_JIP_ERROR_BAD_VAR_INDEX

E_JIP_ERROR_NO_ACCESS

E_JIP_ERROR_BAD_BUFFER_SIZE

E_JIP_ERROR_WRONG_TYPE

E_JIP_ERROR_DISABLED

E_JIP_ERROR_FAILED

eJIP_MulticastSetVar

```
teJIP_Status eJIP_MulticastSetVar(
                                  tsJIP_Context *psJipContext,
                                  tsVar *psVar,
                                  void *pvNewData,
                                  uint32_t u32Size,
                                  tsJIPAddress *psAddress,
                                  int iMaxHops);
```

Description

This function can be used to send a request to set the specified MIB variable on multiple nodes of a WPAN. The target nodes are members of the multicast group corresponding to the specified IPv6 multicast address. The relevant variable is specified using a pointer to a local `tsVar` structure (embedded in the `tsJIP_Context` structure for the WPAN). The particular `tsVar` structure specified can correspond to any node which contains the relevant MIB and variable.

The use of an IPv6 multicast address in this type of update means that the maximum number of IP hops to the WPAN must be specified. This is a standard IPv6 socket parameter - it must be set to a minimum value of 2 in order to traverse the Border-Router and to not more than the maximum legal value of 255.

The network interface used to send this request is specified through the element `iMulticastInterface` of the supplied `tsJIP_Context` structure (the default interface is 'tun0'). Changing this value for each call to the function allows an application to multicast on multiple interfaces. The element `iMulticastSendCount` of the same structure specifies the number of requests that will be sent for each call of this function (the default value is 2).

On receiving the request, a node will carry out the request only if the node is a member of the group corresponding to the specified multicast address (and contains the relevant MIB and variable).

The function is non-blocking, since no responses are received from the target nodes of a multicast.

⚠ ***Caution:** When using this function to update a MIB variable on remote nodes, the locally held context data for the relevant WPAN is not updated with the new data. Therefore, this context data will become desynchronised with the data on the nodes, unless the application is designed to maintain synchronisation in some way (e.g. using **eJIP_GetVar()**).*

Note that it is not possible to use this function to set variables of the 'table of blobs' data type.

Parameters

*psJipContext	Pointer to network context data structure (see Section 5.1)
*psVar	Pointer to any local `tsVar` structure corresponding to MIB variable to be set (see Section 5.5)
*pvNewData	Pointer to data to be assigned to the variable
u32Size	Data size for variable, in bytes
*psAddress	Pointer to structure containing IPv6 multicast address and relevant port number for target nodes
iMaxHops	Maximum number of hops to WPAN (must be at least 2 and must not exceed 255)

Returns

E_JIP_OK

eJIP_TrapVar

```
teJIP_Status eJIP_TrapVar(
                          tsJIP_Context *psJipContext,
                          tsVar *psVar,
                          uint8_t u8NotificationHandle,
                          tprCbVarTrap prCbVarTrap);
```

Description

This function is used to request a MIB variable (on a specific remote node) to be trapped - that is, for the variable to be monitored and a notification of any change in the variable to be sent to the local application. The relevant variable is specified using a pointer to a local `tsVar` structure (embedded in the `tsJIP_Context` structure for the WPAN). The specified `tsVar` structure is unique to the relevant node and MIB on the node.

A handle must be specified which will be subsequently used to refer to any trap notification for the variable. A user-defined callback function must also be provided which will be used to deal with the trap notification. The prototype for this callback function is as follows:

typedef void(*tprCbVarTrap)(struct _tsVar ****psVar*);**

where *psVar* is a pointer to the relevant variable.

The callback function is called in the context of a ‘trap’ thread. The application must ensure that this callback function is thread-safe from the main application thread. Every trap notification is handled in its own thread context. When the callback function is invoked, the relevant node structure (incorporating the MIB and variable) is automatically locked with **eJIP_NodeLock()** and unlocked on completion with **eJIP_NodeUnlock()**.

> ***Caution:*** *A trap set up using this function is not guaranteed to be generated when a change in the variable occurs.*

A MIB variable that has been trapped using this function can be untrapped using the function **eJIP_UntrapVar()**.

Parameters

**psJipContext*	Pointer to network context data structure (see Section 5.1)
***psVar*	Pointer to local `tsVar` structure corresponding to MIB variable to be trapped (this pointer also identifies the node and the MIB) (see Section 5.5)
u8NotificationHandle	Handle to be used to refer to trap notifications for this variable
prCbVarTrap	Pointer to callback function to be invoked when a trap notification for this variable is received

Returns

E_JIP_OK

eJIP_UntrapVar

```
teJIP_Status eJIP_UntrapVar(
                         tsJIP_Context *psJipContext,
                         tsVar *psVar,
                         uint8_t u8NotificationHandle);
```

Description

This function is used to request a MIB variable (on a specific remote node) to be untrapped - that is, for a previously configured trapping of the variable set up using **eJIP_TrapVar()** to be disabled. The relevant variable is specified using a pointer to a local `tsVar` structure (embedded in the `tsJIP_Context` structure for the WPAN). The specified `tsVar` structure is unique to the relevant node and MIB on the node.

The handle used to refer to trap notifications for the variable must also be specified.

Parameters

**psJipContext* Pointer to network context data structure (see Section 5.1)

******psVar* Pointer to local `tsVar` structure corresponding to MIB variable to be untrapped (this pointer also identifies the node and the MIB) (see Section 5.5)

u8NotificationHandle Handle used to refer to trap notifications for this variable

Returns

E_JIP_OK

5. C JIP API Structures

This chapter details the structures representing datatypes used by the C JIP API, which is used to develop applications which run on an IP host of a JenNet-IP system. The structures are defined in the header file **jip.h**.

5.1 tsJIP_Context

This structure contains all the JIP context data for a 'discovered' WPAN as well as internal data used by JIP. It is at the highest level in the hierarchy of structures for a WPAN and acts as the top-level handle (or the 'owner') of the WPAN.

```
typedef struct _tsJIP_Context
{
   tsNetwork  sNetwork;
   void      *pvPriv;
   /* User configurable parameters */
   int        iMulticastInterface;
   int        iMulticastSendCount;
} tsJIP_Context;
```

where:

- sNetwork is a structure (see Section 5.2) containing the JIP context data for the WPAN
- pvPriv is a pointer to internal private data used by JIP
- iMulticastInterface is the index of the network interface from which to send multicast 'set variable' requests using **eJIP_MulticastSetVar()**. Changing this value for each call to the above function allows an application to multicast on multiple interfaces. The default value is 0, corresponding to 'tun0'.
- iMulticastSendCount is the number of times to send each multicast 'set variable' request following a call to **eJIP_MulticastSetVar()**. The default value is 2.

5.2 tsNetwork

This structure contains the JIP context data for a 'discovered' WPAN.

```
typedef struct _tsNetwork
{
    uint32_t               u32NumNodes;
    struct _tsJIP_Context *psOwnerContext;
    tsNode                *psNodes;
} tsNetwork;
```

where:

- u32NumNodes is the number of nodes in the WPAN
- psOwnerContext is a pointer to the top-level context structure (see Section 5.1) for the relevant WPAN (thus identifying the WPAN)
- psNodes is a pointer to a linked list of tsNode structures (see Section 5.3), each containing the context data for a node of the WPAN

5.3 tsNode

This structure contains node-specific JIP context data.

```
typedef struct _tsNode
{
    tsJIPAddress         sNode_Address;
    uint32_t             u32DeviceId;
    uint32_t             u32NumMibs;
    tsMib               *psMibs;
    tsLock               sLock;
    struct _tsNetwork   *psOwnerNetwork;
    struct _tsNode      *psNext;
} tsNode;
```

where:

- sNode_Address is a tsJIPAddress structure (typedef **sockaddr_in6**) representing the JIP address of the node (containing the IPv6 address of the node and port number of the JIP service on the node)
- u32DeviceId is the 32-bit Device ID for the node and is described in the *JenNet-IP WPAN Stack User Guide (JN-UG-3080)*
- u32NumMibs is the number of MIBs on the node
- psMibs is a pointer to a linked list of tsMib structures (see Section 5.4), each containing information about a MIB on the node
- sLock is a structure containing a protective mutex for the node
- psOwnerNetwork is a pointer to the structure (see Section 5.2) containing the context data for the WPAN to which the node belongs (thus identifying the host WPAN)
- psNext is a pointer to the structure for the next node in the linked list

5.4 tsMib

This structure contains information about a MIB on a node.

```
typedef struct _tsMib
{
    char                *pcName;
    uint32_t             u32MibId;
    uint8_t              u8Index;
    uint32_t             u32NumVars;
    tsVar               *psVars;
    struct _tsNode      *psOwnerNode;
    struct _tsMib       *psNext;
} tsMib;
```

where:

- `pcName` is a character string representing the name of the MIB
- `u32MibId` is the identifier of the MIB type
- `u8Index` is the index of the MIB
- `u32NumVars` is the number of variables in the MIB
- `psVars` is a pointer to a linked list of `tsVar` structures (see Section 5.5), each containing information on a variable of the MIB
- `psOwnerNode` is a pointer to the structure (see Section 5.3) for the node to which the MIB belongs (thus identifying the host node and WPAN)
- `psNext` is a pointer to the next MIB in the linked list

5.5 tsVar

This structure contains information about a MIB variable.

```
typedef struct _tsVar
{
   char*              pcName;
   uint8_t            u8Index;
   uint8_t            u8Size;
   teJIP_VarEnable    eEnable;
   teJIP_VarType      eVarType;
   teJIP_AccessType   eAccessType;
   teJIP_Security     eSecurity;
   union
   {
     int8_t*     pi8Data;
     int16_t*    pi16Data;
```

```
            int32_t*   pi32Data;
            int64_t*   pi64Data;
            uint8_t*   pu8Data;
            uint16_t*  pu16Data;
            uint32_t*  pu32Data;
            uint64_t*  pu64Data;
            float*     pfData;
            double*    pdData;
            char*      cData;
            uint8_t*   pbData;
            tsTable*   ptData;
            void*      pvData;
        };
        tprCbVarGet        prCbVarGet;
        tprCbVarSet        prCbVarSet;
        uint8_t            u8TrapHandle;
        tprCbVarTrap       prCbVarTrap;
        struct _tsMib*     psOwnerMib;
        struct _tsVar*     psNext;
    } tsVar;
```

where:

- `pcName` is a character string representing the name of the variable
- `u8Index` is the index of the variable within its MIB
- `u8Size` is the number of bytes in the blob of data (only used for a variable of the blob datatype)
- `eEnable` is used to define whether the variable is enabled or disabled:
  - E_JIP_VAR_DISABLED
  - E_JIP_VAR_ENABLED
- `eVarType` is an enumeration (see Section 5.8) indicating the variable type
- `eAccessType` is an enumeration (Section 5.9) indicating the type of access allowed to the variable (read or write)
- `eSecurity` is an enumeration (see Section 5.10) indicating the type of security applied to the variable
- The union contains pointers for all data types to the data held by the variable - this is a null pointer until the variable is read using the function **eJIP_GetVar()**
- `pvData` is a pointer to the data held by the variable (the pointer should be cast to the correct type, dependent upon `eVarType`) - this is a null pointer until the variable is read using the function **eJIP_GetVar()**
- `prCbVarGet` is a pointer to the user-defined callback function that will be invoked when data is returned as the result of a ‘get variable’ operation. The callback function must set the appropriate data pointer in the above union to

point to the received data. This field can be set to NULL (no callback function) if the application will set this data pointer

- `prCbVarSet` is a pointer to the user-defined callback function that will be invoked as the result of a 'set variable' operation. This field can be set to NULL (no callback function) if the application does not need to be notified of the results of the operation
- `u8TrapHandle` is a handle associated with the trap for this variable
- `prCbVarTrap` is a pointer to the registered user-defined callback function which is used to deal with trap notifications for the variable - the callback function is registered through the function **eJIP_TrapVar()**
- `psOwnerMib` is a pointer to the structure (see Section 5.4) for the MIB to which the variable belongs (thus identifying the host MIB, node and WPAN)
- `psNext` is a pointer to the next variable in the linked list of variables for the MIB

5.6 tsTable

This structure is used to contain a table which is a MIB variable. In this case, the `pvData` field of the `tsVar` structure (see Section 5.5) points to this table structure.

```
typedef struct
{
    uint32_t     u32NumRows;
    tsTableRow  *psRows[];
} tsTable;
```

where:

- `u32NumRows` is the total number of rows in the table
- `psRows[]` is a pointer to an array of pointers to structures, where each structure corresponds to a row of the table (see Section 5.7) - note that an empty row has a NULL row pointer

5.7 tsTableRow

This structure contains information which allows access to a row of a table which is a MIB variable. The structure is pointed to by the psRows field of the tsTable structure (see Section 5.6).

```
typedef struct _tsTableRow
{
    uint32_t  u32Length;
    void     *pvData;
} tsTableRow;
```

where:

- u32Length is the length of a row (in bytes)
- pvData is a pointer to the data of the row of the table

5.8 teJIP_VarType

The following enumerated list contains the possible types for MIB variables.

```
typedef enum
{
    E_JIP_VAR_TYPE_INT8,
    E_JIP_VAR_TYPE_INT16,
    E_JIP_VAR_TYPE_INT32,
    E_JIP_VAR_TYPE_INT64,
    E_JIP_VAR_TYPE_UINT8,
    E_JIP_VAR_TYPE_UINT16,
    E_JIP_VAR_TYPE_UINT32,
    E_JIP_VAR_TYPE_UINT64,
    E_JIP_VAR_TYPE_FLOAT,
    E_JIP_VAR_TYPE_DOUBLE,
    E_JIP_VAR_TYPE_STRING,
    E_JIP_VAR_TYPE_BLOB,
    E_JIP_VAR_TYPE_TABLE_BLOB
} PACK teJIP_VarType;
```

5.8.1 teJIP_VarEnable

The following enumerations are used to define whether a MIB variable is enabled or disabled.

```
typedef enum _eJIP_VarEnable
{
   E_JIP_VAR_DISABLED,
   E_JIP_VAR_ENABLED,
} PACK teJIP_VarEnable;
```

The above enumerations are detailed in the table below.

Enumeration	Description
E_JIP_VAR_DISABLED	Variable is disabled
E_JIP_VAR_ENABLED	Variable is enabled

5.8.2 teJIP_ContextType

The following enumerations are used to indicate the type of context data.

```
typedef enum _eJIP_Context
{
   E_JIP_CONTEXT_CLIENT,
   E_JIP_CONTEXT_SERVER,
} PACK teJIP_ContextType;
```

The above enumerations are detailed in the table below.

Enumeration	Description
E_JIP_CONTEXT_CLIENT	Context data for client (libJIP operating in client mode)
E_JIP_CONTEXT_SERVER	Context data for server (libJIP operating in server mode)

5.9 teJIP_AccessType

The following enumerations are used to indicate the access type of a parameter.

```
enum _eJIP_AccessType
{
    E_JIP_ACCESS_TYPE_CONST,
    E_JIP_ACCESS_TYPE_READ_ONLY,
    E_JIP_ACCESS_TYPE_READ_WRITE,
} PACK;
#ifdef WIN32
typedef uint8 teJIP_AccessType;
#else
typedef enum _eJIP_AccessType teJIP_AccessType;
#endif
```

The above enumerations are detailed in the table below.

Enumeration	Description
E_JIP_ACCESS_TYPE_CONST	Constant - cannot be changed
E_JIP_ACCESS_TYPE_READ_ONLY	Variable is read-only
E_JIP_ACCESS_TYPE_READ_WRITE	Variable is read- and write-enabled

5.10 teJIP_Security

Note that JIP-level security is not currently implemented (but JenNet-level security is available).

```
typedef enum
{
   E_JIP_SECURITY_NONE          /* Security is not implemented */
} PACK teJIP_Security;
```

Part III:
Java JenNet-IP API

6. IP Application Development (Java Version)

The chapter outlines the coding of a Java application that will run on a Linux-based IP host of a JenNet-IP system. This application may typically be used to remotely access a WPAN from a LAN/WAN device, such as a PC or mobile phone. References are made in this chapter to the Java JenNet-IP API (or Java JIP API) methods that you will need to use in your code.

6.1 Overview

A LAN/WAN device (such as PC, tablet or mobile phone) can be used to interact with a WPAN of a JenNet-IP system. The device may be located either:

- remotely from the WPAN, as in the case of a lighting system controlled from a PC in an office in another town, or
- locally to the WPAN, as in the case of a lighting system controlled from a mobile phone within the same building

The application that facilitates this interaction may run on the LAN/WAN device or on the Border-Router connected to the WPAN:

- If implemented directly on the LAN/WAN device, the application provides a dedicated interface for accessing the WPAN
- If implemented on the Border-Router, the application runs on the LAN/WAN side and serves web pages to the LAN/WAN device, where they can be displayed in a normal web browser

The Java JIP API can be used to develop an application that will run on a LAN/WAN device, as described in this chapter.

Note: An application to be run on the Border-Router can be implemented using the C JIP API (described in Part II: C JenNet-IP API), which can also be used to develop an application to be run on a LAN/WAN device.

6.2 API Organisation (Packages, Interfaces, Classes)

This section describes the organisation of the Java JIP API in terms of Java packages, interfaces and classes. The API comprises six Java packages, outlined below.

Note: The **com.nxp.jip** package provides low-level interfaces for implementing JIP operations while the **com.nxp.jip.service** package provides equivalent high-level interfaces. The latter is therefore easier to use.

6.2.1 com.nxp.jip

The **com.nxp.jip** package provides low-level interfaces for implementing JIP operations and is detailed in Chapter 7. It contains the following interfaces and classes:

Interfaces

- JIP
- JipValue
- ModuleList
- ModuleRecord
- Variable
- VariableList
- VariableRecord
- PacketHandler
- PacketListener
- TrapListener

Classes

- JIPImpl
- JipTypes
- PacketHandlerIPv4
- PacketHandlerIPv6

6.2.2 com.nxp.jip.variables

The **com.nxp.jip.variables** package defines the different MIB variable types and is detailed in Chapter 8. It contains the following classes:

Classes

- JipInteger
- JipFloat
- JipDouble
- JipString
- JipTable
- JipBlob

6.2.3 com.nxp.jip.service

The **com.nxp.jip.service** package provides high-level interfaces for implementing JIP operations and is detailed in Chapter 9. It contains the following interfaces and classes:

Interfaces

- JenNetIPNetwork.NodeDiscoveryListener
- Service.TableGetListener

Classes

- JenNetIPNetwork
- Module
- Node
- Service
- VariableInst

6.2.4 com.nxp.jip.service.persist

The **com.nxp.jip.service.persist** package is used for saving and recovering context data, and is detailed in Chapter 10. It contains the following class:

Class

- XmlPersistence

6.2.5 com.nxp.jip.service.cache

The **com.nxp.jip.service.cache** package is used for caching context data, but its details are not required for application development and are therefore not documented here. It contains the following interface and class:

Interface

- Cache

Class

- SimpleCache

6.2.6 com.nxp.jip.exception

The **com.nxp.jip.exception** defines the JIP exceptions, but its details are not required for application development and are therefore not documented here. It contains the following classes.

Classes

- JipError
- JipException
- JipTimeoutException

6.3 JIP Sessions

A communication link between the application and a WPAN is logically represented by a 'JIP session'. If the application opens simultaneous communication links with multiple WPANs, multiple JIP sessions will exist concurrently. Within a JIP session, the application must create and maintain 'context data' for the corresponding WPAN (see Section 5.1). This context data comprises information about the WPAN and is divided into the following two parts:

- General context data about device classes, MIBs and MIB variables (e.g. for each device class, the MIBs that it supports and the variables of each of these MIBs) - this data is held by the Service class
- Network-specific context data for the particular WPAN which corresponds to the JIP session - this data is held by the JenNetIPNetwork class

When a JIP session is created, the corresponding context data is blank. To populate this context data, the application must do one of the following:

- Connect to the Border-Router of the target WPAN and discover the details of the WPAN (nodes, MIBs and MIB variables)
- Recover persisted context data which has previously been discovered and saved to an XML file in non-volatile memory (using the XmlPersistance class)

During the session, the application can then monitor and control the WPAN. Control is achieved by writing to MIB variables on the nodes. Monitoring can be achieved by reading the MIB variables or setting up JIP traps to provide automatic notifications when the variables change.

The Java methods required to implement the above access, control and monitoring are indicated in the rest of this chapter:

- Creating a JIP session (including connecting to the Border-Router of a WPAN) is described in Section 6.4
- Discovering the WPAN is described in Section 6.5
- Monitoring the WPAN is described in Section 6.6
- Accessing MIB variables on wireless nodes is described in Section 6.7
- Persisting the context data for the WPAN is described in Section 6.8

6.4 Initialisation

In order to use the Java JIP API to access a WPAN, a JIP service and then a JIP session must be created, as described in the sub-sections below.

6.4.1 Creating a JIP Service

First, a local IP connection must be pre-configured that will be used to connect to the WPAN. This can be an IPv6 connection or an IPv4 connection, depending on the IP version used by the intervening network. One of the following Java constructors must first be used to create a suitable packet handler for the connection:

- **PacketHandlerIPv6()** from the PacketHandlerIPv6 class to create a packet handler for an IPv6 connection (UDP)
- **PacketHandlerIPv4()** from the PacketHandlerIPv4 class to create a packet handler for an IPv4 connection (UDP or TCP)

Both of the above methods create a socket (associated with a local port) to be used for the communication link.

Using the above packet handler, a JIP implementation instance must now be created using the **JIPImpl()** constructor of the JIPImpl class. This instance adds itself as a ‘packet listener’ to the nominated packet handler, so that it will be notified of any packets received by this handler.

Using the above JIP implementation instance, a JIP service can now be created using the **Service()** constructor of the Service class.

The above three calls can be combined, as in the following example:

```
Service service = new Service(new JIPImpl(new PacketHandlerIPv6()));
```

6.4.2 Creating a JIP Session

Once a JIP service has been created with the required IP connectivity (IPv6 or IPv4), the application must create a JIP session which implements the IP connection to the WPAN.

A JIP session can be started using the **createNetwork()** method of the Service class, by specifying the address of the Border-Router for the relevant WPAN. This method returns a JenNetIPNetwork object which represents the network and subsequent access to this network is implemented by the JenNetIPNetwork class.

An example of the above call is:

```
JenNetIPNetwork network = service.createNetwork(new
InetSocketAddress("fd04:bd3:80e8:2:215:8d00:e:6780", 1873));
```

At this stage, the context data for the session will be blank, but will be automatically populated during the discovery stages.

Note: A JIP service and session created as described above can be closed using the **shutdown()** method of the Service class. This call will stop any threads, free the IP connection, remove any traps and free memory space associated with the session.

6.5 Discovering the WPAN

Once an IP connection has been established to the Border-Router of a WPAN (see Section 6.4), the application can initiate a ‘discovery’ of the nodes in the attached WPAN using the method **discoverNodes()** from the JenNetIPNetwork class. The discovery results are inserted into a list of nodes, where each node is represented by an object based on the Node class.

Note: There are two versions of the method **discoverNodes()**. One version allows the discovery results for individual nodes to be reported to a ‘discovery listener’, based on the interface JenNetIPNetwork.NodeDiscoveryListener.

Once the nodes of the WPAN have been ‘discovered’, information about the nodes can be obtained locally from the node list.

6.5.1 Node Information

Information on the discovered nodes of the WPAN can be extracted (locally) from the created node list:

- From the JenNetIPNetwork class:
    - **getDeviceClassList()** can be used to obtain a list of the Device IDs that are present in the network.
    - **getNodes()** can be used to obtain a list of all the nodes with a particular Device ID in the network.
    - **getNode()** can be used to obtain details of the node with a particular address.
- From the Node class:
    - **getDeviceClass()** can be used to obtain the Device ID of an individual node.
    - **getAddress()** can be used to obtain the address of an individual node.
    - **toString()** can be used to obtain a string containing the Device ID and address of an individual node.

6.5.2 MIB Information

The following methods allow information to be obtained about the MIBs on a node:

- From the JIPImpl class:
    - **queryModules()** can be used to obtain a set of MIB records for a particular node, where these records contain the names and types of the MIBs. The obtained records may only represent a subset of the MIBs on the node - the method can be called multiple times to obtain details of all the MIBs on the node.
- From the Node class:
    - **getModules()** can be used to obtain a list of all the MIBs on a particular node. The method searches the relevant node information held locally.
    - **getModuleByName()** can be used to obtain details of the MIB with a given name on a particular node (if the MIB exists). The method searches the relevant node information held locally.
    - **getModuleById()** can be used to obtain details of the MIB with a given MIB ID on a particular node (if the MIB exists). The method searches the relevant node information held locally.
    - **getModuleByIndex()** can be used to obtain details of the MIB with a given index on a particular node (if the MIB exists). The method searches the relevant node information held locally.
- From the ModuleList interface:
    - **getModules()** can be used to obtain a map of the MIB records in the list of the MIBs on a node, where the records are mapped according to MIB index.

  - **getLastIndex()** and **getModulesRemaining()** can be used to evaluate which MIBs are present on the node but do not appear in the list of MIB records.
- From the ModuleRecord interface:
  - **getModuleName()** can be used to obtain the name of a MIB from its MIB record
  - **getModuleId()** can be used to obtain the ID of a MIB from its MIB record
  - **getModuleIndex()** can be used to obtain the index of a MIB from its MIB record
- From the Module class:
  - **getName()** can be used to obtain the name of a particular MIB
  - **getModuleId()** can be used to obtain the ID of a particular MIB

6.5.3 MIB Variable Information

The following methods allow information to be obtained about the variables on a MIB (note that methods concerned with setting and getting the values of MIB variables are referenced in Section 6.7):

- From the JIPImpl class:
  - **queryVariables()** can be used to obtain a number of MIB Variable records from a particular MIB on a particular node.
- From the Module class:
  - **getVariables()** can be used to obtain a list of the variables in a MIB.
  - **getVariable()** can be used to obtain the variable with a particular name or index within a MIB.
- From the VariableInst class:
  - **getVariableRecord()** can be used to obtain the MIB variable record for a MIB variable.
  - **getVarType()** can be used to obtain the type of a MIB variable.
  - **isReadOnly()** can be used to determine whether a MIB variable has the access type READ_ONLY.
  - **isConstant()** can be used to determine whether a MIB variable has the access type CONST.
  - **isTable()** can be used to determine whether a MIB variable is a table.
  - **isDisabled()** can be used to determine whether a MIB variable is disabled.
- From the JipValue interface:
  - **getType()** can be used to obtain the type of a variable.
- From the Variable interface:
  - **getModuleIndex()** can be used to obtain the index of the MIB which contains a particular variable.

    - **getVarIndex()** can be used to obtain the index of a variable (within a MIB).
    - **getVarType()** can be used to obtain the type of a MIB variable.
    - **isTable()** can be used to determine whether a MIB variable is a table.
    - **isDisabled()** can be used to determine whether a MIB variable is disabled.
- From the VariableList interface:
    - **getModuleIndex()** can be used to obtain the index of the MIB which contains the set of variables indicated in a list of MIB Variable records.
    - **getVariables()** can be used to obtain a map of the records in a list of MIB Variable records, referenced using the MIB Variable index.
    - **getVariablesRemaining()** can be used to obtain the number of MIB Variables present in a MIB after the last one reported in the list.
- In all classes of the **com.nxp.jip.variables** package:
    - **getType()** can be used to obtain the type of a MIB variable.

6.6 Monitoring the WPAN

A WPAN can be monitored for changes such as a node joining or leaving the network, or moving (to a new parent) within the network. This monitoring is initiated using the method **startMonitoring()** from the JenNetIPNetwork class. This method registers a node discovery listener (which will be notified of discovery events) and starts monitoring the network for changes (if it is not already being monitored).

The poll-period for monitoring (in milliseconds) can be set or changed at any time using the method **setMonitorInterval()** from the same class.

Monitoring can subsequently be stopped using the method **stopMonitoring()**. If multiple listeners have been added with **startMonitoring()**, monitoring will only be stopped once all listeners have been removed with **stopMonitoring()**.

6.7 Accessing MIB Variables

The MIB variables on a WPAN node may need to be remotely accessed from a LAN/WAN device in order to read from or write to the variables. In addition, JIP traps can be configured on a MIB variable in order to provide automatic notifications when the variable changes. These types of access are described in the sub-sections below.

6.7.1 Reading from MIB Variables

The following methods allow the value of a MIB variable to be read:

- From the JIPImpl class:
  - **get()** can be used to request the value of a variable of a MIB specified by its ID on a remote node, where the variable may contain a single value or a table of values (two versions of the method are provided). In the latter case, certain rows of the table can be requested.
  - **getByIndex()** can be used to request the value of a variable of a MIB specified by its index on a remote node, where the variable may contain a single value or a table of values (two versions of the method are provided). In the latter case, certain rows of the table can be requested.
- From the VariableInst class:
  - **getValue()** can be used to obtain the value of a MIB variable from the remote node.
- In all classes of the **com.nxp.jip.variables** package:
  - **getValue()** can be used to obtain the value of a MIB variable from a local copy obtained using one of the above methods.

6.7.2 Writing to MIB Variables

The following methods allow a MIB variable to be written to:

- From the JIPImpl class:
  - **set()** can be used to write the value of a variable of a MIB specified by its ID on a remote node, where the variable is a single value.
  - **setByIndex()** can be used to write the value of a variable of a MIB specified by its index on a remote node, where the variable may contain a single value.
- From the VariableInst class:
  - **setValue()** can be used to write the value of a MIB variable on the remote node.
- In all classes of the **com.nxp.jip.variables** package:
  - **setValue()** can be used to write the value of a MIB variable to a local copy (which can later be written to the actual variable on the remote node using one of the above methods).

Alternatively, the method **multicastSet()** from the JIPImpl class can be used to send a write request to multiple nodes in order to update the same MIB variable on these nodes. The request will be received by all nodes in the WPAN but only implemented on nodes in the multicast group with the specified IPv6 multicast address. No responses are issued by the recipient nodes and the method returns immediately after sending the request.

> ***Caution:*** *For all the 'set' methods of the JIPImpl class, the locally held context data for the relevant WPAN is not updated with the new data for the variable and so this context data will become desynchronised with the data on the nodes (unless the application takes steps to maintain synchronisation).*

> **Note:** None of the above 'set' methods can be used to write data to a MIB variable of the type 'table of blobs'.

6.7.3 Using JIP Traps on MIB Variables

The application on a LAN/WAN device or the Border-Router can set up a JIP trap on a MIB variable on a WPAN node. This results in the automatic generation of a notification to the application when the MIB variable on the node is changed in some way.

> ***Caution:*** *A trap that has been configured on a MIB variable is not guaranteed to be generated when a change in the variable occurs.*

Traps on MIB variables can be implemented in the following ways.

Using Methods of the JIPImpl Class

Traps on a MIB variable can be enabled using the following procedure:

1. A trap listener which will receive all trap notifications can be registered using the method **addTrapListener()** from the JIPImpl class (this trap listener can later be removed using the method **removeTrapListener()** from the same class).
2. A JIP trap can then be enabled on a specific MIB variable using the **trap()** method from the JIPImpl class (this trap can later be disabled using the **untrap()** method from the same class). The node, MIB and variable must be specified in this method.

Using Methods of the VariableInst Class

Traps on a MIB variable can be enabled in either of the following ways:

- A trap listener to receive trap notifications can be registered for a specific MIB variable using the method **addListener()** from the VariableInst class (this trap listener can later be removed using the method **removeListener()** from the same class).
- A JIP trap can be enabled on the MIB variable using the **trap()** method from the VariableInst class (this trap can later be disabled using the **untrap()** method from the same class). The trap listener to receive the trap notifications for this variable must be specified as part of the call.

6.8 Persisting Context Data

The context data which reflects the composition of a remote WPAN can be preserved in Non-Volatile Memory (NVM) so that it is still available following a break in execution of the application (e.g. due to a power outage or power cycle). For example, on a PC, the NVM used may be the hard disk.

For the purpose of persisting context data in this way, this data is treated in two parts:

- Network context data comprising basic information about the composition of the network, including the IPv6 addresses and Device IDs of the nodes in the network - see Section 6.8.1
- Node context data which defines the MIBs that reside on nodes with different Device IDs in the network - see Section 6.8.2

The Java methods used in persisting context data are provided by the class XmlPersistence from the package **com.nxp.jip.service.persist**.

6.8.1 Network Context Data

Network context data can be saved to NVM at any time using the method **saveNetwork()** from the XmlPersistence class. The data is written to an XML file.

The saved context data can be retrieved at any time using the method **loadNetwork()** from the XmlPersistence class. This function reads the relevant XML file for the network.

6.8.2 Node Context Data

Node context data contained in a cache instance can be saved to NVM at any time using the method **saveDefinitions()** from the XmlPersistence class. This method saves information concerning the MIBs and associated variables that reside on nodes of each Device ID in the network. Note that the data stored in the MIB variables is not saved. The persisted data is written to an XML file.

The saved context data can be retrieved at any time using the method **loadDefinitions()** from the XmlPersistence class. This method reads the relevant

XML file for the relevant cache instance. The read data is held internally to allow the subsequent rapid discovery of nodes.

Use of these functions to store and retrieve node context data avoids the need to rediscover the nodes of a network following a break in execution of the application.

7. Java Package com.nxp.jip

This chapter details the Java package **com.nxp.jip** which is supplied as part of the Java JIP API. This package consists of the following interfaces:

- **JIP** - see Section 7.1
- **JipValue** - see Section 7.2
- **ModuleList** - see Section 7.3
- **ModuleRecord** - see Section 7.4
- **Variable** - see Section 7.5
- **VariableList** - see Section 7.6
- **VariableRecord** - see Section 7.7
- **PacketHandler** - see Section 7.8
- **PacketListener** - see Section 7.9
- **TrapListener** - see Section 7.10

The referenced sections detail the methods and fields associated with each interface. In addition, the classes of the package are detailed in Section 7.11.

7.1 JIP Interface

This section describes the JIP interface.

7.1.1 JIP Interface Fields

The JIP interface fields are listed and described in the table below.

Field	Type	Description
BUFF_SIZE	static int	Maximum size (in bytes) of message payload.
DEFAULT_PORT	static int	Default port number for sending messages to JenNet-IP nodes

Table 2: JIP Interface Fields

7.1.2 JIP Interface Methods

The JIP interface methods are listed below, along with their page references:

Caution: *With the exception of the multicast set method, all JIP interface methods are blocking. Furthermore, only one can method execute at any one time. Consequently, a method invocation may block the current thread for a very long time.*

Some of the above methods contain an optional *flags* parameter. These methods can be used with or without this parameter, which may be needed when sending a message to an End Device. When used, the parameter takes an EnumSet containing none, either or both of the following `MessageFlag` enumerations:

Enumeration	Description
USE_SLEEPING_DEVICE_TIMEOUT	Allows a longer timeout to be used when transmitting to an End Device that can sleep. The timeout is set using the method **setSleepingDeviceTimeout()**.
REQUEST_STAY_AWAKE	Allows the ‘stay awake’ flag to be set in the message, which requests the target End Device to stay awake to receive further messages. This option is detailed in the *JenNet-IP WPAN Stack User Guide (JN-UG-3080)*.

Table 3: MessageFlag Enumerations

get (single value)

```
Variable get(InetSocketAddress node,
             int moduleId,
             int varIndex,
             EnumSet<MessageFlag> flags);
```

Description

This method can be used to request the value of the specified variable of a particular MIB on the specified node of a WPAN, where the MIB is specified using its identifer.

Parameters

node IP socket address containing IPv6 address of the target node and the port number which provides access to the MIB variable

moduleId ID of the MIB which contains the variable

varIndex Index of the variable within the MIB

flags Optional parameter (can be omitted) which allows one or both of the following options to be selected (see Table 3 on page 114):

USE_SLEEPING_DEVICE_TIMEOUT
REQUEST_STAY_AWAKE

Returns

Variable object retrieved from node

Exceptions

Throws com.nxp.jip.JipException

get (table variable)

Variable get(InetSocketAddress *node***,**
int *moduleId***,**
int *varIndex***,**
short *firstTableEntry***,**
int *entryCount***,**
EnumSet<MessageFlag> *flags***);**

Description

This method can be used to request a number of rows of a table variable of the specified MIB on the specified node of a WPAN, where the MIB is specified using its identifer.

The number of rows returned may be smaller than the number requested.

The list returned may be empty, but it shall not be null.

Parameters

node	IP socket address containing IPv6 address of the target node and the port number which provides access to the MIB variable
moduleId	ID of the MIB which contains the variable
varIndex	Index of the variable within the MIB
firstTableEntry	First row in the table variable to return
entryCount	Maximum number of table rows to return
flags	Optional parameter (can be omitted) which allows one or both of the following options to be selected (see Table 3 on page 114): USE_SLEEPING_DEVICE_TIMEOUT REQUEST_STAY_AWAKE

Returns

Variable object retrieved from node

Exceptions

Throws com.nxp.jip.JipException

getByIndex

```
Variable getByIndex(InetSocketAddress node,
                    int moduleIndex,
                    int varIndex);
```

Description

This method can be used to request the value of the specified variable of a particular MIB on the specified node of a WPAN, where the MIB is specified using its index number.

Parameters

node	IP socket address containing IPv6 address of the target node and the port number which provides access to the MIB variable
moduleIndex	Index of the MIB which contains the variable
varIndex	Index of the variable within the MIB

Returns

Variable object retrieved from node

Exceptions

Throws com.nxp.jip.JipException

set

```
void set(InetSocketAddress node,
            int moduleId,
            int varIndex,
            JipValue var,
            EnumSet<MessageFlag> flags);
```

Description

This method can be used to set the specified value of the specified variable of a particular MIB on the specified node of a WPAN, where the MIB is specified using its identifer.

Parameters

node	IP socket address containing IPv6 address of the target node and the port number which provides access to the MIB variable
moduleId	ID of the MIB which contains the variable
varIndex	Index of the variable within the MIB
var	Value to which variable is to be set
flags	Optional parameter (can be omitted) which allows one or both of the following options to be selected (see Table 3 on page 114): USE_SLEEPING_DEVICE_TIMEOUT REQUEST_STAY_AWAKE

Returns

None

Exceptions

Throws com.nxp.jip.JipException

setByIndex

```
void setByIndex(InetSocketAddress node,
                int moduleIndex,
                int varIndex,
                JipValue var);
```

Description

This method can be used to set the specified value of the specified variable of a particular MIB on the specified node of a WPAN, where the MIB is specified using its index number.

Parameters

node	IP socket address containing IPv6 address of the target node and the port number which provides access to the MIB variable
moduleIndex	Index of the MIB which contains the variable
varIndex	Index of the variable within the MIB
var	Value to which variable is to be set

Returns

None

Exceptions

Throws com.nxp.jip.JipException

multicastSet

```
void multicastSet(InetSocketAddress group,
                  int moduleId,
                  int varIndex,
                  JipValue var);
```

Description

This method can be used to set the specified value of the specified variable of the specified MIB on all nodes in the group that corresponds to the given IPv6 multicast address.

A message broadcast is performed to the specified multicast address. On receiving the message, each node which is a member of the relevant group will attempt to perform the requested operation.

A failure to transmit the broadcast (e.g. due to a socket error) will be detected and reported (via a relevant exception being thrown). Success, on the other hand, does not guarantee delivery.

Note: Unlike most methods of the JIP object, this method is non-blocking and will execute immediately. However, there is no way for the calling application to find out whether the operation succeeded.

Parameters

node IP socket address containing IPv6 multicast address of the target nodes and the port number which provides access to the MIB variable

moduleId ID of the MIB which contains the variable

varIndex Index of the variable within the MIB

var Value to which variable is to be set

Returns

None

Exceptions

Throws com.nxp.jip.JipException

queryModules

```
ModuleList queryModules(InetSocketAddress node,
                        int firstModIndex,
                        int maxRecords,
                        EnumSet<MessageFlag> flags);
```

Description

This method can be used to obtain a number of MIB records from the specified node of a WPAN. A MIB record includes the following information about an individual MIB: identifier, index and name. The method also returns the index of the last MIB for which a record has been returned and the number of remaining (unreported) MIBs on the node.

The method allows the index of the first MIB of interest to be specified, as well as the maximum number of records to return (in one call). Thus, it can be called multiple times to obtain the records for all the MIBs on the target node.

Parameters

node IP socket address containing IPv6 address of the target node and the port number from which to obtain MIB records

firstModIndex Index of first MIB for which record should be obtained

maxRecords Maximum number of MIB records to be obtained

flags Optional parameter (can be omitted) which allows one or both of the following options to be selected (see Table 3 on page 114):

USE_SLEEPING_DEVICE_TIMEOUT
REQUEST_STAY_AWAKE

Returns

List of MIB records returned by the node

Exceptions

Throws com.nxp.jip.JipException

queryVariables

```
VariableList queryVariables(InetSocketAddress node,
                            int moduleIndex,
                            int firstVarIndex,
                            int maxRecords,
                            EnumSet<MessageFlag> flags);
```

Description

This method can be used to obtain a number of MIB Variable records from the specified MIB on the specified node of a WPAN. A MIB Variable record includes the following information about an individual MIB Variable: index, name, type, access type and security flag. The method also returns the number of remaining (unreported) variables within the specified MIB on the node.

The method allows the index of the first variable of interest (within the MIB) to be specified, as well as the maximum number of records to return (in one call). Thus, it can be called multiple times to obtain the records for all variables of the specified MIB on the target node.

Parameters

node	IP socket address containing IPv6 address of the target node and the port number from which to obtain MIB Variable records
moduleIndex	Index of MIB to be queried
firstVarIndex	Index of first variable (in MIB) for which record should be obtained
maxRecords	Maximum number of MIB Variable records to be obtained
flags	Optional parameter (can be omitted) which allows one or both of the following options to be selected (see Table 3 on page 114): USE_SLEEPING_DEVICE_TIMEOUT REQUEST_STAY_AWAKE

Returns

List of MIB Variable records returned by the node

Exceptions

Throws com.nxp.jip.JipException

trap

```
void trap(InetSocketAddress node,
          int moduleIndex,
          int varIndex,
          int handle,
          EnumSet<MessageFlag> flags);
```

Description

This method can be used to register a trap on the specified MIB Variable on the specified node of a WPAN, in order to receive notifications of changes to value of the variable.

Parameters

node IP socket address containing IPv6 address of the target node and the port number of the MIB Variable to be trapped

moduleIndex Index of MIB containing variable to be trapped

varIndex Index of variable (in MIB) for which trap is to be set up

handle User-defined identifier that will be returned in trap notifications

flags Optional parameter (can be omitted) which allows one or both of the following options to be selected (see Table 3 on page 114):

USE_SLEEPING_DEVICE_TIMEOUT
REQUEST_STAY_AWAKE

Returns

None

Exceptions

Throws com.nxp.jip.JipException

untrap

```
void untrap(InetSocketAddress node,
            int moduleIndex,
            int varIndex,
            EnumSet<MessageFlag> flags);
```

Description

This method can be used to unregister an existing trap on the specified MIB Variable on the specified node of a WPAN, in order to disable notifications of changes to value of the variable.

Parameters

node IP socket address containing IPv6 address of the target node and the port number of the MIB Variable to be untrapped

moduleIndex Index of MIB containing variable to be untrapped

varIndex Index of variable (in MIB) for which trap is to be removed

flags Optional parameter (can be omitted) which allows one or both of the following options to be selected (see Table 3 on page 114):

USE_SLEEPING_DEVICE_TIMEOUT
REQUEST_STAY_AWAKE

Returns

None

Exceptions

Throws com.nxp.jip.JipException

addTrapListener

void addTrapListener(TrapListener *listener*);

Description

This method can be used to register a ‘trap listener’ which will receive notifications of all subsequent trap events.

Parameters

listener	Trap listener to be registered

Returns

None

Exceptions

None

removeTrapListener

void removeTrapListener(TrapListener *listener*);

Description

This method can be used to unregister a ‘trap listener’ so that it will no longer receive notifications of trap events.

Parameters

listener	Trap listener to be unregistered

Returns

None

Exceptions

None

setPacketHandler

```
void setPacketHandler(PacketHandler ph);
```

Description

This method can be used to set the packet handler instance to be used in sending and receiving messages.

Parameters

ph	Packet handler to use

Returns

None

Exceptions

Throws java.io.IOException

setRetries

```
void setRetries(int retries);
```

Description

This method can be used to set the number of times that the transmission of a JIP message should be retried before generating a JipException.

Parameters

retries Number of times to re-attempt transmission of JIP message

Returns

None

Exceptions

Throws com.nxp.jip.JipException

setTimeout

void setTimeout(int *milliseconds*);

Description

This method can be used to set the timeout before the transmission of a JIP message is considered to have failed.

If the timeout is reached, the transmission will be retried or a JipException will be generated (depending on the number of retries that have been set with **setRetries()**).

Note: The equivalent timeout for a transmission with the USE_SLEEPING_DEVICE_TIMEOUT option selected is set using the **setSleepingDeviceTimeout()** method.

Parameters

retries	Timeout, in milliseconds

Returns

None

Exceptions

Throws com.nxp.jip.JipException

setSleepingDeviceTimeout

void setSleepingDeviceTimeout(long *t*);

Description

This method can be used to set the timeout before the transmission of a JIP message is considered to have failed, when the USE_SLEEPING_DEVICE_TIMEOUT flag has been set in the method that initiated the transmission (e.g. in the **get()** method).

If the timeout is reached, the transmission will be retried or a JipException will be generated (depending on the number of retries that have been set with **setRetries()**).

Note 1: The equivalent timeout for a transmission without the USE_SLEEPING_DEVICE_TIMEOUT option selected is set using the **setTimeout()** method.

Note 2: The USE_SLEEPING_DEVICE_TIMEOUT option is provided since when sending a message to an End Device that can sleep, a longer timeout may be required.

Parameters

t	Timeout, in milliseconds

Returns

None

Exceptions

Throws com.nxp.jip.JipException

close

```
void close(void);
```

Description

This method can be used to stop any threads and close any sockets used by the implementation.

Parameters

None

Returns

None

Exceptions

None

7.2 JipValue Interface

This section describes the JipValue interface, which consists only of methods.

7.2.1 JipValue Interface Methods

The JipValue interface methods are listed below, along with their page references:

Method	Page
getValue	133
getType	134

getValue

```
Object getValue();
```

Description

This method can be used to obtain the value of a variable. An object of a class appropriate to the value type will be returned.

For example, this method could be used to obtain the value of a MIB variable obtained using the JIP **get()** method.

The type of the variable can be obtained using the method **getType()**.

Parameters

None

Returns

Object representing value of variable

Exceptions

None

getType

```
JipTypes.VariableType getType();
```

Description

This method can be used to obtain the type of a variable.

For example, this method could be used to obtain the type of a MIB variable obtained using the JIP **get()** method.

The value of the variable can be obtained using the method **getValue()**.

Parameters

None

Returns

Type of variable

Exceptions

None

7.3 ModuleList Interface

This section describes the ModuleList interface, which consists only of methods.

7.3.1 ModuleList Interface Methods

The ModuleList interface methods are listed below, along with their page references:

Method	Page
getLastIndex	136
getModules	137
getModulesRemaining	138

getLastIndex

```
int getLastIndex();
```

Description

This method can be used to obtain the index value of the last MIB in a list of MIBs.

For example, this method could be used to obtain the index value of the last MIB in a list of MIB records returned by the JIP **queryModules()** method.

Parameters

None

Returns

Index value of last MIB in list

Exceptions

None

getModules

```
Map<Integer,ModuleRecord> getModules();
```

Description

This method can be used to obtain a map of the records in a list of MIB records, where a record is referenced using the MIB index.

For example, this method could be used to obtain a map of the MIB records in a list returned by the JIP **queryModules()** method.

Parameters

None

Returns

Map containing the index and record for each MIB in the list

Exceptions

None

getModulesRemaining

```
int getModulesRemaining();
```

Description

This method can be used to obtain the number of MIBs present on a node after the last one reported in the list (with index value greater than that of the last MIB in the list).

For example, this method could be used after a call to the JIP **queryModules()** method in order to find the number of unreported MIBs on the node.

Parameters

None

Returns

Number of remaining MIBs on node (not reported in list)

Exceptions

None

7.4 ModuleRecord Interface

This section describes the ModuleRecord interface, which consists only of methods.

7.4.1 ModuleRecord Interface Methods

The ModuleRecord interface methods are listed below, along with their page references:

Method	**Page**
getModuleIndex	140
getModuleId	141
getModuleName	142

getModuleIndex

```
int getModuleIndex();
```

Description

This method can be used to obtain the index value of a MIB from its MIB record.

For example, this method could be used after a call to the JIP **queryModules()** method in order to extract the MIB index value from an individual MIB record.

Parameters

None

Returns

Index value of MIB

Exceptions

None

getModuleId

```
int getModuleId();
```

Description

This method can be used to obtain the identifier of a MIB from its MIB record.

For example, this method could be used after a call to the JIP **queryModules()** method in order to extract the MIB identifier from an individual MIB record.

Parameters

None

Returns

Identifer of MIB

Exceptions

None

getModuleName

```
String getModuleName();
```

Description

This method can be used to obtain the name of a MIB from its MIB record.

For example, this method could be used after a call to the JIP **queryModules()** method in order to extract the MIB name from an individual MIB record.

Parameters

None

Returns

Name of MIB

Exceptions

None

7.5 Variable Interface

This section describes the Variable interface, which consists only of methods.

7.5.1 Variable Interface Methods

The Variable interface methods are listed below, along with their page references:

Method	Page
getValue	144
getVarType	145
getVarIndex	146
getModuleIndex	147
isDisabled	148
isTable	149

getValue

```
JipValue getValue();
```

Description

This method can be used to obtain the value of a variable.

Parameters

None

Returns

Value of variable or NULL if variable is disabled

Exceptions

Throws com.nxp.jip.JipException

getVarType

```
JipTypes.VariableType getVarType();
```

Description

This method can be used to obtain the type of a variable.

Parameters

None

Returns

Type of variable

Exceptions

Throws com.nxp.jip.JipException

getVarIndex

```
int getVarIndex();
```

Description

This method can be used to obtain the index of a variable (within a MIB).

Parameters

None

Returns

Index of variable within MIB

Exceptions

None

getModuleIndex

```
int getModuleIndex();
```

Description

This method can be used to obtain the index of the MIB which contains a particular variable.

Parameters

None

Returns

Index of MIB containing variable

Exceptions

None

isDisabled

```
boolean isDisabled();
```

Description

This method can be used to determine whether a variable is disabled.

Parameters

None

Returns

TRUE if variable is disabled, otherwise FALSE

Exceptions

Throws com.nxp.jip.JipException

isTable

```
boolean isTable();
```

Description

This method can be used to determine whether a variable is a table.

Parameters

None

Returns

TRUE if variable is a table, otherwise FALSE

Exceptions

Throws com.nxp.jip.JipException

7.6 VariableList Interface

This section describes the VariableList interface, which consists only of methods.

7.6.1 VariableList Interface Methods

The VariableList interface methods are listed below, along with their page references:

Method	Page
getVariables	151
getVariablesRemaining	152
getModuleIndex	153

getVariables

```
Map<Integer,VariableRecord> getVariables();
```

Description

This method can be used to obtain a map of the records in a list of MIB Variable records, where a record is referenced using the MIB Variable index.

For example, this method could be used to obtain a map of the MIB Variable records in a list returned by the method **queryVariables()**.

Parameters

None

Returns

Map containing the index and record for each MIB Variable in the list

Exceptions

None

getVariablesRemaining

```
int getVariablesRemaining();
```

Description

This method can be used to obtain the number of MIB Variables present in a MIB on a node after the last one reported in the list (with index value greater than that of the last MIB Variable in the list).

For example, this method could be used after a call to the method **queryVariables()** in order to find the number of unreported variables in the MIB.

Parameters

None

Returns

Number of remaining variables in the MIB (not reported in list)

Exceptions

None

getModuleIndex

```
int getModuleIndex();
```

Description

This method can be used to obtain the index of the MIB which contains the set of variables indicated in a list of MIB Variable records.

For example, this method could be used after a call to the method **queryVariables()** in order to find the index of the MIB which contains the reported variables.

Parameters

None

Returns

Index of MIB containing set of variables

Exceptions

None

7.7 VariableRecord Interface

This section describes the VariableRecord interface, which consists only of methods.

7.7.1 VariableRecord Interface Methods

The VariableRecord interface methods are listed below, along with their page references:

Method	Page
getType	155
getVarIndex	156
getVarName	157
getAccess	158
getSecurity	159

getType

```
JipTypes.VariableType getType();
```

Description

This method can be used to obtain the type of a MIB variable from its MIB Variable record.

For example, this method could be used to obtain the type of a MIB Variable reported following a call to the JIP **queryVariables()** method.

Parameters

None

Returns

Type of variable

Exceptions

None

getVarIndex

```
int getVarIndex();
```

Description

This method can be used to obtain the index of a variable (within a MIB) from its MIB Variable record.

For example, this method could be used to obtain the index of a MIB Variable reported following a call to the JIP **queryVariables()** method.

Parameters

None

Returns

Index of variable within MIB

Exceptions

None

getVarName

```
String getVarName();
```

Description

This method can be used to obtain the name of a MIB Variable from its MIB Variable record.

For example, this method could be used to obtain the name of a MIB Variable reported following a call to the JIP **queryVariables()** method.

Parameters

None

Returns

Name of variable

Exceptions

None

getAccess

```
JipTypes.Access getAccess();
```

Description

This method can be used to obtain the type of access permitted to a MIB Variable.

For example, this method could be used to obtain the access type for a MIB Variable reported following a call to the JIP **queryVariables()** method.

Parameters

None

Returns

Access type for variable: constant, read-only or read-write

Exceptions

None

getSecurity

```
JipTypes.Security getSecurity();
```

Description

This method can be used to obtain the type of security configured for a MIB Variable.

For example, this method could be used to obtain the security type for a MIB Variable reported following a call to the JIP **queryVariables()** method.

Parameters

None

Returns

Security type for variable: (currently) none

Exceptions

None

7.8 PacketHandler Interface

This section describes the PacketHandler interface, which consists only of methods.

7.8.1 PacketHandler Interface Methods

The PacketHandler interface methods are listed below, along with their page references:

Method	Page
open	161
close	162
send	163
addPacketListener	164

open

```
void open();
```

Description

This method can be used to open a packet handler and start listening for received packets.

Parameters

None

Returns

None

Exceptions

Throws com.nxp.jip.JipException

close

```
void close();
```

Description

This method can be used to stop listening for received packets and close an opened packet handler.

Parameters

None

Returns

None

Exceptions

None

send

```
void send(java.net.InetSocketAddress address,
          byte[] data,
          int len);
```

Description

This method can be used to transmit data from the specified buffer to the device with the given IP address.

Parameters

address	IP address of target device
data	Byte array containing data to be sent
len	Length of data to be sent, in bytes

Returns

None

Exceptions

Throws com.nxp.jip.JipException

addPacketListener

```
void addPacketListener(PacketListener packetListener);
```

Description

This method can be used to register a packet listener to be notified of packets received by this handler.

Parameters

packetListener	Packet listener to be registered

Returns

None

Exceptions

None

7.9 PacketListener Interface

This section describes the PacketListener interface, which consists only of a method.

7.9.1 PacketListener Interface Method

The PacketListener interface method is listed below, along with its page reference:

Method	Page
received	166

received

```
void received(byte[] data,
              int length,
              java.net.InetSocketAddress socketAddress);
```

Description

This method should be called for each packet received by a packet handler instance.

Parameters

data	Byte array containing payload of received packet
length	Length of packet payload, in bytes
socketAddress	Source address of packet

Returns

None

Exceptions

None

7.10 TrapListener Interface

This section describes the TrapListener interface, which consists only of a method.

7.10.1 TrapListener Interface Method

The TrapListener interface method is listed below, along with its page reference:

Method	Page
trapUpdate	168

trapUpdate

```
void trapUpdate(java.net.InetSocketAddress address,
                int handle,
                Variable var);
```

Description

This method must be called for each notification of a trap event received by a JIP implementation.

Parameters

address	Address of the remote node that sent the notification
handle	Trap handle of the event
var	Variable instance containing the new value

Returns

None

Exceptions

None

7.11 Classes of com.nxp.jip

This section details the classes of the **com.nxp.jip** package.

The package includes the following classes:

- **JIPImpl** - see Section 7.11.1
- **JipTypes** - see Section 7.11.2
- **PacketHandlerIPv4** - see Section 7.11.3
- **PacketHandlerIPv6** - see Section 7.11.4

7.11.1 JIPImpl Class

The JIPImpl class implements the JIP interface, described in Section 7.1, and includes fields, methods and a constructor, as indicated below.

Fields

This class inherits the JIP interface fields, detailed in Section 7.1.1.

Methods

This class inherits the JIP interface methods, detailed in Section 7.1.2.

It also inherits the following methods from the standard class java.lang.Object: **equals()**, **getClass()**, **hashCode()**, **notify()**, **notifyAll()**, **toString()**, **wait()**.

Constructor

JIPImpl() is used to construct a JIPImpl instance:

```
JIPImpl(PacketHandler packetHandler);
```

- *packetHandler* is the packet handler to be used.

7.11.2 JipTypes Class

The JipTypes class includes methods and nested classes.

Methods

This class inherits the following methods from the standard class java.lang.Object: **equals()**, **getClass()**, **hashCode()**, **notify()**, **notifyAll()**, **toString()**, **wait()**.

Nested Classes

The JipTypes class contains enumerations for various JIP constant values. These enumerations come from four nested classes:

- **JipTypes.Access** - see Section 7.11.2.1
- **JipTypes.Security** - see Section 7.11.2.2
- **JipTypes.Status** - see Section 7.11.2.3
- **JipTypes.VariableType** - see Section 7.11.2.4

7.11.2.1 JipTypes.Access

Enumeration	Description
CONST	Constant - cannot be changed
READ_ONLY	Variable is read-only
READ_WRITE	Variable is read- and write-enabled

Table 4: JipTypes.Access Enumerations

7.11.2.2 JipTypes.Security

There is currently only one security enumeration: NONE (no security).

7.11.2.3 JipTypes.Status

Enumeration	Description
ACCESS_NOT_ALLOWED	Attempted access was not permitted
BAD_BUFFER_SIZE	Buffer size is not appropriate
BAD_MODULE_INDEX	MIB index value is not valid
BAD_VARIABLE_INDEX	MIB variable index value is not valid
DISABLED	Feature is not enabled
ERROR	Operation has failed
SUCCESS	Operation has succeeded
TIMEOUT	Operation has timed out
VALUE_REJECTED	Specified value has not been accepted
WRONG_TYPE	Specified value is not of the correct type

Table 5: JipTypes.Status Enumerations

7.11.2.4 JipTypes.VariableType

Enumeration	Description (Variable Type)
BLOB	blob
DOUBLE	double
FLOAT	float
INT16	int16
INT32	int32
INT64	int64
INT8	int8
STRING	string
TABLE_BLOB	table blob
UINT16	uint16
UINT32	uint32
UINT64	uint64
UINT8	uint8

Table 6: JipTypes.VariableType Enumerations

7.11.3 PacketHandlerIPv4 Class

The PacketHandlerIPv4 class implements the PacketHandler interface, described in Section 7.8, and includes methods and a constructor, as indicated below. The class uses a JIPv4 tunnel to communicate with a JIP Border-Router (through a non-IPv6 network) and then with the associated JIP wireless network.

Methods

This class inherits the PacketHandler interface methods, detailed in Section 7.8.1.

It also inherits the following methods from the standard class java.lang.Object: **equals()**, **getClass()**, **hashCode()**, **notify()**, **notifyAll()**, **toString()**, **wait()**.

Constructor

PacketHandlerIPv4() is used to construct a PacketHandlerIPv4 instance to communicate with the specified JIP Border-Router:

```
PacketHandlerIPv4(
                java.net.InetSocketAddress ipv4Address,
                boolean useTcp);
```

- *ipv4Address* is the IPv4 address of the JIP Border-Router

- *useTcp* indicates whether to tunnel over TCP or UDP:
  - TRUE - use TCP
  - FALSE - use UDP

7.11.4 PacketHandlerIPv6 Class

The PacketHandlerIPv6 class implements the PacketHandler interface, described in Section 7.8, and includes methods and a constructor, as indicated below. The class uses an IPv6 multicast socket to communicate with a JIP wireless network.

Methods

This class inherits the PacketHandler interface methods, detailed in Section 7.8.1.

It also inherits the following methods from the standard class java.lang.Object: **equals()**, **getClass()**, **hashCode()**, **notify()**, **notifyAll()**, **toString()**, **wait()**.

Constructor

PacketHandlerIPv6() is used to construct a PacketHandlerIPv6 instance:

```
PacketHandlerIPv6();
```

8. Java Package com.nxp.jip.variables

This chapter details the Java package **com.nxp.jip.variables** which is supplied as part of the Java JIP API. This package consists of the following classes corresponding to different variable types:

- **JipInteger** - see Section 8.1
- **JipFloat** - see Section 8.2
- **JipDouble** - see Section 8.3
- **JipString** - see Section 8.4
- **JipTable** - see Section 8.5
- **JipBlob** - see Section 8.6

8.1 JipInteger Class

The JipInteger class implements the JipValue interface, described in Section 7.2, and includes methods and constructors, as indicated below.

Methods

This class inherits the JipValue interface methods, detailed in Section 7.2.1. In this JipValue implementation, the **getValue()** method returns an object of the standard Java type **java.math.BigInteger**.

The class also inherits the following methods from the standard class java.lang.Object: **getClass()**, **hashCode()**, **notify()**, **notifyAll()**, **toString()**, **wait()**.

In addition, the class has the following methods of its own, which each converts a value to a particular type.

intValue() converts the value to an **int** (the value may be rounded or truncated during the conversion):

```
int intValue();
```

shortValue() converts the value to a **short** (the value may be rounded or truncated during the conversion):

```
short shortValue();
```

longValue() converts the value to a **long** (the value may be rounded or truncated during the conversion):

```
long longValue();
```

byteValue() converts the value to a **byte** (the value may be rounded or truncated during the conversion):

```
byte byteValue();
```

equals() compares the **int** variable with another **int** variable:

```
boolean equals(java.lang.Object obj);
```

- *obj* is the object representing the **int** variable to compare with

The returned boolean indicates the outcome:

- TRUE - two variables have the same value
- FALSE - two variables do not have the same value

This method overrides **equals()** in the class java.lang.Object.

Constructors

This class has two **JipInteger()** constructors, both used to construct a JipInteger object:

```
JipInteger(JipTypes.VariableType type, long value);
```

- *type* is the JIP integer type of the object
- *value* is the value to be assigned to the object

```
JipInteger(JipTypes.VariableType type,
           java.nio.ByteBuffer buff);
```

- *type* is the JIP integer type of the object
- *buff* is a buffer containing the value to be assigned to the object

8.2 JipFloat Class

The JipFloat class implements the JipValue interface, described in Section 7.2, and includes methods and a constructor, as indicated below.

Methods

This class inherits the JipValue interface methods, detailed in Section 7.2.1. In this JipValue implementation, the **getValue()** method returns an object of the standard Java type **java.lang.Float**.

The class also inherits the following methods from the standard class java.lang.Object: **getClass()**, **hashCode()**, **notify()**, **notifyAll()**, **toString()**, **wait()**.

In addition, the class has the following method of its own.

equals() compares the **float** variable with another **float** variable:

```
boolean equals(java.lang.Object obj);
```

- *obj* is the object representing the **float** variable to compare with

The returned boolean indicates the outcome:

- TRUE - two variables have the same value
- FALSE - two variables do not have the same value

This method overrides **equals()** in the class java.lang.Object.

Constructor

JipFloat() is used to construct a JipFloat object:

```
JipFloat(float value);
```

- *value* is the floating-point value value to be assigned to the object

8.3 JipDouble Class

The JipDouble class implements the JipValue interface, described in Section 7.2, and includes methods and a constructor, as indicated below.

Methods

This class inherits the JipValue interface methods, detailed in Section 7.2.1. In this JipValue implementation, the **getValue()** method returns an object of the standard Java type **java.lang.Double**.

In addition, the class has the following method of its own.

equals() compares the **double** variable with another **double** variable:

```
boolean equals(java.lang.Object obj);
```

- *obj* is the object representing the **double** variable to compare with

The returned boolean indicates the outcome:

- TRUE - two variables have the same value
- FALSE - two variables do not have the same value

This method overrides **equals()** in the class java.lang.Object.

Constructor

JipDouble() is used to construct a JipDouble object:

```
JipDouble(double value);
```

- *value* is the double-precision floating-point value to be assigned to the object

8.4 JipString Class

The JipString class implements the JipValue interface, described in Section 7.2, and includes methods and a constructor, as indicated below.

Methods

This class inherits the JipValue interface methods, detailed in Section 7.2.1. In this JipValue implementation, the **getValue()** method returns an object of the standard Java type **java.lang.String**.

In addition, the class has the following method of its own.

equals() compares the string variable with another string variable:

```
boolean equals(java.lang.Object obj);
```

- *obj* is the object representing the string variable to compare with

The returned boolean indicates the outcome:

- TRUE - two variables have the same value
- FALSE - two variables do not have the same value

This method overrides **equals()** in the class java.lang.Object.

Constructor

JipString() is used to construct a JipString object:

```
JipString(java.lang.String value);
```

value is the string value to be assigned to the object.

8.5 JipTable Class

The JipTable class implements the JipValue interface, described in Section 7.2, and includes methods and a constructor, as indicated below. An object constructed from this class represents a fragment of a table of JIP values.

Methods

This class inherits the JipValue interface methods, detailed in Section 7.2.1. In this JipValue implementation, the **getValue()** method returns an object of the standard Java type **java.lang.Object**.

The class also inherits the following methods from the standard class java.lang.Object: **equals()**, **getClass()**, **hashCode()**, **notify()**, **notifyAll()**, **toString()**, **wait()**.

In addition, the class has the following methods of its own.

getRows() returns a map of the entries in the table, referenced by their index:

```
java.util.Map<java.lang.Short,JipValue> getRows();
```

addRow() adds an entry to the table fragment:

```
void addRow(short index, JipValue value);
```

- *index* is the index of the entry to be added
- *value* is the value of the entry to be added

If the table fragment already contains an entry with the specified index, the existing value of the entry is replaced by the specified value.

addRows() adds a set of rows to the table fragment:

```
void addRows(java.util.Map<java.lang.Short,JipValue> rows);
```

- *rows* is a map of table entries containing the new rows (the map is as returned by **getRows()**).

getRemaining() returns the number of entries remaining in the table after the last entry in the fragment:

```
int getRemaining();
```

getLast() returns the index of the last entry present in the table fragment:

```
int getLast();
```

getVersion() returns the version of the table represented by the object:

```
short getVersion();
```

Constructor

The constructor for this class is:

```
JipTable(short remaining, short version);
```

JipTable() is used to construct a JipTable object (representing a table fragment), where:

- *remaining* is the number of table entries remaining
- *version* is the version number of the table

8.6 JipBlob Class

The JipBlob class implements the JipValue interface, described in Section 7.2, and includes methods and a constructor, as indicated below.

Methods

This class inherits the JipValue interface methods, detailed in Section 7.2.1. In this JipValue implementation, the **getValue()** method returns an object of the type **byte[]** (a byte array).

The class also inherits the following methods from the standard class java.lang.Object: **getClass()**, **hashCode()**, **notify()**, **notifyAll()**, **toString()**, **wait()**.

In addition, the class has the following method of its own.

equals() compares the blob variable with another blob variable:

```
boolean equals(java.lang.Object obj);
```

- *obj* is the object representing the blob variable to compare with

The returned boolean indicates the outcome:

- TRUE - two variables have the same value
- FALSE - two variables do not have the same value

This method overrides **equals()** in the class java.lang.Object.

Constructor

JipBlob() is used to construct a JipBlob object:

JipBlob(byte[] *data***);**

- *data* is the byte array of values to be assigned to the object

9. Java Package com.nxp.jip.service

This chapter details the Java package **com.nxp.jip.service** which is supplied as part of the Java JIP API. This package consists of the following interfaces:

- **JenNetIPNetwork.NodeDiscoveryListener** - see Section 9.1
- **Service.TableGetListener** - see Section 9.2

The referenced sections detail the methods associated with each interface. In addition, the classes of the package are detailed in Section 9.3.

9.1 JenNetIPNetwork.NodeDiscoveryListener Interface

This section describes the JenNetIPNetwork.NodeDiscoveryListener interface, which consists only of methods and defines an object which is capable of receiving node discovery events.

9.1.1 JenNetIPNetwork.NodeDiscoveryListener Interface Methods

The JenNetIPNetwork.NodeDiscoveryListener interface methods are listed below, along with their page references:

Method	**Page**
nodeAdded	182
nodeRemoved	183

nodeAdded

void nodeAdded(Node *node*);

Description

This method is invoked either when a new wireless network node is detected during discovery or when a node joins the network (which is being actively monitored).

Parameters

node	Description of node

Returns

None

Exceptions

None

nodeRemoved

void nodeRemoved(Node *node*);

Description

This method is invoked either when a new wireless network node is not detected during discovery or when a node leaves the network (which is being actively monitored).

Parameters

node	Description of node

Returns

None

Exceptions

None

9.2 Service.TableGetListener Interface

This section describes the Service.TableGetListener interface, which consists only of a method.

9.2.1 Service.TableGetListener Interface Method

The Service.TableGetListener interface method is listed below, along with its page reference:

Method	Page
rowAdded	185

rowAdded

void rowAdded(short *index*, **JipValue** *value***);**

Description

This method is invoked when a new entry is added to a MIB on a wireless network node.

Parameters

index	Index of new entry within MIB
value	Value of new MIB entry

Returns

None

Exceptions

None

9.3 Classes of com.nxp.jip.service

This section details the classes of the **com.nxp.jip.service** package.

The package includes the following classes:

- **JenNetIPNetwork** - see Section 9.3.1
- **Module** - see Section 9.3.2
- **Node** - see Section 9.3.3
- **Service** - see Section 9.3.4
- **VariableInst** - see Section 9.3.5

9.3.1 JenNetIPNetwork Class

The JenNetIPNetwork class has its own methods, as indicated below.

Methods

This class inherits the TrapListener interface method, detailed in Section 7.10.1.

It also inherits the following methods from the standard class java.lang.Object: **equals()**, **getClass()**, **hashCode()**, **notify()**, **notifyAll()**, **toString()**, **wait()**.

In addition, the class has the following methods of its own.

getIPv6Prefix() returns the IPv6 prefix (subnet) of the wireless network.

```
long getIPv6Prefix();
```

getCoordinator() returns a description of the Co-ordinator of the wireless network.

```
Node getCoordinator();
```

getNode() returns a description of the specified node in the wireless network.

```
Node getNode(java.net.InetSocketAddress addr);
```

getAllNodes() returns descriptions of all the currently available nodes in the wireless network, including the Co-ordinator.

```
java.util.Collection<Node> getAllNodes();
```

getAllChildren() returns descriptions all the currently available nodes in the wireless network, excluding the Co-ordinator.

```
java.util.Collection<Node> getAllChildren();
```

getNodes() returns descriptions of all the wireless network nodes whose class matches the specified class(es).

```
java.util.Collection<Node> getNodes(int device_class);
```

- *device_class* is the Device ID filter

getDeviceClassList() returns a list of the distinct Device IDs of the nodes in the wireless network. Note that the Co-ordinator has a special Device ID with the value Service.COORDINATOR_DEVICE_CLASS.

```
java.util.Collection<java.lang.Integer> getDeviceClassList();
```

There are two versions of the method **discoverNodes()**.

This **discoverNodes()** obtains a list of the nodes in the wireless network.

```
java.util.Collection<Node> discoverNodes();
```

This **discoverNodes()** obtains a list of the nodes in the wireless network. As data for each node is fetched, the specified discovery listener will be notified.

```
java.util.Collection<Node> discoverNodes(
JenNetIPNetwork.NodeDiscoveryListener discoveryListener);
```

- *discoveryListener* identifies the discovery listener to be notified (if no listener, set to null)

addNode() adds a new node to the wireless network and returns a description of the node. If a node with the same address has previously been discovered, details of the pre-existing node will be returned instead. A nodeAdded event will be generated for all registered discovery listeners.

```
Node addNode(java.net.InetSocketAddress sockAddr,
                  int deviceClass);
```

- *sockAddr* is the address of the node to be added
- *deviceClass* is the Device ID of the node to be added

removeNode() removes the node with the specified address from the wireless network. The method will return TRUE if the node was successfully removed (FALSE otherwise). A nodeRemoved event will be generated for all registered discovery listeners.

```
boolean removeNode(java.net.InetSocketAddress sockAddr);
```

- *sockAddr* is the address of the node to be removed

startMonitoring() registers a node discovery listener which is to be notified of discovery events and starts monitoring the wireless network for changes (if it is not already being monitored).

```
void startMonitoring(
      JenNetIPNetwork.NodeDiscoveryListener listener);
```

- *listener* is the node discovery listener to be registered

stopMonitoring() unregisters a node discovery listener, so it will no longer be notified of discovery events, and stops monitoring the wireless network for changes (if no listeners remain).

```
void stopMonitoring(
      JenNetIPNetwork.NodeDiscoveryListener listener);
```

- *listener* is the node discovery listener to be unregistered

setMonitorInterval() sets the interval (in milliseconds) at which the wireless network will be polled for changes while being monitored.

```
void setMonitorInterval(long milliseconds);
```

- *milliseconds* is the interval to be set, in milliseconds

shutdown() shuts down the wireless network, stopping all polling and attempting to unregister any outstanding traps.

```
void shutdown();
```

9.3.2 Module Class

The Module class is used identify a MIB and its type. Instances of this class are constructed in and returned by the Node class methods. The Module class has its own methods, as indicated below.

Methods

The class inherits the following methods from the standard class java.lang.Object: **equals()**, **getClass()**, **hashCode()**, **notify()**, **notifyAll()**, **toString()**, **wait()**.

It also has the following methods of its own.

getModuleId() obtains the identifier of a MIB:

```
int getModuleId();
```

getName() obtains the name of a MIB:

```
java.lang.String getName();
```

getVariables() obtains a list of the variables in a MIB (it initiates a query if the MIB does not exist within the cache):

```
java.util.List<VariableInst> getVariables();
```

The above method can throw the exception JipException.

There are two cases of the **getVariable()** method.

This **getVariable()** obtains the variable with the specified name within a MIB (it initiates a query if the MIB variable does not exist within the cache):

```
VariableInst getVariable(java.lang.String varName);
```

- *varName* is a string representing the name of the variable

If no such variable exists, the method will return null.

The above method can throw the exception JipException.

This **getVariable()** obtains the variable with the specified index value within a MIB (it initiates a query if the MIB variable does not exist within the cache):

```
VariableInst getVariable(int index);
```

- *index* is the value of index of the variable within the MIB

If no such variable exists, the method will return null.

The above method can throw the exception JipException.

9.3.3 Node Class

The Node class is used identify a node and its Device ID. Instances of this class are constructed in and returned by the network discovery process. The Node class has its own methods, as indicated below.

Methods

The class inherits the following methods from the standard class java.lang.Object: **getClass()**, **notify()**, **notifyAll()**, **toString()**, **wait()**.

It also has the following methods of its own.

getDeviceClass() obtains the Device ID of a node:

```
int getDeviceClass();
```

getAddress() obtains the address of a node:

```
java.net.InetSocketAddress getAddress();
```

getModules() obtains a list of the MIBs present on a node (it initiates a query if the device does not exist within the cache).

```
java.util.List<Module> getModules();
```

The above method can throw the exception JipException.

getModuleByIndex() obtains the MIB with the given index value on a node (it initiates a query if the MIB does not exist within the cache):

```
Module getModuleByIndex(int moduleIndex);
```

- *moduleIndex* is the index value of the MIB on the node

If no such MIB exists, the method will return null.

The above method can throw the exception JipException.

getModuleByIndex() obtains the MIB with the given identifier on a node (it initiates a query if the MIB does not exist within the cache):

```
Module getModuleById(int moduleId);
```

- *moduleId* is the identifier of the MIB on the node

If no such MIB exists, the method will return null.

The above method can throw the exception JipException.

getModuleByName() obtains the MIB with the given name on a node (it initiates a query if the MIB does not exist within the cache):

```
Module getModuleByName(java.lang.String moduleName);
```

- *moduleName* is a string representing the name of the MIB on the node

If no such MIB exists, the method will return null.

The above method can throw the exception JipException.

toString() returns a string representation of the object's address and Device ID:

```
java.lang.String toString();
```

This method overrides **toString()** in the class java.lang.Object.

hashCode() returns the hash code of an inet address:

```
int hashCode();
```

This method overrides **hashCode()** in the class java.lang.Object.

equals() compares the inet socket addresses of two nodes:

```
boolean equals(java.lang.Object obj);
```

- *obj* is the object representing the node to compare with

The returned boolean indicates the outcome:

- TRUE - nodes have same address
- FALSE - nodes have different addresses

This method overrides **equals()** in the class java.lang.Object.

9.3.4 Service Class

The Service class is used identify a node and its Device ID, which represents the core JenNet-IP service. The Service class has its own field, methods and a constructor, as indicated below.

Field

This class has only one field:

Field	Type	Description
COORDINATOR_DEVICE_CLASS	static int	Represents the Co-ordinator of a JenNet-IP wireless network

Table 7: Service Class Field

Methods

The class inherits the following methods from the standard class java.lang.Object: **equals()**, **getClass()**, **hashCode()**, **notify()**, **notifyAll()**, **toString()**, **wait()**.

It also has the following methods of its own.

createNetwork() finds the Co-ordinator of the JenNet-IP wireless network to which the specified node belongs and returns a JenNetIPNetwork object which describes the network:

```
JenNetIPNetwork createNetwork(
        java.net.InetSocketAddress nodeAddress);
```

- *nodeAddress* is the address of the node for which the network is sought

The above method can throw the exception JipException or java.net.UnknownHostException.

findCoordinator() returns a description of the Co-ordinator of the JenNet-IP wireless network to which the specified node belongs:

```
Node findCoordinator(
       java.net.InetSocketAddress nodeAddress);
```

- *nodeAddress* is the address of the node for which the Co-ordinator is sought

The above method can throw the exception JipException or java.net.UnknownHostException.

getCache() obtains the cache instance used by the service:

```
Cache getCache();
```

shutdown() shuts down the service, stopping all polling and attempting to unregister any outstanding traps.

```
void shutdown();
```

Constructor

Service() used to construct a new Service instance, which uses the supplied JIP API implementation to communicate with the network.

```
Service(JIP jip);
```

- *jip* is the JIP Service instance to use

9.3.5 VariableInst Class

The VariableInst class is used to represent a JIP Variable entity, including the variable's value. Instances of this class are constructed in and returned by the Module class **getVariable()** methods. The VariableInst class has its own methods, as indicated below.

Methods

The class inherits the following methods from the standard class java.lang.Object: **equals()**, **getClass()**, **hashCode()**, **notify()**, **notifyAll()**, **toString()**, **wait()**.

It also has the following methods of its own.

getVarType() obtains the type of the MIB variable:

```
JipTypes.VariableType getVarType();
```

isDisabled() indicates whether the MIB variable is disabled:

```
boolean isDisabled();
```

The returned boolean indicates the outcome:

- TRUE - variable is disabled
- FALSE - variable is enabled

getValue() obtains the value of the MIB variable (if available):

```
JipValue getValue();
```

The above method can throw the exception JipException.

setValue() sets the MIB variable to the specified value:

```
void setValue(JipValue value);
```

- *value* is the value to which the variable will be set

The above method can throw the exception JipException.

setUpdateInterval() sets the polling interval, in milliseconds, for the MIB variable:

```
void setUpdateInterval(long milliseconds);
```

- *milliseconds* is the time-interval, in milliseconds, between consecutive polls of the variable

trap() registers a trap so that the specified trap listener will receive notifications of changes to the MIB variable:

```
void trap(TrapListener listener);
```

- *listener* is the trap listener with which to register trap

untrap() unregisters a trap so that no further notifications of changes to the MIB variable will be generated for the specified trap listener:

```
void trap(TrapListener listener);
```

- *listener* is the trap listener with which to unregister trap

startPoll() starts periodic polling of the MIB variable with the specified period, where the results are passed to the specified trap listener:

```
void startPoll(TrapListener listener, long interval);
```

- *listener* is the trap listener which is to receive polling results
- *interval* is the polling period, in milliseconds

stopPoll() stops periodic polling of the MIB variable:

```
void stopPoll(TrapListener listener);
```

- *listener* is the trap listener which receives the polling results

addListener() registers a trap listener:

```
void addListener(TrapListener listener,
                 long maxInterval,
                 boolean isTrap);
```

- *listener* is the trap listener to be registered
- *maxInterval* is the maximum polling period, in milliseconds, supported by the trap listener
- *isTrap* enables/disables the trap listener to receive trap notifications:
  - TRUE - allow to receive trap notifications
  - FALSE - do not allow to receive trap notifications

removeListener() unregisters a trap listener:

```
void removeListener(TrapListener listener);
```

- *listener* is the trap listener to be unregistered

update() obtains the latest value of the MIB variable:

```
JipValue update();
```

The above method can throw the exception JipException.

isTable() determines whether the MIB variable is a table:

```
boolean isTable();
```

The returned boolean indicates the outcome:

- TRUE - variable is a table
- FALSE - variable is not a table

getName() obtains the name of the MIB variable:

```
java.lang.String getName();
```

getNode() obtains an object representing the node that contains the MIB variable:

```
Node getNode();
```

getVariableRecord() obtains the MIB variable record for the MIB variable:

```
VariableRecord getVariableRecord();
```

getModuleRecord() obtains the MIB record for the MIB which contains the variable:

```
ModuleRecord getModuleRecord();
```

isConstant() determines whether the MIB variable has the access type CONST:

```
boolean isConstant();
```

The returned boolean indicates the outcome:

- TRUE - variable’s access type is CONST
- FALSE - variable’s access type is not CONST

isReadOnly() determines whether the MIB variable has the access type READ_ONLY:

```
boolean isReadOnly();
```

The returned boolean indicates the outcome:

- TRUE - variable’s access type is READ_ONLY
- FALSE - variable’s access type is not READ_ONLY (can be written to)

10. Java Package com.nxp.jip.service.persist

This chapter details the Java package **com.nxp.jip.service. persist** which is supplied as part of the Java JIP API. This package consists of only one class, **XmlPersistence**, which is described in Section 10.1 and is concerned with saving/retrieving wireless network context data to/from an XML file in non-volatile memory.

10.1 XmlPersistence Class

The XmlPersistence class includes methods and a constructor, as indicated below.

Methods

The class inherits the following methods from the standard class java.lang.Object: **equals()**, **getClass()**, **hashCode()**, **notify()**, **notifyAll()**, **toString()**, **wait()**.

It also has the following methods of its own.

saveNetwork() saves data on the nodes of a wireless network to an XML file, where *network* identifies the network:

```
void saveNetwork(JenNetIPNetwork network);
```

loadNetwork() loads the node data from an XML file into a service and returns the identity of the wireless network which contains the nodes:

```
JenNetIPNetwork loadNetwork(Service service);
```

- *service* specifies the service.

The above method can throw the exception java.io.FileNotFoundException.

saveDefinitions() saves the device classes and MIBs stored in a Cache instance to an XML file:

```
void saveDefinitions(Cache cache);
```

- *cache* specifies the cache whose contents are to be saved.

loadDefinitions() loads device classes and MIB definitions from an XML file into a Cache instance:

```
void loadDefinitions(Cache cache);
```

- *cache* specifies the cache.

The above method can throw the exception java.io.FileNotFoundException.

Constructor

XmlPersistence() is used to construct an XmlPersistence instance:

```
XmlPersistence(java.lang.String filename);
```

- *filename* represents the path to the XML file to be used by the instance

Part IV: Appendices

A. JenNet-IP Browser

The JenNet-IP Browser is an example of a generic engineering application which can be used on a LAN/WAN device in order to monitor and control a JenNet-IP WPAN via an IP connection. A Java version of the application, which can be run on the LAN/WAN device (such as a PC), is supplied as an executable in the JenNet-IP SDK:

JenNet-IP-Browser-x.y.z.jar

A C-version of the application is provided in the firmware of the Linksys or Buffalo router for JenNet-IP demonstration systems and runs on the router. Assuming your PC has an IP connection to the router, this version of the application can be accessed by directing your web browser to:

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

You can write your own versions of these applications using the Java JIP API and C JIP API, detailed in this manual.

This appendix provides useful preliminary information for getting started with the Java version of the JenNet-IP Browser.

> **Note:** The Java JenNet-IP Browser is fully described in an online manual which is embedded in the application and which can be accessed via **Help > Online manual**.

A.1 Browser Functionality

The JenNet-IP Browser application allows you to:

- Browse nodes in the wireless network of a JenNet-IP system
- View the MIBs on a node
- Monitor changing MIB variable values, using trap, poll and manual methods
- Write new values to MIB variables (write permissions allowing)
- Diagnose node issues using log details generated by the browser

A.2 Pre-requisites

To run the Java version of the JenNet-IP Browser application, your must have the following on your PC/workstation:

- Windows (XP, Vista or 7), Linux or Mac OSX
- Java 1.6 or higher

Normally, an IPv6 connection is used between the PC/workstation and WPAN. Preparing the IPv6 connection is described in Appendix A.2.1.

Alternatively, an IPv4 connection can be used between the PC/workstation and WPAN. Configuring an IPv4 connection is described in Appendix A.2.2.

A.2.1 Preparing an IPv6 Connection

In order to use an IPv6 connection between the PC/workstation and WPAN, you will need:

- IPv6 enabled on the machine (enabled by default in Windows Vista and 7)
- IPv6 address of the WPAN Co-ordinator

Procedures for these requirements are presented below.

To enable IPv6 in Windows XP

This procedure may only be required if you are using Windows XP, as IPv6 is enabled by default in Windows Vista and 7.

1. Launch a command window on your PC/workstation.
2. Enter the following at the command prompt:

```
netsh interface ipv6 install
```

3. Press the <Enter> key and wait for the command prompt to re-appear.

To obtain and enter the IPv6 address of the Co-ordinator

This procedure assumes an IP connection between your PC/workstation and a Linksys or Buffalo router used for JenNet-IP demonstration systems.

1. Access the web version of the JenNet-IP Browser by entering the following address in your web browser: **http://192.168.11.1/cgi-bin/Browser.cgi**
2. Once this browser has detected and displayed the nodes in the wireless network, click on **Border-Router** (which also acts as the Co-ordinator).
3. Copy or make a note of the IPv6 address for the Co-ordinator, which is shown in the orange bar near the top of the resulting page.
4. Run the Java version of the JenNet-IP Browser, press the **Discover** button and enter the IPv6 address into the **IP address** field of the resulting window.

A.2.2 Preparing an IPv4 Connection

In order to use an IPv4 connection between the PC/workstation and WPAN, follow the procedure below:

1. Run the Java version of the JenNet-IP Browser.
2. Follow the menu path **Configure > Network**. The **Configure network properties** dialogue box appears.
3. In the dialogue box:
   - Tick the **IPv4** checkbox.
   - In the **IPv4 address** field, enter the IPv4 address of the Border-Router of the wireless network.

   Leave all other fields at their default values (unless specific values are required).
4. Click **OK**.

B. JenNet-IP (JIP) CLI

This appendix details the Command Line Interface (CLI) which is supplied with the JenNet-IP C software. The JenNet-IP (JIP) CLI is intended for use during system development to allow a JenNet-IP WPAN to be accessed (monitored and controlled) from the command line on a LAN/WAN device, such as a PC.

The JIP CLI commands are issued on a LAN/WAN device and may be executed on this or another LAN/WAN device that has an IP connection to the target WPAN. Typically, the commands are:

1. entered by the user on a LAN/WAN device via Secure Shell (SSH), using a client such as PuTTY or Tera Term
2. executed on the LAN/WAN side of the the Border-Router of the target WPAN

A typical arrangement is illustrated in Figure 9 below.

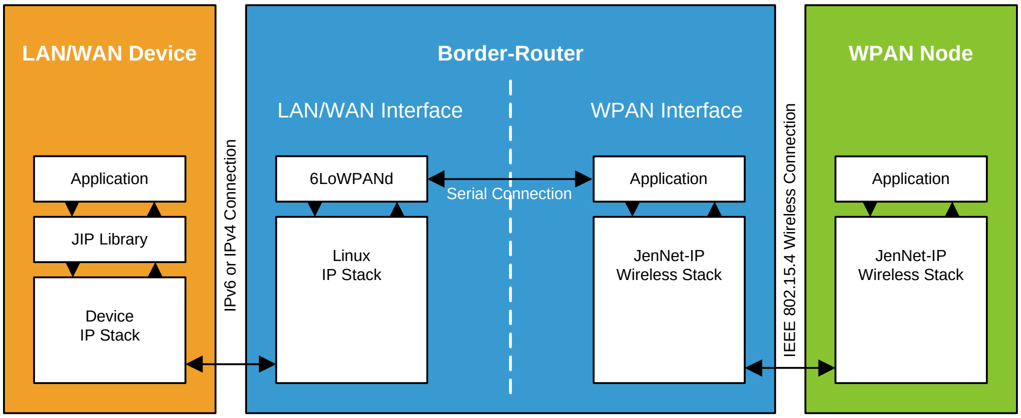

Figure 9: JIP CLI in JenNet-IP System

No special installation of the JIP CLI is required. It can be run on any LAN/WAN device on which the JIP C library software is installed.

The CLI is initially run by entering the following on the command line:

```
JIP -6 <IPv6 address of Co-ordinator>
```

The required commands can then be entered on the command line, as required. The commands can also be concatenated to form a script which performs a series of operations on the WPAN.

The JIP CLI commands are detailed in Appendix B.1 and examples of their use are provided in Appendix B.2.

B.1 Commands

The JIP CLI commands are listed and described in Table 8 below.

A list of brief command descriptions can also be displayed in the command window using the `help` command.

The commands and parameters can be tab-completed.

Command	Parameter	Description
`JIP -6`	`<IPv6 address>`	Run the JIP CLI in order to interact with the wireless network which has the Co-ordinator with the specified IPv6 address
`quit`		Terminate execution of the JIP CLI
`help`		Display the Help which provides a list of commands and brief descriptions
`?`		Display the Help which provides a list of commands and brief descriptions
`discover`		Perform a network discovery to obtain information about the nodes of the WPAN, including: • IPv6 address • Device ID • MIBs - names and IDs • MIB variables - names, indices, access types and values
`save`	`<filename>`	Save the current network context data (as returned by the `discover` command) to the specified file in the file-system
`load`	`<filename>`	Load the previously saved network context data from the specified file (in the file-system) into RAM
`ipv6`	`<IPv6 address>`	Filter the network information that has been previously obtained using the `discover` or `load` command by retaining only the information about the node(s) with the specified IPv6 address - if this is a multicast address, information about multiple nodes will be retained (subsequently this information can only be used with the `set` command)
`device`	`<Device ID>`	Filter the network information that has been previously obtained using the `discover` command by retaining only the information about the node(s) with the specified Device ID
`mib`	`<MIB name or ID>`	Filter the network information that has been previously obtained using the `discover` command by retaining only the information the MIB with the specified name or ID (which may be present on multiple nodes)

Table 8: JIP CLI Commands

Command	Parameter	Description
`var`	`<MIB variable name or index>`	Filter the network information that has been previously obtained using the `discover` command by retaining only the information about the MIB variable with the specified name or index value (which may be present in multiple MIBs and/or on multiple nodes)
`print`		Print the network information that has been previously obtained using the `discover` command and possibly filtered using the `ipv6`, `device`, `mib` and `var` commands
`get`		Get the MIB variable values that have been previously obtained using the `discover` command and that remain after any filtering has been applied using the `ipv6`, `device`, `mib` and `var` commands - the values are obtained from the network nodes
`set`	`<value>`	Set the values of the MIB variables that have been previously detected using the `discover` command and that remain after any filtering has been applied using the `ipv6`, `device`, `mib` and `var` commands - the value is set on the network node(s)

Table 8: JIP CLI Commands

Note: Once one of the filter commands `ipv6`, `device`, `mib` or `var` has been issued, the applied filter can be removed by issuing the command again but with no parameter.

B.2 Example Usage

This section describes the use of the JIP CLI commands that are listed and described in Appendix B.1.

B.2.1 Running the JIP CLI

The JIP CLI is run by entering the following command in the command window:

```
JIP -6 <IPv6 address>
```

where `<IPv6 address>` is the IPv6 address of the Co-ordinator of the wireless network that the CLI will be used to access.

For example:

```
JIP -6 `cat /tmp/6LoWPANd.tun0`
```

Once the CLI is running, the command prompt will become:

```
JIP : >
```

Note: Execution of the JIP CLI can be stopped at any time using the command `quit`.

B.2.2 Discovering the Wireless Network

The contents of the wireless network can be ‘discovered’ using the command:

```
discover
```

While the discovery is taking place, the following will be displayed in the command window (and no other commands can be entered):

```
Running discovery...
```

Once the discovery has completed, the number of nodes discovered and the time taken by the discovery (in milliseconds) will be displayed, and the command prompt will then be displayed again. For example:

```
Discovered 2 devices (Time: 3331 ms)
```

The `discover` command obtains the following information about each discovered node in the network:

- IPv6 address of node
- Device ID of node
- List of MIBs on node - for each MIB, the following information is provided:
  - MIB name
  - MIB ID
  - List of MIB variables - for each variable, the following information is provided: name, index value within MIB, permissible access types and data type (note that the variables are not read)

Once the discovery has completed, this ‘network context data’ is held in RAM on the LAN/WAN device (and is not automatically displayed). The information can then be:

- saved to the local file-system, as described in Appendix B.2.3
- printed to the screen, as described in Appendix B.2.5, but may first be filtered by IPv6 address, Device ID, MIB or MIB variable, as described in Appendix B.2.4

B.2.3 Persisting Network Context Data

Once the wireless network has been discovered, as described in Appendix B.2.2, the obtained network context data can be saved from RAM into the local file-system, so that this data can be recovered if it is lost from RAM.

The data can be saved to a file in NVM using the following command:

```
save <filename>
```

where `<filename>` is the path and name of the file in which the data is to be saved.

The saved data can later be loaded from NVM into RAM using the following command:

```
load <filename>
```

Use of these commands may be useful if the JIP CLI is stopped and then restarted:

1. While the JIP CLI is running, the `save` command is used to store the network context data in NVM.
2. Execution of the CLI is stopped using the `quit` command - the network context data held in RAM is then lost.
3. The JIP CLI is later run again using the `JIP -6` command.
4. The saved network context data can then be recovered from NVM and loaded into RAM using the `load` command, avoiding the need to discover the network again.

B.2.4 Applying Filters

The network context data that is held in RAM can be filtered such that subsequent commands only apply to the remaining data. For example, the data following a discovery can be filtered such that only data for nodes with a certain Device ID remain. Following this filtering, a `print` command (see Appendix B.2.5) will only display the subset of data remaining.

Note 1: After a filter has been applied, the full network context data is still held in RAM but only a subset of this data is 'active' and accessible to subequent commands.

Note 2: Once one of the filter commands `ipv6`, `device`, `mib` or `var` has been issued, the applied filter can be removed by issuing the command again but with no parameter.

The network context data can be filtered by IPv6 address, Device ID, MIB and MIB variable, as described below. In each case, the filter applies to the currently 'active' network context data (that remains accessible after any previous filtering).

IPv6 Address Filter

The currently active data can be filtered by IPv6 address using the following command:

```
ipv6 <IPv6 address>
```

where `<IPv6 address>` is the IPv6 address of the nodes for which context data is to remain active.

Normally, specifying an IPv6 address will result in the data for only one node remaining active. However, the specified address can be a multicast address, in which case the data for all nodes in the corresponding multicast group will remain active. Subsequently, this active multicast data can only be used with the `set` command, when setting the same MIB variable value on multiple nodes.

Device ID Filter

The currently active data can be filtered by Device ID using the following command:

```
device <Device ID>
```

where `<Device ID>` is the Device ID of the nodes for which context data is to remain active.

Thus, only data on the nodes with the specified Device ID will then be accessible.

MIB Filter

The currently active data can be filtered by MIB using the following command:

```
mib <MIB name or ID>
```

where `<MIB name or ID>` is the name or ID of the MIB for which node context data is to remain active.

If the specified MIB is present on multiple nodes for which context data was previously active, the relevant data for all these nodes will remain active.

For any node which contains the specified MIB, the context data that will remain active includes:

- IPv6 address of node
- Device ID of node
- Details of specified MIB, including MIB variables

MIB Variable Filter

The currently active data can be filtered by MIB variable using the following command:

```
var <Variable name or index>
```

where `<Variable name or index>` is the name or index value of the MIB variable for which node context data is to remain active.

If the specified MIB variable is present in MIBs on multiple nodes for which context data was previously active, the relevant data for all these nodes will remain active.

For any node which contains the specified MIB variable, the context data that will remain active includes:

- IPv6 address of node
- Device ID of node

- Details of relevant MIB, including the specified MIB variable but excluding all other MIB variables

B.2.5 Printing Network Information

The currently active network context data (see Appendix B.2.4) can be displayed in the command window using the following command:

```
print
```

If no filtering has been applied to the network context data held in RAM, the full network context data will be displayed, as detailed in Appendix B.2.2.

Example output is:

```
Node: fd04:bd3:80e8:2:215:8d00:12:147d    Device ID: 0x80100001
  Mib: 'Node', ID 0xffffff00
    Var: 'MacAddr', Index 0
      E_JIP_VAR_TYPE_UINT64, E_JIP_ACCESS_TYPE_CONST, E_JIP_SECURITY_NONE
        Value: ?

    Var: 'DescriptiveName', Index 1
      E_JIP_VAR_TYPE_STR, E_JIP_ACCESS_TYPE_READ_WRITE, E_JIP_SECURITY_NONE
        Value: ?

    Var: 'Version', Index 2
      E_JIP_VAR_TYPE_STR, E_JIP_ACCESS_TYPE_CONST, E_JIP_SECURITY_NONE
        Value: ?

    Var: 'TxPower', Index 3
      E_JIP_VAR_TYPE_UINT8, E_JIP_ACCESS_TYPE_CONST, E_JIP_SECURITY_NONE
        Value: ?

  Mib: 'JenNet', ID 0xffffff01
    Var: 'DeviceType', Index 0
      E_JIP_VAR_TYPE_UINT32, E_JIP_ACCESS_TYPE_CONST, E_JIP_SECURITY_NONE
        Value: ?

    Var: 'Parent Interface', Index 1
      E_JIP_VAR_TYPE_UINT64, E_JIP_ACCESS_TYPE_READ_ONLY, E_JIP_SECURITY_NONE
        Value: ?

    Var: 'TreeVersion', Index 2
      E_JIP_VAR_TYPE_UINT32, E_JIP_ACCESS_TYPE_READ_ONLY, E_JIP_SECURITY_NONE
        Value: ?

    Var: 'SubTreeNodes', Index 3
      E_JIP_VAR_TYPE_UINT32, E_JIP_ACCESS_TYPE_READ_ONLY, E_JIP_SECURITY_NONE
        Value: ?

    Var: 'NetworkTable', Index 4
      E_JIP_VAR_TYPE_TABLE_BLOB, E_JIP_ACCESS_TYPE_READ_ONLY, E_JIP_SECURITY_NONE
        Value:

    Var: 'LastChange', Index 5
      E_JIP_VAR_TYPE_BLOB, E_JIP_ACCESS_TYPE_READ_ONLY, E_JIP_SECURITY_NONE
        Value: ?
    Var: 'NeighbourTable', Index 6
      E_JIP_VAR_TYPE_TABLE_BLOB, E_JIP_ACCESS_TYPE_READ_ONLY, E_JIP_SECURITY_NONE
        Value: ?

    Var: 'Depth', Index 7
      E_JIP_VAR_TYPE_UINT16, E_JIP_ACCESS_TYPE_READ_ONLY, E_JIP_SECURITY_NONE
        Value: ?

  Mib: 'Groups', ID 0xffffff02
    Var: 'Groups', Index 0
      E_JIP_VAR_TYPE_TABLE_BLOB, E_JIP_ACCESS_TYPE_READ_ONLY, E_JIP_SECURITY_NONE
        Value: ?

    Var: 'AddGroup', Index 1
      E_JIP_VAR_TYPE_BLOB, E_JIP_ACCESS_TYPE_READ_WRITE, E_JIP_SECURITY_NONE
        Value: ?

    Var: 'RemoveGroup', Index 2
      E_JIP_VAR_TYPE_BLOB, E_JIP_ACCESS_TYPE_READ_WRITE, E_JIP_SECURITY_NONE
        Value: ?

    Var: 'ClearGroups', Index 3
      E_JIP_VAR_TYPE_UINT8, E_JIP_ACCESS_TYPE_READ_WRITE, E_JIP_SECURITY_NONE
        Value: ?

  Mib: 'OND', ID 0xffffff03
    Var: 'Images', Index 0
      E_JIP_VAR_TYPE_TABLE_BLOB, E_JIP_ACCESS_TYPE_READ_ONLY, E_JIP_SECURITY_NONE
        Value: ?

    Var: 'DeviceID', Index 1
      E_JIP_VAR_TYPE_UINT32, E_JIP_ACCESS_TYPE_READ_WRITE, E_JIP_SECURITY_NONE
        Value: ?

    Var: 'ChipSet', Index 2
      E_JIP_VAR_TYPE_UINT16, E_JIP_ACCESS_TYPE_READ_WRITE, E_JIP_SECURITY_NONE
        Value: ?

    Var: 'Revision', Index 3
      E_JIP_VAR_TYPE_UINT16, E_JIP_ACCESS_TYPE_READ_WRITE, E_JIP_SECURITY_NONE
        Value: ?

    Var: 'Download', Index 4
      E_JIP_VAR_TYPE_UINT8, E_JIP_ACCESS_TYPE_READ_WRITE, E_JIP_SECURITY_NONE
        Value: ?

    Var: 'LoadImage', Index 5
      E_JIP_VAR_TYPE_UINT8, E_JIP_ACCESS_TYPE_READ_WRITE, E_JIP_SECURITY_NONE
        Value: ?
```

B.2.6 Accessing MIB Variables

The JIP CLI provides commands for accessing MIB variables on network nodes.

The `get` command is used to read the current values of MIB variables and has no parameters.

The `set` command is used to write new values to MIB variables and has the following format:

```
set <value>
```

where `<value>` is the value to be set.

The commands apply to all MIB variables that are included in the currently active network context data (see Appendix B.2.4). Therefore, before using these commands, the particular MIB variable(s) to access must be selected by filtering out the irrelevant MIB variables. Example filtering and accesses are provided below.

Example 1: Reading all MIB variables of a MIB on a particular node

In order to read all MIB variables of a certain MIB on one particular node, the following filtering may be applied (after network discovery):

1. Use the `ipv6` command to filter out all other nodes of the network.
2. Use the `mib` command to filter out all other MIBs on the node.
3. Use the `get` command to read the values of all MIB variables on the selected MIB on the selected node.

Example output from the above procedure is shown below.

```
Reading Node 'fd04:bd3:80e8:2:215:8d00:12:147d' MiB 'OND' Var
'Images': Success (Time: 15ms)
Reading Node 'fd04:bd3:80e8:2:215:8d00:12:147d' MiB 'OND' Var
'DeviceID': Success (Time: 7ms)
Reading Node 'fd04:bd3:80e8:2:215:8d00:12:147d' MiB 'OND' Var
'ChipSet': Success (Time: 7ms)
Reading Node 'fd04:bd3:80e8:2:215:8d00:12:147d' MiB 'OND' Var
'Revision': Success (Time: 7ms)
Reading Node 'fd04:bd3:80e8:2:215:8d00:12:147d' MiB 'OND' Var
'Download': Success (Time: 7ms)
Reading Node 'fd04:bd3:80e8:2:215:8d00:12:147d' MiB 'OND' Var
'LoadImage': Success (Time: 7ms)
```

Example 2: Writing to a certain MIB variable on a particular node

In order to write to a certain MIB variable on one particular node, the following filtering may be applied (after network discovery):

1. Use the `ipv6` command to filter out all other nodes of the network.
2. Use the `mib` command to filter out all irrelevant MIBs on the node.
3. Use the `var` command to filter out all other MIB variables in the relevant MIB on the node.
4. Use the `set` command to write a new value to the selected MIB variable on the node.

Example output from the above procedure is shown below.

```
Setting Node 'fd04:bd3:80e8:2:215:8d00:12:147d' MiB 'Groups' Var 'AddGroup' to
'1500f00f': Success (Time: 11ms)
```

Example 3: Reading a certain MIB variable on all nodes (where present)

In order to read a certain MIB variable on all network nodes where it is present, the following filtering may be applied (after network discovery):

1. Use the `mib` command to filter out all irrelevant MIBs on all network nodes.
2. Use the `var` command to filter out all other MIB variables in the relevant MIB on the nodes.
3. Use the `get` command to read the value of the selected MIB variable on all nodes.

B.2.7 Sequencing and Scripting Commands

It may be necessary to issue the JIP CLI commands in a certain order to obtain the desired results. In particular, the sequence in which the filtering commands (see Appendix B.2.4) are issued may be important.

For example, in order to print the details of a particular MIB on all nodes that have a particular Device ID, the following filtering may be applied (after network discovery):

1. Use the `device` command to filter out all irrelevant nodes of the network.
2. Use the `mib` command to filter out all irrelevant MIBs on all network nodes.
3. Use the `print` command to display the details of the selected MIB on all nodes.

More examples of sequencing commands are provided in Appendix B.2.6.

In addition, commands can be concatenated to form a script on the command line. Within a script:

- All commands appear on a single command line
- Individual commands are separated by semi-colons
- The set of commands is preceded by the command line option `-e`
- The set of commands is enclosed by quotes

For example:

```
JIP -6 ‘cat /tmp/6LoWPANd.tun0’ -e "discover; mib BulbControl; var Mode; set 1"
```

In this example:

- `JIP -6 ‘cat /tmp/6LoWPANd.tun0’` launches the JIP CLI and immediately executes the script which follows
- `discover` performs a network discovery
- `mib BulbControl` selects the MIB called BulbControl on all relevant nodes
- `var Mode` selects the variable called Mode from this MIB
- `set 1` assigns the value ‘1’ to this MIB variable on all relevant nodes

On completion of script execution, the JIP CLI automatically terminates.

This facility allows JIP functionality to be easily embedded in shell scripts.

C. Glossary

The main terms used within this document are defined below.

Term	Description
Address	A numeric value that is used to identify a network device.
API	Application Programming Interface: A set of programming functions that can be incorporated in application code to provide an easy-to-use interface to underlying functionality and resources.
Application	The program that deals with the input/output/processing requirements of the host device, as well as high-level interfacing to the network.
Border-Router	Also known as an Edge-Router. A device which provides a single point of interaction between two networks. The device may perform translation of address or protocol information. In a JenNet-IP system, a Border-Router sits between each WPAN and the LAN.
Channel	A narrow frequency range within the designated radio band - for example, the IEEE 802.15.4 2400-MHz band is divided into 16 channels. A wireless network operates in a single channel which is determined at network initialisation.
Child	A network node which is connected directly to a parent node and for which the parent node provides routing functionality. A child can be an End Device or Router. Also see Parent.
Cluster	A wireless cluster in a JenNet-IP system is a WPAN which is connected to a LAN via a Border-Router device.
Context Data	Data which reflects the current state of a network node. The context data must be preserved during sleep mode.
Co-ordinator	The node through which a wireless network is started, initialised and formed - the Co-ordinator acts as the seed from which the network grows, as it is joined by other nodes. The Co-ordinator also usually provides a routing function. All networks must have one and only one Co-ordinator.
Device ID	32-bit value that indicates the non-networking functionality of a JenNet-IP wireless node (e.g. a type of lamp). Comprises Manufacturer ID, Product ID and Base Type.
End Device	A wireless network node which has no networking role (such as routing) and is only concerned with data input/output/processing. As such, an End Device cannot be a parent.
Host	Generic term for an IP device that creates or consumes data packets.
IPv4	Internet Protocol version 4: The original protocol used on the Internet, still widely used today, employing a 32-bit addressing scheme.
IPv6	Internet Protocol version 6: The latest Internet Protocol (used by 6LoWPAN and JenNet-IP) employing a 128-bit addressing scheme.
IEEE 802.15.4	A standard wireless network protocol that is used as the lowest level of the JenNet-IP software stack. Among other functionality, it provides the physical interface to the wireless network's transmission medium (radio).

Term	Description
JenNet	NXP’s proprietary wireless network protocol which sits on IEEE 802.15.4 in the software stack and provides multi-hop functionality.
Joining	The process by which a device becomes a node of a network. The device transmits a joining request. If this is received and accepted by a parent node (Co-ordinator or Router), the device becomes a child of the parent.
MIB	Management Information Base - a database comprising a table of local variables, held in memory on a wireless network node.
Network Application ID	An application-level network identifier comprising 32 bits unique to the application, defined by the application developer
PAN ID	Personal Area Network Identifier: A 16-bit value that uniquely identifies the wireless network in that all neighbouring networks must have different PAN IDs.
Parent	A network node which allows other nodes (children) to connect to it and provides a routing function for these child nodes. A maximum number of children can be accepted (this limit is user-configurable). A parent can be a Router or the Co-ordinator. Also see Child.
Router	A wireless network node which provides routing functionality (in addition to input/output/processing), if used as a parent node. Also see Routing.
Routing	The ability of a network node to pass messages from one node to another, acting as a stepping stone from the source node to the target node. Routing functionality is provided by Routers and the Co-ordinator. Routing is handled by the network level software and is transparent to the application on the node.
Sleep Mode	An operating state of a node in which the device consumes minimal power. During sleep, the only activity of the node is to time the sleep duration to determine when to wake up and resume normal operation. The total sleep duration is user-configurable. Normally, only End Devices sleep.
Stack	The collection of software layers used to operate a system. The high-level user application is at the top of the stack and the low-level interface to the transmission medium is at the bottom of the stack.
UDP	User Datagram Protocol: Simple message-based connectionless protocol used in IP. Messages in a JenNet-IP system are implemented as UDP packets embedded in the payloads of IPv6 packets.
WPAN	Wireless PAN: A Personal Area Network (PAN) implemented wirelessly through radio communication between nodes.

Revision History

Version	Date	Comments
1.0	6-July-2012	First release
1.1	18-Sept-2012	Name of Java version of JenNet-IP Browser corrected and IPv6 multicast address format clarified
1.2	10-Jan-2013	Updated for the JN516x devices
1.3	15-Aug-2013	Updated for JenNet-IP v1.1 as follows: • C functions eJIP_GroupJoin() and eJIP_GroupLeave() added • Parameter added to C function eJIP_Init() • C structures tsJIP_Context and tsVar modified • Java method setSleepingDeviceTimeout() added to JIP Interface • Optional Flags parameter added to some Java JIP Interface methods

Important Notice

Limited warranty and liability - Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the *Terms and conditions of commercial sale* of NXP Semiconductors.

Right to make changes - NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use - NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

Applications - Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Export control - This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

NXP Laboratories UK Ltd
(Formerly Jennic Ltd)
Furnival Street
Sheffield
S1 4QT
United Kingdom

Tel: +44 (0)114 281 2655
Fax: +44 (0)114 281 2951

For the contact details of your local NXP office or distributor, refer to:

www.nxp.com

For online support resources, visit the Wireless Connectivity TechZone:

www.nxp.com/techzones/wireless-connectivity