JenNet-IP WPAN Stack User Guide

JN-UG-3080
Revision 1.5
11 September 2014

Contents

Preface 13

Organisation 13
Conventions 15
Acronyms and Abbreviations 15
Related Documents 16
Support Resources 16
Trademarks 16

Part I: Concept Information

1. Introduction 19

1.1 Wireless IP 19
1.2 6LoWPAN 21
1.3 Software Architecture 22
1.4 JenNet-IP Software 24
1.4.1 Required Hardware Platforms 24
1.4.2 Development Software 24
1.4.3 Toolchain Software 26
1.5 JenNet-IP User Documentation 26
1.6 Where Now? 27

2. Wireless Network Concepts 29

2.1 Wireless Operation 29
2.1.1 Radio Communication 29
2.1.2 Battery Power 30
2.2 Network Communications 30
2.3 Network Node Types 31
2.4 Network Topology 32
2.5 Network Identity 33
2.6 Node Addressing 34
2.7 Routing 34
2.7.1 Neighbour and Routing Tables 35
2.7.2 Routing Process on a Node 35
2.8 Network Formation and Operation 36
2.9 Other Network Operations 38
2.9.1 Auto-ping 38
2.9.2 Sleep Mode 39

2.9.3 Data Polling 39

3. JenNet-IP System Overview 41

3.1 Hardware Architecture and Components 41

3.1.1 WPAN (Wireless Cluster) 42

3.1.2 LAN 42

3.1.3 Border-Router (WPAN-LAN Router) 43

3.1.4 WAN 43

3.1.5 IP Host 44

3.2 Software Architecture and Components 45

3.2.1 Software Overview 45

3.2.2 Software Components (IPv6 Case) 46

3.2.3 Software Components (IPv4 Case) 49

3.2.4 JenNet-IP Browser 50

3.3 JenNet-IP WPAN Stack 51

3.3.1 Application Level 51

3.3.2 Network Level 52

3.3.3 Physical/Data Link Level 53

3.4 Essential JenNet-IP Concepts 54

3.4.1 MIBs and MIB Variables 55

3.4.2 Traps 55

3.5 Network Data and Standard MIBs 56

3.6 Network Security 56

3.7 JenNet Network Profiles 57

3.8 Fundamental Operations in JenNet-IP 59

3.9 Low-Energy Devices 60

3.9.1 Principles of Low-Energy Devices 60

3.9.2 Configuration of Low-Energy Devices 61

3.9.3 Registering a Low-Energy Device with a WPAN 61

Part II: JenNet-IP Embedded API

4. WPAN Application Development 65

4.1 Starting and Forming a WPAN 65

4.1.1 Performing a Cold Start 65

4.1.2 Performing a Warm Start 69

4.1.3 Fast Commissioning Mode 71

4.1.3.1 Principles of Fast Commissioning 71

4.1.3.2 Coding Fast Commissioning 72

4.2 Storing and Transferring Data 72

4.2.1 Creating a MIB and its Variables 73

4.2.2 Remotely Discovering MIBs 74

4.2.2.1 Obtaining List of MIBs 74

4.2.2.2 Obtaining List of Variables in a MIB 75
4.2.3 Remotely Setting MIB Variable Values 75
4.2.4 Remotely Obtaining MIB Variable Values 76
4.2.5 Remote Monitoring of MIB Variables (using Traps) 77
4.3 Forming Multicast Groups 79
4.4 Obtaining Error Reports 80
4.5 Handling Events 80
4.5.1 Stack Events 80
4.5.2 Data Events 81
4.5.3 Peripheral Events 82
4.6 Entering and Leaving Sleep Mode 82
4.6.1 Entering Sleep Mode 82
4.6.2 Leaving Sleep Mode 83
4.6.3 ‘Stay Awake’ Request 83
4.7 Data Polling 84
4.7.1 Polling Methods 84
4.7.2 Polling Events 84
4.8 Persisting Context Data 86
4.9 Using Low-Energy Devices 86
4.9.1 Implementation on Low-Energy Device 86
4.9.2 Implementation in JenNet-IP WPAN 87
4.9.2.1 On the Co-ordinator 87
4.9.2.2 On a Router 88
4.9.2.3 On a Target Node 89
4.9.3 Persisting Low-Energy Device Registration Data 90

5. JIP Embedded API General Functions 91

5.1 Stack Management Functions 91
v6LP_InitHardware 92
eJIP_Init 93
iJIP_ResumeStack 94
vJIP_Tick 95
vJIP_Sleep 96
u32JIP_GetErrNo 97
vJIP_EnableSecurity 98
vApi_DeleteChild 100
vApi_ConfigureFastCommission 101
eApi_SendNetworkAnnounceEnhanced 102
eApi_SendLowEnergyInform 103
5.2 Stack Mode Functions 104
vApi_SetStackMode 105
u16Api_GetStackMode 106
5.3 Network Profile Functions 107
bJnc_SetJoinProfile 108

bJnc_SetRunProfile 109
vJnc_GetNwkProfile 110
u8GetCurJoinProfile 111
u8GetCurRunProfile 112
bJnc_ChangeJoinProfile 113
5.4 Data Transfer Functions 114
eJIP_Poll 115
i6LP_RecvFrom 116
5.5 IPv6 Address Functions 117
iJIP_CreateInterfaceIdFrom64 118
iJIP_GetOwnDeviceAddress 119
iJIP_GetLastDestinationAddr 120
iJIP_GetLastSourceAddr 121
bJIP_AddGroupAddr 122
bJIP_RemoveGroupAddr 123
5.6 IP Functions 124
vJIP_SetDefaultMaxHopCount 125
vJIP_SetPacketDefragTimeout 126

6. JIP Embedded API MIB Functions 127

6.1 MIB Macros 127
6.1.1 MIB Type Definition Macros 127
START_DEFINE_MIB 128
DEFINE_VAR 129
END_DEFINE_MIB 130
6.1.2 MIB Declaration Macros 131
JIP_START_DECLARE_MIB 132
JIP_CALLBACK 133
JIP_END_DECLARE_MIB 134
6.2 MIB Initialisation Function 135
eJIP_RegisterMib 136
6.3 Local Variable Access Functions 137
vJIP_NotifyChanged 138
vJIP_SetEnabled 139
eJIP_PacketAddData 140
eJIP_AddTrap 141
eJIP_RemoveTrap 142
6.4 Remote Variable Access Functions 143
eJIP_Remote_ID_Set 144
eJIP_Remote_TableGet 146
eJIP_Remote_Trap 148
eJIP_Remote_Untrap 149
eJIP_Remote_QueryMib 150
eJIP_Remote_QueryVar 151

7. JIP Embedded API Callback Functions 153

7.1 General Callback Functions 153

vJIP_ConfigureNetwork 154

bJIP_GroupCallback 155

vJIP_PeripheralEvent 156

vJIP_StackEvent 157

v6LP_DataEvent 159

vJIP_StayAwakeRequest 160

7.2 MIB and Trap Callback Functions 161

vJIP_Remote_SetResponse 162

vJIP_Remote_GetResponse 163

vJIP_Remote_TableGetResponse 164

vJIP_Remote_TrapResponse 166

vJIP_Remote_TrapNotify 167

vJIP_Remote_QueryMibResponse 168

vJIP_Remote_QueryVarResponse 169

vJIP_Remote_DataSent 171

8. JIP Embedded API Structures and Enums 173

8.1 Data Types 173

8.1.1 tsJIP_InitData 173

8.1.2 tsNwkInfo 175

8.1.3 MAC_ExtAddr_s 176

8.1.4 ts6LP_SockAddr 176

8.1.5 tsJIP_StackGroupChange 177

8.1.6 tsJIP_MibDef 177

8.1.7 tsJIP_VarDef 178

8.1.8 tsJIP_MibInst 178

8.1.9 tsJIP_QueryMibResponse 179

8.1.10 tsJIP_QueryVarResponse 179

8.1.11 prSet 180

8.1.12 prGet 181

8.1.13 tsJIP_TableData 182

8.1.14 tsAssocNodeInfo 182

8.1.15 EUI64_s 183

8.1.16 in6_addr 183

8.1.17 tsStackReset 184

8.2 Enumerations 185

8.2.1 teJIP_Device 185

8.2.2 teJIP_VarType 185

8.2.3 teJIP_Access 186

8.2.4 teJIP_AccessType 186

8.2.5 teJIP_PollResponse 187

8.2.6 teJIP_Security 187

8.2.7 teLowEnergyStatus 188

8.3 Events 188

8.3.1 teJIP_StackEvent 188

8.3.2 teJIP_DataEvent 189

8.4 Return Codes 190

8.4.1 teJIP_Status 190

8.4.2 teJenNetStatusCode 191

8.5 Error Codes and Enumerations 192

8.5.1 te6LP_ErrorCode 192

8.5.2 te6LP_ErrorInfo 194

9. JenNet-IP Parameters 197

9.1 JenNet Network Parameters (tsNetworkConfigData) 197

9.2 JenNet Network Profile Parameters (tsNwkProfile) 201

9.3 Stack Parameters 204

Part III: Optional Features

10. Over-Network Download (OND) 209

10.1 OND Terminology 209

10.2 OND Features 209

10.3 General Operation 210

10.4 Image Storage 211

10.5 Multi-Image Bootloader 212

10.6 OND Restrictions for JN5164 212

10.7 OND Process 213

10.7.1 Initiating an OND 213

10.7.2 Downloading an Image 213

10.7.3 Recovering Image Blocks 215

10.8 Incorporating OND into an Application 215

10.8.1 Configuration in Application 215

10.8.2 Initialisation in Application 216

10.8.3 Performing a Download 216

10.9 OND Initialisation Functions 219

eOND_SrvInit 220

eOND_DevInit 221

eOND_SleepingDevInit 222

vOND_SleepConfigure 223

10.10 Building an Application with OND 224

10.10.1 Makefile Modifications 224

10.10.2 Post-Build Modifications (using Checksum Tool) 224

10.10.3 OND Checksum Tool 225

11. Standalone WPAN **227**
11.1 Architecture and Operation 227
11.2 WPAN Formation 227
11.3 IP Extension 228

Part IV: Appendices

A. Notes on JenNet Initialisation **233**
A.1 Routing 233
A.2 Losing a Parent Node (Orphaning) 234
A.2.1 Detecting Orphaning 234
A.2.2 Re-joining the Network 235
A.3 Losing a Child Node 235
A.3.1 End Device Children 235
A.3.2 Router Children 237
A.4 Auto-polling 237

B. Buffer and Ping Recommendations **238**
B.1 Stack Queues and Buffers 238
B.2 End Device Ping Interval 238

C. Handling ICMP Messages **239**

D. Identifiers **241**
D.1 Device ID 241
D.2 Device Type ID 242
D.3 MIB ID 242

E. Network Application ID **243**
E.1 Channel Scan 243
E.2 Route Establishment 244
E.3 Functions 244
vApi_SetUserBeaconBits 245
vApi_RegBeaconNotifyCallback 246
v_6LP_SetUserData 247
v_6LP_SetNwkCallback 248

F. JenNet-IP Data Packet Format **249**

G. JenNet-IP Principles **251**
G.1 Introduction 251
G.1.1 JIP Modules 251
G.1.2 JIP Variables 252
G.1.3 JIP Commands 252
G.2 Discovery 253
G.3 Standard Modules 255
G.3.1 Node Module 255
G.3.2 JenNet Module 256

G.3.3 Groups Module 257
G.3.4 OND Module 259
G.3.5 DeviceID Module 260
G.4 Standard Commands 261
G.4.1 'Get' Request 262
G.4.2 'Get by ID' Request 263
G.4.3 'Get' Response 263
G.4.4 'Set' Request 265
G.4.5 'Set by ID' Request 266
G.4.6 'Set' Response 267
G.4.7 'Query Modules' Request 267
G.4.8 'Query Modules' Response 268
G.4.9 'Query Variables' Request 268
G.4.10 'Query Variables' Response 269
G.4.11 'Trap' Request 270
G.4.12 'Untrap' Request 270
G.4.13 'Trap' Response 270
G.4.14 Trap Notifications 271
G.5 Low-Energy Frames 272
G.6 Enumerations 273
G.6.1 Variable Type Enumerations 273
G.6.2 Access Type Enumerations 274
G.6.3 Status Enumerations 274
H. JenNet-IP Browser 275
H.1 Browser Functionality 275
H.2 Pre-requisites 275
H.2.1 Preparing an IPv6 Connection 276
H.2.2 Preparing an IPv4 Connection 276
I. Memory Heap 277
I.1 Heap Organisation and Use 277
I.2 Heap Error Conditions 277
J. Example Over-Network Download (OND) 278
K. Exception Handling 280
K.1 Exception Types 280
K.2 Exception Handlers 281
K.3 Handler Registration 281
K.4 Stack Frame 282
L. MicroMAC for Low-Energy Devices 283
L.1 Enabling the MicroMAC 283
L.2 Application Coding for the MicroMAC 284
L.2.1 Initialisation 284
L.2.2 Transmitting Frames 284
L.2.3 Receiving Frames 286
L.3 MicroMAC API 287
L.3.1 Initialisation Functions 288
vMMAC_Enable 289
vMMAC_EnableInterrupts 290

vMMAC_ConfigureRadio 291
vMMAC_SetChannel 292
L.3.2 Transmit Functions 293
vMMAC_SetTxParameters 294
vMMAC_SetTxStartTime 295
vMMAC_StartMacTransmit 296
vMMAC_StartPhyTransmit 298
u32MMAC_GetTxErrors 299
L.3.3 Receive Functions 300
vMMAC_SetRxAddress 301
vMMAC_SetRxStartTime 302
vMMAC_StartMacReceive 303
vMMAC_StartPhyReceive 305
bMMAC_RxDetected 306
u32MMAC_GetRxTime 307
u32MMAC_GetRxErrors 308
L.3.4 Timing Function 309
u32MMAC_GetTime 310
vMMAC_SetCutOffTimer 311
vMMAC_SynchroniseBackoffClock 312
L.3.5 Interrupt Control Functions 313
vMMAC_ConfigureInterruptSources 314
u32MMAC_PollInterruptSource 315
u32MMAC_PollInterruptSourceUntilFired 316
L.3.6 Miscellaneous Functions 317
vMMAC_RadioOff 318
vMMAC_GetMacAddress 319
L.4 Structures 320
L.4.1 tsMacFrame 320
L.4.2 tsPhyFrame 321
L.4.3 MAC_Addr_u 321
L.4.4 MAC_ExtAddr_s 322
L.5 Enumerations 322
L.5.1 ‘Transmit Options’ Enumerations 322
L.5.2 ‘Transmit Status’ Enumerations 323
L.5.3 ‘Receive Options’ Enumerations 324
L.5.4 ‘Receive Status’ Enumerations 325
L.5.5 ‘Interrupt Status’ Enumerations 326
L.6 MAC and PHY Transceiver Modes 327
L.6.1 MAC Mode 327
L.6.2 PHY Mode 327
M. Glossary 328

Preface

This manual is the main reference resource in developing applications for devices in the wireless network of an NXP JenNet-IP system. The manual first introduces the basic principles of a JenNet-IP system, with particular attention to the wireless part. It then describes the Application Programming Interface (API) that can be used to develop applications for the NXP microcontrollers on which the wireless network nodes are based. The API resources (functions, network parameters, enumerations, structures, etc) are fully detailed. The manual should be used as a reference resource throughout JenNet-IP wireless application development.

Note 1: The development of applications for the LAN/WAN part of a JenNet-IP system is described in the *JenNet-IP LAN/WAN Stack User Guide (JN-UG-3086)*.

Note 2: JenNet-IP can be run on the NXP JN5168 and JN5164 wireless microcontrollers. These devices are collectively referred to as JN516x or JN51xx devices in this manual.

Note 3: The version of the JenNet protocol referenced in this manual is v2.0 and is not the same as the version described in the *JenNet Stack User Guide (JN-UG-3041)*.

Note 4: In the JenNet-IP stack (SDK JN-SW-4165) to which this manual relates, NXP's MiniMAC layer is used instead of the standard IEEE 802.15.4 MAC layer in order to reduce code-size. The functional and implementation changes are minimal but you should refer to the SDK Release Notes (JN-RN-0052) for more details. You are also advised to start your JenNet-IP application development from a relevant NXP Application Note (JN-AN-1190 or JN-AN-1162).

Organisation

The manual is divided into 4 parts:

- Part I: Concept Information comprises 3 chapters providing background information for JenNet-IP:
  - Chapter 1 introduces 6LoWPAN and NXP's JenNet-IP.
  - Chapter 2 outlines the wireless network concepts that you will need for an understanding of JenNet-IP systems.
  - Chapter 3 describes a JenNet-IP system.

- Part II: JenNet-IP Embedded API comprises 6 chapters detailing the JenNet-IP Embedded API (JIP Embedded API), which is used to develop applications for the JN51xx-based wireless nodes of a JenNet-IP system:
  - Chapter 4 details the main tasks to implement in a JenNet-IP wireless network application, including the necessary API function calls.
  - Chapter 5 details the general functions of the JIP Embedded API.
  - Chapter 6 details the MIB functions of the JIP Embedded API.
  - Chapter 7 details the user-defined callback functions of the JIP Embedded API.
  - Chapter 8 details the structures and enumerations of the JIP Embedded API.
  - Chapter 9 describes the network and stack parameters used in a JenNet-IP system.
- Part III: Optional Features comprises 2 chapters describing two optional features of a JenNet-IP system:
  - Chapter 10 describes the Over-Air Download (OND) feature for performing software upgrades on wireless nodes
  - Chapter 11 describes the operation of a WPAN as a standalone network without an IP connection
- Part IV: Appendices describes the following miscellaneous topics:
  - The roles of certain JenNet parameters used in a JenNet-IP system
  - Recommendations on buffering and pinging
  - ICMP message handling
  - Device ID
  - Network Application ID
  - Data packet format
  - Low-level principles of JenNet-IP
  - JenNet-IP Browser
  - Memory Heap
  - Example of OND
  - Exception handling
  - MicroMAC stack used in low-energy devices
  - 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
JMP	JenNet Management Protocol
LAN	Local Area Network
LQI	Link Quality Indicator
MIB	Management Information Base
MLD	Multicast Listener Discovery
MTU	Maximum Transmission Unit
NVM	Non-Volatile Memory
OND	Over-Network Download

PAN	Personal Area Network
PER	Packet Error Rate
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-3086	JenNet-IP LAN/WAN Stack User Guide
JN-UG-3098	BeyondStudio for NXP Installation and User Guide
JN-UG-3093	JN516x-EK001 Evaluation Kit User Guide
JN-UG-3087	JN516x Integrated Peripherals API User Guide
JN-UG-3024	IEEE 802.15.4 Stack User Guide
JN-AN-1190	JenNet-IP Application Template Application Note
JN-AN-1162	JenNet-IP Smart Home Application Note
JN-AN-1110	JenNet-IP Border-Router Application Note
JN-AN-1059	Wireless Network Deployment Guidelines Application Note

***Caution:** You should **not** refer to the JenNet Stack User Guide (JN-UG-3041), as this describes a different version of the JenNet protocol from the version used in JenNet-IP.*

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

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. Introduction

This chapter describes the motivation for JenNet-IP, provides a high-level view of the software architecture, and introduces the JenNet-IP product and user documentation.

1.1 Wireless IP

The Internet and Personal Area Networks (PANs) traditionally operate at opposite geographical scales - the Internet is a worldwide network while a PAN has a relatively limited operating space. Wireless PANs (WPANs) which operate through radio communication are now well established through recent technologies such as ZigBee, IEEE 802.15.4 and NXP's own JenNet protocol. Wireless networks offer clear advantages when compared with wired solutions, in terms of ease of implementation and cost. However, the operating environment of a WPAN is normally restricted to several hundred metres by the limited ranges of the small radio transceivers employed in the network nodes.

A powerful and highly flexible PAN solution can be achieved by combining Internet and WPAN technologies to allow a much expanded WPAN arena. This integrated approach facilitates the following types of system:

- A WPAN can be controlled and monitored remotely over a Wide Area Network (WAN), such as the Internet. As an example, a home heating/lighting WPAN could be remotely accessed over the Internet from a PC, allowing the house to be prepared for the occupant's return from a business trip. This type of system is illustrated in Case A in Figure 1 below.
- Two or more physically separate but neighbouring WPANs can operate as a single system by allowing them to communicate over a Local Area Network (LAN), such as an Ethernet bus. As an example, security WPANs on different floors of the same building can be connected via a LAN and centrally controlled from a single security console for the building, which could be remotely located. This type of system is illustrated in Case B in Figure 1 below.
- Two or more geographically separate WPANs can operate as a single system by allowing them to communicate over a WAN, such as the Internet. As an example, security WPANs on factory sites in different cities could be connected via the Internet to effectively form a single security system for the manufacturing company. This type of system is illustrated in Case C in Figure 1 below.

These integrated systems operate using the Internet Protocol (IP). Data is transported in IP packets in the LAN/WAN domain of the system. Within a WPAN, the IP packets are transmitted wirelessly between nodes using an established wireless transport protocol, such as IEEE 802.15.4. The marriage of IP and WPAN has given rise to 6LoWPAN (IPv**6** over **Lo**w power **W**ireless **P**ersonal **A**rea **N**etworks). Note that the '6' comes from the fact that version 6 of the Internet Protocol is used in WPAN communication (while version 4 is still the predominant technology used on the Internet). JenNet-IP is based on the 6LoWPAN technology.

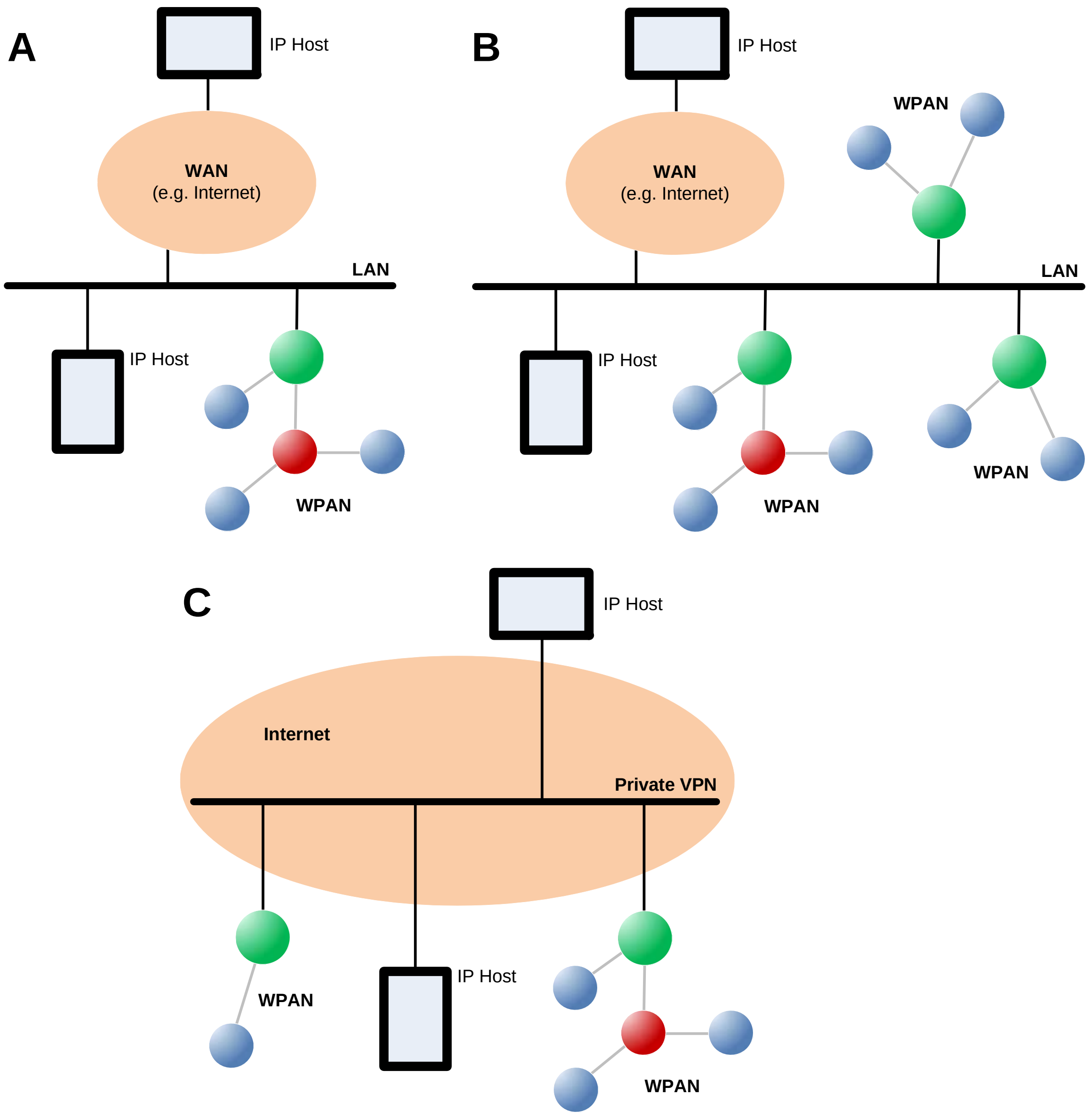


Figure 1: Example 6LoWPAN Systems

IP Host = Remote device (e.g. PC, tablet or phone) used to access a WPAN (via WAN/LAN)

Note: A WPAN in a 6LoWPAN system is also referred to as a ‘wireless cluster’.

1.2 6LoWPAN

In a 6LoWPAN system, information is communicated between nodes (which may be in different wireless networks) by means of IP datagrams or packets. In a wireless network (WPAN) which is part of a 6LoWPAN system, IPv6 packets are used.

IPv6 is a delivery protocol for transferring packets across data networks, including the Internet, Ethernet and Personal Area Networks (PANs). Due to its explosive growth, the Internet faces the problem that the supply of IPv4 addresses is now effectively exhausted. This is the driving force behind the adoption of IPv6, which supports a much larger address space (128 bits for IPv6 compared with 32 bits for IPv4), providing an almost inexhaustible supply of addresses for network nodes.

IPv6 packets are transported between the nodes of a WPAN using one of the established wireless network protocols (e.g. JenNet), built on the IEEE 802.15.4 wireless network standard (see Section 1.3). An IPv6 packet is carried in the payload of an IEEE 802.15.4 data frame, which is passed between network nodes. However, a raw IPv6 packet may be too large to fit into the payload of an IEEE 802.15.4 frame. 6LoWPAN is an adaptation layer which enables IPv6 packets to fit into IEEE 802.15.4 frame payloads. 6LoWPAN uses compression and fragmentation techniques to deal with packets created by the protocols in the Internet Protocol Suite. For the full specification of IPv6 over IEEE 802.15.4, refer to RFC 4944 available from the IETF (**www.ietf.org**).

Note: The devices in a WPAN are referred to as ‘nodes’, while the devices in an IP-based network are referred to as ‘hosts’. In this manual, we will refer to the WPAN devices as nodes, even though they may be accessed across an IP-based network.

1.3 Software Architecture

This section presents a simplified view of the software architecture implemented in a 6LoWPAN system. More detailed software architectures for JenNet-IP are presented in Section 3.3.

The software stack shown in Figure 2 comprises three basic levels which are present on both the wireless network nodes and the IP-based devices of a 6LoWPAN system. The details of the levels differ between the WPAN and LAN/WAN domains of the system.

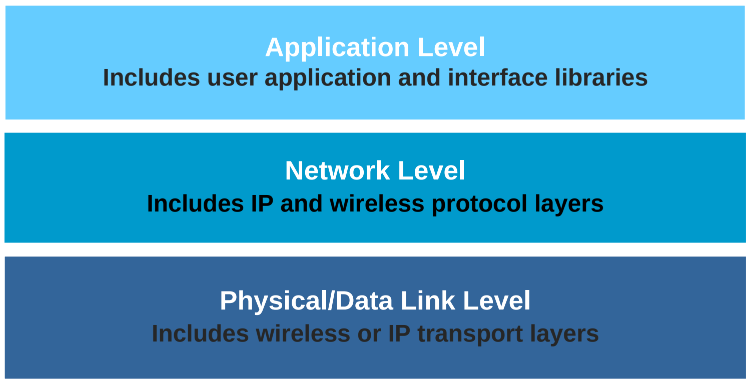

Application Level
Includes user application and interface libraries
Network Level
Includes IP and wireless protocol layers
Physical/Data Link Level
Includes wireless or IP transport layers

Figure 2: Basic 6LoWPAN Software Stack

The 6LoWPAN stack levels illustrated in Figure 2 are described below (from top to bottom):

- **Application level:** This level contains the user applications which are responsible for collecting/reporting data (e.g. temperature measurements), as well as initiating data transmissions and handling received data. The user application interacts with the lower levels of the stack via dedicated interfaces in the form of function libraries.
- **Network level:** This level is responsible for managing communications with the network and comprises the following:
    - **Internet Protocols:** These protocols (UDP, TCP) are concerned with assembling IPv6 packets to be sent and disassembling received IPv6 packets. The Internet Protocols are described further in Chapter 3. JenNet-IP uses UDP as well as ICMP, which is mainly concerned with management and error reporting.
    - **6LoWPAN:** This layer is present only in the WPAN domain and is concerned with compressing IPv6 packets before they are inserted into the wireless network data frames to be transmitted and decompressing IPv6 packets extracted from received data frames.
    - **Wireless Network Protocol:** This layer is present only in the WPAN domain and is concerned with wireless network management such as starting/joining a network, message (frame) addressing and routing, and applying security (encryption/decryption) to messages. In JenNet-IP, this layer is provided by NXP’s own JenNet protocol.
- **Physical/Data Link level:** This level is responsible for assembling frames to be transmitted and disassembling received frames, and interacting with the physical transmission medium.
    - In the WPAN domain, this level is based on the IEEE 802.15.4 wireless network protocol and is concerned with handling IEEE 802.15.4 data frames (referred to as MAC frames). This involves inserting compressed IPv6 packets into the payloads of data frames to be transmitted and extracting such packets from the payloads of received data frames.
    - In the LAN/WAN domain, this level may be based on IPv4 or IPv6 operating over a WAN (e.g. Internet) and/or LAN (e.g. Ethernet or WiFi).

Note: The JenNet-IP version of the 6LoWPAN stack is fully introduced in Section 3.3.

1.4 JenNet-IP Software

This section outlines the NXP JenNet-IP software, which is based on the 6LoWPAN technology introduced in Section 1.2. The JenNet-IP software includes components that run on the WPAN and LAN/WAN sides of the system. A detailed introduction to this software is presented in Section 3.2.

The software and documentation referenced below can be obtained free-of-charge from NXP, as indicated in "Support Resources" on page 16.

1.4.1 Required Hardware Platforms

The hardware platforms for the JenNet-IP WPAN and LAN/WAN software are as follows:

- The JenNet-IP WPAN software currently runs on the following NXP JN516x wireless microcontrollers:
  - JN5168-001
  - JN5164-001

  These devices integrate a 2.4-GHz radio transceiver, a 32-bit processor core and a wide range of on-chip peripherals, providing sufficient memory to run user application software.
- The JenNet-IP LAN/WAN software runs on an IP Host, such as a PC, tablet or mobile phone. Specific support is provided for developing applications for Linux-based platforms.

1.4.2 Development Software

The JenNet-IP software is provided in the *JN516x JenNet-IP SDK (JN-SW-4165)* installer. This software includes:

- Application Programming Interfaces (APIs) for easy JenNet-IP application development for the following platforms:
  - JN516x wireless microcontrollers (on nodes in the WPAN domain)
  - IP Hosts such as PCs, tablets and mobile phones (in the LAN/WAN domain)
- Stack software required to implement a JenNet-IP system:
  - WPAN stack, which includes the JenNet protocol software that is used on top of IEEE 802.15.4, where the latter includes NXP's MiniMAC layer
  - LAN/WAN stack, which supports both IPv6 and IPv4 connectivity
  - Low-energy device stack, which includes NXP's MicroMAC layer
- Software which runs on the Border-Router, the interface between the WPAN and LAN/WAN domains

A more specific list of the contents of the *JN516x JenNet-IP SDK (JN-SW-4165)* is provided in the table below.

Components	Comments
JenNet-IP WPAN APIs and stack software	Needed for applications that run on the JN516x devices of a wireless network
JenNet-IP LAN/WAN APIs and stack software	Needed for applications that run on IP host devices on a LAN or WAN
JenNet-IP Border-Router applications	Includes the following applications that run on the Border-Router: **6LoWPANd**, **JIPd** and **FWDISTRIBUTION**
JenOS Persistent Data Manager (PDM) API	Needed for saving/retrieving context data to/ from Flash memory on a JN516x module
802.15.4 Stack API	Needed for developing applications directly on top of IEEE 802.15.4 stack layers
JN516x Integrated Peripherals and Board APIs	Needed for hardware control (JN516x and evaluation kit boards)

Table 1: Contents of JenNet-IP SDK

Note: To aid your JenNet-IP application development, an example application is available in the Application Note *JenNet-IP Smart Home (JN-AN-1162)* and an application template is available in the Application Note *JenNet-IP Application Template (JN-AN-1190)*.

Features of the JenNet-IP software include:

- Support for wireless Tree networks using JenNet (with IEEE 802.15.4)
- Socket formation and data transfer services via an IPv6 UDP socket layer
- Packet fragmentation and re-assembly (when an IP packet is so large, it must be broken up and transported in multiple IEEE 802.15.4 frames)
- IP level translation between 6LoWPAN wireless network and the Ethernet, via a WPAN-LAN router (Border-Router)
- Support for unicast, multicast and broadcast addressing

1.4.3 Toolchain Software

In addition, NXP provide a set of application development tools in the *BeyondStudio for NXP (JN-SW-4141)* installer, including an Integrated Development Environment (IDE), a C code compiler and a Flash programmer for the JN516x device .

Note: Installation instructions for the JN516x JenNet-IP SDK and the toolchain are provided in the *BeyondStudio for NXP Installation and User Guide (JN-UG-3098)*.

1.5 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 (this manual)	Provides a general introduction to JenNet-IP and details the software resources for developing applications that run on devices on the WPAN side of a JenNet-IP system
JN-UG-3086	JenNet-IP LAN/WAN Stack User Guide	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 LAN/WAN and WPAN sides of a JenNet-IP system

Table 2: JenNet-IP User Documentation

The above documents can be obtained from NXP as indicated in "Support Resources" on page 16.

1.6 Where Now?

This manual is designed to provide an introduction to JenNet-IP as well as detailed information on developing wireless network applications for a JenNet-IP system.

Software designers who are reading this manual may be involved in developing applications for the WPAN side and/or LAN/WAN side of a JenNet-IP system. The next steps for these developers are outlined below.

WPAN Application Development

You are recommended to study all chapters of this manual:

- You should first read the remaining chapters in Part I: Concept Information to provide the necessary background information for your JenNet-IP application development.
- You should then refer to Part II: JenNet-IP Embedded API throughout your application development.

LAN/WAN Application Development

You are recommended to study the following:

- You should first read the remaining chapters in Part I: Concept Information to provide the necessary background information for your JenNet-IP application development.
- You should then refer to the *JenNet-IP LAN/WAN Stack User Guide (JN-UG-3086)* throughout your application development.

2. Wireless Network Concepts

This chapter describes the fundamental concepts needed for an understanding of Wireless Personal Area Networks (WPANs). A 6LoWPAN system contains one or more WPANs which may be accessed from the Internet, where a constituent WPAN is also referred to as a 'wireless cluster'. In this chapter, we concentrate purely on a WPAN in isolation from the rest of the 6LoWPAN system. The following concepts are described: wireless operation (including radio communication and battery power), communication routing, network topologies and node types, network identity, node addressing, software architecture, network formation and operation.

Note: In this chapter, references are made to the JenNet protocol - this is an NXP software layer required to form a multi-hop network.

2.1 Wireless Operation

The idea of a wireless network is to use radio links to replace the cables that connect the nodes of a traditional network - thus, the nodes exchange data via radio communications. However, cable replacement may be extended to include the power cabling for certain nodes. These issues are expanded upon in the sub-sections below.

2.1.1 Radio Communication

The NXP JenNet-IP software is designed to run on JN51xx microcontrollers (see Section 1.4), which feature an integrated radio transceiver operating in the 2400-MHz radio frequency (RF) band. This band is available for unlicensed use in most geographical areas (check your local radio communication regulations). The basic characteristics of this RF band for the IEEE 802.15.4 protocol are as follows:

Frequency Range	2405 to 2480 MHz
Channel Numbers	11-26 (16 channels)
Data Rate	250 kbps

Thus, this RF band is split into 16 channels. It is possible to automatically select the best channel (that with least detected activity) at system start-up.

The range of a radio transmission is dependent on the operating environment (inside or outside a building), the NXP hardware module used (that carries the JN51xx microcontroller) and the type of antenna used. Using a JN51xx standard-power module fitted with an external dipole antenna, a range of 1 km can typically be achieved in an open area. Inside a building, this can be reduced due to absorption, reflection, diffraction and standing wave effects caused by walls and other solid objects. A JN51xx high-power module can achieve a range which is a factor of five greater than that of a standard-power module.

Tip: For guidance on the deployment of radio devices, refer to the Application Note *Wireless Network Deployment Guidelines (JN-AN-1059)*, available from NXP (see "Support Resources" on page 16).

2.1.2 Battery Power

One of the objectives of the wireless network protocols is the reduction of power cabling by allowing the autonomy of certain nodes through battery power and even solar power. This brings the advantages of easier and cheaper network installation, more flexible siting of nodes and relocation of nodes.

Low-capacity batteries are often used and their use is optimised by restricting the time for which energy is required. To this end, data is transmitted infrequently (perhaps once per hour or even per week), with the device reverting to low-power sleep mode the rest of the time. However, it is not normally feasible for all network devices to be battery-powered, since some nodes must be left on all the time (see Section 2.3).

2.2 Network Communications

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

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

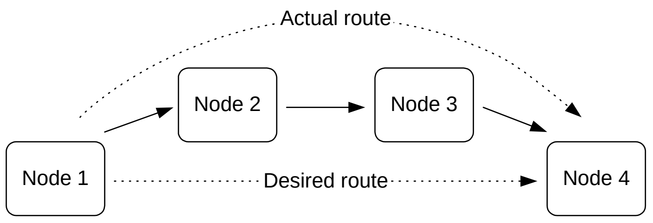


Figure 3: Routing between Network Nodes

The process of receiving data destined for another node and passing it on is known as ‘routing’ (see Section 2.7). The application running on the routing node is not aware that the data is being routed, as the process is completely automatic and transparent to the application.

2.3 Network Node Types

A wireless network can be made up from nodes of three types:

- Co-ordinator
- Router
- End Device

These node types and their roles are summarised in Table 3 below. Note that every wireless network must initially have a Co-ordinator to start and form the network.

Node Type	Role
Co-ordinator	The Co-ordinator is an essential node and plays a fundamental role at system initialisation, when its tasks are to: • Select the radio channel to be used by the network • Start the network • Allow other nodes to connect to it (that is, to join the network) In addition to running applications, the Co-ordinator may provide message routing, security management and other services.
Router	In addition to running applications, the main tasks of a Router are to: • Relay messages from one node to another (routing) • Allow other nodes to connect to it (that is, to join the network) A Router must remain active and therefore cannot sleep.
End Device	The main tasks of an End Device at the network level are sending and receiving messages. An End Device cannot allow other nodes to connect to it. It can be battery-powered and, when not transmitting or receiving, can sleep in order to conserve power.

Table 3: Node Types and Roles

The application on each node configures the node as a Co-ordinator, Router or End Device. The application on the Co-ordinator can also pre-configure the desired radio channel for the network (or enable an automated search for the best channel).

Note: A ‘low-energy device’ can also be used with the wireless network of a JenNet-IP system. This type of device has very limited energy resources (e.g. an ‘energy harvesting’ device) and is not a full member of the JenNet-IP network, but can send messages to the network. Low-energy devices are more fully introduced in Section 3.9.

2.4 Network Topology

A wireless network in an NXP JenNet-IP system has a Tree topology, which determines how the nodes are linked and how messages propagate through the network.

A Tree topology consists of a Co-ordinator, Routers and End Devices. The Co-ordinator is linked to a set of Routers and End Devices - its children. A Router may then be linked to more Routers and End Devices - its children. This can continue to a number of levels.

Note: A Router can be used in place of an End Device in a Tree network, but the message relay functionality of the Router will not be used - only its application will be relevant.

This hierarchy can be visualised as a tree structure with the Co-ordinator at the top, as illustrated in the figure below.

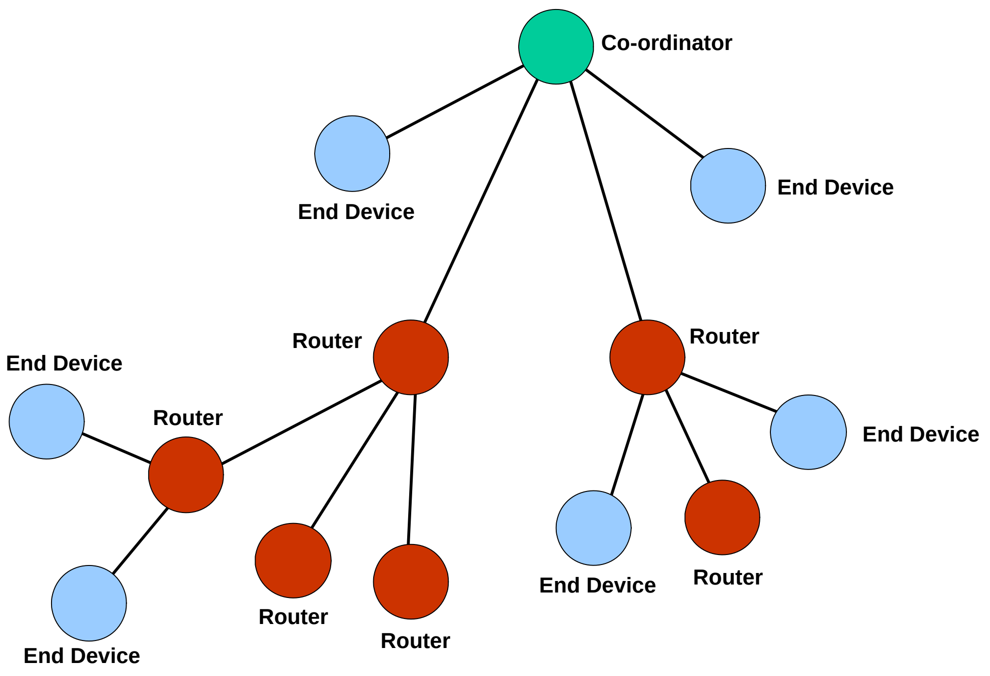


Figure 4: Tree Topology

Note that:

- The Co-ordinator and Routers can have children, and can therefore be parents.
- End Devices cannot have children, and therefore cannot be parents.

The communication rules in a Tree topology are:

- A node can directly communicate only with its parent and with its own children (if any).
- In sending a message from one node to another, the message must travel from the source node up the tree to the nearest common ancestor and then down the tree to the destination node.

While in a Tree network there is no alternative route if a necessary link fails, the JenNet protocol provides the facility to automatically repair failed routes.

Note: It is important when designing and deploying a Tree network that all child nodes stay within range of at least one Router, so that reliable communication can occur.

2.5 Network Identity

It must be possible to identify a wireless network uniquely in order to manage situations in which multiple wireless networks are operating in the same space or neighbouring spaces. A node must be able to identify the network to which it belongs. In a wireless network which is part of a JenNet-IP system, the PAN (Personal Area Network) ID is used for this identification.

The PAN ID is a 16-bit value used by the IEEE 802.15.4 protocol. It is used by the lower levels of the software stack to identify the network - for example, in the delivery of messages sent between nodes. It should be unique within the operating environment - that is, it should not clash with the PAN ID of a neighbouring network. A value for the PAN ID can be pre-set in the user application code of the Co-ordinator. In JenNet-IP:

- If the PAN ID is pre-set to 0xFFFF, the Co-ordinator will choose an initial PAN ID at random and then check for its uniqueness by “listening” for the PAN IDs of other networks (it will repeatedly choose a random PAN ID until it finds one that does not clash with that of another network)
- If the PAN ID is pre-set to any other valid value, the Co-ordinator will use this PAN ID (irrespective of whether it clashes with the PAN ID of another network)

Routers and End Device will subsequently learn the PAN ID when they join the network.

Note: A wireless network may also implement its own network identifier at the application level, in addition to the PAN ID at the stack level. The implementation of a Network Application ID in JenNet-IP applications is described in Appendix E.

2.6 Node Addressing

The basic way of referring to a node in a network is by means of a numeric address.

In a wireless network, the IEEE or MAC address of the device is commonly used. This is a 64-bit address, allocated by the IEEE, which uniquely identifies the device – this address is fixed for the lifetime of the device and no two devices in the world can have the same IEEE address. It is also sometimes called the ‘extended’ address.

In a 6LoWPAN system, a wireless network node is identified by means of its IPv6 address. This is a 128-bit address in which the leading 64 bits (bits 127-64) identify the network and the trailing 64 bits (bits 63-0) identify the device. In JenNet-IP, this second part of the address, known as the Host Interface ID, is derived from the device’s MAC address - the Host Interface ID is taken to be the MAC address with bit 57 inverted. IPv6 addresses are described in more detail in Section 3.3.

2.7 Routing

A message sent from one node to another in a wireless network usually needs to pass through one or more intermediate nodes before reaching its final destination. The role of passing a message on (without processing its contents) is known as routing. The nodes that can perform routing are the Routers and the Co-ordinator.

In a tree network, in one transmission a message can only be passed up the tree to the parent node or down the tree to a child node (from where it may be passed on). The message is passed up the tree in these ‘hops’ until it reaches the first common ancester of the source node and the destination node, when it will be passed back down the tree via another branch until it reaches its destination. The message will only reach the top of the tree if the Co-ordinator is the only common ancestor.

A message contains two addresses for routing purposes - the address of the destination node and the address of the “next hop” node. The latter is modified by the routing node as the message propagates through the network, and becomes the same as the destination address for the final hop.

Note 1: In JenNet-IP, tree routing is only respected for unicasts. For a broadcast, the source node transmits the message to all nodes within radio range - each receiving Router node then re-broadcasts the message (but only the first time it receives the message). A multicast to a group of nodes is handled as a broadcast with a multicast address corresponding to the group - multicasts are described in more detail in Section 4.3.

Note 2: Although broadcast/multicast messages may be received by an End Device (that is awake), these messages are discarded by the device. Therefore, a message should only ever be unicast to an End Device.

2.7.1 Neighbour and Routing Tables

The routing mechanism requires routing information to be stored in the Routers and Co-ordinator. This information includes node addresses and is stored on the node in two tables:

- **Neighbour table:** Contains entries for all immediate children as well as the node's parent.
- **Routing table:** Contains entries for all descendant nodes (lower in the tree) that are not immediate children.

Together, these tables give a Router knowledge of all descendant nodes in the tree. Since the Co-ordinator is at the root of the tree, it has knowledge of all nodes in the network. These tables are assembled automatically by the stack as the network is initialised and formed.

Note: In a JenNet-IP system, the Neighbour table of a wireless network node is held in a standard Management Information Base (MIB) on the node - see Section 3.5.

2.7.2 Routing Process on a Node

On receiving a message, a Router node implements the following routing process:

1. The Router first checks the final destination address to determine whether the message was intended for itself and, if this is the case, processes the contents of the message.
2. If the above check failed, the Router checks its Neighbour table to determine whether the message is destined for one of its immediate children and, if this is the case, passes the message to the relevant child node.
3. If the previous check failed, the Router checks its Routing table to determine whether the message is destined for one its other descendants and, if this is the case, passes the message to the relevant intermediate child (Router).
4. If the previous check failed, the Router passes the message up the tree to its parent for further routing.

For the Co-ordinator, the routing mechanism is similar except the message cannot be passed further up the tree.

2.8 Network Formation and Operation

The creation of a wireless network starts with the Co-ordinator. The procedure for starting and forming the network is as follows:

On Co-ordinator

In order to form a wireless network, the Co-ordinator must be programmed with the details of the nodes that will potentially join the network, possibly supplied in the form of a ‘white list’ of all nodes that are allowed to join. A valid node may be identified by its IEEE/MAC address or an abbreviated identifier. A unique ‘commissioning key’ for the node must also be provided, which will be used to secure the joining process. This information can be provided to the Co-ordinator at any time by an ‘out-of-band’ (non-wireless) means - for example, from a remote device via the Internet.

1. **Radio Channel Selected:** The Co-ordinator searches for a suitable radio channel (usually the one which has least activity). This can be limited to those known to be usable - for example, avoiding frequencies where it is known a wireless LAN is operating.
2. **PAN ID Allocated:** The Co-ordinator assigns a 16-bit PAN ID to the network. The PAN ID is pre-set by the system developer but if this is set to the value 0xFFFF, the Co-ordinator selects a PAN ID at random which does not clash with that of a neighbouring network (see Section 2.5).
3. **Network Ready for Joining:** The Co-ordinator now ‘listens’ (in the chosen channel) for requests from other nodes (Routers and End Devices) to join the network.

On Other Devices

4. **Networks Searched:** A node (Router or End Device) wishing to join the network first scans the available channels to find potential networks to join. In doing so, the node transmits beacon requests in these channels and waits for beacons from potential parents.
5. **Best Parent Selected:** The Co-ordinator will initially be the only potential parent of a new node. However, once the network has partially formed, the joining device may be able to 'see' the Co-ordinator and one or more Routers of the network. In this case, it uses the following criteria to choose its parent (in the given order of precedence on a Router and in reverse order on an End Device):
   - Depth in tree (preference given to parent highest up the tree)
   - Number of children (preference given to parent with fewest children)
   - Signal strength (preference given to parent with strongest signal)

   The applications may also implement their own network identifier which can be used in the selection of an appropriate network/parent - see Appendix E.
6. **Join Request Sent:** The node then sends a message to the selected parent (Co-ordinator or Router), encrypted with the node’s commissioning key, asking to join the network through it. The selected parent initially rejects the join request but checks whether the requesting node is allowed in the network - this information must be obtained from the Co-ordinator. If the node is valid,

the Co-ordinator supplies the node's commissioning key to all Router nodes in the network.

7. **Join Request Resent:** The node then sends another encrypted join request to a potential parent. Provided that the node has been successfully validated by the Co-ordinator (in Step 6), the join request can be decrypted with the commissioning key supplied by the Co-ordinator and the request can be accepted. If the request is rejected, the node will perform another search (but an End Device will sleep before starting this search).
8. **Route Established to Co-ordinator:** The node sends an Establish Route message to the Co-ordinator, which replies to confirm the node's membership of the network. This exchange of messages results in the necessary entries for the node being added to the Routing tables between the node and Co-ordinator.
9. **Network Key Received and Saved:** In the response to the Establish Route message (Step 8), the Co-ordinator includes the network security key. The node must save this network key, which must be used to encrypt/decrypt all future network communications in which the node participates (the node may save the network key in non-volatile memory to aid any future rejoins of the network).

Once all nodes have joined the network, data can be sent between the nodes. This data is encrypted using the network key and can contain any kind of information, as it is not interpreted by the networking software.

The commissioning key and network key are summarised in Section 3.6.

Tip: For more information on how a wireless network forms and operates at the IEEE 802.15.4 level, refer to the *IEEE 802.15.4 Stack User Guide (JN-UG-3024)*.

2.9 Other Network Operations

This section describes a number of network tasks that must either be configured or implemented in your application code.

2.9.1 Auto-ping

In a Tree network, a node may lose its parent and be unaware of this loss, particularly if data exchanges with its parent are infrequent. In JenNet, an auto-ping mechanism (enabled by default) is employed to periodically verify that the parent node is still present (during periods when application data is not being exchanged). On each ping, the node sends a message to its parent:

- If the parent is still present and recognises the node as its child, it sends a response.
- Otherwise, one of two error situations exist:
    - If the parent is not present, no response is sent. If a certain number (five, by default) of consecutive pings are unacknowledged, the child considers its parent to be lost and the child must attempt to re-join the network.
    - If the parent is present but has disowned the child, an “Unknown-Node” message is sent back (which an End Device must obtain by polling - see Section 2.9.3). In this case, the child must attempt to re-join the network.

Note 1: In a busy network, pinging is not essential since the loss of a parent will be noticed through failed data communications. To avoid unnecessary traffic in such networks, when data is received from the parent node, the ping timer is re-started.

Note 2: Further information and advice on use of the auto-ping mechanism are provided in Appendix A.2.1 and Appendix B.2.

2.9.2 Sleep Mode

A node which does not need to regularly send data, receive data, take input or produce output can conserve power during its inactive phases by entering a low-power mode, called sleep mode. Since Routers and the Co-ordinator must constantly remain active for routing and joining purposes, normally only End Devices are allowed to sleep.

Two forms of sleep mode are available:

- **Sleep with memory held:** On-chip volatile memory remains powered during sleep, allowing memory contents to be preserved. This mode permits both application and stack context data to be preserved during sleep, allowing stack operation to be resumed (rather than re-started from scratch) on waking.
- **Sleep without memory held:** On-chip volatile memory is not powered during sleep, meaning memory contents are lost. Therefore, stack context data is lost and stack operation must re-start from scratch on waking. However, it is possible to save application context data in non-volatile memory (e.g. Flash memory) before entering sleep mode and then to retrieve this data on exiting sleep

For information on implementing sleep mode using the JenNet-IP API functions, refer to Section 4.6.

2.9.3 Data Polling

An End Device can sleep for a good proportion of the time in order to conserve power. Therefore, when data arrives for the End Device from another node, it may not be possible to deliver the data immediately, since the destination node may be in sleep mode. Therefore, the parent of the destination node buffers the data until the End Device is out of sleep mode and ready to receive data. It is the responsibility of the End Device to poll its parent to check whether there is pending data waiting to be delivered.

The data polling mechanisms employed in JenNet-IP are detailed in Section 4.7. By default, auto-polling is enabled, in which the End Device polls periodically with a configured period and also immediately after waking from sleep - the configuration of auto-polling is described in Appendix A.4.

3. JenNet-IP System Overview

This chapter describes a JenNet-IP system, which is based on the 6LoWPAN system introduced in Chapter 1. The description here assumes that you are already familiar with the wireless network concepts presented in Chapter 2 and basic IP concepts (outlined in the *JenNet-IP LAN/WAN Stack User Guide (JN-UG-3086)*).

3.1 Hardware Architecture and Components

A JenNet-IP system contains one or more WPANs connected to a LAN, such as an Ethernet bus, which may be connected to a WAN, such as the Internet (example systems are presented in Section 1.1). This allows:

- The WPAN(s) to be monitored and controlled remotely, e.g. over the Internet.
- Multiple WPANs to communicate with each other via the LAN/WAN domain (thus, a node in one WPAN can send a message to a node in another WPAN).

A typical system is illustrated in Figure 5 below.

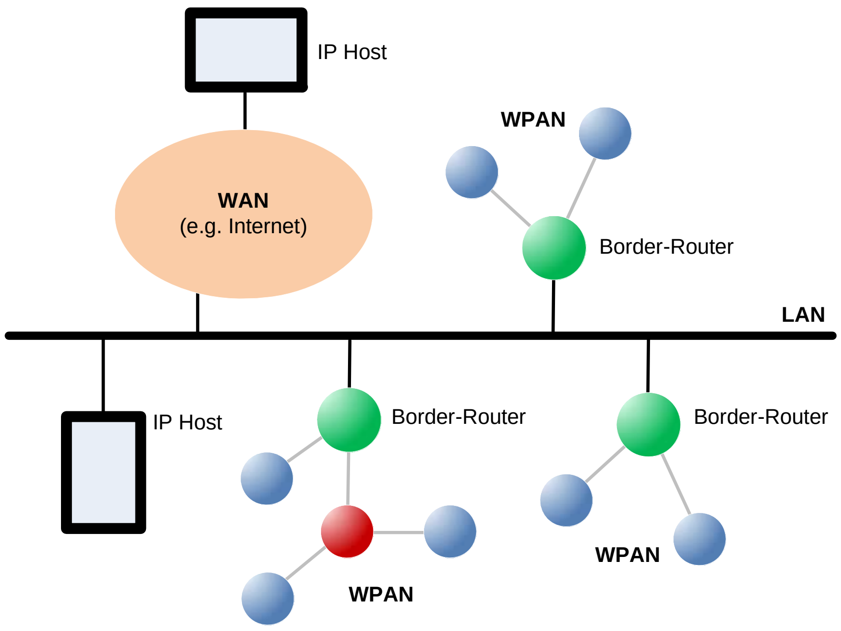


Figure 5: Typical JenNet-IP System

The main components of a JenNet-IP system, as illustrated in Figure 5, are as follows:

- **WPAN:** A wireless network (which may be one of many in the system)
- **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 in the LAN/WAN domain with an IP connection to the system - for example, this may be a PC, tablet or mobile phone from which a WPAN is monitored and controlled

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

Note: A WPAN can also operate in isolation without an IP connection via a Border-Router, but with the capability to add a Border-Router if and when IP connectivity is required - refer to Section 3.1.1 and Chapter 11.

3.1.1 WPAN (Wireless Cluster)

A WPAN in a JenNet-IP system is referred to as a ‘wireless cluster’. A JenNet-IP system may contain multiple WPANs, where each is an autonomous wireless network. Each WPAN contains a single Co-ordinator node and a number of other nodes (End Devices and/or Routers) - see Section 2.3. The Co-ordinator is normally incorporated in the Border-Router device, described in Section 3.1.3.

The WPANs of a JenNet-IP system can have distinct pre-set PAN IDs or the same pre-set PAN ID. However, if the same pre-set PAN ID is used in multiple networks with operating spaces that overlap, the PAN ID of a network may be dynamically changed at network start-up in order to achieve distinct PAN IDs - see Section 2.5.

Messages are sent between the wireless network 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 (irrespective of whether the destination node is inside or outside the WPAN of the source node).

A WPAN can operate alone as a simple wireless network without a Border-Router. In this case, IP connectivity is likely to be available as an option by adding a Border-Router. Such a network operates without a full Co-ordinator, consisting only of Routers and a pseudo-Co-ordinator, which is used to establish the network. For more information on this type of network, refer to Chapter 11.

3.1.2 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).

3.1.3 Border-Router (WPAN-LAN Router)

The Border-Router is a device used to connect a WPAN to the LAN, where each WPAN has its own Border-Router. 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 frames). 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.

3.1.4 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.

3.1.5 IP Host

An IP host is connected to the JenNet-IP system via the LAN or a WAN (such as the Internet). It may have either of the following roles.

Remote Access Host

This IP host is used to access the JenNet-IP system remotely via a WAN (e.g. the Internet). It can be a standard PC, tablet or mobile phone. The device may be equipped with specific software for JenNet-IP system monitoring and control. Alternatively, the device may access the system via a web server implemented on a separate IP host (see Data Management Host below).

The JenNet-IP software includes APIs that can be used to develop applications to monitor and control a JenNet-IP system from a remote IP host. These APIs are detailed in the *JenNet-IP LAN/WAN Stack User Guide (JN-UG-3086)*.

Data Management Host

This IP host can be used as an intermediary for remote accesses to a JenNet-IP system. In this sense, the device may act as a web server for other IP hosts (see Remote Access Host above). In practice, the device may be incorporated in the Border-Router.

The specific roles of the device are as follows:

- Provides a web server for interactions with IP hosts requiring remote access to a JenNet-IP system:
  - The device handles requests from other IP hosts for status information on the system components and for configuration changes to the system components (see below).
  - The device must be able to generate web-based output to be displayed in a web browser on the requesting device and accept web-based input from the latter device.
- Interacts with individual nodes within a WPAN, in order to deal with configuration and monitoring requests from an IP host (see above) - for example, it may be required to service a request to obtain information from the MIB (Management Information Base) on a particular node or to modify the MIB on a node (MIBs are described in Section 3.4.1).

3.2 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 3.2.1, and then taking a more detailed view in Section 3.2.2 (IPv6 case) and Section 3.2.3 (IPv4 case).

3.2.1 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.

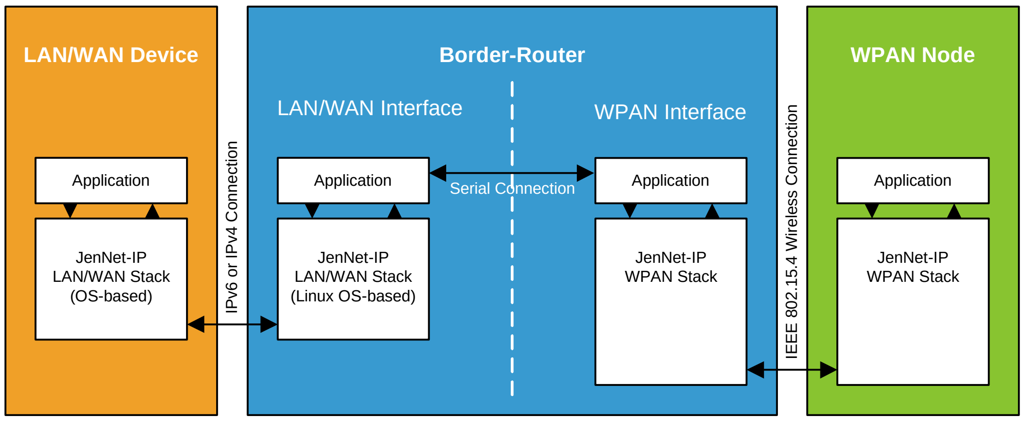


Figure 6: Software Divisions in JenNet-IP System

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 3.2.2 and Section 3.2.3.

3.2.2 Software Components (IPv6 Case)

This section provides more details of the JenNet-IP software components introduced in Section 3.2.1, in the case of an IPv6 connection to the LAN/WAN domain.

Note: The JenNet-IP software components that are required in the case of an IPv4 connection to the LAN/WAN domain are outlined in Section 3.2.3.

The figure below is a more detailed version of Figure 6, showing the contents of the JenNet-IP stacks (below the applications) and other software components required in the Border-Router.

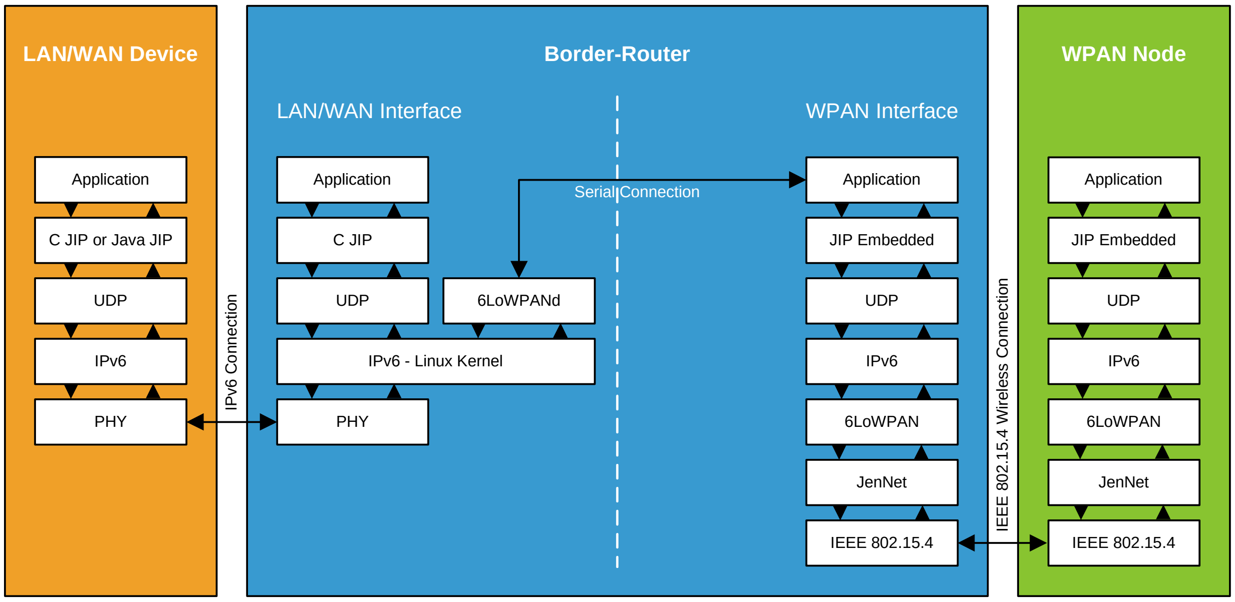


Figure 7: Software Components in JenNet-IP System (IPv6 Case)

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

WPAN Node

The following software runs on the NXP JN51xx microcontroller in a node of a WPAN:

- **Application:** This software is developed using C APIs provided in the *JN516x JenNet-IP SDK (JN-SW-4165)*. In particular, the JenNet-IP Embedded API is needed (described in Part II: JenNet-IP Embedded API).
- **JenNet-IP WPAN Stack:** This software stack is also provided in the *JN516x JenNet-IP SDK (JN-SW-4165)*. It consists of the stack layers indicated in Figure 7 and detailed in Section 3.3.

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 domain 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 (described in Section 3.3), 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 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 JIP API, which is provided in the *JN516x JenNet-IP SDK (JN-SW-4165)* and described in the *JenNet-IP LAN/WAN Stack User Guide (JN-UG-3086)*. 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 3.3. 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 case of the JN516x-EK001 Evaluation Kit, the LAN/WAN stack is implemented in a Linksys router and the WPAN stack is implemented on a JN5168-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 and control the WPAN. It can be developed using the C JIP API or the Java JIP API, both detailed in the *JenNet-IP LAN/WAN Stack User Guide (JN-UG-3086)*. Alternatively, a standard test application known as the JenNet-IP Browser can be used which is supplied as a Java executable with the JenNet-IP SDK (and is introduced in Section 3.2.4). 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 7 and detailed in the *JenNet-IP LAN/WAN Stack User Guide (JN-UG-3086)*.

3.2.3 Software Components (IPv4 Case)

This section provides more details of the JenNet-IP software components introduced in Section 3.2.1, in the case of an IPv4 connection to the LAN/WAN domain.

Note: The JenNet-IP software components that are required in the case of an IPv6 connection to the LAN/WAN domain are described in Section 3.2.2.

The figure below is a more detailed version of Figure 6, showing the contents of the JenNet-IP stacks (below the applications) and other software components required in the Border-Router.

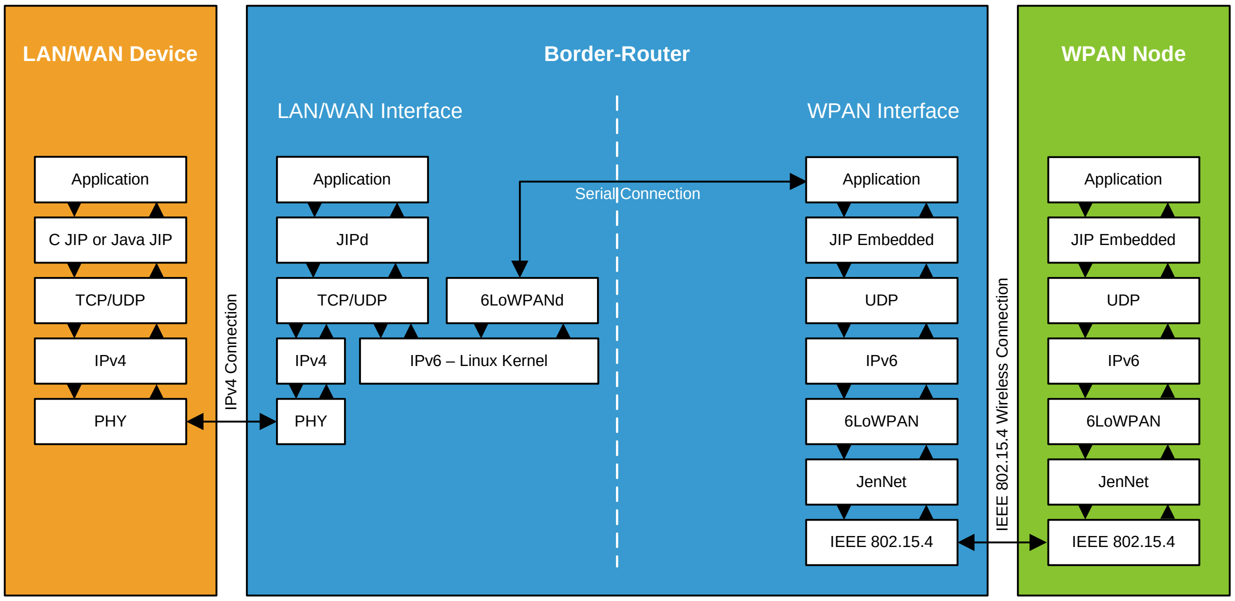


Figure 8: Software Components in JenNet-IP System (IPv4 Case)

The software depicted in Figure 8 is similar to that described for the IPv6 case in Section 3.2.2, 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 on this device:
    - **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)

3.2.4 JenNet-IP Browser

The JenNet-IP Browser is an example application that may be used from a LAN/WAN device to interact with the WPAN nodes of a JenNet-IP system. This application provides a generic engineering interface to the WPAN, allowing MIB variables on nodes to be inspected and/or edited.

The application may be run on either of the following:

- remote LAN/WAN device
- LAN/WAN side of the Border-Router - in this case, the application may serve web pages that can be viewed in a web browser on the LAN/WAN device

In either case, it is the application that sits above the JenNet-IP LAN/WAN stack on the appropriate device in Figure 6, Figure 7 and Figure 8.

A Java version of the application is supplied in the JenNet-IP SDK as an executable that can be run directly on a LAN/WAN device (e.g. a PC or workstation) with an IP connection to the Border-Router of a WPAN. It represents an example of a test application that a developer may design using the Java JIP API (described in the *JenNet-IP LAN/WAN Stack User Guide (JN-UG-3086)*) to control and monitor the WPANs of a JenNet-IP system.

Note 1: Use of the Java version 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.

Note 2: The C JIP API (described in the *JenNet-IP LAN/WAN Stack User Guide (JN-UG-3086)*) can alternatively be used to develop a similar application for a Linux-based platform.

A C-version of the application is provided in the firmware of the Linksys and Buffalo routers used in JenNet-IP demonstration systems (and runs on the router). This application was developed using the C JIP API. It is accessed from a normal web browser running on the LAN/WAN device. This application is 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).*

Use of the JenNet-IP Browser is described further in Appendix H.

3.3 JenNet-IP WPAN Stack

In Section 1.3, the 6LoWPAN software architecture was introduced as a stack comprising three basic levels: Application level, Network level and Physical/Data Link level. This section presents a more detailed view of the JenNet-IP stack that runs on devices on the WPAN side of a system.

Note: The JenNet-LAN/WAN stack that runs on devices on the LAN/WAN side of a system is detailed in the *JenNet-IP LAN/WAN Stack User Guide (JN-UG-3086)*.

The diagram in Figure 9 below repeats the three basic stack levels but provides a more detailed view of the layers inside for the WPAN side of a JenNet-IP system.

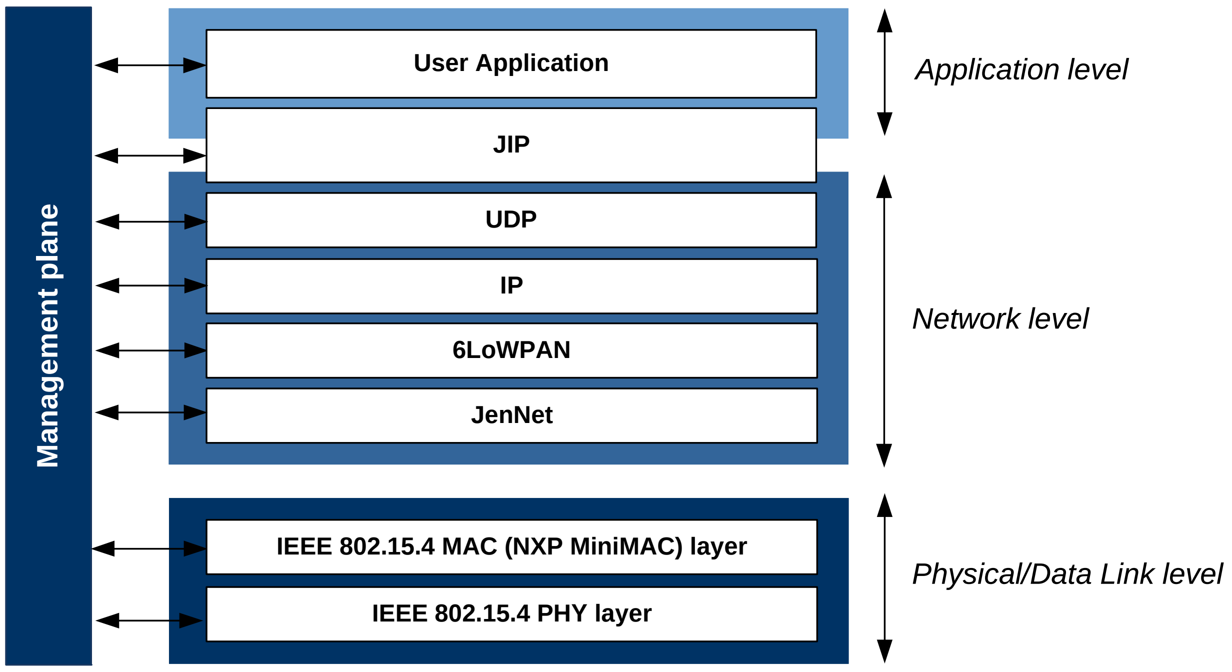


Figure 9: NXP JenNet-IP Stack - WPAN side

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

3.3.1 Application Level

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

User Application

This layer makes use of services provided by the node. The user application interacts with the network through the JIP layer (described below).

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 uses a single communications port (not one of the commonly used ports) on the local node to allow a remote device to set and retrieve values in a MIB (Management Information Base) on the node.

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 3.4.

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

3.3.2 Network Level

The Network level manages communications with the network and comprises the following layers.

Internet Protocols

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

- **UDP:** The User Datagram Protocol (UDP) layer is a simple message-based connectionless protocol. Messages in a JenNet-IP system are implemented as UDP packets embedded in the payloads of IPv6 packets. Thus, this layer is concerned wth assembling/disassembling UDP packets.
- **IP:** The Internet Protocol (IP) layer provides functionality for delivering packets over a network. It is responsible for assembling/disassembling IPv6 packets by inserting/extracting UDP packets, and handling the IPv6 packet headers.

The JenNet-IP Embedded API (introduced in Section 3.4) allows the user application to interact with the UDP and IP layers. Most operations are performed through interactions with the UDP layer.

6LoWPAN

The 6LoWPAN layer provides data compression/decompression and fragmentation services. Messages are transported over the wireless network of a JenNet-IP system inside IEEE 802.15.4 data frames. An IPv6 packet containing the message data is embedded in the payload of an IEEE 802.15.4 frame. However, an IPv6 packet is normally too large to fit in the frame payload. The 6LoWPAN stack layer compresses the packet before it is inserted into the IEEE 802.15.4 frame. If the compressed packet is still too large, 6LoWPAN fragments the compressed packet for transportation in two or more frames. The layer also decompresses the packet extracted from a received frame (and combines fragmented packets, if necessary).

JenNet

JenNet is NXP’s proprietary protocol, which provides the multi-hop capability of the JenNet-IP protocol stack. The JenNet protocol handles network addressing and routing by invoking actions in the IEEE 802.15.4 MAC layer (implemented as the NXP MiniMAC - see Section 3.3.3). Its tasks include:

- Starting the network
- Adding devices to and removing them from the network
- Routing messages (IEEE 802.15.4 data frames) to their intended destinations
- Applying security to outgoing messages

3.3.3 Physical/Data Link Level

This level is provided by the IEEE 802.15.4 standard and consists of two separate layers - the Physical layer and the Data Link layer. These layers together handle the transmission and reception of packets (messages) between two WPAN nodes within radio range of each other.

Tip: In order to develop JenNet-IP WPAN applications, no knowledge of IEEE 802.15.4 is required. However, if you do require more information on IEEE 802.15.4, refer to the *IEEE 802.15.4 Stack User Guide (JN-UG-3024)*.

Data Link Layer

This layer is provided by the IEEE 802.15.4 MAC (Media Access Control) layer. In JenNet-IP, this layer is implemented as the NXP ‘MiniMAC’, which is a cut-down MAC layer to help reduce code-size. The layer is responsible for message delivery, as well as for assembling IEEE 802.15.4 data frames (referred to as MAC frames) to be transmitted and for decomposing received MAC frames.

Physical Layer

This layer is provided by the IEEE 802.15.4 PHY (Physical) layer. It is concerned with the interface to the physical transmission medium, exchanging data bits with this medium, as well as exchanging data bits with the layer above (the Data Link layer).

3.4 Essential JenNet-IP Concepts

JenNet-IP or JIP is NXP's proprietary protocol which provides the user application with access to device functionality. This protocol has associated APIs comprising functions (and associated resources) which facilitate this access:

- **JIP Embedded API:** This is a C API used to develop applications to run on a JN51xx device on nodes of a WPAN. This API is detailed in Part II: JenNet-IP Embedded API.
- **C JIP API and Java JIP API:** These C and Java APIs can be used to develop applications that will run on a LAN/WAN device, such as a PC, tablet or mobile phone. The C JIP API can also be used to develop an application for the LAN/WAN side of a Border-Router and can only be used on Linux-based platforms. These APIs are detailed in the *JenNet-IP LAN/WAN Stack User Guide (JN-UG-3086)*.

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 an appendix of the *JenNet-IP LAN/WAN Stack User Guide (JN-UG-3086).*

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 JenNet-IP stack. MIBs are described further in Section 3.4.1 below.

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 Appendix G.

3.4.1 MIBs and MIB Variables

A MIB (Management Information Base) is a database containing local variables and their values, held in memory on a WPAN node. A MIB allows variables to be collected into a logical group. Up to 255 MIBs can exist on each node. The stack creates five standard MIBs (described in Appendix G.3) and, therefore, the (local) application can create up to 250 MIBs.

The application 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 JIP. 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 (JIP Embedded 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 that defines a MIB type (and the variables within it) and creates a MIB. However, remote applications 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 3.4.2 below.

3.4.2 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 3.4.1) 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.

3.5 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 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 Appendix G.3.

Information held in the standard MIBs on a node can be read by a WPAN application as described in Section 4.2.4 or by a LAN/WAN device application as described in the *JenNet-IP LAN/WAN Stack User Guide (JN-UG-3086)*.

3.6 Network Security

Communications within a WPAN of a JenNet-IP system are mostly secured through encryption based on one of two security keys:

- Commissioning key
- Network key

These two keys are explained below. Note that the node join process, referred to below, is described more fully in Section 2.8.

Commissioning Key

The commissioning key is unique to a particular node and is only used when the node joins the network. This key is held on the node and must also be provided to the Border-Router before the node joins the network, possibly as part of a ‘white list’ of all nodes that are allowed to join the network. This information can be supplied to the Border-Router by an ‘out-of-band’ (non-wireless) means - for example, from a remote device via the Internet.

When a Router or the Co-ordinator receives a join request from a potential child node, the request is encrypted using the node’s commissioning key and is initially rejected by the prospective parent. The latter device then obtains the relevant commissioning key from the Border-Router (see Note below). Another similarly encrypted join request is then received from the same node and is decrypted using the commissioning key obtained. The decryption must be successful in order for the join process for the node to continue and the parent shares the network key with the child node, encrypted using the commissioning key. All subsequent communication is encrypted using the network key (see below).

Note: In practice, a Router will request a commissioning key from the Co-ordinator. The application on the Co-ordinator then obtains the relevant key from the Border-Router and broadcasts it to all Router nodes.

Network Key

The network key is known to all nodes of the network and is used to encrypt all internal network communications in which a node is involved after it has joined the network. This key is supplied by the Co-ordinator as part of the node join process - it is included in the Co-ordinator's response to an Establish Route message from the node (sent after the node has been accepted by a parent). Note that the Establish Route message and its response are themselves encrypted with the network key between the parent and the Co-ordinator, but with the joining node's commissioning key between the node and its parent (since the network key is not known to the node at this stage). Once a node has the network key, it may save the key to non-volatile memory in order to make any future rejoins more efficient.

3.7 JenNet Network Profiles

The operational properties of a WPAN in a JenNet-IP system are pre-configured via a set of JenNet network parameters (detailed in Chapter 9). Nine of these parameters are collected together in a network profile, which defines a combination of well-matched values for these parameters. Each profile has a unique index in the range 0-255. Thus, rather than setting each parameter value individually, a profile allows a group of parameter values to be set collectively by simply referencing the profile index.

The network profile is set on the Co-ordinator. Other devices inherit these parameter values from their parent when they join the network. It is, however, possible to override these parameter values from the application (see below).

A set of ten standard network profiles are supplied with the JenNet-IP software. They are numbered 0-9, where profile 0 is the default profile (and is used if no other profile is selected) and profiles 8-9 are for the standalone WPANs described in Chapter 11.

Note: The JenNet network profile parameters and the standard network profiles are described in Section 9.2.

The most appropriate profile to choose depends on both the network size (the total number of nodes in the network) and the tree type. The tree type is an indication of the density of the network - it is recommended that a profile for a tree type of:

- "Sparse" is selected for a network with up to 5 nodes per room
- "Bushy" is selected otherwise

Refer to Table 8 in Section 9.2 for details of the standard “Sparse” and “Bushy” profiles. However, these are only guidelines and experimentation with other settings may yield benefits in particular environments.

The network profile can be selected either from a device on the LAN/WAN via the Border-Router or from within the applications on the WPAN nodes.

Remotely via Border-Router

The network profile can be chosen remotely from a device such as a PC on the LAN/WAN side of the JenNet-IP system. This selection is performed using an interface which allows interaction with the Border-Router for the network - for example, the NXP JenNet-IP Border-Router Configuration interface which is built into the Linksys router provided in NXP evaluation kits that support JenNet-IP. In this case, the relevant option is provided on the **6LoWPANd** sub-tab of the **JenNet-IP** tab.

This is the preferred profile selection method for a network with a Border-Router.

The profile is passed to each node as it joins the network. Therefore, if the profile is changed, it may be necessary to re-start the network for the change to take effect.

However, when using standard profiles (only), it is possible to change the profile used by a running network at any time through the Co-ordinator without having to re-start the network. The JenNet-IP Border-Router Configuration interface (mentioned above) uses this facility to implement an automatic profile selection option - if selected, the most appropriate standard profile is automatically set based on the number of nodes in the network and their rate of joining (the profile is updated as the network evolves).

Within the WPAN Applications

The application on a node can over-ride the network profile that is passed to the node as it joins the network. If this profile selection method is used, it is important to ensure that all nodes in the network are set to use the same profile.

Since the profile is set in the application, this method is not suitable for an application that may be deployed in a variety of environments, unless there is a way for the user to configure the setting on each node prior to adding it to the network.

Note: Functions to set and change the network profile are described in Section 5.3.

3.8 Fundamental Operations in JenNet-IP

The fundamental operations on devices in a JenNet-IP system are as follows:

- The control of devices is achieved by writing to MIB variables on the devices
- The monitoring of devices is achieved by reading MIB variables on the devices

Therefore, the applications that run on the devices to be monitored/controlled and on the devices to perform the monitoring/control must facilitate these operations, as follows:

- On a device to be monitored and/or controlled (normally a WPAN node):
    - To facilitate monitoring, the application must create MIBs and associated variables and react to write requests for these MIB variables
    - To facilitate control, the application must create MIBs and associated variables, populate them with data and react to read requests for these MIB variables
- On a device to perform monitoring and/or control (normally a LAN/WAN device):
    - To facilitate monitoring, the application must identify the IPv6 addresses of devices to be monitored and either submit read requests for MIB variables on the target devices or configure and receive traps on the MIB variables
    - To facilitate control, the application must identify the IPv6 addresses of devices to be controlled and submit write requests for MIB variables on the target devices

The coding of these operations in an application is described in Section 4.2 for a WPAN node and in the *JenNet-IP LAN/WAN Stack User Guide (JN-UG-3086)* for a LAN/WAN device.

3.9 Low-Energy Devices

JenNet-IP provides support for WPAN ‘low-energy devices’ which have very limited energy resources. These devices include:

- Devices that are completely self-powered through energy harvesting
- Battery-powered devices that require ultra-long battery life

Typical devices of this type are switches (e.g. light-switch), panic/emergency buttons, detectors and sensors. The energy harvesting devices can be ‘bursting energy harvesters’ which generate and store energy in a very short time by electromechanical means (such as flipping a switch) or ‘trickling energy harvesters’ which generate and store energy over a long period of time (such as from solar cells).

3.9.1 Principles of Low-Energy Devices

JenNet-IP minimises the power demands on low-energy devices by:

- Employing IEEE 802.15.4 frames that carry the minimum payload necessary to be useful and secure, thus minimising the amount of energy needed for each frame transmission
- Not requiring these devices to be full members of the network and allowing them to only transmit data when they need to (e.g. when a button on the device is pressed)

In order to minimise energy (and memory) usage, a low-energy device employs a reduced software stack. It does not run the JenNet-IP stack and transmit JenNet-IP frames. Instead, a special cut-down version of the IEEE 802.15.4 stack is used in which the MAC layer is replaced by an NXP-adapted ‘MicroMAC’ layer.

The use of a low-energy device in conjunction with a JenNet-IP network is illustrated in Figure 10 below.

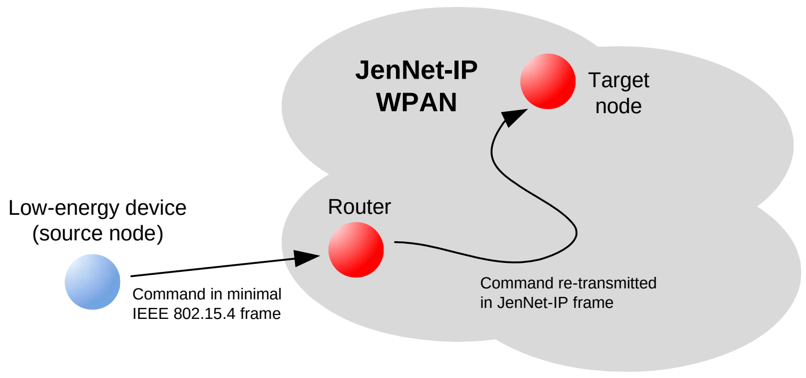

JenNet-IP
WPAN
Target node
Router
Low-energy device (source node)
Command in minimal IEEE 802.15.4 frame
Command re-transmitted in JenNet-IP frame

Figure 10: Low-Energy Device and JenNet-IP WPAN

A command from a low-energy device is forwarded within the WPAN as a multicast in a JenNet-IP frame.

Low-energy devices cannot receive frames from a JenNet-IP network.

3.9.2 Configuration of Low-Energy Devices

Low-energy devices are not full members of the network and cannot be configured from within the JenNet-IP system. They must be pre-configured (either in the factory or during installation using hardware switches or a tool) with the following:

- 64-bit IEEE/MAC address to be used to identify the device (always factory-set)
- 128-bit security key to be used to authenticate communications with the WPAN
- Fixed 2.4-GHz radio channel (11-26) for communication with the WPAN

The WPAN will need to operate in the fixed radio channel of the low-energy devices.

The IEEE/MAC address is used to generate an IPv6 multicast address that will be used to identify the destination nodes for transmissions from the device.

3.9.3 Registering a Low-Energy Device with a WPAN

The WPAN Co-ordinator requires prior knowledge of the low-energy devices that will be permitted to operate with the network. For this purpose, the Co-ordinator must have access to a ‘white list’ of the IEEE/MAC addresses of these devices as well as their security keys.

A low-energy device must first be registered with the network as follows:

1. The low-energy device transmits an IEEE 802.15.4 frame (containing its IEEE/MAC address and security key) to the WPAN.
2. A WPAN Router which receives this frame will recognise it as a ‘low-energy frame’ and will send a ‘Low Energy Request’ (containing the device’s IEEE/MAC address) in a JenNet-IP frame to the Co-ordinator.
3. On reaching the Co-ordinator, the request will be passed to the application. It is then the responsibility of the application to determine whether the device’s IEEE/MAC address is in the white list of permissible low-energy devices.

   The Co-ordinator may, alternatively, receive the low-energy frame directly, in which case it will generate the ‘Low Energy Request’ itself.
4. If the device is accepted, the application on the Co-ordinator must broadcast a message through the network, informing other nodes of the admission of the low-energy device (and passing on its IEEE/MAC address and security key).

Subsequently, when a Router receives a frame from the low-energy device, it will check that the device is in the list of registered low-energy devices. If this is the case, the Router will re-transmit the payload in a JenNet-IP frame - this will be a multicast with a multicast address derived from the IEEE/MAC address of the low-energy device.

Note: Full details of incorporating low-energy devices in a JenNet-IP system are provided in Section 4.9. Implementing the MicroMAC stack on a low-energy device is described in Appendix L.

Part II:
JenNet-IP Embedded API

4. WPAN Application Development

The chapter outlines the main tasks that you are likely to incorporate in a WPAN application. References are made to the JIP Embedded API functions that you will need to use in your code. Note that different application programs will be required for different network nodes, according to the node type (Co-ordinator, Router, End Device) and functionality (e.g. environmental monitor, light sensor, fire detector).

The essential application tasks covered in this chapter are:

- Starting and forming a JenNet-IP WPAN - see Section 4.1
- Storing data on nodes and transferring data from one node to another (perhaps in a different WPAN) - Section 4.2
- Forming multicast groups (of nodes) - see Section 4.3
- Obtaining error information arising from network operation - see Section 4.4
- Handling events generated during network operation - Section 4.5
- Entering and leaving sleep mode - see Section 4.6
- Polling for data (by an End Device) - see Section 4.7
- Persisting context data - see Section 4.8
- Using low-energy devices - see Section 4.9

Note: Details of all the API functions and associated resources referenced in this chapter can be found in Chapter 5 to Chapter 8.

4.1 Starting and Forming a WPAN

The process for starting and forming a WPAN requires a 'cold start' to be performed on each of the network nodes - first on the Co-ordinator, then on the other nodes, which each joins the network by associating with either the Co-ordinator or a previously joined Router.

The exception to this type of start is on a device which has woken from sleep with memory held (see Section 2.9.2), and needs to re-start and re-take its place in the network - this situation requires a 'warm start' to be performed on the device.

The cold start and warm start cases are covered separately in the subsections below.

4.1.1 Performing a Cold Start

A cold start is performed on a node which is starting (Co-ordinator) or joining a WPAN. The application code for a cold start is similar for all the nodes in the network, but the data provided for the stack initialisation depends on the node type.

On node power-up (or reset), the node’s wireless microcontroller calls the user-defined **AppColdStart()** routine which forms the entry point into the application, allowing you to perform hardware and software initialisation, and start the main application loop. The essential function calls in the **AppColdStart()** routine are illustrated in Figure 11 and described below.

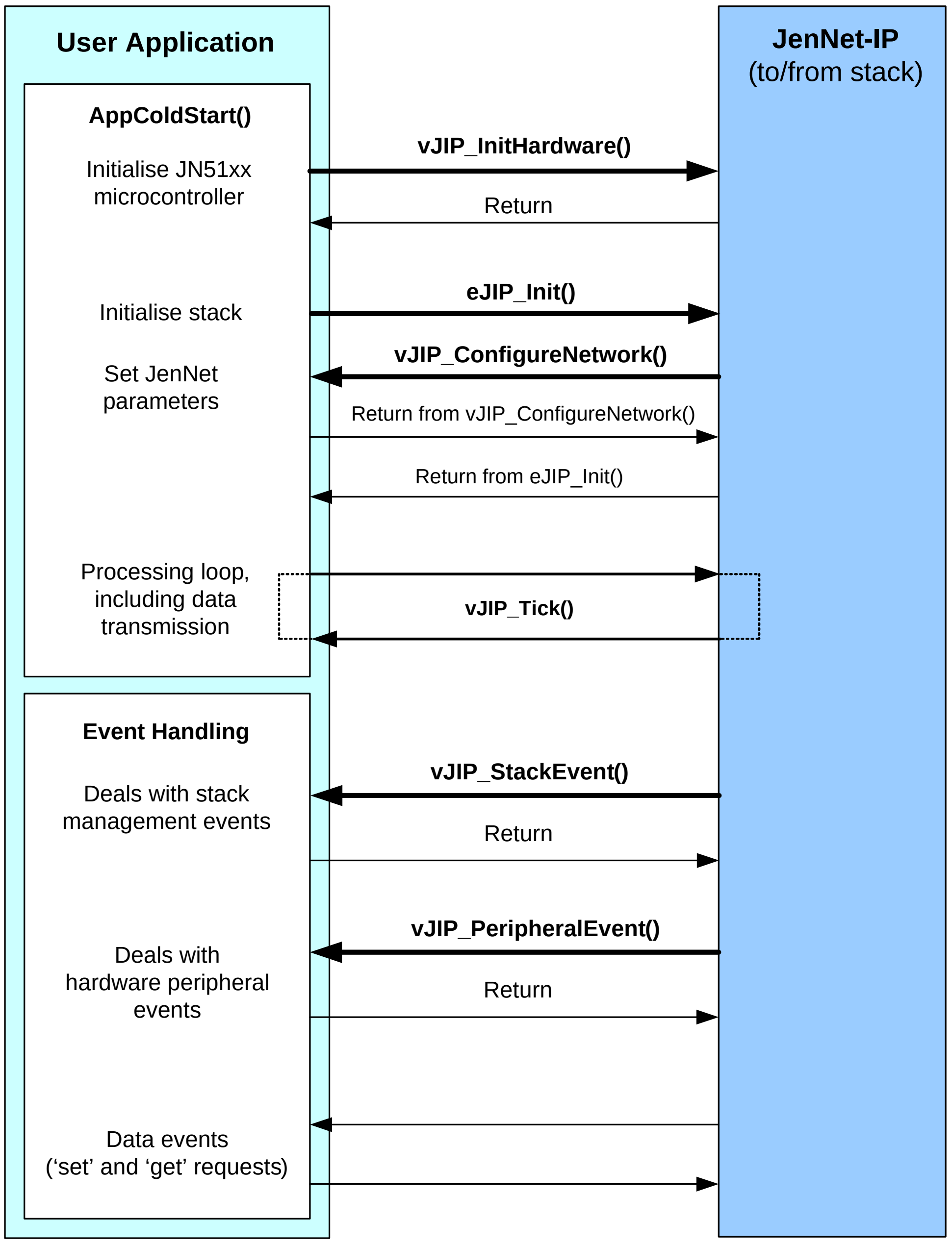


Figure 11: Function Calls in a Cold Start

Within the **AppColdStart()** routine, the following functions must be called:

1. **v6LP_InitHardware():** This function initialises the JN51xx microcontroller. It requires no parameter values.
2. **eJIP_Init():** This function initialises the protocol stack and accepts a `tsJIP_InitData` data structure containing stack initialisation data. The data supplied through this structure depends on the node type. The elements of the structure include the following (among others):
   - **Device type:** Co-ordinator, Router or End Device.
   - **PAN ID:** In the case of the Co-ordinator, this is the PAN ID to be used for the network (if the value 0xFFFF is used, the Co-ordinator will choose a random PAN ID which does not clash with the PAN IDs of other networks). For other nodes, the initial value set for the PAN ID is not important.
   - **Radio channel:** In the case of the Co-ordinator, this can be either a fixed 2.4-GHz channel on which the network will operate or a set of channels from which the Co-ordinator will determine the best channel for the network. Similarly, for other nodes, this can be either an individual channel or a set of channels on which the node will search for a potential parent.
   - **IPv6 address prefix (Co-ordinator only):** A 64-bit prefix that will be applied in assigning IPv6 addresses to all nodes (see Section 3.3.2) - this prefix will be passed from the Co-ordinator to all joining nodes.

   Full details of the `tsJIP_InitData` structure and its elements are given in Section 8.1.1.

   Provided that the JenNet protocol is enabled, the **eJIP_Init()** function invokes the callback function **vJIP_ConfigureNetwork()**. This is a user-defined callback function used to set values for the JenNet network parameters (listed and described in Section 9.1 and Section 9.2).

Note 1: Some JenNet network parameters are included in a profile (see Section 3.7 and Section 9.2). Functions are provided for setting and accessing these profile parameters, and are detailed in Section 5.3. If required, these functions should be called in the callback function **vJIP_ConfigureNetwork()**.

Note 2: The JenNet-IP stack uses one of the JN51xx on-chip timers. By default, this is the Tick Timer, but an alternative timer can be selected using the JenNet parameter `u8InternalTimer` (see Section 9.1). Whichever timer is used by the stack, the application must not use this timer for any other purpose.

Note 3: If JenNet security is to be implemented (see Section 3.6), the function **vJIP_EnableSecurity()** must also be called within the callback function **vJIP_ConfigureNetwork()**.

Use of the **v6LP_InitHardware()** and **eJIP_Init()** functions is illustrated in the code fragment below.

```
PUBLIC void AppColdStart(void)
{
    /* Stack initialisation data structure */
    tsJIP_InitData sJipInitData;

    /* Initialise hardware */
    v6LP_InitHardware();

    /* Initialise application hardware... */

    /* Configure stack */
    sJipInitData.u64AddressPrefix       = 0x1234ULL;
    sJipInitData.u32Channel             = (1 << 11) | (1 << 12);
    sJipInitData.u16PanId               = 0xffff;
    sJipInitData.u16MaxIpPacketSize     = 0;
    sJipInitData.u16NumPacketBuffers    = 4;
    sJipInitData.u8UdpSockets           = 2;
    sJipInitData.eDeviceType            = E_JIP_DEVICE_COORDINATOR;
    sJipInitData.u32RoutingTableEntries = 200;
    sJipInitData.u32DeviceId            = 0x12345678;
    sJipInitData.u8UniqueWatchers       = 16;
    sJipInitData.u8MaxTraps             = 64;
    sJipInitData.u8QueueLength          = 16;
    sJipInitData.u8MaxNameLength        = 16;
    sJipInitData.u16Port                = 1873;
    sJipInitData.u16JMP_Port            = 1875;
    sJipInitData.pcVersion              = "VersionString";

    /* Attempt to initialise the JenNet-IP stack */
    if (eJIP_Init(&sJipInitData) != E_JIP_OK)
    {
        /* Stack initialisation failed, stop */
        while(1);
    }

    /* Call main application code... */
}
```

Following a cold start, the application must call the function **vJIP_Tick()** in the main processing loop. This function generates the necessary stack and data events, which must be handled by the application as described in Section 4.5.

If JenNet security has been enabled using **vJIP_EnableSecurity()**, the stack event E_STACK_NODE_AUTHORISE is generated on the Co-ordinator when a Router node needs the commissioning key of another node that is attempting to join it (see Section 3.6). The Co-ordinator application must then obtain the relevant commissioning key from the Border-Router and pass the key into the network using the JenNet function **eApi_CommissionNode()**, described on page 99.

Tip: The network formation/joining process can be speeded up by employing fast commissioning mode, described in Section 4.1.3.

4.1.2 Performing a Warm Start

A warm start is performed on an End Device that wakes from sleep with memory held and allows the node to resume its previous operation in the network.

Note: A warm start will only be performed following sleep with memory held. The latter can optionally be enabled in the call to the function **vJIP_Sleep()**. If memory is not held during sleep, a cold start will be performed on waking, as described in Section 4.1.1.

On waking from sleep with memory held, the node's JN51xx microcontroller calls the user-defined **AppWarmStart()** routine, which forms the entry point into the application, allowing the application to perform hardware and software re-initialisation, and start the main application loop. The essential function calls in the **AppWarmStart()** routine are illustrated in Figure 12 and described below.

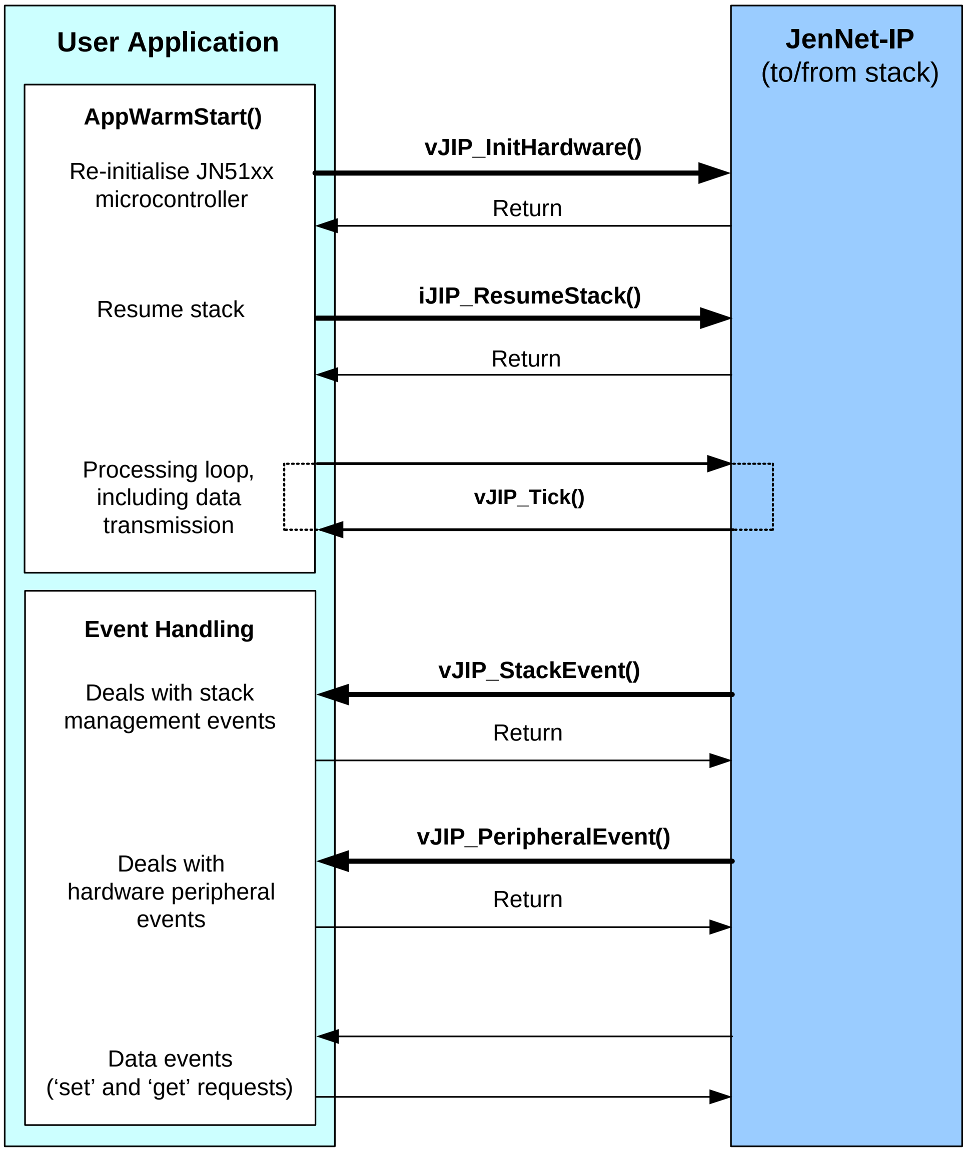


Figure 12: Function Calls in a Warm Start

Within the **AppWarmStart()** routine, the following functions must be called:

1. **v6LP_InitHardware():** This function initialises the JN51xx microcontroller. It requires no parameter values.
2. **iJIP_ResumeStack():** This function resumes the protocol stack from the state it was in before the node entered sleep mode. The function assumes that stack context data has been preserved in on-chip memory during sleep.

Use of the **iJIP_ResumeStack()** function is illustrated in the code fragment below.

```
PUBLIC void AppWarmStart(void)
{
    if (iJIP_ResumeStack() != 0)
    {
        /* Failed to start */
    }

    /* Application can now resume... */
}
```

Following a warm start, the application must call the function **vJIP_Tick()** in the main processing loop. This function generates the necessary stack and data events, which must be handled by the application as described in Section 4.5.

4.1.3 Fast Commissioning Mode

In the network formation/joining process described in Section 4.1.1, a joining node scans a pre-configured set of channels in searching for a network to join (see Section 2.8). Depending on the channel in which the network operates, this scan may take a significant length of time to find the network. Fast commissioning mode reduces this time by using a pre-configured fixed channel for commissioning, which is known by all potential nodes and is different from the channel used for normal network operation.

Fast commissioning mode can be used to add Router nodes to a network (but not End Devices). Therefore, the joining device is always a Router.

4.1.3.1 Principles of Fast Commissioning

The basic principles of fast commissioning mode are as follows:

- **Co-ordinator:** The Co-ordinator enters fast commissioning mode (as the result of a user input) and transmits Network Announce messages in the fixed fast commissioning channel for a certain length of time. This message contains the PAN ID, network key and operational channel of the network, as well as a special pre-configured fast commissioning PAN ID and security key.
- **Router:** The joining node ‘listens’ for Network Announce messages in the fast commissioning channel. On receiving a Network Announce message containing a valid network key as well as the correct fast commissioning PAN ID and security key, the node will attempt to join the source node of the message (it will need to receive two such messages from the same source to properly configure security). The join attempt will be performed using the normal operational channel and PAN ID of the network.

On the joining node, fast commissioning mode is implemented between normal channel scans. Normally, after scanning one channel, there is a random ‘back-off’ period before scanning another channel. If enabled, fast commissioning (listening for Network Announce messages) is performed during this back-off period.

If a join attempt fails, the joining node will return to scanning and fast commissioning.

4.1.3.2 Coding Fast Commissioning

If required, fast commissioning mode must be pre-configured by the application. This configuration is performed in the user-defined callback function **vJIP_ConfigureNetwork()**, which is invoked by the initialisation function **eJIP_Init()**. This must be done on both the Co-ordinator node and the (potential) Router nodes.

To enable and configure fast commissoning mode, the following functions must be called within the above callback function:

- **vApi_ConfigureFastCommission():** This function allows the following fast commissioning parameters to be specified:
    - **Channel** - this is the fixed channel in which fast commissioning will be performed (that is, in which Network Announce messages will be sent and received)
    - **PAN ID** - this is a fixed 16-bit PAN ID which identifies a network that is in fast commissoning mode
- **vSecurityUpdateKey():** This function is used to specify the fast commissioning security key which is used to authenticate messages exchanged during fast commissioning. This is a JenNet function which is described on page 98.

On the Co-ordinator node, fast commissioning mode should be initiated by a user input (e.g. pressing a button) which causes the application to issue a series of Network Announce messages. Each message is transmitted by calling the function **eApi_SendNetworkAnnounceEnhanced()**.

4.2 Storing and Transferring Data

This section describes how data is stored within the nodes of a JenNet-IP WPAN and transferred between nodes (where the source and destination nodes may be in different WPANs).

On a JenNet-IP node, data is stored as variables in a MIB (MIBs and MIB variables are introduced in Section 3.4.1). Transferring data to/from a JenNet-IP node involves writing/reading MIB variables on the node. A node may have more than one MIB (up to 255 MIBs, in fact), where each MIB is of a particular MIB type (with unique MIB type identifier) defined as described in Section 4.2.1 - for example, a MIB type may contain environmental data, such as the last temperature, humidity and wind-speed readings.

The JIP Embedded API is used to manage MIBs, providing functionality to:

- Create a MIB (and its associated variables)
- Discover MIBs and their variables
- Remotely set MIB variable values
- Remotely retrieve MIB variable values
- Monitor MIB variables locally or remotely

The above MIB management tasks are described in the sub-sections below.

Note: Data delivery in a JenNet-IP system requires a destination IPv6 address, UDP port and JMP port. A ‘socket’ mechanism is implemented to ensure that data arriving for a particular IPv6 address and port is routed to the relevant application. Both outgoing and incoming data packets are buffered by the stack on the local node. Sockets and buffering are both handled by the JIP Embedded API, and are transparent to the application.

4.2.1 Creating a MIB and its Variables

A MIB is based on a MIB type, which is statically defined as part of the application initialisation. One or more MIBs of a defined MIB type can be created on a node (however, the current release is restricted to only one MIB of each type per node).

Defining a MIB Type

A MIB type is defined as a `tsJIP_MibDef` structure (see Section 8.1.6) which contains a 32-bit MIB type identifier and the MIB variables. A MIB type can contain up to 255 variables, where each variable is defined in a `tsJIP_VarDef` structure (see Section 8.1.7) containing information which includes:

- Name (character string) for the variable
- 8-bit identifier for the variable
- Type of variable
- Access permissions
- Value indicating valid lifetime of variable (and therefore cache refresh rate)
- Value indicating security applied to the variable

Macros are supplied by the JIP Embedded API to aid in the definition of a MIB type and its variables - the macros will fill in the relevant structures. The MIB type definition macros (to be used in the given order) are:

START_DEFINE_MIB() to start the definition process

DEFINE_VAR() to define a variable (must be used for each variable)

END_DEFINE_MIB() to finish the definition process

The above macros are detailed in Section 6.1.1.

Creating and Registering a MIB

Once a MIB type (and its variables) has been defined, a MIB based on the type can be created through a `tsJIP_MibInst` structure which includes:

- Index value which identifies MIB
- Pointers to user-defined callback functions used to Set and Get variable values

Macros are supplied by the JIP Embedded API to aid in the creation of a MIB - the macros will fill in the relevant structure. The MIB declaration macros (to be used in the given order) are:

JIP_START_DECLARE_MIB() to start the declaration process

JIP_CALLBACK() to declare a variable of the MIB and specify the Set/Get callback functions for the variable (must be used for each variable of the MIB)

JIP_END_DECLARE_MIB() to finish the declaration process

The above macros are detailed in Section 6.1.2.

A MIB must be registered with JenNet-IP using the **eJIP_RegisterMib()** function. Up to 255 MIBs can be registered per node, each through a separate call to this function.

4.2.2 Remotely Discovering MIBs

The JIP Embedded API provides functions which allow an application to obtain information on the MIBs on a remote node and the variables within the MIBs.

4.2.2.1 Obtaining List of MIBs

The function **eJIP_Remote_QueryMib()** can be used to obtain a list of the MIBs on a remote node (an IPv6 address and port must be specified). It may not be possible to return the full list of all MIBs from a single call to this function (due to a limit on the payload of the UDP packet in which the results are returned). Therefore, more than one call to the function may be required to return the full MIB list, with each call requesting a specific range of consecutive MIBs. In each call, you must specify the:

- index value (in the range 0-255) of the first MIB to be reported in the resulting list - the first time the function is called, this parameter should be set to zero and then increased accordingly for subsequent calls
- maximum number of MIBs to be returned in the list (but this itself will be limited by the UDP payload size)

This function simply submits a request for a list of MIBs on the remote node and returns immediately (the function is non-blocking). Feedback on the success of this request is handled by two user-defined callback functions, as follows:

- The callback function **vJIP_Remote_DataSent()** is called by the stack to report the outcome of the attempt to send the ‘query MIB’ request, i.e. whether the request was successfully transmitted. Note that this function is also used when sending other JIP requests.
- The callback function **vJIP_Remote_QueryMibResponse()** is called by the stack to report whether the request was successful on the remote node, i.e. whether the list of MIBs was successfully obtained. The function also reports the list obtained and indicates the number of MIBs which remain unreported (requiring further calls to **eJIP_Remote_QueryMib()**).

4.2.2.2 Obtaining List of Variables in a MIB

The function **eJIP_Remote_QueryVar()** can be used to obtain a list of the variables in a specified MIB on a remote node. It may not be possible to return the full list of MIB variables from a single call to this function (due to a limit on the payload of the UDP packet in which the results are returned). Therefore, more than one call to the function may be required to return the full list, with each call requesting a specific range of consecutive MIB variables. In each function call, you must specify the:

- index value (in the range 0-255) of the first MIB variable to be reported in the resulting list - the first time the function is called, this parameter should be set to zero and then increased accordingly for subsequent calls
- maximum number of variables to be returned in the list (but this itself will be limited by the UDP payload size)

This function simply submits a request for a list of MIB variables and returns immediately (the function is non-blocking). Feedback on the success of this request is handled by two user-defined callback functions, as follows:

- The callback function **vJIP_Remote_DataSent()** is called by the stack to report the outcome of the attempt to send the ‘query variable’ request, i.e. whether the request was successfully transmitted. Note that this function is also used when sending other JIP requests.
- The callback function **vJIP_Remote_QueryVarResponse()** is called by the stack to report whether the request was successful on the remote node, i.e. whether the list of MIB variables was successfully obtained. The function also reports the list obtained and indicates the number of variables which remain unreported (requiring further calls to **eJIP_Remote_QueryVar()**).

4.2.3 Remotely Setting MIB Variable Values

MIB variable values should be updated by the application on the local node - for example, when a sensor reading changes, the value of the corresponding local MIB variable should be updated. However, the JIP Embedded API allows the value of a MIB variable to be set by a remote application using **eJIP_Remote_ID_Set()**. This function simply submits a request to set the value of a remote MIB variable and returns immediately (the function is non-blocking). Feedback on the success of this request is handled by two user-defined callback functions, as follows:

- The callback function **vJIP_Remote_DataSent()** is called by the stack to report the outcome of the attempt to send the ‘Set variable’ request, i.e. whether the request was successfully transmitted. Note that this function is also used when sending other JIP requests.
- The callback function **vJIP_Remote_SetResponse()** is called by the stack to report whether the request was successful on the remote node, i.e. whether the remote MIB variable was successfully set.

Use of the above functions in remotely setting the value of a MIB variable is illustrated in Figure 13 below.

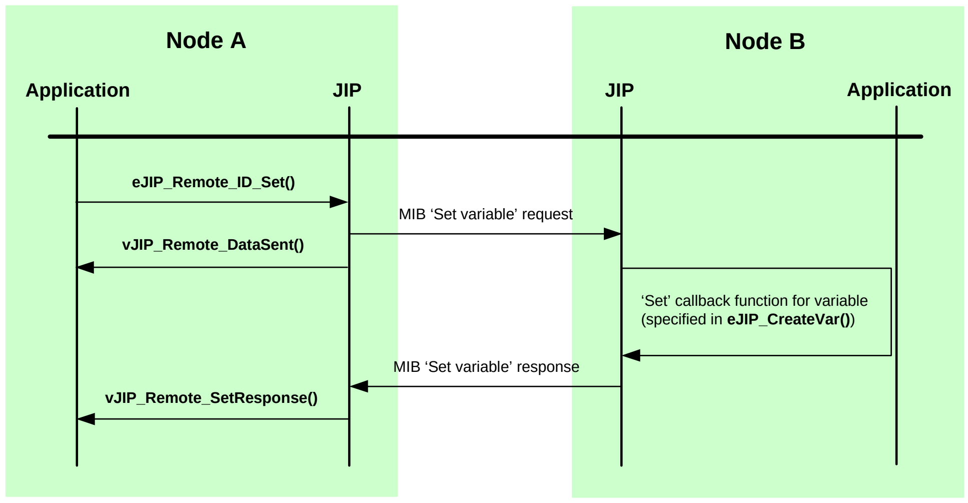

Node A
Application
JIP
Node B
JIP
Application
eJIP_Remote_ID_Set()
MIB 'Set variable' request
vJIP_Remote_DataSent()
'Set' callback function for variable (specified in eJIP_CreateVar())
MIB 'Set variable' response
vJIP_Remote_SetResponse()

Figure 13: Remotely Setting a MIB Variable

4.2.4 Remotely Obtaining MIB Variable Values

The value of a MIB variable can be remotely obtained using the function **eJIP_Remote_TableGet()**. This function simply submits a request to get the value of a remote MIB variable, which may or may not be of the table datatype, and returns immediately (the function is non-blocking). Feedback on the success of this request is handled by two user-defined callback functions, as follows:

- The callback function **vJIP_Remote_DataSent()** is called by the stack to report the outcome of the attempt to send the 'Get variable' request, i.e. whether the request was successfully transmitted. Note that this function is also used when sending other JIP requests.
- The callback function **vJIP_Remote_TableGetResponse()** is called by the stack only if the MIB variable is a table. The function is called for each table entry received in the 'Get variable' response in order to report the entry. It is called at least once, even if no table entries were returned in the response (e.g. the requested entry is not present).
- The callback function **vJIP_Remote_GetResponse()** is called by the stack to report whether the request was successful on the remote node, i.e. whether the remote MIB variable was successfully retrieved. For a MIB variable which is not the table datatype, the function also reports the obtained value.

Use of the above functions in remotely getting the value of a MIB variable is illustrated in Figure 14 below.

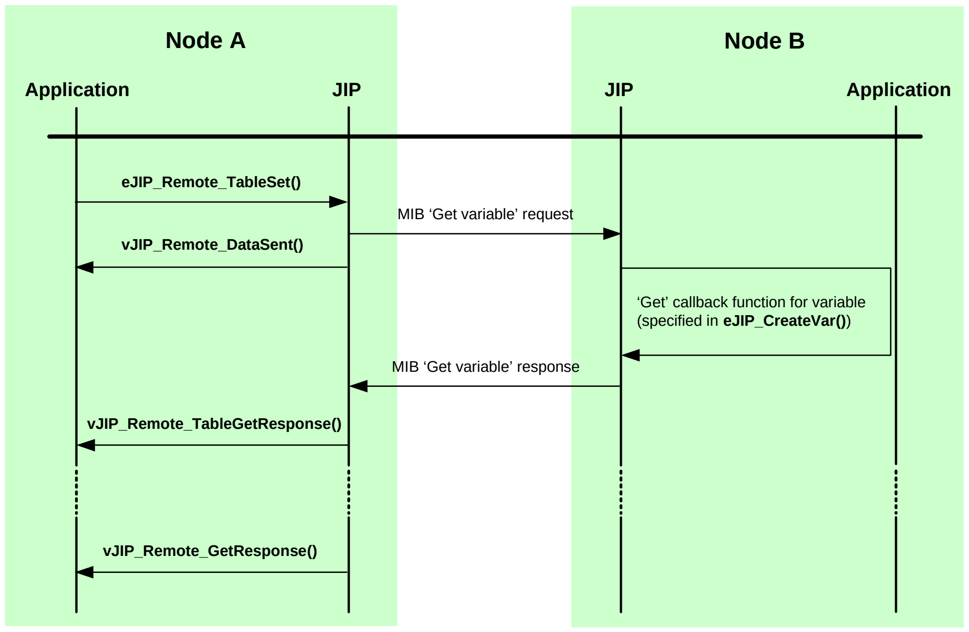

Node A
Node B
Application
JIP
JIP
Application
eJIP_Remote_TableSet()
MIB 'Get variable' request
vJIP_Remote_DataSent()
'Get' callback function for variable
(specified in eJIP_CreateVar())
MIB 'Get variable' response
vJIP_Remote_TableGetResponse()
vJIP_Remote_GetResponse()

Figure 14: Remotely Getting a MIB Variable

4.2.5 Remote Monitoring of MIB Variables (using Traps)

The variables in a MIB can be automatically monitored by remote applications using JIP 'traps', which work on a similar principle to the industry-standard SNMP traps. A trap is a mechanism which is associated with a MIB variable and which generates a notification whenever the value of the variable is changed in the MIB. This notification is sent to all remote nodes that have registered an interest in the variable.

To use traps, actions must be performed on the local node and on the remote nodes, as described below.

On Local Node

The generation of a trap notification for a MIB variable is triggered by calling the function **vJIP_NotifyChanged()** on the local node. This function may be called by the application whenever the value of a local MIB variable is changed or whenever the enabled state of the variable is changed (which determines whether or not the variable can be accessed). In order to reduce network traffic, the application may call this function selectively - for example, only when a variable value changes by at least a certain minimum amount. The trap notification message generated will be then be sent to any remote nodes that are trapping the variable.

On Remote Node

In order to automatically receive trap notifications when the value of a MIB variable is updated, the remote node must register an interest in the variable by calling the function **eJIP_Remote_Trap()**. This function simply submits a request to trap the MIB variable and returns immediately (the function is non-blocking). Feedback on the success of this request is handled by two user-defined callback functions, as follows:

- The callback function **vJIP_Remote_DataSent()** is called by the stack to report the outcome of the attempt to send the 'remote trap' request, i.e. whether the request was successfully transmitted. Note that this function is also used when sending other JIP requests.
- The callback function **vJIP_Remote_TrapResponse()** is called by the stack to report whether the request was successful on the remote node, i.e. whether the remote MIB variable was successfully trapped.

When a notification for a trapped variable is received on the remote node, the stack calls the user-defined callback function **vJIP_Remote_TrapNotify()** to handle the trap. This function will report the new value of the trapped variable. For more information on the generation of trap notifications, see "On Local Node" above.

A trapped MIB variable can later be untrapped by calling **eJIP_Remote_Untrap()**. Again, this function simply submits a request to untrap the remote MIB variable and returns immediately (the function is non-blocking). Feedback on the success of this request is handled by the above two user-defined callback functions.

Caution: *A trap that has been set up on a remote MIB variable is lost if either the remote node or the local device is restarted.*

Note: As an alternative to the above, a trap can be set up on a local MIB variable using the function **eJIP_AddTrap()**. In this case, trap notifications will be sent to a specified IPv6 multicast address and processed by all nodes in the corresponding multicast group. For details of multicast groups, see Section 4.3.

4.3 Forming Multicast Groups

An IPv6 packet can be sent to (selected) multiple devices. This type of transmission is referred to as a ‘multicast’.

IPv6 provides the facility of a multicast group, which has a unique IPv6 multicast address (described in the *JenNet-IP LAN/WAN Stack User Guide (JN-UG-3086)*). The stack on each device that belongs to a multicast group stores the IPv6 multicast address of the group. A device can belong to more than one multicast group.

An IPv6 packet containing a multicast address is actually broadcast by the source node on each of its links. The extent of the broadcast is determined by the specified ‘scope’ - see the *JenNet-IP LAN/WAN Stack User Guide (JN-UG-3086)*. The stack on each receiving device determines whether the device is a member of the group corresponding to the multicast address in the packet (and therefore whether to accept the packet). If the receiving device is a Router, it will also need to pass on the packet. An IP Router 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.

Functions are provided to add the local node to and remove it from a multicast group:

- **bJIP_AddGroupAddr()** is used to add the local node to a multicast group
- **bJIP_RemoveGroupAddr()** is used to remove the local node from a multicast group

Before processing the add or remove request, both of these functions invoke the user-defined callback function **bJIP_GroupCallback()** to authorise (or refuse) the request. If the callback function returns an authorisation, the original function modifies the relevant MIB to fulfil the request.

Therefore, to form a multicast group, an IPv6 multicast address must be assigned to the group and the function **bJIP_AddGroupAddr()** must be called on each node which is to be a member of the group.

Remote access to the IPv6 multicast addresses of the groups to which a node belongs is provided via the Groups module (MIB) on the node (for details of this MIB, refer to Appendix G.3.3).

Note 1: The Groups MIB on a node can be used to remotely add the node to and remove the node from multicast groups.

Note 2: Although multicast messages may be received by an End Device (that is awake), these messages are discarded by the device. Therefore, End Devices should not be added to a multicast group.

4.4 Obtaining Error Reports

Some functions of the JIP Embedded API return a value which, if non-zero, signifies that an error has occurred and that 'extended error' information is available. Details of this error can then be obtained using the function **u32JIP_GetErrNo()**. When a function call has produced an error, to obtain the extended error information **u32JIP_GetErrNo()** must be called immediately - that is, it must be called before any other function (since the next function call will reset the extended error value).

The extended error information returned by **u32JIP_GetErrNo()** is contained in a 32-bit value comprising three distinct parts:

- Bits 7-0 give an 'error code' which indicates the source of the error
- Bits 15-8 give 'error information' which supplements the error code
- Bits 31-16 are reserved for future use

Details of the error codes and information are provided in Section 8.5.

4.5 Handling Events

Events are generated by the function **vJIP_Tick()**, which must be called in the main processing loop of the application. The following event types may be generated and require a user-defined event handler to be implemented as a callback function:

- Stack events
- Data events
- Peripheral events

These event categories and their handlers are described in the sub-sections below.

Note: Application callback functions relating to stack and data events are executed within the context of the function **vJIP_Tick()**. However, callback functions relating peripheral events are executed in interrupt context (see Section 4.5.3).

4.5.1 Stack Events

The following stack events can be generated on a node (also refer to Section 8.3.1):

- E_STACK_STARTED: Stack has started
- E_STACK_JOINED: Local node has joined a parent
- E_STACK_NODE_JOINED: A child node has joined the local node
- E_STACK_NODE_LEFT: A child node has left the local node
- E_STACK_TABLES_RESET: Routing tables have been reset (on Co-ordinator or Router) and stack set-up is complete (on any node type)

- E_STACK_RESET: Stack has been reset
- E_STACK_POLL: Local End Device has polled the parent node for data
- E_STACK_NODE_JOINED_NWK (generated on the Co-ordinator): A node has joined the network
- E_STACK_NODE_LEFT_NWK (generated on the Co-ordinator): A node has left the network
- E_STACK_NODE_AUTHORISE (generated on the Co-ordinator): A node is attempting to join the network and its commissioning key needs to be obtained from the Border-Router
- E_STACK_ROUTE_CHANGE (generated on the Co-ordinator): A node has moved in the network
- E_STACK_GROUP_CHANGE: One or more multicast addresses have been added to or removed from network

All of the above events must be handled by a single user-defined callback function, **vJIP_StackEvent()**. In this function call, the stack identifies the specific event (from the above) that has occurred. The stack may also provide additional information about the event. The callback function is executed in the context of **vJIP_Tick()**.

4.5.2 Data Events

The following data events can be generated on a node (also refer to Section 8.3.2):

- E_DATA_SENT: A data packet has been sent successfully
- E_DATA_SEND_FAILED: An attempt to send a data packet failed
- E_DATA_RECEIVED: A data packet has been received
- E_IP_DATA_RECEIVED: A data packet has been received at the IP layer
- E_6LP_ICMP_MESSAGE: An ICMP message has been passed up to the application

All of the above events must be handled by a single user-defined callback function, **v6LP_DataEvent()**, which is executed in the context of **vJIP_Tick()**.

Note: When the MIB 'Remote Variable Access' functions (described in Section 6.4) are used to send and receive data packets, the **v6LP_DataEvent()** function will not be invoked because the stack contains a dedicated handler for the above events. However, the **v6LP_DataEvent()** function must still be included within the application to catch events not handled by the stack.

4.5.3 Peripheral Events

A peripheral event can be generated by one of the integrated peripherals of the JN51xx microcontroller (e.g. ADC, timer). These peripherals and their associated events are described in the *JN516x Integrated Peripherals API User Guide (JN-UG-3087)*. All peripheral events must be handled by a single user-defined callback function, **vJIP_PeripheralEvent()**. This function call identifies the peripheral that generated the event and the source of this event within the peripheral. The callback function is executed in interrupt context.

Peripheral events should be queued for processing in the main application loop if handling them will take a significant amount of time, if the callback function needs to make calls into the stack or if the callback function needs to access Non-Volatile Memory (NVM).

4.6 Entering and Leaving Sleep Mode

Some network devices, particularly End Devices, may only need to be active for a small proportion of time in order to collect and transmit/receive data (e.g. once per hour or once per day). Since such devices are often powered by batteries, it is desirable to conserve power when the device is not active by putting it into a low-power sleep mode. For more information on sleep modes, refer to Section 2.9.2.

4.6.1 Entering Sleep Mode

Sleep mode is entered on a device by calling the function **vJIP_Sleep()**. This function allows the sleep duration to be specified. It also provides the option to preserve the contents of on-chip volatile memory during sleep (sleep with memory held), and therefore to preserve both application and stack context data.

Note 1: If the device enters sleep without memory held, context data can be preserved by storing it in Non-Volatile Memory (NVM) before entering sleep (and then retrieving this data on waking from sleep). The application can do this using the supplied JenOS Persistent Data Manager (PDM) - refer to Section 4.8.

Note 2: A device sending a series of packets to an End Device Sleep can request sleep to be postponed to allow all the packets to be received before the End Device enters sleep mode - refer to Section 4.6.3.

4.6.2 Leaving Sleep Mode

The actions taken on waking from sleep depend on whether the contents of on-chip volatile memory were preserved during sleep:

- For sleep with memory held, waking from sleep results in a warm start through the routine **AppWarmStart()**, as described in Section 4.1.2.
- For sleep without memory held, waking from sleep results in a cold start through the routine **AppColdStart()**, as described in Section 4.1.1.

Note 1: If the device wakes from sleep without memory held but application context data has been stored in Flash memory (see Note above), code to retrieve this data must be included in **AppColdStart()** using the JenOS PDM functions (see Section 4.8) or the Flash functions of the Integrated Peripherals API.

Note 2: While an End Device is asleep, data arriving for the End Device is buffered by its parent. On waking from sleep, the End Device should poll its parent for any pending data. This polling is normally performed automatically but is configurable - see Appendix A.4.

4.6.3 ‘Stay Awake’ Request

If a device needs to send a series of packets to an End Device, the source device can request the End Device to stay awake (postpone entering sleep mode) in order to receive the packets. This request is implemented by setting the ‘stay awake’ flag in a packet. Thus, this flag may be set in all packets except the final one of the series (setting the flag requests the End Device to stay awake long enough to receive at least one further packet).

The ‘stay awake’ flag is bit 7 in the handle of the JIP command within the packet (for command formats, see Appendix G.4). This handle is a user-defined identifier for the command, except bit 7 is reserved for the ‘stay awake’ flag. The handle is specified in the MIB function which is called to send the command (e.g. **eJIP_Remote_ID_Set()**).

On receiving a packet, the stack on the End Device checks the ‘stay awake’ flag. If this bit is set, the user-defined callback function **vJIP_StayAwakeRequest()** is invoked, which takes the appropriate action:

- Since the End Device is not obliged to stay awake, the request may be ignored (for example, the End Device may not be able to accept such requests due to limited power).
- To honour the request, a timer should be implemented to postpone sleep.

If the End Device does not stay awake to receive further packets, the packets will be buffered on the parent of the End Device to be collected by data polling on wake-up (see Section 4.7).

4.7 Data Polling

Data polling in wireless networks was introduced in Section 2.9.3. An End Device that goes through sleep episodes (in order to conserve power) may not be able to receive data sent to it. Therefore, data destined for the End Device is buffered on its parent and the End Device must poll its parent for this data while awake.

4.7.1 Polling Methods

There are two methods that an End Device can employ to poll its parent:

- **Auto-polling:** The End Device periodically polls its parent with a pre-configured period (set to 5 seconds by default) and also automatically polls its parent on waking from sleep. Auto-polling is enabled by default on an End Device in a JenNet-IP network. The configuration of auto-polling is described in Appendix A.4.
- **Manual polling:** The application on the End Device uses the function **eJIP_Poll()** to poll its parent.

In both of the above cases, a single poll may not retrieve all the pending data for the End Device. Therefore, after each poll (manual or auto), the function **eJIP_Poll()** should be called (at least once) until there is no further pending data.

The parent will store a pending packet indefinitely, but there is limited buffer space for packet storage on the parent node. If the buffer is full when a new packet arrives, the oldest packet will be automatically deleted to make room for the new one. However, network-level packets always take priority over data packets.

4.7.2 Polling Events

The events that are generated following a poll are as follows:

- If a poll results in data being received, the stack event E_STACK_POLL of the type E_JIP_POLL_DATA_READY is generated. The received data is usually handled by the stack and is not passed to the application (if the data is destined for the application, the relevant data event will also be generated).
- If a poll results in no data, the stack event E_STACK_POLL of the type E_JIP_POLL_NO_DATA is generated.

Therefore, if a poll (manual or auto) results in an E_JIP_POLL_DATA_READY event, there may be more pending data on the parent and the function **eJIP_Poll()** should be called repeatedly until the event E_JIP_POLL_NO_DATA is generated.

The above events and the IEEE 802.15.4 messages that result from polling are illustrated in Figure 15 below - in this example, the polling is initiated manually by a call to **eJIP_Poll()**.

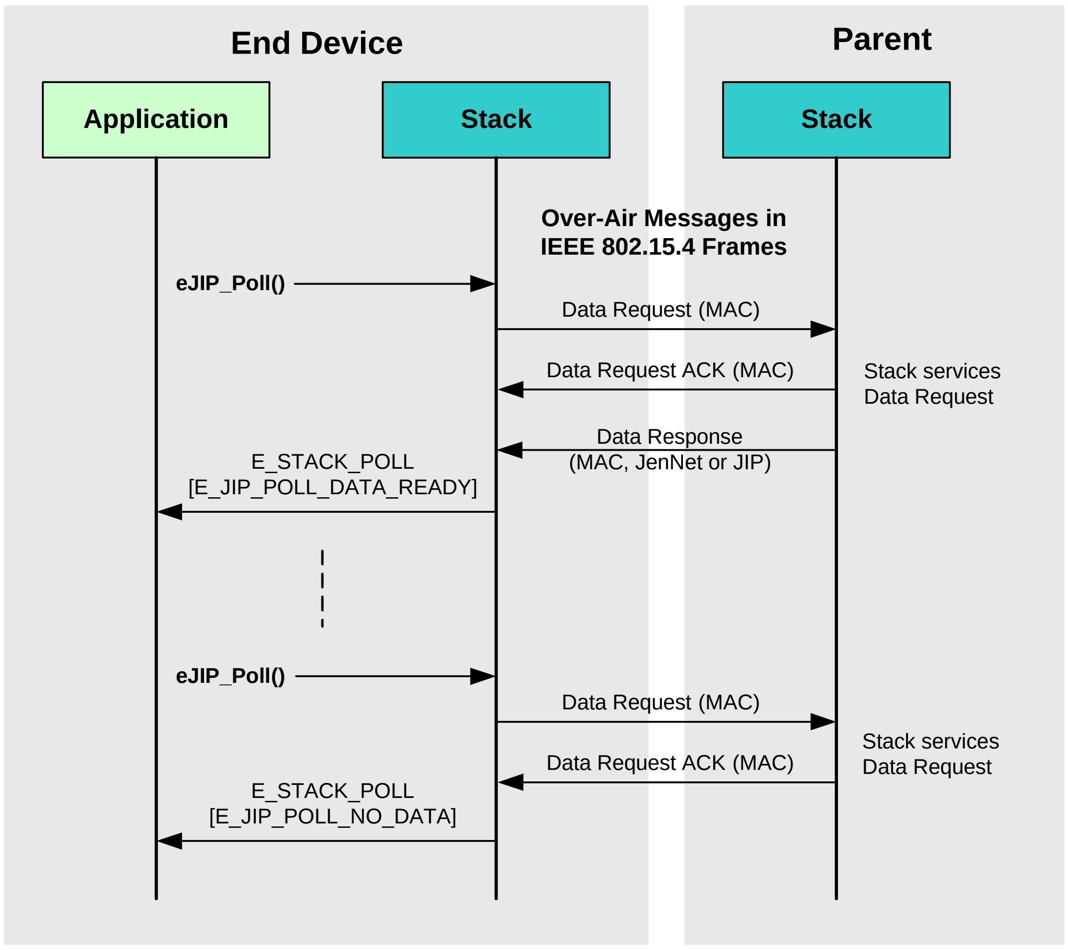

End Device
Parent
Application
Stack
Stack
Over-Air Messages in IEEE 802.15.4 Frames
eJIP_Poll()
Data Request (MAC)
Data Request ACK (MAC)
Stack services Data Request
Data Response (MAC, JenNet or JIP)
E_STACK_POLL [E_JIP_POLL_DATA_READY]
eJIP_Poll()
Data Request (MAC)
Data Request ACK (MAC)
Stack services Data Request
E_STACK_POLL [E_JIP_POLL_NO_DATA]

Figure 15: Events and Messages Resulting from Data Polling

Note: Polling may be the main activity of an End Device while it is awake. The messages depicted in Figure 15 may therefore be important in battery life estimates for the End Device. In order to perform these calculations, you will also need transceiver power figures from your wireless microcontroller datasheet as well as frame details from the IEEE 802.15.4 standard.

4.8 Persisting Context Data

Data needed for the operation of a wireless network node is normally stored in on-chip RAM. This includes data that may evolve during node operation, e.g. context data for the network stack and application data. This data is only maintained in RAM while the node is powered and will normally be lost during an interruption to the power supply (e.g. power failure or battery replacement).

In order for the node to recover from a power interruption with continuity of service, provision must be made for storing a back-up of context data in Non-Volatile Memory (NVM), normally Flash memory. This data can then be recovered during a re-boot following power loss, allowing the node to resume its role in the network.

The storage and recovery of context data can be handled using the Persistent Data Manager (PDM), which is provided in the JenNet-IP software. The supplied PDM is identical to the PDM module in JenOS (Jennic Operating System). Therefore, for details of the PDM, you should refer to the *JenOS User Guide (JN-UG-3075)*, although you do not need to use JenOS in conjunction with JenNet-IP.

Note: The *JenOS User Guide* states that stack context data will automatically be saved to NVM when changes take place. This is not the case for the JenNet-IP stack - the application must store and retrieve stack context data as part of the application context data.

4.9 Using Low-Energy Devices

Low-energy devices were introduced in Section 3.9. This section provides the implementation details for using a low-energy device in conjunction with a JenNet-IP WPAN. The low-energy device is not formally a part of the JenNet-IP WPAN but its use with the network requires some pre-configuration and application coding on both the low-energy device and the WPAN nodes.

4.9.1 Implementation on Low-Energy Device

A low-energy device must be pre-configured with an IEEE/MAC address, a 128-bit security key and a fixed 2.4-GHz radio channel (see Section 3.9.2). This configuration is manufacturer-specific but may be conducted in the factory or during installation using hardware switches on the device or a configuration tool.

The software stack used on a low-energy device is a reduced version of the IEEE 802.15.4 stack in which the MAC layer is replaced by NXP's 'MicroMAC' layer. Thus, the low-energy device does not need any JenNet-IP software. However, the MicroMAC is supplied in the *JN516x JenNet-IP SDK (JN-SW-4165)*.

Details of how to implement the MicroMAC on a low-energy device are provided in Appendix L. This information includes how to send an IEEE 802.15.4 frame to the JenNet-IP WPAN. A frame from a low-energy device is always interpreted within the

WPAN as a JenNet-IP 'Set by ID' command to remotely set a MIB variable on a WPAN node in order to achieve the desired action (e.g. switching on a light). It is the responsibility of the application to manage the 4-byte frame counter - the three most significant bytes of the frame counter are inserted into the frame payload and the least significant byte is used as the sequence counter in the frame header. The low-energy frame format is detailed in Appendix G.5.

4.9.2 Implementation in JenNet-IP WPAN

The low-energy device implementation details for the WPAN nodes are provided in the sub-sections below for the Co-ordinator, a Router and any target node of a frame.

Note 1: The process for registering a low-energy device with a WPAN is outlined in Section 3.9.3. It may be useful to refer to this process in studying this section.

Note 2: Since End Devices do not use multicast messages, they should not be targets for commands from low-energy devices.

4.9.2.1 On the Co-ordinator

The Co-ordinator plays a vital role in registering a low-energy device with the WPAN.

Pre-configuration

In order to use one or more low-energy devices with the WPAN, certain pre-configuration is necessary on the Co-ordinator, as follows:

- **Radio channel:** The 2.4-GHz radio channel to be used by the WPAN must be fixed by the Co-ordinator to match the channel used by the low-energy devices (therefore, no channel search should be performed during network formation). The channel is set through **eJIP_Init()** during initialisation - see Section 4.1.1.
- **IEEE/MAC addresses:** A 'white list' must be created on the Co-ordinator (or on the LAN/WAN side of the Border-Router) containing the IEEE/MAC addresses (and security keys) of all the low-energy devices that could potentially be registered with the system. This white list is application-specific.
- **Security keys:** For each low-energy device in the white list, the 128-bit security key of the device must also be stored in advance. This key is used to encrypt and decrypt the payloads of frames transmitted from the low-energy device to the WPAN (but not within the WPAN, where JenNet-IP security is used).
- **Registered device list:** The stack on the Co-ordinator maintains a list of registered low-energy devices. The maximum number of devices that can be stored in this list can be set by the application through the following variable:

  ```
  extern PUBLIC uint8 u8JNT_LowEnergyDevices;
  ```

  The default value is 10. If another value is required, this must be set before initialising the stack.

Registration

The first time a low-energy device sends an IEEE 802.15.4 frame to the WPAN, the device is registered with the Co-ordinator. The Co-ordinator may receive the low-energy frame directly or may receive a 'Low Energy Request' in a JenNet-IP frame from a Router (see Section 4.9.2.2). In either case, the JenNet-IP stack on the Co-ordinator generates an E_STACK_LOW_ENERGY_SEEN event, which is passed to the application via the **vJIP_StackEvent()** callback function. This event contains the IEEE/MAC address of the low-energy device (in a `MAC_ExtAddr_s` structure).

It is then the responsibility of the application to perform the following steps:

1. Check for the IEEE/MAC address in the white list to validate that the source device is an authorised low-energy device.

2. If the low-energy device is valid, send its IEEE/MAC address and security key to the rest of the WPAN in a broadcast message, sent using the function **eApi_SendLowEnergyInform()** - also see Section 4.9.2.2.

   As a result of the call to **eApi_SendLowEnergyInform()**, the stack on the Co-ordinator adds the device to the local list of registered low-energy devices.

Note: The function **eApi_SendLowEnergyInform()** can also be used to unregister a low-energy device from the WPAN. In this case, the device is removed from the list of registered low-energy devices.

4.9.2.2 On a Router

Any Router in the WPAN could potentially receive a frame from a low-energy device. Therefore, a Router must be prepared to handle these frames.

Pre-configuration

In order to use one or more low-energy devices with the WPAN, certain pre-configuration is necessary on each Router, as follows:

- **Radio channel:** The 2.4-GHz radio channel to be used by the Router must be fixed to match the channel used by the low-energy devices and Co-ordinator (therefore, no channel search should be performed during network joining). The channel is set through **eJIP_Init()** during initialisation - see Section 4.1.1.
- **Registered device list:** The stack on the Router maintains a list of registered low-energy devices. This list is not accessible to the application but the maximum number of devices that can be stored in the list can be set by the application through the following variable:

```
extern PUBLIC uint8 u8JNT_LowEnergyDevices;
```

  The default value is 10. If another value is required, this must be set before initialising the stack.

Registration/Operation

When a Router receives an IEEE 802.15.4 frame, the stack on the Router checks whether the source IEEE/MAC address of the frame corresponds to a low-energy device in the local registered device list:

- If the IEEE/MAC address in the frame header is not in the registered device list and the frame length is within the expected range for a low-energy frame, the Router forwards the device address to the Co-ordinator in a 'Low Energy Request'. The Co-ordinator handles this request as described in Section 4.9.2.1. If the device is accepted by the Co-ordinator, the Router eventually receives an acceptance message from the Co-ordinator (as a result of the call to **eApi_SendLowEnergyInform()**) and the stack on the Router adds the device to the local list of registered low-energy devices.

Note 1: All the Routers in the WPAN will receive the acceptance message from the Co-ordinator and add the device to the local list of registered low-energy devices.

Note 2: The frame transmitted by the low-energy device to register with the WPAN may contain a valid command (e.g. to switch on a light), but this command may not be acted on (beyond the registration process).

- If the IEEE/MAC address is already in the registered device list, the Router forwards the command in a multicasted JenNet-IP frame. It derives the multicast address from the IEEE/MAC address of the source device (see Section 4.9.2.3).

4.9.2.3 On a Target Node

The target WPAN node for a command from a low-energy device can be a Router or the Co-ordinator. The command is forwarded within the WPAN in a multicasted JenNet-IP frame. Therefore, the target node cannot be an End Device since these devices discard multicast frames.

The IPv6 multicast address for a frame from a low-energy device is generated from the IEEE/MAC address of the device, as follows.

```
0xFF15::<IEEE/MAC address>:0000
```

If a node (Router or Co-ordinator) is to be a member of the multicast group for a low-energy device then the node must add itself to the multicast group. The application on the node can do this using the function **bJIP_AddGroupAddr()**.

4.9.3 Persisting Low-Energy Device Registration Data

On a WPAN node which has registered low-energy devices, these registrations will normally be lost if the device is re-started or the power to the device is interrupted. However, the facility is available to preserve this registration data in non-volatile memory so that the node does not lose the data. In this case, the JenOS PDM module is used to persist the data (see Section 4.8).

To support the persistence of low-energy device registration data, the application on the node must include the following declaration to point to the database of registered low-energy devices:

```
extern PUBLIC tsLowEnergyDevice *psJNT_LowEnergyDeviceDatabase;
```

This line provides a pointer to the start of the database, which is an array of type **tsLowEnergyDevice**. The number of entries in the array is given by the variable `u8JNT_LowEnergyDevices` (see Section 4.9.2.1 and Section 4.9.2.2). Therefore, these two variables are used by the application to specify the database to be persisted in non-volatile memory by PDM.

Note that the pointer is populated during stack initialisation and the array space is, by default, taken from the heap. It is also possible to assign space to the database before stack initialisation by setting `psJNT_LowEnergyDeviceDatabase` from the application - if so, the array space will not be taken from the heap.

5. JIP Embedded API General Functions

This chapter details all the functions of the JIP Embedded API that are not related to MIBs. These functions are defined in the header files **jip.h** and **api.h**.

These general functions are divided into the following categories:

- Stack management functions, detailed in Section 5.1
- Stack mode functions, detailed in Section 5.2
- Network profile functions, detailed in Section 5.3
- Data transfer functions, detailed in Section 5.4
- IPv6 address functions, detailed in Section 5.5
- IP functions, detailed in Section 5.6

5.1 Stack Management Functions

This section describes the functions for managing the stack, including initialising it and putting the device into sleep mode, as well as a function for obtaining error information on the last function call.

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

Function	**Page**
v6LP_InitHardware	92
eJIP_Init	93
iJIP_ResumeStack	94
vJIP_Tick	95
vJIP_Sleep	96
u32JIP_GetErrNo	97
vJIP_EnableSecurity	98
vApi_DeleteChild	100
vApi_ConfigureFastCommission	101
eApi_SendNetworkAnnounceEnhanced	102
eApi_SendLowEnergyInform	103

v6LP_InitHardware

```
void v6LP_InitHardware(void);
```

Description

This function initialises the JN51xx microcontroller, and must be called as part of the application’s cold start routine **AppColdStart()** and warm start routine **AppWarmStart()**.

Parameters

None

Returns

None

eJIP_Init

teJIP_Status eJIP_Init(tsJIP_InitData **psInitData***);**

Description

This function initialises the JenNet-IP stack. You must call this function after calling **v6LP_InitHardware()** and before calling other JIP Embedded API functions. The initialisation data is passed into the function by means of a structure of type **tsJIP_InitData** - for details of this structure, refer to Section 8.1.1. This initialisation data includes values for some of the JenNet parameters detailed in Chapter 9.

Before passing this structure into the function, you are advised to first perform a 'memset' operation on the structure, in order to set its elements to zero values, and then set individual elements to the desired values (if required).

Provided JenNet is enabled in the stack initialisation structure, this function will invoke the callback function **vJIP_ConfigureNetwork()**. This callback function sets the JenNet parameters and will over-write those that have already been set by this function, **eJIP_Init()**.

***Caution:** If using a JN51xx high-power module, the function **vAHI_HighPowerModuleEnable()** from the Integrated Peripherals API must be called after **eJIP_Init()**. If it is called before the stack is initialised, the radio will not transmit.*

Parameters

**psInitData*	Pointer to a structure containing the stack initialisation data (see Section 8.1.1).

Returns

E_JIP_ERROR_FAILED

E_JIP_OK

iJIP_ResumeStack

```
int iJIP_ResumeStack(void);
```

Description

This function is used to resume the protocol stack after a device wakes from sleep with memory held.

Include this function in the device’s warm start routine **AppWarmStart()**, which is called by the device when it wakes from sleep with memory held. You must call this function after calling **v6LP_InitHardware()** and before calling other JIP Embedded API functions (for cold or warm start).

Parameters

None

Returns

0 if successful; any other value indicates an error

vJIP_Tick

```
void vJIP_Tick(void);
```

Description

This function must be called in the main processing loop of the application to generate stack and data events. These events must then be handled as described in Section 4.5.

Parameters

None

Returns

None

vJIP_Sleep

```
void vJIP_Sleep(bool_t bMemoryHold,
                uint32 u32SleepPeriodInMs);
```

Description

This function puts a network device to sleep for the specified period of time. The function is normally used with End Devices.

The function provides the option to preserve the contents of memory during sleep (sleep with memory held), to allow the node to easily resume operation on waking from sleep. If memory contents are not preserved (sleep without memory held), the stack will initiate a cold start on waking from sleep and attempt to re-join the network as a new node.

In the case of sleep without memory held, the application can store its context data in non-volatile memory (such as Flash memory) while asleep. To save and later retrieve context data, the application can use the Persistent Data Manager (PDM) provided in the JenNet-IP software. The supplied PDM is identical to the PDM module in JenOS (Jennic Operating System) - for details of the PDM, you should refer to the *JenOS User Guide (JN-UG-3075)*. Also refer to Section 4.8.

Parameters

bMemoryHold	Determines whether contents of memory will be preserved during sleep: TRUE: Preserve memory contents FALSE: Do not preserve memory contents
u32SleepPeriodInMs	Sleep duration, in milliseconds

Returns

None

u32JIP_GetErrNo

```
uint32 u32JIP_GetErrNo(void);
```

Description

This function returns extended error information for the most recent error.

The function is used to obtain the error conditions arising from an API function call. It must be called once the API function has returned and before another API function is called. This is because each API function resets the extended error information to zero and then sets it the appropriate value at the end of execution.

A 32-bit value is returned by this function, but only bits 15-0 are valid (see below), with bits 31-16 reserved for future use.

Parameters

None

Returns

32-bit value in which the lower 16 bits (bits 15-0) represent extended error information - bits 7-0 give an error code and 15-8 give further error information, as defined in the tables in Section 8.5.

vJIP_EnableSecurity

```
void vJIP_EnableSecurity(void);
```

Description

This function can be used to enable JenNet security on the local node. If required, the function must be called during initialisation in the user-defined callback function **vJIP_ConfigureNetwork()**. Security is described in Section 2.8 and Section 3.6.

The JenNet function **vSecurityUpdateKey()** (described below) can be used to set the network key (on the Co-ordinator) or a commissioning key.

The JenNet function **eApi_CommissionNode()** (described on the next page) can be used on the Co-ordinator to pass a node's commissioning key into the network.

Parameters

None

Returns

None

vSecurityUpdateKey()

```
void vSecurityUpdateKey(uint8 u8KeyIndex,
                        tsSecurityKey *psSecurityKey);
```

Description

This JenNet function can be used to set a security key on the local node.

Parameters

u8KeyIndex	Type of key to set: 0 - network key (on Co-ordinator only) 1 - commissioning key 2 - fast commissioning key (on Co-ordinator or Router only)
**psSecurityKey*	Pointer to following structure containing 128-bit key to be set:

```
typedef struct
{
  uint32 u32KeyVal_1; /* Least significant word */
  uint32 u32KeyVal_2;
  uint32 u32KeyVal_3;
  uint32 u32KeyVal_4; /* Most significant word */
} tsSecurityKey;
```

Returns

None

eApi_CommissionNode()

```
teJenNetStatusCode eApi_CommissionNode(
                                  MAC_ExtAddr_s *psDeviceAddr,
                                  tsSecurityKey *psSecurityKey);
```

Description

This JenNet function can be used on the Co-ordinator to distribute a commissioning key (of a particular node) throughout the network.

The function should be called after the stack event E_STACK_NODE_AUTHORISE has been generated on the Co-ordinator to indicate that a Router has requested the commissioning key of another node that is attempting to join it. The application must obtain the requested key from the Border-Router and then call this function to pass the obtained key into the network (and therefore back to the requesting Router). This is illustrated in the code example below.

Parameters

****psDeviceAddr***	Structure containing IEEE/MAC address of node for which commissioning key is being supplied (see Section 8.1.3)
****psSecurityKey***	Pointer to following structure containing 128-bit key:

```
typedef struct
{
  uint32 u32KeyVal_1; /* Least significant word */
  uint32 u32KeyVal_2;
  uint32 u32KeyVal_3;
  uint32 u32KeyVal_4; /* Most significant word */
} tsSecurityKey;
```

Returns

E_JENNET_SUCCESS
E_JENNET_DEFERRED
E_JENNET_ERROR

Example

```
PUBLIC void vJIP_StackEvent(teJIP_StackEvent eEvent, uint8 *pu8Data, uint8 u8DataLen)
{
    switch (eEvent)
    {
    case E_STACK_NODE_AUTHORISE:
        {
            MAC_ExtAddr_s sCommNodeAddr;
            /* Get MAC address from incoming data (memcpy to avoid any alignment issues) */
            memcpy((uint8 *)&sCommNodeAddr, pu8Data, 8);
            /* Find commissioning key for this address (not shown here) */
            /* Use JenNet to send key to network */
            (void)eApi_CommissionNode(&sCommNodeAddr, &sCommissioningKey);
        }
        break;
    }
}
```

vApi_DeleteChild

void vApi_DeleteChild(MAC_ExtAddr_s ****psAddr*);**

Description

This function can be used to remove a child node from the network. It must be called on the immediate parent of the child. The child node is specified using its IEEE/MAC address.

The function is particularly useful when operating a JenNet-IP network in standalone mode (see Chapter 11). The pseudo-Co-ordinator of a standalone network can have a maximum number of children, determined by the JenNet Parameter `u8MaxChildren` (see Section 9.2) which is set to 10 by default. The pseudo-Co-ordinator is normally a remote control unit and, once commissioned, a node does not need to remain a child of the pseudo-Co-ordinator in order to be controlled. Therefore, this function can be used to break this parent-child relationship to ensure that sufficient child places are available to allow further nodes to join and be commissioned by the remote control unit.

Parameters

**psAddr*	Pointer to structure containing IEEE/MAC address of child node to be removed

Returns

None

vApi_ConfigureFastCommission

```
void vApi_ConfigureFastCommission(uint8 u8Channel,
                                  uint16 u16PanId);
```

Description

This function can be used to configure fast commissioning mode on the Co-ordinator or on a Router which may potentially join the network. The radio channel in which fast commissioning will be conducted must be specified as well as a 16-bit PAN ID that will be used by the network when in fast commissioning mode (these values are different from those used in normal operational mode). The function must be called within the user-defined callback function **vJIP_ConfigureNetwork()**.

Note: A fast commissioning security key must also be pre-defined using the JenNet function **vSecurityUpdateKey()**, described on page 98.

Fast commissioning mode is described in Section 4.1.3.

Parameters

u8Channel	Radio channel, in the range 11-26, to be used for fast commissioning
u16PanId	PAN ID to be used for fast commissioning

Returns

None

eApi_SendNetworkAnnounceEnhanced

```
teJenNetStatusCode
            eApi_SendNetworkAnnounceEnhanced(void);
```

Description

This function can be used on the Co-ordinator to transmit a Network Announce message when in fast commissioning mode. The sent message contains the PAN ID, network key and operational channel of the network, as well as the pre-configured fast commissioning PAN ID and security key. The last two items must be defined as follows:

- The fast commissioning PAN ID must be pre-defined using the function **vApi_ConfigureFastCommission()**
- The fast commissioning security key must be pre-defined using the JenNet function **vSecurityUpdateKey()**

The above two functions must be called within the user-defined callback function **vJIP_ConfigureNetwork()**.

Fast commissioning mode is described in Section 4.1.3.

Parameters

None

Returns

E_JENNET_SUCCESS

E_JENNET_DEFERRED

E_JENNET_ERROR

eApi_SendLowEnergyInform

```
teJenNetStatusCode eApi_SendLowEnergyInform(
                                  MAC_ExtAddr_s *psAddr,
                                  uint8 *pu8Key,
                                  teLowEnergyStatus eStatus);
```

Description

This function can be called on the Co-ordinator to inform the rest of the WPAN that a low-energy device is to be registered with or unregistered from the JenNet-IP system. As a result, the Co-ordinator and all Router nodes should add or remove the device in the local list of registered low-energy devices (this list maintenance is performed by the stack on the nodes).

The message that is sent as a result of this function call contains the IEEE/MAC address and security key of the low-energy device, which are specified in this function.

Low-energy devices are introduced in Section 3.9 and their implementation is described in Section 4.9.

Parameters

psAddr	Pointer to structure containing the 64-bit IEEE/MAC address of the low-energy device
pu8Key	Pointer to a location containing the 128-bit security key of the low-energy device
eStatus	Action to take on Router nodes, one of: E_LEF_ADD (Add low-energy device to local list) E_LEF_DELETE (Remove low-energy device from local list)

Returns

E_JENNET_SUCCESS

E_JENNET_DEFERRED

E_JENNET_ERROR

5.2 Stack Mode Functions

This section describes the functions concerned with the JenNet-IP stack mode. The stack mode value is a 16-bit bitmap containing various data items (see the function descriptions). The value is transmitted as part of the beacon payload and is used by joining devices as part of the criteria for filtering received beacons - a joining device will discard beacons from other nodes that do not use the same stack mode.

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

Function	**Page**
vApi_SetStackMode	105
u16Api_GetStackMode	106

vApi_SetStackMode

```
void vApi_SetStackMode(uint16 u16ModeMask);
```

Description

This function can be used to set the stack mode value on the local node. This value is a 16-bit bitmap containing the following data items:

Bits	Name	Description	Default
15-8	Stack version	These bits should be set to zero, as the stack modifies these bits itself as required. Currently, the internally set value is 1	1
7-2	-	These bits are unused and should be set to zero	0
1	Commissioning mode	1 - enable commissioning mode 0 - disable commissioning mode	0
0	Standalone mode	1 - enable standalone (non-gateway) mode 0 - disable standalone (non-gateway) mode If set to 0, beacon responses will be automatically sent when a device joins the network	0

Parameters

u16ModeMask	Bitmap containing the stack mode value to be set (see above). The following macros are also available: NONE_GATEWAY_MODE (0x0001) CMSN_MODE (0x0002)

Returns

None

u16Api_GetStackMode

```
uint16 u16Api_GetStackMode(void);
```

Description

This function can be used to obtain the stack mode value of the local node. This value is a 16-bit bitmap containing the following data items:

Bits	Name	Description	Default
15-8	Stack version	These bits are set by the stack. Currently, the internally set value is 1	1
7-2	-	These bits are unused and should read as zero	0
1	Commissioning mode	1 - commissioning mode enabled 0 - commissioning mode disabled	0
0	Standalone mode	1 - standalone (non-gateway) mode enabled 0 - standalone (non-gateway) mode disabled If set to 0, beacon responses are automatically sent when a device joins the network	0

Parameters

None

Returns

16-bit bitmap containing the obtained stack mode value (see above)

5.3 Network Profile Functions

This section describes the functions concerned with JenNet network profiles and their parameters (described in Section 9.2). If required, these functions should be called in the callback function **vJIP_ConfigureNetwork()**. Normally, the desired profile should be set on the Co-ordinator - other devices inherit the profile parameter values from their parent when they join the network, but the application can over-ride these parameter values using these functions.

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

Function	**Page**
bJnc_SetJoinProfile	108
bJnc_SetRunProfile	109
vJnc_GetNwkProfile	110
u8GetCurJoinProfile	111
u8GetCurRunProfile	112
bJnc_ChangeJoinProfile	113

Note 1: JenNet network profiles are introduced in Section 3.7. The network profile parameters and the standard profiles are detailed in Section 9.2.

Note 2: The network profile parameters are divided into two categories - 'join parameters' and 'run parameters'. It is possible to use the join parameters of one profile with the run parameters of another profile.

bJnc_SetJoinProfile

```
bool_t bJnc_SetJoinProfile(uint8 u8ProfileIndex,
                           const tsNwkProfile *psProfile);
```

Description

This function can be used to set the network ‘join profile’ on the Co-ordinator (the parameter values in this profile will be inherited by devices that join the network). Only the join parameters of a network profile are set by this function (`u8MinBeaconLQI`, `u16ScanBackOffMin`, `u16ScanBackOffMax`, `u16EstRtBackOffMin`, `u16EstRtBackOffMax`).

A standard profile or a custom profile can be specified. The ten standard profiles supplied with the JenNet-IP software are numbered 0 to 9.

- If the function parameter *u8ProfileIndex* is set to a value in the range 0-9, the corresponding standard profile will be used and the function parameter *psProfile* will be ignored.
- If *u8ProfileIndex* is set to PROFILE_USER (255), a custom profile will be used which must be specified through *psProfile*.
- All other values of *u8ProfileIndex* are invalid.

Parameters

u8ProfileIndex	Index value of network profile to be used: 0-9: Use standard profile with specified index PROFILE_USER (255): Use custom profile specified via *psProfile* Any other value: Undefined
psProfile	Pointer to custom network profile to use (this parameter is ignored if a standard profile is specified via *u8ProfileIndex*)

Returns

TRUE if join profile successfully set, FALSE otherwise

bJnc_SetRunProfile

```
bool_t bJnc_SetRunProfile(uint8 u8ProfileIndex,
                          const tsNwkProfile *psProfile);
```

Description

This function can be used to set the network 'run profile' on the Co-ordinator (the parameter values in this profile will be inherited by devices that join the network). Only the join parameters of a network profile are set by this function (`u8MaxChildren`, `u8MaxSleepingChildren`, `u8MaxFailedPkts`, `u8MaxBcastTTL`, `u16RouterPingPeriod`).

A standard profile or a custom profile can be specified. The ten standard profiles supplied with the JenNet-IP software are numbered 0 to 9.

- If the function parameter *u8ProfileIndex* is set to a value in the range 0-9, the corresponding standard profile will be used and the function parameter *psProfile* will be ignored.
- If *u8ProfileIndex* is set to PROFILE_USER (255), a custom profile will be used which must be specified through *psProfile*.
- All other values of *u8ProfileIndex* are invalid.

Parameters

u8ProfileIndex	Index value of network profile to be used: 0-9: Use standard profile with specified index PROFILE_USER (255): Use custom profile specified via *psProfile* Any other value: Undefined
psProfile	Pointer to custom network profile to use (this parameter is ignored if a standard profile is specified via *u8ProfileIndex*)

Returns

TRUE if run profile successfully set, FALSE otherwise

vJnc_GetNwkProfile

void vJnc_GetNwkProfile(tsNwkProfile ****psProfile*);**

Description

This function can be used to obtain the network profile that is currently in use (the full profile, including both join and run parameters). The function fills in the supplied structure with the obtained profile parameter values.

Parameters

**psProfile*	Pointer to structure to receive current network profile

Returns

None

u8GetCurJoinProfile

```
uint8 u8GetCurJoinProfile(void);
```

Description

This function can be used to obtain the index of the network profile to which the current 'join parameter' values belong (`u8MinBeaconLQI`, `u16ScanBackOffMin`, `u16ScanBackOffMax`, `u16EstRtBackOffMin`, `u16EstRtBackOffMax`). If they are from a standard profile, the function will return the relevant profile index in the range 0-9, otherwise a custom index will be returned.

Parameters

None

Returns

Index of network profile to which current 'join parameter' values belong

u8GetCurRunProfile

```
uint8 u8GetCurRunProfile(void);
```

Description

This function can be used to obtain the index of the network profile to which the current 'run parameter' values belong (`u8MaxChildren`, `u8MaxSleepingChildren`, `u8MaxFailedPkts`, `u8MaxBcastTTL`, `u16RouterPingPeriod`). If they are from a standard profile, the function will return the relevant profile index in the range 0-9, otherwise a custom index will be returned.

Parameters

None

Returns

Index of network profile to which current run parameter values belong

bJnc_ChangeJoinProfile

```
bool_t bJnc_ChangeJoinProfile(uint8 u8ProfileIndex,
                              uint8 u8NumBroadcasts,
                              uint8 u8BroadcastPeriod);
```

Description

This function can be called by the application on the Co-ordinator to change the JenNet network profile used by the WPAN (while the network is running). The new profile specified in this function must be a standard profile (0-9).

The Co-ordinator must inform the other network nodes of the change in profile and all nodes should switch to the new profile at the same time. Therefore, the profile switch on the Co-ordinator does not occur immediately. Calling this function will first cause the Co-ordinator to transmit a series of broadcasts to notify the other network nodes of the profile change - the number of broadcasts in the series and the period between consecutive broadcasts are specified in the function call through the parameters *u8NumBroadcasts* and *u8BroadcastPeriod* respectively.

The profile switch should occur once the series of broadcasts is complete - that is, after a time delay (from the moment that the function was called) which is equal to *u8NumBroadcasts* x *u8BroadcastPeriod.* Timing information to ensure a synchonised profile switch (across all nodes) is conveyed in the broadcasts.

On all nodes, the switch is handled by the JenNet-IP stack and is transparent to the application on the node.

Note: If a node that is currently using a custom profile receives a notification to change to a standard profile, it will ignore the notification and continue to use the custom profile. However, the node will pass on the profile change information in its beacon payload.

Parameters

u8ProfileIndex	Index value of (standard) network profile to be used: 0-9: Use standard profile with specified index Any other value: Unused
u8NumBroadcasts	Number of broadcasts to inform other nodes of the new profile
u8BroadcastPeriod	Time-period between consecutive broadcasts, in tenths of a second

Returns

TRUE if function returned successfully, FALSE otherwise

5.4 Data Transfer Functions

This section describes functions that are concerned with receiving data.

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

Function	**Page**
eJIP_Poll	115
i6LP_RecvFrom	116

eJIP_Poll

```
teJIP_PollResponse eJIP_Poll(void);
```

Description

This non-blocking (but synchronous) function is used on an End Device to generate a manual poll request to its parent, in order to retrieve any pending data that may be waiting for the device on its parent (the data may have arrived while the End Device was sleeping).

The function returns immediately and indicates whether the request is being successfully processed - if E_JIP_POLL_PENDING is returned, an E_STACK_POLL stack event can be subsequently expected, with data field containing a 1-byte value of type **teJIP_PollResponse** (see Section 8.2.5) with the following meanings:

E_JIP_POLL_NO_DATA - Poll complete but no data pending

E_JIP_POLL_DATA_READY - Poll complete and data received

E_JIP_POLL_TIMEOUT - Poll timed out and no data received

E_JIP_POLL_ERROR - Problem with request

If data is received as a result of this function call, the data will be handled automatically by the JenNet-IP stack.

A single call to this function may not obtain all pending data and multiple calls may be required to retrieve all the data. If E_JIP_POLL_DATA_READY results from a function call, there may be more data and the function should be called again. A result of E_JIP_POLL_NO_DATA indicates that all pending data has been retrieved and there is no need to call the function again.

In addition to manual polling, this function should also be used when auto-polling is enabled on an End Device (see Appendix A.4). An auto-poll may not retrieve all the pending data from the parent and this function should be called (once) following an auto-poll to request any further data.

Parameters

None

Returns

One of:

E_JIP_POLL_ERROR (problem with request)

E_JIP_POLL_PENDING (request accepted but not complete yet)

i6LP_RecvFrom

```
int i6LP_RecvFrom(int iSocket,
                  uint8 *pu8Data,
                  uint16 u16DataLen,
                  uint32 u32Flags,
                  ts6LP_SockAddr *psSrcAddr,
                  uint8 *pu8AddrLen);
```

Description

This non-blocking (but synchronous) function retrieves a received packet from the specified local socket and stores it in the specified application buffer.

Note: The Berkeley version of this function is blocking.

The function should be called after receiving the data event E_DATA_RECEIVED which indicates that data is available on the socket. It is also used to receive the ICMP event E_6LP_ICMP_MESSAGE by specifying the special ICMP socket SIXLP_ERROR_SOCKET through the parameter *iSocket*. For more information on handling ICMP messages, refer to Appendix C. ICMP messages are also described in RFC 4443 available from the IETF (www.ietf.org).

Note that the incoming packet can be discarded by calling this function with the parameter **pu8Data* set to NULL.

Parameters

iSocket	Socket identifier
**pu8Data*	Pointer to the application buffer to receive packet data. If set to NULL, the packet is discarded
u16DataLen	Available space in buffer, in bytes
u32Flags	Flags (can be left as 0)
**psSrcAddr*	Pointer to structure to receive packet source address and port
**pu8AddrLen*	Pointer to location to receive length of address/port structure, in bytes

Returns

Size of received payload, in bytes. -1 indicates that nothing has been received, the receiver buffer is too small for the packet or some other error.

Note that if -1 is returned because the receive buffer is too small, the buffer will not be freed by this function. In this case, the buffer should be freed by calling the function again with **pu8Data* set to NULL.

5.5 IPv6 Address Functions

This section describes the functions which are used to obtain IPv6 addresses and create host interface IDs. IPv6 addresses are introduced in Section 3.3.

The IPv6 address functions are listed below, along with their page references:

Function	Page
iJIP_CreateInterfaceIdFrom64	118
iJIP_GetOwnDeviceAddress	119
iJIP_GetLastDestinationAddr	120
iJIP_GetLastSourceAddr	121
bJIP_AddGroupAddr	122
bJIP_RemoveGroupAddr	123

iJIP_CreateInterfaceIdFrom64

```
int iJIP_CreateInterfaceIdFrom64(
                                    EUI64_s *psDeviceInterfaceId,
                                    EUI64_s *psDeviceEUI64);
```

Description

This function creates a host interface ID (the lower 64 bits of an IPv6 address) from a MAC address. The host interface ID is taken to be the MAC address with bit 57 inverted and is returned in a structure pointed to by **psDeviceInterfaceId.*

The structure `EUI64_s` used in this function is described in Section 8.1.15.

Parameters

**psDeviceInterfaceId*	Pointer to structure to receive host interface ID
**psDeviceEUI64*	Pointer to structure containing the MAC address

Returns

0 (success) or -1 (failure - for example, if the pointers passed in were not valid)

iJIP_GetOwnDeviceAddress

```
int iJIP_GetOwnDeviceAddress(
                        in6_addr *psDeviceAddress,
                        bool_t bUseGlobal);
```

Description

This function obtains the 128-bit IPv6 address of the local device.

The obtained address comprises two parts:

- 64-bit address prefix, which can be a global address prefix or a link-local address prefix (the required prefix type must be specified through the Boolean parameter).
- 64-bit host interface ID

The address prefix is specified at the time of network formation through the stack initialisation data structure, `tsStackInitData` (described in Section 8.1.1), used by **eJIP_Init()**.

Parameters

**psDeviceAddress*	Pointer to structure to receive the IPv6 address of the local device - see Section 8.1.16
bUseGlobal	Required address prefix type: TRUE: Use global prefix FALSE: Use link-local prefix

Returns

0 (success) or –1 (error)

iJIP_GetLastDestinationAddr

```
int iJIP_GetLastDestinationAddr(ts6LP_SockAddr *psAddr);
```

Description

This function obtains the destination IPv6 address of the last packet received by the local node. The function should return the (unicast) address of the local node or a multicast address.

Parameters

**psAddr*	Pointer to structure to receive the destination address information (see Section 8.1.4)

Returns

0 (success) or –1 (error)

iJIP_GetLastSourceAddr

```
int iJIP_GetLastSourceAddr(ts6LP_SockAddr *psAddr);
```

Description

This function obtains the source IPv6 address of the last packet received by the local node.

Parameters

**psAddr*	Pointer to structure to receive the source address information (see Section 8.1.4)

Returns

0 (success) or –1 (error)

bJIP_AddGroupAddr

```
bool_t bJIP_AddGroupAddr(in6_addr *psAddr);
```

Description

This function can be used to add the local node to the multicast group with the specified IPv6 multicast address. The multicast address is stored locally so that the node can recognise IP packets with this destination address as being for itself (the IPv6 address of the multicast group is added to the MIB for the 'Groups module' on the node - see Appendix G.3.3).

Note: Although multicast messages may be received by an End Device (that is awake), these messages are discarded by the device. Therefore, an End Device should not be added to a multicast group.

The function will cause the application callback function **bJIP_GroupCallback()** to be invoked, which must provide authorisation before the change is implemented.

The node can subsequently be removed from the multicast group using the function **bJIP_RemoveGroupAddr()**.

Note that a node can simultaneously be a member of more than one multicast group. The maximum number of groups to which a node can belong is determined by the stack parameter `u8SocketMaxGroupAddrs` (see Section 9.3) - the default maximum is 8 groups but this maximum can be set to any value in the range 0-255.

Parameters

***psAddr*	Pointer to IPv6 address of multicast group to which the local node is to be added

Returns

TRUE if node successfully added to multicast group, FALSE otherwise

bJIP_RemoveGroupAddr

```
bool_t bJIP_RemoveGroupAddr(in6_addr *psAddr);
```

Description

This function can be used to remove the local node from the multicast group with the specified IPv6 multicast address (the IPv6 address of the multicast group is removed from the MIB for the ‘Groups module’ on the node - see Appendix G.3.3).

The function will cause the application callback function **bJIP_GroupCallback()** to be invoked, which must provide authorisation before the change is implemented.

The function can only be used to remove the node from a multicast group to which it has previously been added using the function **bJIP_AddGroupAddr()**.

Parameters

psAddr	Pointer to IPv6 address of multicast group to which the local node is to be removed

Returns

TRUE if node successfully removed from multicast group, FALSE otherwise

5.6 IP Functions

This section details the functions that allow certain aspects of IP packet delivery to be configured.

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

Function	**Page**
vJIP_SetDefaultMaxHopCount	125
vJIP_SetPacketDefragTimeout	126

vJIP_SetDefaultMaxHopCount

```
void vJIP_SetDefaultMaxHopCount(
                              uint8 u8userDefaultHopCount);
```

Description

This function sets a value for the maximum number of hops in the IPv6 headers of outgoing packets. The default value is 255.

Each time the packet is forwarded towards its destination by an intermediate IP host, the hop count in the IPv6 header is decremented by one. If the hop count falls below zero, the packet is discarded.

Note that the hop count is not decremented within the wireless network (which is considered to be a single hop at the IP level). Thus, the first time the count is decremented is at the Border-Router, on entering the LAN/WAN domain.

Parameters

u8userDefaultHopCount	Maximum hop count to set (in the range 1-255)

Returns

None

vJIP_SetPacketDefragTimeout

```
void vJIP_SetPacketDefragTimeout(
                        uint8 u8userTimeoutInSeconds);
```

Description

This function sets the timeout period after which incomplete packets will be discarded in order to free up buffer space. Packets arriving fragmented will be held by the stack until all the parts are available for defragmenting or until the timeout period expires.

By default, the timeout period is set to 60 seconds, in accordance with the 6LoWPAN specification. However, you are advised to set it to 1 second in order to avoid exhausting the buffer pool.

Parameters

u8userTimeoutInSeconds	Timeout period (in seconds) for which the stack waits before discarding packet fragments (default timeout period is 60 seconds)

Returns

None

6. JIP Embedded API MIB Functions

This chapter details the functions and macros of the JIP Embedded API that are used to manage MIBs and their variables, including the use of JIP traps to monitor MIB variables. The functions are defined in the header file **jip.h** and the macros are defined in **jip_define_mib.h**.

The function/macro descriptions are divided into the following sub-sections.

- Section 6.1 details the macros for defining MIB types and declaring MIBs
- Section 6.2 details the MIB initialisation function
- Section 6.3 details the functions for locally accessing MIB variables
- Section 6.4 details the functions for remotely accessing MIB variables

6.1 MIB Macros

The MIB macros are used to define a MIB type (and its variables) and then create one or more MIBs based on the MIB type (currently, only one instance of each MIB type per node is supported). The macros are divided two sets:

- MIB type definition macros, described in Section 6.1.1
- MIB declaration macros, described in Section 6.1.2

Defining MIB types and creating MIBs are described in Section 4.2.1.

6.1.1 MIB Type Definition Macros

This set of macros is used to define a MIB type and its variables. The macros automatically fill in the relevant structures.

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

Macro	**Page**
START_DEFINE_MIB	128
DEFINE_VAR	129
END_DEFINE_MIB	130

START_DEFINE_MIB

START_DEFINE_MIB(*ID*, *NAME***)**

Description

This macro can be used to begin the process of defining a MIB type. A unique identifier and name (character string) must be specified for the MIB type. The macro automatically creates a `tsJIP_MibDef` structure for the MIB type.

Use of this macro must be followed by the macros **DEFINE_VAR()** and **END_DEFINE_MIB()**.

Parameters

ID	Unique 32-bit identifier of MIB type
NAME	Character string representing unique name for MIB type

DEFINE_VAR

DEFINE_VAR(*ID*, *TYPE*, *NAME*, *DISPLAY_NAME*,
FLAGS, *ACCESS*, *CACHE*, *SECURITY***)**

Description

This macro can be used to define a variable for a MIB type. A unique identifier and name must be specified for the variable, as well as a number of other properties. The macro automatically creates a `tsJIP_VarDef` structure for the variable and also fills in the relevant fields of the `tsJIP_MibDef` structure for the MIB type.

The macro creates only one variable at a time and so must be used repeatedly to create multiple variables. Use of this macro must follow **START_DEFINE_MIB()** and precede **END_DEFINE_MIB()**.

Parameters

ID	Unique 8-bit identifier for the variable within the MIB
TYPE	Data type of variable - one of the types listed in Section 8.2.2
NAME	Character string representing unique name for variable, used to identify the variable within the code
DISPLAY_NAME	Character string representing a human-readable name to be used in query responses relating to the variable (NULL setting indicates that string specified in *NAME* is to be used)
FLAGS	Reserved for future use
ACCESS	Type of access allowed to the variable - one of the access types listed in Section 8.2.3
CACHE	Value indicating the valid lifetime of the variable when cached (see below)
SECURITY	Security applied to the variable (currently must always be set to E_JIP_SECURITY_NONE)

Cache Macros

Any one of the following macros can be used in the *CACHE* parameter to indicate the valid lifetime of the variable (indicating the maximum time for which the variable can be cached and therefore the minimum refresh rate):

Cache Macro	Comments
E_JIP_CACHE_NONE	Variable should not be cached - for future use
E_JIP_CACHE_SECONDS(*TIME***)**	Specifies cache time-limit in seconds (through *TIME*) - for future use
E_JIP_CACHE_MINUTES(*TIME***)**	Specifies cache time-limit in minutes (through *TIME*) - for future use
E_JIP_CACHE_HOURS(*TIME***)**	Specifies cache time-limit in hours (through *TIME*) - for future use
E_JIP_CACHE_DAYS(*TIME***)**	Specifies cache time-limit in days (through *TIME*) - for future use
E_JIP_CACHE_CONST	The variable is a constant, so there is no cache time-limit

END_DEFINE_MIB

END_DEFINE_MIB(*NAME***)**

Description

This macro can be used to finish the process of defining a MIB type. The macro automatically fills in the relevant fields of the `tsJIP_MibDef` structure for the MIB type.

Use of this macro must follow the macros **START_DEFINE_MIB()** and **DEFINE_VAR()**.

Parameters

NAME	Character string representing name of the MIB type

6.1.2 MIB Declaration Macros

This set of macros is used to create a MIB based on a (pre-defined) MIB type. The macros automatically fill in the relevant structure.

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

Macro	Page
JIP_START_DECLARE_MIB	132
JIP_CALLBACK	133
JIP_END_DECLARE_MIB	134

JIP_START_DECLARE_MIB

```
JIP_START_DECLARE_MIB(DEF_NAME,
                      INST_NAME)
```

Description

This macro can be used to create a MIB based on a (pre-defined) MIB type. A unique name (character string) must be specified for the MIB. The macro automatically creates a `tsJIP_MibInst` structure for the MIB.

Use of this macro must be followed by the macros **JIP_CALLBACK()** and **JIP_END_DECLARE_MIB()**.

Parameters

DEF_NAME	Name of pre-defined MIB type
INST_NAME	Character string representing unique name for MIB

JIP_CALLBACK

JIP_CALLBACK(*NAME*, *SET*, *GET*, *DATA***)**

Description

This macro can be used to declare a variable of a MIB and specify the user-defined callback functions that will be used to perform read and write accesses to the variable.

The macro applies to only one variable at a time and so must be used repeatedly for multiple variables. Use of this macro must follow **JIP_START_DECLARE_MIB()** and precede **JIP_END_DECLARE_MIB()**.

Parameters

NAME	Character string representing name of variable
SET	Name of callback function to be used to set (write) value of variable
GET	Name of callback function to be used to get (read) value of variable
DATA	Pointer to custom data to be passed to callback functions

JIP_END_DECLARE_MIB

```
JIP_END_DECLARE_MIB(INST_NAME,
                    INST_HANDLE)
```

Description

This macro can be used to finish the process of creating a MIB. The macro automatically fills in the relevant fields of the `tsJIP_MibInst` structure for the MIB. A handle must be specified which will be used to refer to the MIB in function calls.

Use of this macro must follow the macros **JIP_START_DECLARE_MIB()** and **JIP_CALLBACK()**.

Parameters

INST_NAME	Character string representing name of the MIB
INST_HANDLE	Character string representing handle for the MIB

6.2 MIB Initialisation Function

A MIB initialisation function is provided to register a MIB with JenNet-IP.

The function is listed below, along with its page reference:

Function	Page
eJIP_RegisterMib	136

eJIP_RegisterMib

```
teJIP_Status eJIP_RegisterMib(thJIP_Mib hMib);
```

Description

This function is used to register a MIB with JenNet-IP, where the MIB has already been created by the application (using the macros described in Section 6.1). The MIB is specified by passing its handle into the function (this is the handle that was specified for the MIB in the **JIP_END_DECLARE_MIB()** macro).

Parameters

hMib	Handle of MIB to be registered

Returns

E_JIP_OK

6.3 Local Variable Access Functions

The local variable access functions are concerned with accesses to MIB variables on the local node. Most are concerned with the generation of notification messages that result from changes in MIB variables and that are used by JIP traps.

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

Function	Page
vJIP_NotifyChanged	138
vJIP_SetEnabled	139
eJIP_PacketAddData	140
eJIP_AddTrap	148
eJIP_RemoveTrap	149

vJIP_NotifyChanged

```
void vJIP_NotifyChanged(thJIP_Mib hMib,
                                    uint8 u8VarID);
```

Description

This function generates a trap notification of a change in the specified MIB variable. The application may call this function when the value of the variable changes and a notification of this change must be communicated to other nodes that are trapping the variable.

Calling this function results in sending a trap notification message to any remote nodes that are trapping the specified variable. On receiving this notification, a remote node will invoke the callback function **vJIP_Remote_TrapNotify()**. For more information on variable trapping, refer to Section 4.2.5.

Parameters

hMib	Handle of MIB which contains the variable that has changed
u8VarID	Identifier of the variable that has changed

Returns

None

vJIP_SetEnabled

```
void vJIP_SetEnabled(thJIP_Mib hMib,
                     uint8 u8VarID,
                     bool_t bEnabled);
```

Description

This function can be used to change the enabled state of a MIB variable and to generate a trap notification to indicate this change. The enabled state of a MIB variable determines whether access to the variable is allowed.

Calling this function results in sending a trap notification message to any remote nodes that are trapping the specified variable. On receiving this notification, a remote node will invoke the callback function **vJIP_Remote_TrapNotify()**. For more information on variable trapping, refer to Section 4.2.5.

Parameters

hMib	Handle of MIB which contains the variable that has changed
u8VarID	Identifier of the variable that has changed
bEnabled	The enabled state of the variable, one of: TRUE - the variable can be accessed FALSE - the variable cannot be accessed

Returns

None

eJIP_PacketAddData

```
teJIP_Status eJIP_PacketAddData(
                                    thJIP_Packet hHandle,
                                    void *pvData,
                                    uint32 u32Len,
                                    uint32 u32Entry);
```

Description

This function can be used to add a read MIB variable value to the response of a Get request (issued on a remote node using **eJIP_Remote_TableGet()**). The function must insert the obtained data and the relevant MIB entry index (if relevant) into the response. It should be used in the user-defined callback function that handles the Get request and generates the response.

While this function should normally be called once in the callback function, it must be called multiple times for variables that are tables, in order to add multiple entries to the response.

Parameters

hHandle	Handle of response packet
pvData	Pointer to data to be added to response packet
u32Len	Length of data to be added, in bytes
u32Entry	Index of MIB entry from which data obtained (only relevant to table type variables)

Returns

E_JIP_OK

eJIP_AddTrap

```
teJIP_Status eJIP_AddTrap(thJIP_Mib phMib,
                          uint8 u8VarID,
                          ts6LP_SockAddr *psAddr,
                          uint8 u8Handle);
```

Description

This function can be used to set a trap on the specified local MIB variable. The trap notifications will be sent to the multicast group corresponding to the specified IPv6 multicast address.

This trap can be removed using the function **eJIP_RemoveTrap()**.

Parameters

phMib	Handle of MIB which contains the variable to be trapped
u8VarID	Identifier of the variable to be trapped
psAddr	Pointer to a structure containing the IPv6 multicast address to which trap notifications will be sent (see Section 8.1.4)
u8Handle	A user-defined handle for the local trap request

Returns

E_JIP_OK

E_JIP_ERROR_FAILED

eJIP_RemoveTrap

```
teJIP_Status eJIP_RemoveTrap(thJIP_Mib phMib,
                             uint8 u8VarID,
                             ts6LP_SockAddr *psAddr);
```

Description

This function can be used to remove a trap on the specified local MIB variable, for which trap notifications are sent to the specified IPv6 multicast address.

Parameters

phMib	Handle of MIB which contains the variable to be untrapped
u8VarID	Identifier of the variable to be untrapped
psAddr	Pointer to a structure containing the IPv6 multicast address to which trap notifications are sent (see Section 8.1.4)

Returns

E_JIP_OK

E_JIP_ERROR_FAILED

6.4 Remote Variable Access Functions

The remote variable access functions are used to send requests to remote nodes to set, get, trap and query MIB variables, as well as query the available MIBs.

The requests sent using these functions are handled on the remote node by the user-defined callback functions described in Section 7.2.

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

Function	**Page**
eJIP_Remote_ID_Set	144
eJIP_Remote_TableGet	146
eJIP_Remote_Trap	148
eJIP_Remote_Untrap	149
eJIP_Remote_QueryMib	150
eJIP_Remote_QueryVar	151

eJIP_Remote_ID_Set

```
teJIP_Status eJIP_Remote_ID_Set(
                        ts6LP_SockAddr *psAddr,
                        uint8 u8Handle,
                        uint32 u32MibID,
                        uint8 u8VarIndex,
                        teJIP_VarType eVarType,
                        void *pvVal,
                        uint32 u32ValSize
                        bool_t bUseMibIndex);
```

Description

This function sends a request to set or change the value of a MIB variable on a remote node. A call to this function results in events which generate calls to the following callback functions:

- **vJIP_Remote_DataSent()**: Callback function which returns an enumeration value that can be used to check the result of the attempt to send a request to a remote node.
- **vJIP_Remote_SetResponse()**: Callback function which returns an enumeration value that can be used to check the result of the attempt to set the value of the remote variable.

The relevant MIB can be specified using either an 8-bit MIB index or 32-bit MIB identifier, as selected through the *bUseMibIndex* parameter (the MIB ID is associated with a MIB type but can be used to specify a MIB here since only one MIB of each type is permitted in the current JenNet-IP release).

As alternatives to this function, macros are provided that automatically set the *bUseMibIndex* parameter:

- **eJIP_Remote_Set()** sets *bUseMibIndex* to TRUE, so that the MIB index is used
- **eJIP_Remote_MIB_Set()** sets *bUseMibIndex* to FALSE, so that the MIB ID is used

All other parameters for these macros are the same as for **eJIP_Remote_ID_Set()** - for the prototypes, refer to the header file **JIP.h**.

Parameters

**psAddr*	Pointer to a structure containing the address and port number for the remote node. In practice, all nodes will run JIP on the same port, therefore the remote port will be the same as the local port used for JIP.
u8Handle	A user-defined handle for the set request. This handle is then used by **vJIP_Remote_SetResponse()**, allowing the request and its response to be matched. Note that bit 7 is reserved for a ‘stay awake’ flag (see Section 4.6.3)
u32MibID	Identifier or index (see *bUseMibIndex* below) of the MIB containing the variable on the remote node.
u8VarIndex	Index of the variable within the MIB on the remote node.
eVarType	The type of variable being set (must match the type on the remote node).

**pvVal*	Pointer to the object holding the value to send.
u32ValSize	Size, in bytes, of the object pointed to by **pvVal*. If this is the wrong size for the type of variable then the behaviour is undefined.
bUseMibIndex	Indicates whether MIB index or ID to be used to identify MIB in *u32MibID* parameter (above), one of: TRUE - Use 8-bit MIB index FALSE - Use 32-bit MIB ID

Returns

E_JIP_OK (Success)

E_JIP_ERROR_FAILED (Unable to send the request through the UDP layer).

eJIP_Remote_TableGet

```
teJIP_Status eJIP_Remote_TableGet(
                    ts6LP_SockAddr *psAddr,
                    uint8 u8Handle,
                    uint32 u32MibID,
                    uint8 u8VarIndex,
                    uint16 u16FirstEntry,
                    uint8 u8EntryCount,
                    bool_t bUseMibIndex);
```

Description

This function sends a request to retrieve the value of a MIB variable on a remote node. If the MIB variable is of the table datatype, the first table entry and the number of table entries to be returned must be specified. A call to this function results in events which generate calls to the following callback functions:

- **vJIP_Remote_DataSent()**: Callback function which returns an enumeration value that can be used to check the result of the attempt to send a request to a remote node.
- **vJIP_Remote_TableGetResponse()**: Callback function which is invoked for each table entry in the response (and is invoked at least once, even if no entries are returned).
- **vJIP_Remote_GetResponse()**: Callback function which returns an enumeration value that can be used to check the status of the attempt to get the value of a remote variable.

The relevant MIB can be specified using either an 8-bit MIB index or 32-bit MIB identifier, as selected through the *bUseMibIndex* parameter (the MIB ID is associated with a MIB type but can be used to specify a MIB here since only one MIB of each type is permitted in the current JenNet-IP release).

As alternatives to this function, macros are provided that automatically set the *bUseMibIndex, u16FirstEntry* and *u8EntryCount* parameters:

- **eJIP_Remote_Get()** sets *bUseMibIndex* to TRUE, so that the MIB index is used, *u16FirstEntry* to 0 and *u8EntryCount* to 255
- **eJIP_Remote_MIB_Get()** sets *bUseMibIndex* to FALSE, so that the MIB ID is used, *u16FirstEntry* to 0 and *u8EntryCount* to 255

All other parameters for these macros are the same as for **eJIP_Remote_TableGet()** - for the prototypes, refer to the header file **JIP.h**.

Parameters

**psAddr*	Pointer to a structure containing the address and port number for the remote node. In practice, all nodes will run JIP on the same port, therefore the remote port will be the same as the local port used for JIP.
u8Handle	A user-defined handle to the get request. This handle is then used by **vJIP_Remote_TableGetResponse()** and **vJIP_Remote_GetResponse()**. The stack does not use this value - it is provided so the application can match the request

	with the response. Note that bit 7 is reserved for a ‘stay awake’ flag (see Section 4.6.3)
u32MibID	Identifier or index (see *bUseMibIndex* below) of the MIB containing the variable on the remote node.
u8VarIndex	Index of the variable within the MIB on the remote node.
u16FirstEntry	Index of first table entry requested (ignored if variable is not a table datatype) - required when accessing a table with more entries than can be received in a single request.
u8EntryCount	Number of table entries to return (ignored if variable is not a table datatype - treated as a maximum, the remote node may return fewer entries.
bUseMibIndex	Indicates whether MIB index or ID to be used to identify MIB in *u32MibID* parameter (above), one of: TRUE - Use 8-bit MIB index FALSE - Use 32-bit MIB ID

Returns

E_JIP_OK (Success)

E_JIP_ERROR_FAILED (Unable to send the request through the UDP layer).

eJIP_Remote_Trap

```
teJIP_Status eJIP_Remote_Trap(
                        ts6LP_SockAddr *psAddr,
                        uint8 u8Handle,
                        uint8 u8NotificationHandle,
                        uint8 u8MibIndex,
                        uint8 u8VarIndex);
```

Description

This function sets up a remote trap on a MIB variable on a remote node so that changes to the variable can be monitored. A call to this function results in events which generate calls to the following callback functions

- **vJIP_Remote_DataSent()**: Callback function which returns an enumeration value that can be used to check the result of the attempt to send a request to a remote node.
- **vJIP_Remote_TrapResponse()**: Callback function which returns an enumeration value that can be used to check the result of the attempt to set a remote variable trap.

Parameters

**psAddr*	Pointer to a structure containing the address and port number for the remote node. In practice, all nodes will run JIP on the same port, therefore the remote port will be the same as the local port used for JIP.
u8Handle	A user-defined handle for the remote trap request. This handle is then used by **vJIP_Remote_TrapResponse()**, allowing the request and its response to be matched. Note that bit 7 is reserved for a 'stay awake' flag (see Section 4.6.3)
u8NotificationHandle	A user-defined handle for the trap notification request. This handle is then used by **vJIP_Remote_TrapNotify()**.
u8MibIndex	Index of the MIB containing the variable on the remote node.
u8VarIndex	Index of the variable within the MIB on the remote node.

Returns

E_JIP_OK (Success)

E_JIP_ERROR_FAILED (Unable to send the request through the UDP layer).

eJIP_Remote_Untrap

```
teJIP_Status eJIP_Remote_Untrap(
                    ts6LP_SockAddr *psAddr,
                    uint8 u8Handle,
                    uint8 u8MibIndex,
                    uint8 u8VarIndex);
```

Description

This function sends a request to unregister an interest in a trapped MIB variable on a remote node. A call to this function results in events which generate calls to the following callback functions:

- **vJIP_Remote_DataSent()**: Callback function which returns an enumeration value that can be used to check the result of the attempt to send a request to a remote node.
- **vJIP_Remote_TrapResponse()**: Callback function which returns an enumeration value that can be used to check the result of the attempt to untrap a remote variable.

Parameters

**psAddr*	Pointer to a structure containing the address and port number for the remote node. In practice, all nodes run JIP on the same port, therefore the remote port will be the same as the local port used for JIP.
u8Handle	A user-defined handle for the remote untrap request. This handle is then used by **vJIP_Remote_TrapResponse()**. Note that bit 7 is reserved for a 'stay awake' flag (see Section 4.6.3)
u8MibIndex	Index of the MIB containing the variable on the remote node.
u8VarIndex	Index of the variable within the MIB on the remote node.

Returns

E_JIP_OK (Success)

E_JIP_ERROR_FAILED (Unable to send the request through the UDP layer)

eJIP_Remote_QueryMib

```
teJIP_Status eJIP_Remote_QueryMib(
                    ts6LP_SockAddr *psAddr,
                    uint8 u8Handle,
                    uint8 u8MibStartIndex,
                    uint8 u8NumMibs);
```

Description

This function sends a request for a list of the MIBs that are present on a remote node. A call to this function results in events which generate calls to the following callback functions:

- **vJIP_Remote_DataSent()**: Callback function which returns an enumeration value that can be used to check the result of the attempt to send a request to a remote node.
- **vJIP_Remote_QueryMibResponse()**: Callback function which returns a query status enumeration for the MIB query request, the total number of MIBs on the node, and the number remaining to be reported.

Parameters

**psAddr*	Pointer to a structure containing the address and port number for the remote node. In practice, all nodes will run JIP on the same port, therefore the remote port will be the same as the local port used for JIP.
u8Handle	A user-defined handle for the query request. This handle is then used by **vJIP_Remote_QueryMibResponse()**. Note that bit 7 is reserved for a ‘stay awake’ flag (see Section 4.6.3)
u8MibStartIndex	Index of the first MIB to return on the remote node.
u8NumMibs	The maximum number of MIBs to return in the list. Because of the maximum size of a UDP packet, the number of MIBs returned may be less than this.

Returns

E_JIP_OK (Success)

E_JIP_ERROR_FAILED (Unable to send the request through the UDP layer)

eJIP_Remote_QueryVar

```
teJIP_Status eJIP_Remote_QueryVar(
                    ts6LP_SockAddr *psAddr,
                    uint8 u8Handle,
                    uint8 u8MibIndex,
                    uint8 u8VarStartIndex,
                    uint8 u8NumVars);
```

Description

This function sends a request to query the variables in a particular MIB on a remote node - that is, to obtain a list of the variables in the MIB. The function allows you to specify the number of variables and the index of the first variable to include in the returned list (useful when the function must be called several times to report all variables of the MIB). This function call results in calls to the following callback functions:

- **vJIP_Remote_DataSent()**: Callback function which returns an enumeration value that can be used to check the result of the attempt to send a request to a remote node.
- **vJIP_Remote_QueryVarResponse()**: Callback function which returns a query status enumeration value and a pointer to an array containing the variable details.

Parameters

**psAddr*	Pointer to a structure containing the address and port number for the remote node. In practice, all nodes will run JIP on the same port, therefore the remote port will be the same as the local port used for JIP.
u8Handle	A user-defined handle for the remote query variable request. This handle is then used by **vJIP_Remote_QueryVarResponse()**. Note that bit 7 is reserved for a 'stay awake' flag (see Section 4.6.3)
u8MibIndex	Index of the MIB containing the variable on the remote node.
u8VarStartIndex	Index of the first variable within the remote MIB to return.
u8NumVars	The maximum number of variables to return in the list. Because of the maximum size of a UDP packet, the number of variables returned may be less than this.

Returns

E_JIP_OK (Success)
E_JIP_ERROR_FAILED (Unable to send the request through the UDP stack)

7. JIP Embedded API Callback Functions

This chapter describes functions that must be defined by the user as callback (stack to application) functions to deal with data, stack and peripheral events occurring in the stack.

The callback functions are mostly called in application context, resulting from application calls into the stack - only **vJIP_PeripheralEvent()** is called in interrupt context.

Note: If a call is made into the stack from within one of the callback functions then the callback function may be re-entered.

The JIP Embedded API callback functions are divided into two categories:

- General callback functions, detailed in Section 7.1
- MIB and trap callback functions, detailed in Section 7.2

7.1 General Callback Functions

The general callback functions deal with network configuration, multicast groups and event handling.

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

Function	**Page**
vJIP_ConfigureNetwork	154
bJIP_GroupCallback	155
vJIP_PeripheralEvent	156
vJIP_StackEvent	157
v6LP_DataEvent	159
vJIP_StayAwakeRequest	160

vJIP_ConfigureNetwork

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

Description

This function sets the JenNet parameters. The required values of these parameters are passed into the function by means of a structure of type **tsNetworkConfigData**. For details of this structure and descriptions of the parameters, refer to Chapter 9.

This callback function is invoked during stack initialisation which results from a call to **eJIP_Init()**. The latter function sets the default values for all the JenNet parameters before **vJIP_ConfigureNetwork()** is invoked, but this callback function will over-write the previously set values.

Parameters

***psNetworkConfigData*	Pointer to structure containing values to be set

Returns

None

bJIP_GroupCallback

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

Description

This function is called when a request has been initiated to add the local node to or remove it from a multicast group through a call to **bJIP_AddGroupAddr()** or **bJIP_RemoveGroupAddr().** The purpose of the function is to authorise the request - if the function returns TRUE then JenNet-IP will perform the necessary modifications to satisfy the request.

Parameters

eEvent	Type of request, one of: E_JIP_GROUP_JOIN (request to join group) E_JIP_GROUP_LEAVE (request to leave group)
***psAddr*	Pointer to IPv6 multicast address of relevant group

Returns

TRUE if the request is authorised, FALSE if the request is refused

vJIP_PeripheralEvent

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

Description

This function handles an interrupt generated by an on-chip peripheral.

Note: This function is called in interrupt context (unlike the other callback functions, which are called in application context).

Refer to the *JN516x Integrated Peripherals API User Guide (JN-UG-3087*) for further information on peripheral interrupts, including the possible values of the function parameters.

Parameters

u32Device	Peripheral device that generated interrupt
u32ItemBitmap	Individual interrupt source within peripheral

Returns

None

vJIP_StackEvent

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

Description

This function handles a stack management event from the IEEE 802.15.4 or JenNet layer of the stack. For some stack events (see below), data accompanies the event and is made available through the function parameters.

Parameters

eEvent	Type of stack management event received, one of: E_STACK_STARTED E_STACK_JOINED E_STACK_NODE_JOINED E_STACK_NODE_LEFT E_STACK_TABLES_RESET E_STACK_RESET E_STACK_POLL E_STACK_NODE_JOINED_NWK E_STACK_NODE_LEFT_NWK E_STACK_NODE_AUTHORISE E_STACK_ROUTE_CHANGE E_STACK_GROUP_CHANGE
**pu8Data*	Pointer to additional data for some events (for details, see table below)
u8DataLen	Length of additional data (if relevant), in bytes

The table below details the stack events for which additional data is returned through *pu8Data*:

Stack Event	***pu8Data***	***u8DataLen* (bytes)**
E_STACK_STARTED	Points to `tsNwkInfo` structure (detailed in Section 8.1.2)	Size of `tsNwkInfo`
E_STACK_JOINED	Points to `tsNwkInfo` structure (detailed in Section 8.1.2)	Size of `tsNwkInfo`
E_STACK_POLL	Points to the poll-response byte which is of the enumerated type `teJIP_PollResponse` (detailed in Section 8.2.2).	1

Stack Event	*pu8Data*	*u8DataLen* (bytes)
E_STACK_NODE_JOINED	Points to tsAssocNodeInfo structure. The sMacAddr and u32DeviceClass elements of the structure contain useful data (e.g. address of node that has joined or left) but u16NetworkAddr is always 0xFFFE	Size of tsAssocNodeInfo
E_STACK_NODE_LEFT		
E_STACK_NODE_JOINED_NWK		
E_STACK_NODE_LEFT_NWK		
E_STACK_ROUTE_CHANGE		
E_STACK_RESET	Points to tsStackReset structure (detailed in Section 8.1.17), containing information relating to the stack reset	Size of tsStackReset
E_STACK_NODE_AUTHORISE	Points to MAC_ExtAddr_s structure, containing the address of the node which is attempting to gain authorisation	Size of MAC_ExtAddr_s
E_STACK_GROUP_CHANGE	Points to tsJIP_StackGroupChange structure, containing a list of multicast groups which node has been added to or removed from (allows application to keep track of changes initiated remotely)	Size of tsJIP_StackGroupChange

Returns

None

v6LP_DataEvent

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

Description

This function handles a data management event by servicing the buffer associated with the local socket on which the event occurred.

After receiving an E_DATA_RECEIVED, E_IP_DATA_RECEIVED or E_6LP_ICMP_MESSAGE data event on a socket, the application must ensure that the associated buffer is dealt with and released by calling **i6LP_RecvFrom()** on the socket. If the packet is not needed, the buffer can simply be released by calling **i6LP_RecvFrom()**, setting the *pu8Data* parameter to NULL.

An E_6LP_ICMP_MESSAGE event contains an ICMP message that needs to be handled by the application (normally, ICMP messages are handled by the stack). The message can be one of a number of types (indicated in its header) and can be read using the **i6LP_RecvFrom()** function. For more information on handling ICMP messages, refer to Appendix C. ICMP messages are also described in RFC 4443 available from the IETF (www.ietf.org).

Parameters

iSocket	Socket identity (for E_IP_DATA_RECEIVED event, this can be ignored).
eEvent	Type of data management event received, one of: E_DATA_SENT E_DATA_SEND_FAILED E_DATA_RECEIVED E_IP_DATA_RECEIVED E_6LP_ICMP_MESSAGE
**psAddr*	Pointer to structure containing the address/port (either source or destination, depending on event)
u8AddrLen	Length of address/port structure, in bytes

Returns

None

vJIP_StayAwakeRequest

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

Description

This function is called on an End Device when a packet is received in which the ‘stay awake’ flag is set (bit 7 of the handle). This bit is set by the source device of the packet to request the target End Device to stay awake in order to receive further packets.

The End Device is not obliged to stay awake, so this callback function may ignore the request (for example, the End Device may not be able to accept such requests due to limited power). Otherwise, to honour the request, the callback function should implement a timer to postpone sleep.

For a full description of the ‘stay awake’ request, refer to Section 4.6.3.

Parameters

None

Returns

None

7.2 MIB and Trap Callback Functions

The MIB and trap callback functions are used to react to requests relating to MIB and trap management.

These callback functions are invoked following calls to the functions detailed in Section 6.4 - that is, to deal with responses to requests to set, get, trap and query MIB variables, as well as query the available MIBs. For example, the callback function **vJIP_Remote_GetResponse()** is invoked as the result of a call to the function **eJIP_Remote_TableGet()** that requests information on a variable.

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

Function	Page
vJIP_Remote_SetResponse	162
vJIP_Remote_GetResponse	163
vJIP_Remote_TableGetResponse	164
vJIP_Remote_TrapResponse	166
vJIP_Remote_TrapNotify	167
vJIP_Remote_QueryMibResponse	168
vJIP_Remote_QueryVarResponse	169
vJIP_Remote_DataSent	171

vJIP_Remote_SetResponse

void vJIP_Remote_SetResponse(
ts6LP_SockAddr **psAddr*,
uint8 *u8Handle*,
uint8 *u8MibIndex*,
uint8 *u8VarIndex*,
teJIP_Status *eStatus***);**

Description

This callback function is invoked to handle the response to a remote variable Set request issued by **eJIP_Remote_ID_Set()**. The response provides the result of the Set request.

Parameters

**psAddr*	Pointer to a structure containing the address and port number for the remote node.
u8Handle	Handle for the remote Set response callback. This handle matches the user-defined handle passed into the original **eJIP_Remote_ID_Set()** call.
u8MibIndex	Index of the MIB containing the variable on the remote node.
u8VarIndex	Index of the variable within the MIB on the remote node.
eStatus	The result of attempting to set the value, one of: • E_JIP_OK (Success) • E_JIP_ERROR_BAD_MIB_INDEX (MIB index is out of range) • E_JIP_ERROR_BAD_VAR_INDEX (Variable index is out of range for the specified MIB) • E_JIP_ERROR_NO_ACCESS (Variable is read-only or constant) • E_JIP_ERROR_BAD_BUFFER_SIZE (For string types, string is too long; for blob types, data is wrong length) • E_JIP_ERROR_WRONG_TYPE (Variable is of a different type from the one in the request) • E_JIP_ERROR_BAD_VALUE (Application on the remote host rejected the value) • E_JIP_ERROR_DISABLED (Remote variable has been disabled so cannot be set) • E_JIP_ERROR_FAILED (Unknown error)

Returns

None

vJIP_Remote_GetResponse

```
void vJIP_Remote_GetResponse(
                    ts6LP_SockAddr *psAddr,
                    uint8 u8Handle,
                    uint8 u8MibIndex,
                    uint8 u8VarIndex,
                    teJIP_Status eStatus,
                    teJIP_VarType eVarType,
                    const void *pvVal,
                    uint32 u32ValSize);
```

Description

This callback function is invoked to handle the response to a remote variable Get request issued by **eJIP_Remote_TableGet()**. The response provides the result of the Get request, including the value of the variable.

Parameters

**psAddr*	Pointer to a structure containing the address and port number for the remote node.
u8Handle	Handle for the remote Get response callback. This handle matches the user-defined handle passed into the original **eJIP_Remote_TableGet()** call.
u8MibIndex	Index of the MIB containing the variable on the remote node.
u8VarIndex	Index of the variable within the MIB on the remote node.
eStatus	The result of attempting to get the value, one of: • E_JIP_OK (Success) • E_JIP_ERROR_BAD_MIB_INDEX (MIB index is out of range) • E_JIP_ERROR_BAD_VAR_INDEX (Variable index is out of range for that MIB) • E_JIP_ERROR_DISABLED (Remote variable has been disabled so cannot be retrieved) • E_JIP_ERROR_FAILED (Unknown error)
**pvVal*	Pointer to the object holding the value returned
u32ValSize	Size, in bytes, of the object pointed to by **pvVal*.

Returns

None

vJIP_Remote_TableGetResponse

```
void vJIP_Remote_TableGetResponse(
                    ts6LP_SockAddr *psAddr,
                    uint8 u8Handle,
                    uint8 u8MibIndex,
                    uint8 u8VarIndex,
                    teJIP_Status eStatus,
                    teJIP_VarType eVarType,
                    uint16 u16Entry,
                    uint16 u16Remaining,
                    uint8 u8PacketRemaining,
                    const void *pvVal,
                    uint32 u32ValSize);
```

Description

This callback function is invoked for each table entry (of a table variable) in a response to a remote variable Get request issued by **eJIP_Remote_TableGet()**. The function reports the value of the table entry.

It is called at least once, even if there are no table entries in the response - for example, the end of the table has been reached. In this case, the parameters *u16Entry*, *u8PacketRemaining* and *u32ValSize* will be zero, and **pvVal* will be NULL.

Parameters

**psAddr*	Pointer to a structure containing the address and port number for the remote node.
u8Handle	Handle for the remote Get response callback. This handle matches the user-defined handle passed into the original **eJIP_Remote_TableGet()** call
u8MibIndex	Index of the MIB containing the variable on the remote node.
u8VarIndex	Index of the variable within the MIB on the remote node.
eStatus	The result of attempting to get the value, one of: • E_JIP_OK (Success) • E_JIP_ERROR_BAD_MIB_INDEX (MIB index is out of range) • E_JIP_ERROR_BAD_VAR_INDEX (Variable index is out of range for that MIB) • E_JIP_ERROR_DISABLED (Remote variable has been disabled so cannot be retrieved) • E_JIP_ERROR_FAILED (Unknown error)
eVarType	Data type of the variable.
u16Entry	Entry number for the entry being reported.
u16Remaining	Number of entries in the table after this entry.
u8PacketRemaining	Number of entries returned in this packet after this entry.
**pvVal*	Pointer to the object holding the entry value returned.

u32ValSize	Size, in bytes, of the object pointed to by **pvVal*.

Returns

None

vJIP_Remote_TrapResponse

```
void vJIP_Remote_TrapResponse(
                    ts6LP_SockAddr *psAddr,
                    uint8 u8Handle,
                    uint8 u8MibIndex,
                    uint8 u8VarIndex,
                    teJIP_Status eStatus);
```

Description

This callback function is invoked to handle the response to a trap request issued by **eJIP_Remote_Trap()** or an untrap request issued by **eJIP_Remote_Untrap()**. The response provides the result of the trap request.

Parameters

**psAddr*	Pointer to a structure containing the address and port number for the remote node.
u8Handle	Handle for the remote trap response callback. This handle matches the user-defined handle passed into the original **eJIP_Remote_Trap()** or **eJIP_Remote_Untrap()** call.
u8MibIndex	Index of the MIB containing the variable on the remote node.
u8VarIndex	Index of the variable within the MIB on the remote node.
eStatus	The result of attempting to trap the value, one of: • E_JIP_OK (Success) • E_JIP_ERROR_BAD_MIB_INDEX (MIB index is out of range) • E_JIP_ERROR_BAD_VAR_INDEX (Variable index is out of range for the specified MIB) • E_JIP_ERROR_DISABLED (Remote variable has been disabled. A notification will however be issued with a status of “Success” when the variable is enabled) • E_JIP_ERROR_FAILED (Unknown error)

Returns

None

vJIP_Remote_TrapNotify

```
void vJIP_Remote_TrapNotify(
                    ts6LP_SockAddr *psAddr,
                    uint8 u8Handle,
                    uint8 u8MibIndex,
                    uint8 u8VarIndex,
                    teJIP_Status eStatus,
                    teJIP_VarType eVarType,
                    void *pvVal,
                    uint32 u32ValSize);
```

Description

This callback function issues a notification to the local application as the result of a trap notification received from a remote node, indicating either of the following:

- the value of a trapped variable on the remote node has changed
- the enabled state of a trapped variable on the remote node has been changed

The response provides the results of the trap, including the type and value of the variable. A trap for the variable will have been previously set up using **eJIP_Remote_Trap()**.

Parameters

**psAddr*	Pointer to a structure containing the address and port number for the remote node.
u8Handle	Handle for the trap notification. This handle matches the user-defined handle passed into the original **eJIP_Remote_Trap()** call.
u8MibIndex	Index of the MIB containing the variable on the remote node.
u8VarIndex	Index of the variable within the MIB on the remote node.
eStatus	The result of attempting to retrieve the value, one of: • E_JIP_OK (Success) • E_JIP_ERROR_DISABLED (Remote variable has been disabled. A notification will be issued with a status of “Success” when the variable is re-enabled) • E_JIP_ERROR_FAILED (Unknown error)
eVarType	Type of the variable returned.
**pvVal*	Pointer to the object holding the value returned.
u32ValSize	Size, in bytes, of the object pointed to by **pvVal*.

Returns

None

vJIP_Remote_QueryMibResponse

```
void vJIP_Remote_QueryMibResponse(
                    ts6LP_SockAddr *psAddr,
                    uint8 u8Handle,
                    teJIP_Status eStatus,
                    uint16 u16MibsOutstanding,
                    uint8 u8MibsRemaining,
                    tsJIP_QueryMibResponse *psMib);
```

Description

This callback function is invoked to handle the response to a MIB Query request issued by **eJIP_Remote_QueryMib()**, which is used to obtain a list of the MIBs on a remote node. The callback function returns the details of one MIB reported in the response.

The function returns a structure containing the identification information for a MIB (see Section 8.1.9). However, the response may contain information on multiple MIBs and so the stack may call this function multiple times (until the parameter *u8MibsRemaining* becomes zero). In addition, the response may not report all the MIBs on the remote node - there may be more MIBs to be reported (as indicated by a non-zero value of the parameter *16MibsOutstanding*). If this is the case, the application must call **eJIP_Remote_QueryMib()** again to query the remaining MIBs on the remote node.

Parameters

**psAddr*	Pointer to a structure containing the address and port number for the remote node.
u8Handle	Handle for the remote MIB query response callback. This handle matches the user-defined handle passed into the original **eJIP_Remote_QueryMib()** call.
eStatus	The result of attempting to query, one of: • E_JIP_OK (Success) • E_JIP_ERROR_BAD_MIB_INDEX (*u8MibStartIndex* parameter in **eJIP_Remote_QueryMib()** is out of range) • E_JIP_ERROR_FAILED: (Unknown error)
u16MibsOutstanding	The number of MIBs that remain to be queried after this response. If this is non-zero then the application may need to make further calls to **eJIP_Remote_QueryMib()**.
u8MibsRemaining	Number of MIBs that remain to be extracted from this response.
**psMib*	Pointer to structure containing the extracted MIB information (see Section 8.1.9).

Returns

None

vJIP_Remote_QueryVarResponse

```
void vJIP_Remote_QueryVarResponse(
                         ts6LP_SockAddr *psAddr,
                         uint8 u8Handle,
                         uint8 u8MibIndex,
                         teJIP_Status eStatus,
                         uint16 u16VarsOutstanding,
                         uint8 u8VarsRemaining,
                         tsJIP_QueryVarResponse *psVar);
```

Description

This callback function is invoked to handle the response to a Remote Variable Query request issued by **eJIP_Remote_QueryVar()**, which is used to obtain a list of the variables in a MIB on a remote node. The callback function returns the details of one MIB variable reported in the response.

The function returns a structure containing the information on a MIB variable (see Section 8.1.10). However, the response may contain information on multiple variables and so the stack may call this function multiple times (until the parameter *u8VarsRemaining* becomes zero). In addition, the response may not report all the variables of the relevant MIB - there may be more variables to be reported (as indicated by a non-zero value of the parameter *u16VarsOutstanding*). If this is the case, the application must call **eJIP_Remote_QueryVar()** again to query the remaining variables of the MIB.

Parameters

**psAddr*	Pointer to a structure containing the address and port number for the remote node.
u8Handle	Handle for the remote query variable response callback. This handle matches the user-defined handle passed into the original **eJIP_Remote_QueryVar()** call.
u8MibIndex	Index of the MIB containing the variables on the remote node.
eStatus	The result of attempting to query variable, one of: • E_JIP_OK (Success) • E_JIP_ERROR_BAD_MIB_INDEX (*u8MibIndex* parameter to **eJIP_Remote_QueryVar()** is out of range) • E_JIP_ERROR_BAD_VAR_INDEX (*u8VarStartIndex* parameter to **eJIP_Remote_QueryVar()** is out of range) • E_JIP_ERROR_FAILED (Unknown error)
u16VarsOutstanding	The number of variables in the MIB that remain to be queried after this response. If this is non-zero then the application may need to make further calls to **eJIP_Remote_QueryVar()**.
u16VarsReturned	The number of variables in the MIB that remain to be extracted from this response.
**psVar*	Pointer to structure describing the extracted MIB variable.

Returns

None

vJIP_Remote_DataSent

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

Description

This callback function is invoked following a function call to send a request to a MIB on a remote node. The function reports the status (success or failure) of the attempt to send the request.

Parameters

psAddr	Pointer to a structure containing the address and port number for the remote node.
eStatus	Status of send, one of: • E_JIP_OK (Success - request was sent) • E_JIP_ERROR_FAILED (Failure - request could not be sent)

Returns

None

8. JIP Embedded API Structures and Enums

This chapter details the structures and enumerations used by the functions of the JIP Embedded API described in this manual.

8.1 Data Types

8.1.1 tsJIP_InitData

This structure contains stack initialisation data and is used in the function **eJIP_Init()**. Note that some of the elements of this structure can be subsequently changed by the callback function **vJIP_ConfigureNetwork()**.

```
typedef struct
{
    uint64        u64AddressPrefix;
    uint32        u32Channel;
    uint16        u16PanId;
    uint16        u16MaxIpPacketSize;
    uint16        u16NumPacketBuffers;
    uint8         u8UdpSockets;
    teJIP_Device  eDeviceType;
    uint32        u32RoutingTableEntries;
    uint32        u32DeviceId;
    uint8         u8UniqueWatchers;
    uint8         u8MaxTraps;
    uint8         u8QueueLength;
    uint8         u8MaxNameLength;
    uint16        u16Port;
    uint16        u16JMP_Port;
    const char   *pcVersion;
} tsJIP_InitData;
```

The elements of the above structure are described in the table below.

tsStackInitData Element	**Description**	**Range**
u64AddressPrefix	IPv6 address prefix - see Section 3.3.2 **Specified on Co-ordinator only**	-
u32Channel	2.4-GHz channel(s) to be used by the network or in a channel scan. Can be set in one of two ways: • Fixed integer value in the range 11-26, directly representing the 2.4-GHz channel to use. • Bitmap containing the set of channels to scan, where any or all of bits 11 to 26 are set to indicate the channels to scan, e.g. if bit 15 is set to '1', channel 15 is included in the scan.	11-26 0x00000800 - 0x07FFF800
u16PanId	Desired 16-bit PAN ID to use or search for - note that the Co-ordinator may select an alternative value for the network. The special value 0xFFFF indicates that the Co-ordinator must generate a PAN ID which does not clash with that of neighbouring networks or that an End Device/Router can join any detected network.	0-0xFFFF (0xFFFF indicates Co-ordinator must generate PAN ID)
u16MaxIpPacketSize	Maximum IP-packet size, in bytes. Should be set to: maximum payload size + 40.	0 and 256-1280 (0 gives default value of 1280)
u16NumPacketBuffers	Number of individual packet buffers in buffer area. This value is set by the stack and the value can be read back by the application after the call to **eJIP_Init()** returns.	-
u8UdpSockets	Maximum number of UDP sockets on node. At least one socket is required for JIP.	-
eDeviceType	Device type, one of: • Co-ordinator • Router • End Device	E_JIP_COORDINATOR E_JIP_ROUTER E_JIP_END_DEVICE
u32RoutingTableEntries	Number of entries in Routing table. Should be set according to the maximum number of nodes in the network, in order to limit memory space required for the table. **Not applicable to End Devices**	25-1000
u32DeviceId	32-bit Device ID - see Appendix D.1.	-
u8UniqueWatchers	Maximum number of remote nodes that can simultaneously trap MIB variables on the local node (this is the maximum for the node, not per variable).	-
u8MaxTraps	Maximum number of traps that can be simultaneously registered for MIB variables on the local node (this is the maximum for the node, not per variable).	-

`u8QueueLength`	Maximum number of items in the JIP queue which contains JIP communications that will lead to packets in the transmit buffers. Examples of these communications are trap notifications and command responses. A single item in this queue may yield more than one outgoing packet. For example, a trap notification for a variable only adds one item to the queue for all traps registered for this variable, but may lead to multiple outgoing packets. Therefore, this parameter can have a smaller value than `u8UniqueWatchers` and `u8MaxTraps`.	-
`u8MaxNameLength`	Maximum length (in bytes) of the DescriptiveName variable in the Node MIB.	-
`u16Port`	Number of UDP port on which to receive JIP packets. The default port number is 1873.	-
`u16JMP_Port`	Number of the UDP port on which to send and receive JenNet Management Protocol (JMP) packets. These packets are used for management messages between devices. The default port number is 1875.	-
`pcVersion`	Pointer to character string which provides the contents of the Version variable in the Node MIB.	-

8.1.2 tsNwkInfo

This structure contains details of the wireless network to which the local node belongs and the node’s place in the network.

```
typedef struct
{
    MAC_ExtAddr_s   sLocalAddr;
    MAC_ExtAddr_s   sParentAddr;
    uint16          u16Depth;
    uint16          u16PanID;
    uint8           u8Channel;
    uint8          *pau8ExtData;
}tsNwkInfo;
```

where:

- `sLocalAddr` is a `MAC_ExtAddr_s` structure containing the local node's IEEE/MAC address (see Section 8.1.3)
- `sParentAddr` is a `MAC_ExtAddr_s` structure containing the parent node's IEEE/MAC address (see Section 8.1.3)
- `u16Depth` is the depth of the local node within the network tree (number of hops from the Co-ordinator)
- `u16PanID` is the PAN ID of the network to which the local node belongs
- `u8Channel` is the number of the 2.4-GHz radio channel used

- `pau8ExtData` is a pointer to the user data passed in the Establish Route message (may be NULL)

8.1.3 MAC_ExtAddr_s

This structure stores the 64-bit IEEE/MAC address of a node as two 32-bit words.

```
typedef struct
{
   uint32 u32L;
   uint32 u32H;
} MAC_ExtAddr_s;
```

where:

- `u32L` contains the least-significant 32-bit word of the IEEE/MAC address
- `u32H` contains the most-significant 32-bit word of the IEEE/MAC address

8.1.4 ts6LP_SockAddr

This structure contains address and port information, and is typically used to specify the address of a node and the number of the port on which JIP runs on the node.

```
typedef struct
{
    int       sin6_family;
    in_port_t sin6_port;
    uint32    sin6_flowinfo;
    in6_addr  sin6_addr;
    uint32    sin6_scope_id;
} ts6LP_SockAddr;
```

where:

- `sin6_family` is the ‘address family' - should always be set to E_6LP_PF_INET6
- `sin6_port` is the IPv6 port number
- `sin6_flowinfo` contains flow information - this is not used by the stack and the value set is unimportant
- `sin6_addr` is the IPv6 address
- `sin6_scope_id` contains scope information - this is not used by the stack and the value set is unimportant

8.1.5 tsJIP_StackGroupChange

This structure is passed with the E_STACK_GROUP_CHANGE event, when the set of multicast addresses registered on a node has been modified (added to or removed from).

```
typedef struct
{
    MAC_ExtAddr_s  sDeviceAddress;
    in6_addr      *psAddressList;
    uint8          u8AddressListLen;
    bool_t         bAddNotRemove;
} tsJIP_StackGroupChange;
```

where:

- `sDeviceAddress` is a `MAC_ExtAddr_s` structure (see Section 8.1.3) containing the IEEE/MAC address of device on which the change has occured (it is set to zero if the change occured on the local device)
- `psAddressList` is the list of multicast addresses in the network
- `u8AddressListLen` is the number of entries in the list of multicast addresses
- `bAddNotRemove` is a boolean indicating whether an address has been added or removed:
    - TRUE - address(es) added
    - FALSE - address(es) removed

8.1.6 tsJIP_MibDef

This structure defines a MIB.

```
typedef const struct {
      uint32                  u32MibID;
      uint8                   u8Variables;
      tsJIP_VarDef           asVariables[];
} PACK tsJIP_MibDef;
```

where:

- `u32MibID` is the 32-bit identifier for the MIB type
- `u8Variables` is the number of variables in the MIB
- `asVariables[]` is the array of the variables in the MIB

8.1.7 tsJIP_VarDef

This structure defines a MIB variable.

```
typedef const struct {
      const char              *pcName;
      uint8                    u8VarID;
      uint8                    u8Flags;
      teJIP_VarType           eVarType;
      teJIP_Access            eAccess;
      uint8                    u8CacheHint;
      teJIP_Security          eSecurity;
} PACK tsJIP_VarDef;
```

where:

- `pcName` is a pointer to a character string containing the name of the variable
- `u8VarID` is an 8-bit identifier for the variable within the MIB
- `u8Flags` is a flags bitmap (reserved for future use)
- `eVarType` is the data type of the variable (see Section 8.2.2)
- `eAccess` is a bitmap which indicates the permitted types of access to the variable (see Section 8.2.3)
- `u8CacheHint` indicates the duration for which the variable value should be cached - this is set using one of the CACHE macros described on page 129
- `eSecurity` indicates the type of security applied to the variable (see Section 8.2.6)

8.1.8 tsJIP_MibInst

This structure defines a MIB (an instance of a MIB type).

```
typedef struct {
    thJIP_Mib         psNext;
    char              *pcName;
    tsJIP_MibDef     *psDef;
    uint8              u8Index;
    tsJIP_VarDynamic  asFuncs[];
} PACK tsJIP_MibInst;
```

where:

- `psNext` is a pointer for internal use only
- `pcName` is a pointer to a character string containing the name of the MIB
- `u8Index` is an 8-bit index value for the MIB on the node

- `asFuncs[]` is an array of pointers to callback functions for writing and reading the MIB variables, where these functions are specified in the macro **JIP_CALLBACK()** - this array is automatically populated by the macro

8.1.9 tsJIP_QueryMibResponse

This structure contains the results of a query on a MIB (yielding the MIB ID, MIB index and name).

```
typedef struct
{
    uint32          u32MibID;
    uint8           u8MibIndex;
    const char     *pcName;
} tsJIP_QueryMibResponse;
```

where:

- `u32MibID` is the MIB ID (indicating the MIB type)
- `u8MibIndex` is the MIB index value (on the node)
- `pcName` is a pointer to a character string containing the name of the MIB

8.1.10 tsJIP_QueryVarResponse

This structure contains details of a MIB variable.

```
typedef struct
{
   uint8               u8VarIndex;
   const char         *pcName;
   teJIP_VarType       eVarType;
   teJIP_AccessType   eAccessType;
   teJIP_Security      eSecurity;
} tsJIP_QueryVarResponse;
```

where:

- `u8VarIndex` is the index number of the variable within the MIB
- `pcName` is a pointer to a character string representing the name of the variable
- `eVarType` is a value indicating the data type of the variable (enumerations are provided in Section 8.2.2)
- `eAccessType` is a value indicating the types of access to the variable that are permitted (enumerations are provided in Section 8.2.4)
- `eSecurity` is a value indicating the level of JIP security applied to the variable (enumerations are provided in Section 8.2.6)

8.1.11 prSet

For each MIB variable type (see Section 8.2.2), a user-defined callback function must be provided which can be used to set the value of the variable. A pointer to this ‘Set’ callback function is contained in an internal structure for the variable type, along with a pointer to the equivalent ‘Get’ callback function (see Section 8.1.12).

The ‘Set’ callback function has the following prototype, according to the variable type:

Variable Type	‘Set’ Callback Function Prototype
E_JIP_VAR_TYPE_INT8	**teJIP_Status (*prSet)(int8** *i8Val*, **void ***pvCbData***);**
E_JIP_VAR_TYPE_INT16	**teJIP_Status (*prSet)(int16** *i16Val*, **void ***pvCbData)***;**
E_JIP_VAR_TYPE_INT32	**teJIP_Status (*prSet)(int32** *i32Val*, **void ***pvCbData***);**
E_JIP_VAR_TYPE_INT64	**teJIP_Status (*prSet)(int64** *i64Val,* **void ***pvCbData***);**
E_JIP_VAR_TYPE_UINT8	**teJIP_Status (*prSet)(uint8** *u8Val*, **void ***pvCbData***);**
E_JIP_VAR_TYPE_UINT16	**teJIP_Status (*prSet)(uint16** *u16Val*, **void ***pvCbData***);**
E_JIP_VAR_TYPE_UINT32	**teJIP_Status (*prSet)(uint32** *u32Val*, **void ***pvCbData***);**
E_JIP_VAR_TYPE_UINT64	**teJIP_Status (*prSet)(uint64** *u64Val*, **void ***pvCbData***);**
E_JIP_VAR_TYPE_FLOAT	**teJIP_Status (*prSet)(float** *fVal*, **void ***pvCbData***);**
E_JIP_VAR_TYPE_DOUBLE	**teJIP_Status (*prSet)(double** *dVal*, **void ***pvCbData***);**
E_JIP_VAR_TYPE_STRING	**teJIP_Status (*prSet)(const char ***pcVal*, **uint8** *u8Len*, **void ***pvCbData***);**
E_JIP_VAR_TYPE_BLOB	**teJIP_Status (*prSet)(const uint8 ***pu8Val*, **uint8** *u8Len*, **void ****pvCbData***);**
E_JIP_VAR_TYPE_ TABLE_BLOB	**teJIP_Status (*prSet)(const uint8 ***pu8Val*, **uint8** *u8Len*, **void ****pvCbData*, **uint16** *u16Entry***);**

Table 4: ‘Set’ Variable Callback Function Prototypes

In the above functions:

- The value to be written to the variable is specified in the parameter *i8Val*, *i16Val*, *i32Val*, *i64Val*, *u8Val*, *u16Val*, *u32Val*, *u64Val*, *fVal*, or *dVal*, or in the location pointed to by *pcVal* or *pu8Val*, as appropriate.
- Custom user data (not used by JenNet-IP) can be specified in the location pointed to by *pvCbData* (if there is no data, a null pointer is specified).
- For a string or blob, the length of the string/blob is specified through the parameter *u8Len*.
- For a table blob, the table entry number of the variable to be set is specified through the parameter *u16Entry*.

8.1.12 prGet

For each MIB variable type (see Section 8.2.2), a user-defined callback function must be provided which can be used to set the value of the variable. A pointer to this 'Get' callback function is contained in an internal structure for the variable type, along with a pointer to the equivalent 'Set' callback function (see Section 8.1.11).

The 'Get' callback function has the following prototype, according to the variable type:

Variable Type	'Get' Callback Function Prototype
E_JIP_VAR_TYPE_INT8	**void (*prGet)(thJIP_Packet** *hPacket*, **void** **pvCbData***);**
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	**void (*prGet)(thJIP_Packet** *hPacket*, **void** **pvCbData*, **tsJIP_TableData** **psTableData***);**

Table 5: 'Get' Variable Callback Function Prototypes

In the above functions:

- The handle of the packet in which the obtained value will be inserted is specified through the parameter *hPacket* (this handle is internal and transparent to the application).
- Custom user data (not used by JenNet-IP) can be specified in the location pointed to by *pvCbData*.
- For a table blob, information on the table is passed to the callback function in the structure pointed to by *psTableData* (this structure is described in Section 8.1.13).

8.1.13 tsJIP_TableData

This structure contains information on a table variable and is used when a request is received to read the variable.

```
typedef struct
{
    uint16  u16FirstEntry;
    uint8   u8EntryCount;
    uint16  u16RemainingEntries;
    uint16  u16TableVersion;
} tsJIP_TableData;
```

where:

- `u16FirstEntry` is the index of the first requested entry (populated by JIP)
- `u8EntryCount` is the number of entries requested (populated by JIP)
- `u16RemainingEntries` is the number of entries in the table after the last entry read and added to the response packet (populated by application)
- `u16TableVersion` is the table version, used to detect inconsistencies between multiple ‘table read’ response packets (populated by application)

8.1.14 tsAssocNodeInfo

This structure contains information about a node that is joining the network.

```
typedef struct
{
    MAC_ExtAddr_s sMacAddr;
    uint32        u32DeviceClass;
    uint16        u16NetworkAddr;
} tsAssocNodeInfo;
```

where:

- `sMacAddr` contains the 64-bit IEEE/MAC address of the joining node
- `u32DeviceClass` is the Device ID of the joining node
- `u16NetworkAddr` is the 16-bit network address of the joining node (will always be 0xFFFE in JenNet-IP)

8.1.15 EUI64_s

This structure stores a 64-bit value as two 32-bit words and can be used to hold any 64-bit value.

```
struct EUI64_s_Tag
{
    uint32    u32L;
    uint32    u32H;
} EUI64_s;
```

where:

- u32L is the least-significant 32-bit word of the 64-bit value
- u32H is the most-significant 32-bit word of the 64-bit value

8.1.16 in6_addr

This structure contains a uniion that can be used to store a 128-bit IPv6 address as an array with one of the following sizes/formats:

- sixteen 8-bit elements
- eight 16-bit elements
- four 32-bit elements

```
typedef struct
{
    union {
        uint8  u6_addr8[16];
        uint16 u6_addr16[8];
        uint32 u6_addr32[4];
    } in6_u;
    #define s6_addr     in6_u.u6_addr8
    #define s6_addr16   in6_u.u6_addr16
    #define s6_addr32   in6_u.u6_addr32
} in6_addr;
```

where:

- u6_addr8[16] is the address as sixteen 8-bit values
- u6_addr16[8] is the address as eight 16-bit values
- u6_addr32[4] is the address as four 32-bit values

The same IPv6 address storage format should be used throughout an application, as otherwise endianness will become an issue.

8.1.17 tsStackReset

This structure contains information which relates to the cause of a stack reset.

```
typedef struct
{
    uint8 u8StackCurrentState;
    uint8 u8StackResumeState;
    uint8 u8LostPacketCount;
    uint8 u8UnackPings;
    uint8 u8EstablishRouteAttempts;
} tsStackReset;
```

where:

- `u8StackCurrentState` is the JenNet state when the reset was initiated (for internal stack use only)
- `u8StackResumeState` is the JenNet state after the reset has completed (for internal stack use only)
- `u8LostPacketCount` is the number of failed packet transmissions (only valid on an End Device)
- `u8UnackPings` is the number of unacknowledged ping requests
- `u8EstablishRouteAttempts` is the number of route establishments attempted

8.2 Enumerations

8.2.1 teJIP_Device

The wireless network device type (Co-ordinator, Router or End Device) is specified in the stack initialisation data (`tsJIP_InitData`) using the enumerations below.

```
typedef enum PACK
{
    E_JIP_DEVICE_COORDINATOR = E_6LP_COORDINATOR,
    E_JIP_DEVICE_ROUTER      = E_6LP_ROUTER,
    E_JIP_DEVICE_END_DEVICE  = E_6LP_END_DEVICE
} teJIP_Device;
```

8.2.2 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;
```

8.2.3 teJIP_Access

The following list defines the types of access available for a MIB variable (a MIB variable can have more than one access type by combining two or more of these enumerations in a logical-OR operation).

```
typedef enum
{
    E_JIP_ACCESS_TYPE_READ              = 0x01,
    E_JIP_ACCESS_TYPE_WRITE             = 0x02,
    E_JIP_ACCESS_TYPE_TRAP              = 0x04,
} PACK teJIP_Access;
```

The above enumerations are detailed in the table below.

Access Type	Description
E_JIP_ACCESS_TYPE_READ	Variable can be read remotely
E_JIP_ACCESS_TYPE_WRITE	Variable can be set remotely
E_JIP_ACCESS_TYPE_TRAP	Variable can be trapped

8.2.4 teJIP_AccessType

The following enumerations are used to specify the accessibility 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.

Access Type	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

8.2.5 teJIP_PollResponse

```
typedef enum PACK
{
   E_JIP_POLL_NO_DATA = 0,
   E_JIP_POLL_DATA_READY,
   E_JIP_POLL_TIMEOUT,
   E_JIP_POLL_ERROR,
   E_JIP_POLL_PENDING
} teJIP_PollResponse;
```

The above enumerations are detailed in the table below.

Enumeration	Description
E_JIP_POLL_NO_DATA	Poll complete but no data pending
E_JIP_POLL_DATA_READY	Poll complete and data received
E_JIP_POLL_TIMEOUT	Poll timed-out, no data received
E_JIP_POLL_ERROR	Problem with request
E_JIP_POLL_PENDING	Request accepted but not complete yet

8.2.6 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;
```

8.2.7 teLowEnergyStatus

The following enumerations are used to instruct Routers in the maintenance of the local list of registered low-energy devices.

```
typedef enum PACK
{
    E_LEF_ADD,
    E_LEF_DELETE
} teLowEnergyStatus;
```

The above enumerations are detailed in the table below.

Enumeration	Description
E_LEF_ADD	Add a device to the local list of registered low-energy devices
E_LEF_DELETE	Remove a device from the local list of registered low-energy devices

8.3 Events

This section details events that the application may need to handle on a wireless node of a JenNet-IP system - stack and data events are detailed here, while peripheral events are covered in the *JN516x Integrated Peripherals API User Guide (JN-UG-3087)*.

8.3.1 teJIP_StackEvent

The stack events that are passed up to the **vJIP_StackEvent()** callback function are listed and described below.

```
typedef enum
{
    E_STACK_STARTED,
    E_STACK_JOINED,
    E_STACK_NODE_JOINED,
    E_STACK_NODE_LEFT,
    E_STACK_TABLES_RESET,
    E_STACK_RESET,
    E_STACK_POLL,
    E_STACK_NODE_JOINED_NWK,
    E_STACK_NODE_LEFT_NWK,
    E_STACK_NODE_AUTHORISE,
    E_STACK_ROUTE_CHANGE,
    E_STACK_GROUP_CHANGE
} teJIP_StackEvent;
```

The enumerations in the above structure are described in the table below.

Stack Event	Description
E_STACK_STARTED	Co-ordinator has started (see tsNwkInfo in Section 8.1.2)
E_STACK_JOINED	This node has joined the network (see tsNwkInfo in Section 8.1.2)
E_STACK_NODE_JOINED	A device has joined this parent (see tsAssocNodeInfo in Section 8.1.14)
E_STACK_NODE_LEFT	A device has left this parent (see tsAssocNodeInfo in Section 8.1.14)
E_STACK_TABLES_RESET	Tables have been reset (e.g. Neighbour table on a Router or the Co-ordinator). Used as part of initialisation on all node types (including End Devices which have no tables) and is generated just before the stack starts
E_STACK_RESET	Stack has been reset (see tsStackReset in Section 8.1.17)
E_STACK_POLL	A deferred poll response has arrived (see teJIP_PollResponse in Section 8.2.5)
E_STACK_NODE_JOINED_NWK	A device has joined the network (only seen at Co-ordinator) (see tsAssocNodeInfo in Section 8.1.14)
E_STACK_NODE_LEFT_NWK	A device has left the network (only seen at Co-ordinator) (see tsAssocNodeInfo in Section 8.1.14)
E_STACK_NODE_AUTHORISE	The device with the specified address is trying to join but its commissioning key must be obtained from the Border-Router (see MAC_ExtAddr_s in Section 8.1.3)
E_STACK_ROUTE_CHANGE	A device has moved (only seen at Co-ordinator) (see tsAssocNodeInfo in Section 8.1.14)
E_STACK_GROUP_CHANGE	One or more multicast addresses have been added or removed from network (see tsJIP_StackGroupChange in Section 8.1.5). The event is generated on the Co-ordinator for all changes within the wireless network and is also triggered on the node where a modification is made.

8.3.2 teJIP_DataEvent

The data events that are passed up to the **v6LP_DataEvent()** callback function are listed and described below.

```
typedef enum
{
   E_DATA_SENT,
   E_DATA_SEND_FAILED,
   E_DATA_RECEIVED,
   E_IP_DATA_RECEIVED,
   E_6LP_ICMP_MESSAGE
} teJIP_DataEvent;
```

The enumerations in the above structure are described in the table below.

Data Event	Description
E_DATA_SENT	Packet sent successfully
E_DATA_SEND_FAILED	Packet send failed (see **u32JIP_GetErrNo()** function)
E_DATA_RECEIVED	Packet received
E_IP_DATA_RECEIVED	Packet received at IP layer
E_6LP_ICMP_MESSAGE	ICMP message passed up to the application

8.4 Return Codes

8.4.1 teJIP_Status

The following status responses are returned by the JIP functions.

```
typedef enum
{
   E_JIP_OK                         = 0x00,
   E_JIP_ERROR_TIMEOUT              = 0x7f,
   E_JIP_ERROR_BAD_MIB_INDEX        = 0x8f,
   E_JIP_ERROR_BAD_VAR_INDEX        = 0x9f,
   E_JIP_ERROR_NO_ACCESS            = 0xaf,
   E_JIP_ERROR_BAD_BUFFER_SIZE      = 0xbf,
   E_JIP_ERROR_WRONG_TYPE           = 0xcf,
   E_JIP_ERROR_BAD_VALUE            = 0xdf,
   E_JIP_ERROR_DISABLED             = 0xef,
   E_JIP_ERROR_FAILED               = 0xff
} PACK teJIP_Status;
```

The above enumerations are detailed in the table below.

Enumeration	Description
E_JIP_OK	Operation was successful
E_JIP_ERROR_TIMEOUT	Not applicable (not used by the stack)
E_JIP_ERROR_BAD_MIB_INDEX	Specified MIB index value not found in registered MIBs
E_JIP_ERROR_BAD_VAR_INDEX	Variable with specified index not found in MIB
E_JIP_ERROR_NO_ACCESS	Attempt to set a variable for which write access is not permitted
E_JIP_ERROR_BAD_BUFFER_SIZE	Requested operation would have exceeded available buffer space
E_JIP_ERROR_WRONG_TYPE	Variable type in requested set operation did not match the type of the variable itself
E_JIP_ERROR_BAD_VALUE	Value in requested set operation was not appropriate (e.g. out of range)
E_JIP_ERROR_DISABLED	Attempt has been made to access a variable that is disabled
E_JIP_ERROR_FAILED	Non-specific failure

8.4.2 teJenNetStatusCode

This structure contains JenNet status codes.

```
typedef enum
{
    E_JENNET_SUCCESS,
    E_JENNET_DEFERRED,
    E_JENNET_ERROR
} teJenNetStatusCode;
```

The above enumerations are detailed in the table below.

Enumeration	Description
E_JENNET_SUCCESS	Operation completed successfully
E_JENNET_DEFERRED	Operation is on-going (e.g. packet transmission)
E_JENNET_ERROR	Operation failed

8.5 Error Codes and Enumerations

This section details the error enumerations returned by the function **u32JIP_GetErrNo()**. This function returns a 32-bit value, in which bits 7-0 represent an error code and bits 15-8 represent further error information (bits 31-16 are reserved).

8.5.1 te6LP_ErrorCode

An error code appears as bits 7-0 of the value returned by **u32JIP_GetErrNo()**.

```
typedef enum PACK
{
    E_6LP_NETWORK_FORMATION                 // 0x01
    E_6LP_6LP_BUILD,                        // 0x02
    E_6LP_SEND_FAIL,                        // 0x03
    E_6LP_ICMP_BUILD,                       // 0x04
    E_6LP_COMPRESS_FAIL,                    // 0x05
    E_6LP_CHECKSUM_FAIL,                    // 0x06
    E_6LP_TRANSMIT_FAIL,                    // 0x07
    E_6LP_BITSTREAM_FAIL,                   // 0x08
    E_6LP_IPV6_BUFFER_FAIL,                 // 0x09
    E_6LP_IPV6_HEADER_BUILD,                // 0x0a
    E_6LP_IPV6_HEADER_INFO,                 // 0x0b
    E_6LP_PING_FAIL,                        // 0x0c
    E_6LP_RX_FAIL,                          // 0x0d
    E_6LP_RX_DEFRAG_TIMER_FAIL,             // 0x0e
    E_6LP_SOCKET_FAIL,                      // 0x0f
    E_6LP_SOCKET_BIND_FAIL,                 // 0x10
    E_6LP_SOCKET_SENDTO_FAIL,               // 0x11
    E_6LP_SOCKET_GET_NEXT_PACKET_FAIL,      // 0x12
    E_6LP_SOCKET_IP_SENDTO_FAIL,            // 0x13
    E_6LP_SOCKET_RECV_FROM_FAIL,            // 0x14
    E_6LP_SOCKET_IP_RECV_FROM_FAIL,         // 0x15
    E_6LP_SOCKET_CLOSE,                     // 0x16
    E_6LP_SOCKET_COMPRESSION_OPT_FAIL,      // 0x17
    E_6LP_SOCKET_SET_NEXT_HEADER_FAIL,      // 0x18
    E_6LP_SOCKET_EVT_HANDLER,               // 0x19
    E_6LP_IP_SENDTO_FAIL,                   // 0x1a
    E_6LP_IP_RECV_FROM_FAIL,                // 0x1b
} te6LP_ErrorCode;
```

The enumerations in the above structure are described in the table below.

Error Code (bits 7-0 of returned value)	**Description**
E_6LP_NETWORK_FORMATION	Network formation error
E_6LP_6LP_BUILD	6LoWPAN stack creation error
E_6LP_SEND_FAIL	Attempt to send IPv6 packet has failed
E_6LP_ICMP_BUILD	ICMP stack module error
E_6LP_COMPRESS_FAIL	6LoWPAN compression failed
E_6LP_CHECKSUM_FAIL	Failure in 6LoWPAN checksum generation
E_6LP_TRANSMIT_FAIL	Attempt to transmit 802.15.4 frame has failed
E_6LP_BITSTREAM_FAIL	6LoWPAN bitstream module error
E_6LP_IPV6_BUFFER_FAIL	IPv6 buffer error
E_6LP_IPV6_HEADER_BUILD	IPv6 header creation error
E_6LP_IPV6_HEADER_INFO	IPv6 header access function error
E_6LP_PING_FAIL	Reserved
E_6LP_RX_FAIL	IPv6 receive module error
E_6LP_RX_DEFRAG_TIMER_FAIL	Receive defragmentation timer error
E_6LP_SOCKET_FAIL	Socket creation failure
E_6LP_SOCKET_BIND_FAIL	Socket binding failure
E_6LP_SOCKET_SENDTO_FAIL	‘Send to’ function call has failed
E_6LP_SOCKET_GET_NEXT_PACKET_FAIL	No next packet in the system for specified socket
E_6LP_SOCKET_IP_SENDTO_FAIL	IP-layer ‘Send to’ function call has failed
E_6LP_SOCKET_RECV_FROM_FAIL	‘Receive from’ function call has failed
E_6LP_SOCKET_IP_RECV_FROM_FAIL	IP-layer ‘Receive from’ function call has failed
E_6LP_SOCKET_CLOSE	Socket close error
E_6LP_SOCKET_COMPRESSION_OPT_FAIL	Reserved for future use
E_6LP_SOCKET_SET_NEXT_HEADER_FAIL	‘Set next header’ function call has failed
E_6LP_SOCKET_EVT_HANDLER	Socket event handler error
E_6LP_IP_SENDTO_FAIL	Attempt by IP layer to send IPv6 packet failed
E_6LP_IP_RECV_FROM_FAIL	Attempt by IP layer to receive IPv6 packet failed

8.5.2 te6LP_ErrorInfo

Error information appears as bits 15-8 of the value returned by **u32JIP_GetErrNo()**.

```
typedef enum PACK
{
    I_6LP_NONE  = 0,
    I_6LP_PARAM,                            // 0x01
    I_6LP_START_FAIL,                       // 0x02
    I_6LP_MTU_ERROR,                        // 0x03
    I_6LP_BUFFER_UNKNOWN,                   // 0x04
    I_6LP_DEVICE_UNKNOWN,                   // 0x05
    I_6LP_SOCKET_PROTOCOL_UNSUPPORTED,      // 0x06
    I_6LP_PACKET_TOO_LARGE,                 // 0x07
    I_6LP_NO_TCP_HANDLER,                   // 0x08
    I_6LP_TCP_ERROR,                        // 0x09
    I_6LP_NEXT_HEADER_UNSUPPORTED,          // 0x0a
    I_6LP_BUFFER_TOO_SMALL,                 // 0x0b
    I_6LP_BUFFER_RECOVERY,                  // 0x0c
    I_6LP_BUFFER_NONE,                      // 0x0d
    I_6LP_PING_ID_ERROR,                    // 0x0e
    I_6LP_PING_TIMER_FAIL,                  // 0x0f
    I_6LP_PING_NO_FREE_SERVER_INSTANCES,    // 0x10
    I_6LP_PING_NO_FREE_BUFFERS,             // 0x11
    I_6LP_PING_SEND_PING_FAILED,            // 0x12
    I_6LP_FRAME_FAIL,                       // 0x13
    I_6LP_MESH_CB_FAIL,                     // 0x14
    I_6LP_BCAST_CB_FAIL,                    // 0x15
    I_6LP_IPV6_HEADER_FAIL,                 // 0x16
    I_6LP_6LP_FRAGMENT_FAIL,                // 0x17
    I_6LP_SOCKETS_NONAVAILABLE,             // 0x18
    I_6LP_SOCKET_NOT_FOUND,                 // 0x19
    I_6LP_SOCKET_RANGE,                     // 0x1a
    I_6LP_SOCKET_FAMILY,                    // 0x1b
    I_6LP_DEST_PREFIX_MISMATCH,             // 0x1c
    I_6LP_NO_COMP_OPTIONS,                  // 0x1d
} te6LP_ErrorInfo;
```

The enumerations in the above structure are described in the table below.

Error Information (bits 15-8 of returned value)	**Description**
I_6LP_NONE	No error
I_6LP_PARAM	Function parameter error (user error)
I_6LP_START_FAIL	6LoWPAN network has failed to start
I_6LP_MTU_ERROR	Size of MTU out of valid range
I_6LP_BUFFER_UNKNOWN	System buffer does not exist
I_6LP_DEVICE_UNKNOWN	Device type unrecognised on network creation
I_6LP_SOCKET_PROTOCOL_UNSUPPORTED	Socket has been configured to unsupported protocol
I_6LP_PACKET_TOO_LARGE	Data packet too large for system to handle
I_6LP_NO_TCP_HANDLER	TCP protocol plug-in is not present
I_6LP_TCP_ERROR	TCP protocol error
I_6LP_NEXT_HEADER_UNSUPPORTED	Unrecognised IPv6 next header
I_6LP_BUFFER_TOO_SMALL	User data buffer too small
I_6LP_BUFFER_RECOVERY	Buffer recovery failed after transmit buffers exhausted
I_6LP_BUFFER_NONE	No system buffers available
I_6LP_PING_ID_ERROR	Reserved
I_6LP_PING_TIMER_FAIL	Reserved
I_6LP_NO_FREE_SERVER_INSTANCES	Reserved
I_6LP_NO_FREE_BUFFERS	Reserved
I_6LP_PING_SEND_PING_FAILED	Reserved
I_6LP_FRAME_FAIL	Received 802.15.4 frame is invalid
I_6LP_MESH_CB_FAIL	No mesh header callback installed
I_6LP_BCAST_CB_FAIL	No broadcast header callback installed
I_6LP_IPV6_HEADER_FAIL	Received IPv6 header is corrupted
I_6LP_6LP_FRAGMENT_FAIL	Illegal 6LoWPAN fragment received
I_6LP_SOCKETS_NONAVAILABLE	No socket free on the device
I_6LP_SOCKET_NOT_FOUND	Specified socket ID does not exist on device
I_6LP_SOCKET_RANGE	Specified socket ID is out of range

I_6LP_SOCKET_FAMILY	Socket family value is not supported
I_6LP_DEST_PREFIX_MISMATCH	Reserved for future use
I_6LP_NO_COMP_OPTIONS	Reserved for future use

9. JenNet-IP Parameters

This chapter details the parameters that can be configured to determine the properties and behaviour of the wireless network. These parameters are divided into three categories:

- JenNet network configuration parameters (see Section 9.1)
- JenNet network profile parameters (see Section 9.2)
- Stack parameters (see Section 9.3)

9.1 JenNet Network Parameters (tsNetworkConfigData)

This section details the JenNet parameters that can be set on individual nodes using the **vJIP_ConfigureNetwork()** callback function (otherwise their default values will be used). Values should be specified only for those parameters that are to be changed from their default values. For advice on using some of these parameters, refer to Appendix A.

These parameters are contained in the following structure.

```
typedef struct
{
    void    *pvDoNotChange;
    uint16  u16PanID;
    uint8   u8Channel;
    uint32  u32ScanChannels;
    bool_t  bPurgeInactiveED;
    uint32  u32RoutePurgeInterval;
    uint32  u32RouteImportInterval;
    bool_t  bSleepDuringBackoff;
    uint8   u8EndDevicePingInterval;
    uint32  u32EndDeviceScanTimeout;
    uint32  u32EndDeviceScanSleep;
    uint32  u32EndDevicePollPeriod;
    uint32  u32EndDeviceActivityTimeout;
    uint32  u32RouterActivityTimeout;
    bool_t  bPermitExtNwkPkts;
    uint32  u32RoutingTableEntries;
    void    *pvRoutingTableSpace;
    uint8   u8InternalTimer;
    bool_t  bRecoveredFromJPDM;
    uint16  u16CommWindow;
} tsNetworkConfigData;
```

The elements of this structure are described in the table below (note that parameters tagged with @ are also included in the structure `tsJIP_InitData` described in Section 8.1.1).

JenNet Parameter	Description	Default Value	Range
`u16PanID @`	16-bit PAN ID to identify network (if no existing network with same PAN ID). **Co-ordinator only**	0xAAAA	0-0xFFFE
`u8Channel`	The 2.4-GHz channel to be used by the network or in a channel scan (see `u32ScanChannels` below). **Co-ordinator only**	0	0: Channel scan 11-26: Fixed channel
`u32ScanChannels @`	Bitmap (32 bits) of the set of channels to consider when performing a scan of the 2.4-GHz band for a suitable channel to use. The Co-ordinator will select the quietest channel from those available. Other node types will scan the possible channels to search for the network. A scan must have been enabled using `u8Channel` (=0).	0x07FFF800 (all channels)	0x00000800 - 0x07FFF800 (Bit 11 set ⇒ Ch 11, Bit 12 set ⇒ Ch 12,...)
`bPurgeInactiveED`	Enables automatic removal of an End Device child if the child has not exchanged data with the parent within the timeout period specified through `u32EndDeviceActivityTimeout`. **Co-ordinator and Routers only**	TRUE (Enabled)	TRUE: Enable FALSE: Disable
`u32RoutePurgeInterval`	Time interval between maintenance checks of two consecutive entries in a Routing table. If a route remains unused during two complete Routing table maintenance cycles, the validity of the route is checked. If the route is invalid, its entry is removed. Set in units of 100 ms. **Co-ordinator and Routers only**	10 (1 second)	10-100
`u32RouteImportInterval`	Time interval between consecutive route importation requests, used by the Co-ordinator to learn new routes from Routers. Set in units of 100 ms. **Co-ordinator only**	10 (1 second)	10-100
`bSleepDuringBackoff`	Enables sleep for an End Device during start-up. This helps avoid network congestion. **End Devices only**	FALSE (Disabled)	TRUE: Sleep FALSE: Do not sleep
`u8EndDevicePingInterval`	Number of sleep cycles between auto-pings generated by an End Device (to its parent). **End Devices only**	1	0-255 Zero value disables pings

Table 6: JenNet Parameters in Network Configuration

`u32EndDeviceScanTimeout`	Timeout after which device abandons a failed radio-channel scan.	5 seconds	Do not change from default value
`u32EndDeviceScanSleep`	Amount of time following a failed scan that an End Device waits before starting another scan. Set in milliseconds. **End Devices only**	10000 or 0x2710 (10 seconds)	1-0x07FFFFFF Values below 0x3E8 (1 second) are not recommended
`u32EndDevicePollPeriod`	Time between auto-poll data requests sent from an End Device (while awake) to its parent. Set in units of 10 ms. **End Devices only**	500 or 0x1F4 (5 seconds)	0-0xFFFFFFFF Setting this value to 0 means the device will not auto-poll
`u32EndDeviceActivityTimeout`	Timeout period for communication (excluding data polling) from an End Device child. If no message is received from the End Device within this period, the child is assumed lost and is removed from the Neighbour table (and Routing tables higher in the network). This timeout period must be greater than the period between consecutive pings. **Co-ordinator and Routers only**	600 (60 seconds)	0-0xFFFFFFFF Timeout is value set multiplied by 100 ms
`u32RouterActivityTimeout`	Timeout period for communication from a Router child. If no message is received from the Router within this period, the child is assumed lost and is removed from the Neighbour table (and Routing tables higher in the network). **Co-ordinator and Routers only**	`u16RouterPingPeriod x u8MaxFailedPkts`	0-0xFFFFFFFF Timeout is value set multiplied by 10 ms
`bPermitExtNwkPkts`	Allows packets from other 802.15.4-based wireless networks to be received.	FALSE (Disallowed)	TRUE: Allow FALSE: Disallow
`u32RoutingTableEntries @`	Number of elements in array used to store the Routing table. Should be set according to the maximum number of nodes in the network, in order to limit memory space required for the table. **Co-ordinator and Routers only**	-	25-1000
`*pvRoutingTableSpace`	Pointer to the location in memory at which the Routing table array is stored. The Routing table is an array of structures. Storage for the table is allocated by the stack. This pointer is provided in case the application needs to access the Routing table directly. **Co-ordinator and Routers only**	NULL	-

Table 6: JenNet Parameters in Network Configuration

`u8InternalTimer`	On-chip timer to be used by JenNet layer of stack. The application must not use this timer for any other purpose.	E_AHI_DEVICE_TICK_TIMER	E_AHI_DEVICE_TIMER0 or E_AHI_DEVICE_TIMER1 or E_AHI_DEVICE_TICK_TIMER
`bRecoveredFromJPDM`	Not supported in JenNet-IP	0	0
`u16CommWindow`	Amount of time for which a commissioning key is held by a node during the commissioning process (a joining node must perform route establishment within this time). The time is expressed in units of 100 ms.	600 (1 minute)	0-65535

Table 6: JenNet Parameters in Network Configuration

@ indicates parameter also included in structure `tsJIP_InitData` (see Section 8.1.1).

9.2 JenNet Network Profile Parameters (tsNwkProfile)

This section details the parameters that are set on the Co-ordinator for the whole wireless network and are inherited by other devices when they join the network. A given set of parameter values constitute a network profile, each profile having a unique index. Standard profiles are provided in the JenNet-IP software (see Table 8).

Note: JenNet network profiles are introduced in Section 3.7. Functions for setting and accessing the profile parameters are detailed in Section 5.3.

```
typedef struct
{
    uint8 u8MaxChildren;          /* Set in bJnc_SetRunProfile */
    uint8 u8MaxSleepingChildren;  /* Set in bJnc_SetRunProfile */
    uint8 u8MaxFailedPkts;        /* Set in bJnc_SetRunProfile */
    uint8 u8MaxBcastTTL;          /* Set in bJnc_SetRunProfile */
    uint16 u16RouterPingPeriod;   /* Set in bJnc_SetRunProfile */
    uint8 u8MinBeaconLQI;         /* Set in bJnc_SetJoinProfile */
    uint16 u16ScanBackOffMin;     /* Set in bJnc_SetJoinProfile */
    uint16 u16ScanBackOffMax;     /* Set in bJnc_SetJoinProfile */
    uint16 u16EstRtBackOffMin;    /* Set in bJnc_SetJoinProfile */
    uint16 u16EstRtBackOffMax;    /* Set in bJnc_SetJoinProfile */
}tsNwkProfile;
```

The elements of this structure are described in Table 7 below.

JenNet Parameter	Description	Default Value	Range
`u8MaxChildren`	Maximum number of children the node can have. **Co-ordinator and Routers only**	10	0-16
`u8MaxSleepingChildren`	Maximum number of children of a node that can be End Devices (nodes capable of sleeping). This value must be less than or equal to `u8MaxChildren`. The remaining child nodes are reserved exclusively for Routers, although any number of children can be Routers. **Co-ordinator and Routers only**	8	0-`u8MaxChildren`
`u8MaxFailedPkts`	Number of missed communications (MAC acknowledgments) before parent or child considered to be lost.	5	0-255 Zero value disables the feature

Table 7: JenNet Parameters in Network Profile

`u8MaxBcastTTL`	The maximum number of hops that a broadcast message can make.	5	1-255
`u16RouterPingPeriod`	Time between auto-pings generated by a Router (to its parent). Set in units of 10 ms. The same value should be set in all routing nodes in the network. **Co-ordinator and Routers only**	500 (5 seconds)	0-65535 Zero value disables pings
`u8MinBeaconLQI`	Radio signal strength threshold below which beacons from potential parent will be rejected. **Routers and End Devices only**	0 (All accepted)	0-255
`u16ScanBackOffMin`	The minimum value of the random delay applied before scanning for a network. This time is expressed in units of 10 ms. **Routers and End Devices only**	100 (1 second)	0-65535
`u16ScanBackOffMax`	The maximum value of the random delay applied before scanning for a network. This time is expressed in units of 10 ms. **Routers and End Devices only**	1000 (10 seconds)	0-65535
`u16EstRtBackOffMin`	The minimum value of the random delay applied before requesting route establishment. This time is expressed in units of 10 ms. **Routers and End Devices only**	100 (1 second)	0-65535
`u16EstRtBackOffMax`	The maximum value of the random delay applied before requesting route establishment. This time is expressed in units of 10 ms. **Routers and End Devices only**	1000 (10 seconds)	0-65535

Table 7: JenNet Parameters in Network Profile

The above network profile parameters are divided into two categories:

- **Join parameters:** `u8MinBeaconLQI`, `u16ScanBackOffMin`, `u16ScanBackOffMax`, `u16EstRtBackOffMin`, `u16EstRtBackOffMax`
- **Run parameters:** `u8MaxChildren`, `u8MaxSleepingChildren`, `u8MaxFailedPkts`, `u8MaxBcastTTL`, `u16RouterPingPeriod`

It is possible to use the ‘join parameters’ of one profile with the ‘run parameters’ of another profile (set using the functions described in Section 5.3).

The standard network profiles provided in the JenNet-IP software are listed and detailed in Table 8 below.

Parameter/Property	Profile Index									
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
`u8MaxChildren`	10	16	10	16	10	16	10	16	3	3
`u8MaxSleepingChildren`	8	12	8	12	8	12	8	12	0	0
`u8MaxFailedPkts`	7	7	5	5	5	5	5	5	5	5
`u8MaxBcastTTL`	16	16	12	12	10	10	8	8	16	16
`u16RouterPingPeriod`	1500	1500	1000	1000	700	700	500	500	0	0
`u8MinBeaconLQI`	55	55	45	45	40	40	35	35	55	55
`u16ScanBackOffMin`	1 s	1 s	1 s	1 s	1 s	1 s	1 s	1 s	1 s	1 s
`u16ScanBackOffMax`	10 s	10 s	5 s	5 s	5 s	5 s	3 s	3 s	3 s	10 s
`u16EstRtBackOffMin`	1 s	1 s	1 s	1 s	1 s	1 s	1 s	1 s	1 s	1 s
`u16EstRtBackOffMax`	10 s	10 s	5 s	5 s	5 s	5 s	3 s	3 s	3 s	10 s
Network Size *	> 250	> 250	150 - 250	150 - 250	50 - 150	50 - 150	< 50	< 50	< 50	150 - 250
Tree Type	Sparse	Bushy	Sparse	Bushy	Sparse	Bushy	Sparse	Bushy	-	-

Table 8: Standard Network Profiles

* Network sizes indicated above are for guidance only and should not be assumed to be rigid

Note: Profiles 0-7 are intended for full JenNet-IP systems (with both WPAN and LAN/WAN domains) while profiles 8 and 9 are intended for standalone WPANs, described in Chapter 11. Profile 0 is the default.

9.3 Stack Parameters

This section details parameters that are pre-set in the stack but can be modified at run-time before initialising the stack. Some of these parameters do not appear in the header files and therefore the parameters should be explicitly declared before use. They are mostly concerned with performance tuning and should normally be left at their default values.

Caution: *All testing has been conducted with the default values for these parameters and changes may affect network performance and/or stability.*

The parameters are listed and detailed in the table below.

Parameter	Type	Values/Units	Description
b6LP_AlwaysBroadcast	bool_t	Boolean Default: FALSE	Indicates whether all transmissions within the wireless network will be treated as broadcasts: TRUE to transmit all packets as JenNet broad-casts, FALSE to transmit a unicast packet as a real unicast but a multicast packet as a JenNet broadcast. If set to TRUE, communication is still possible with an individual node
u8_6LP_GCastTimeout	uint8	Default: 60 secs Max: 255 secs	Time between MLD* transmissions
u8_6LP_GCastShortTimeout	uint8	Default: 5 secs Max: 255 secs	Time between failed MLD* transmission and retry
u8_6LP_GCastAddrStoreEntries	uint8	Default: 12 Max: 255	Number of addresses that can be stored and for-warded, for MLD*. Each address is 16 bytes long
u8SocketMaxGroupAddrs	uint8	Default: 8 Max: 255	Total number of multicast addresses that can be registered to sockets. Each address is 16 bytes long.
u8_6LPQSize	uint8	Default: 8 Max: 255	Size of internal queue for non-timer events between lower stack layers and JIP layer. Increase if function **v_6LP_Tick()** is called infre-quently. Each entry is 140 bytes long
u8_6LPTimerQSize	uint8	Default: 8 Max: 255	Size of internal queue for timer events between lower stack layers and JIP layer. Increase if function **v_6LP_Tick()** is called infrequently. Each entry is 8 bytes long

Table 9: Stack Parameters

Parameter	Type	Values/Units	Description
u16ClocksPerTick	unint16	Default: 0 Max: 65535	Configures throttling mechanism for **v_6LP_Tick()**. If left at 0, the 6LPQ and 6LPTimerQ queues are processed every time **v_6LP_Tick()** is called. Otherwise, they are only processed when u16ClocksPerTick symbol clock periods have passed since they were processed. This allows **v_6LP_Tick()** to be called frequently whilst limiting how often it does anything. This improves reliability of broadcast and fragment transmissions. Unlike the other parameters in this table, this value can be altered at any time
u8OND_SectorsAvailable	uint8	Default: 4 Max: 255	Number of NVM sectors available for the storage of application images. Needed for OND only
u8OND_SectorSize	uint8	Default: 64 KB Max: 255 KB	Size of each NVM sector (this value is dictated by the NVM device used). Needed for OND only
u8OND_SrvMaxServers	uint8	Default: 2 Max: 16	Number of entries in the list of OND servers held on the Co-ordinator
gMAC_u8MaxBuffers	uint8	Default: 4 or 5 Max: 255	**MAC:** Number of transmit and receive buffers (one of the buffers is reserved for receive). The default value is 5 **MiniMAC:** Number of buffers available for queued frames at the MAC layer, waiting to be transmitted. The default value is 4
u8JNT_IndirectTxBuffers	uint8	Default: 3 Max: 16	Used on a Router to specify the number of MAC buffers that can be used to queue received frames for sleeping End Devices. It is advised that this variable is set before starting the stack and that the value is lower than *gMAC_u8MaxBuffers* (since these frames may be queued for a long time, it is not desirable for all MAC buffers to be used for this as it limits the throughput of messages between Routers)
u8_6LP_SpeculativeBroadcasts PerSec	uint8	Default: 0 Max: 255	Used on the Co-ordinator to put a limit on the number of ‘speculative broadcasts’ per second. If a message arrives at the Co-ordinator for a device without a route (possibly because the device has left and rejoined the network), the Co-ordinator can send a speculative broadcast to the device in case it is within radio range. This limit is intended to prevent network flooding. If set to zero, speculative broadcasts are disabled
bLastPktDirect	bool_t	Boolean	Indicates whether a received packet came directly from the originator (TRUE) or came via one or more hops (FALSE). The parameter can be read by the application in a callback function invoked by the stack and is only valid for the duration of the callback function execution.

Table 9: Stack Parameters

Parameter	Type	Values/Units	Description
u8LastPktLqi	uint8	Min: 0 Max: 255	Indicates the link quality (LQI value) of received over-air data. The valid range is 0 (poorest quality) to 255 (highest quality). The parameter can be read by the application in a callback function invoked by the stack and is only valid for the duration of the callback function execution.

Table 9: Stack Parameters

* MLD = Multicast Listener Discovery

Part III:
Optional Features

10. Over-Network Download (OND)

This Chapter describes the optional Over-Network Download (OND) facility that is available in a JenNet-IP system. OND provides the capability to upgrade application software on the nodes of a JenNet-IP WPAN by:

- distributing the replacement software through a WPAN from a “server” device in the LAN/WAN domain
- updating the software in a node with minimal interruption to node operation

If required, OND must be integrated into the applications that run on the nodes of the WPAN. Functions and other resources are provided to aid this integration.

10.1 OND Terminology

The following key terms/concepts are used in the description of OND:

Term	Description
Image	Binary file containing stack and application code (and data) to be run on a device
Server	Device that stores an image for other devices and distibutes it via OND

10.2 OND Features

The main features of OND are summarised below:

- OND occurs in the background, so the network continues to operate normally during a download:
  - Takes up to an hour to update an image across a 100-node network
  - Download time does not vary much with network size
  - Rate of download can be adjusted as required
- A new image can be:
  - Automatically sent to all interested nodes at once, or to specific nodes
  - Pushed into the network from outside
  - Pulled by individual nodes
- Servers can be located anywhere in the IP part of the system
- All JenNet-IP routing nodes can act as intermediate image servers:
  - Can serve their own image or one for a different device type
  - Network traffic is kept to a minimum
- Nodes can be configured to reset automatically once the new image is downloaded, or under control of their application, or under command from a remote device

- Border-Router provides interfaces to load new images, to push them into the network and to initiate a reset
    - Web interface
    - Command line interface
- JIP MIB allows monitoring of downloads and requests for new images on each device
- API is also provided for local administration by the node's application

10.3 General Operation

OND allows the application software on a JenNet-IP wireless node to be upgraded with minimal disruption to node operation and without physical intervention by the user/installer (e.g. no need for a cabled connection to the node).

The replacement software is distributed from an OND server, which is a device located in the LAN/WAN domain. The image is distributed to the WPAN via the Co-ordinator, which is also termed a server. The wireless Routers are able to propogate the image through the network to the relevant nodes. This is illustrated in the figure below.

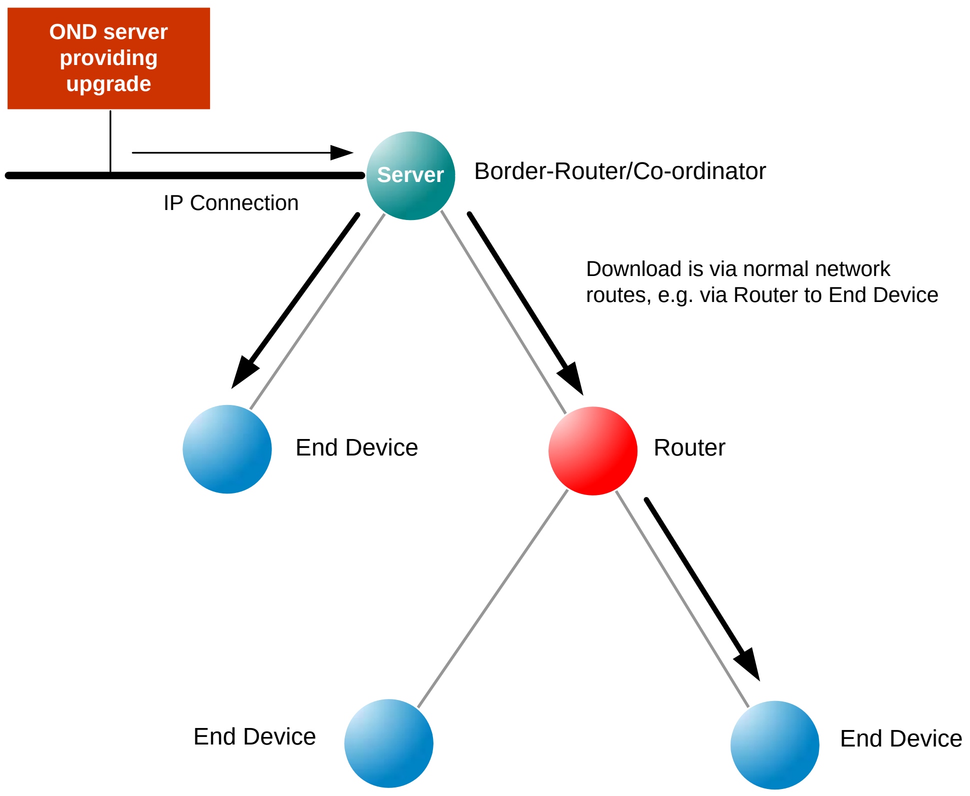

OND server providing upgrade
Server
Border-Router/Co-ordinator
IP Connection
Download is via normal network routes, e.g. via Router to End Device
End Device
Router
End Device
End Device

Figure 16: OND Routing

Note that the JN5164 device can only act as an intermediary in the OND process and cannot update its own application image via OND - for details, refer to Section 10.6.

10.4 Image Storage

An application image is stored in the Non-Volatile Memory (NVM) associated with the JN51xx wireless microcontroller, normally Flash memory. The JN516x device has internal Flash memory but can also have an external Flash device.

The application is run directly from internal Flash memory. A node with the capability to participate in OND can store upgrade images in either internal or external Flash memory. By default, internal Flash memory is used, which may contain more than one application image - the currently running image and one or more upgrade images.

The upgrade images that may be stored in the Flash memory (internal or external) of a device participating in OND are as follows:

- A newer replacement for the current image in one of the following states:
  - Not yet completely received
  - Complete but not verified
  - Complete and verified but not yet running
- An image intended for a different device type (may be partial or complete)

Space is reserved in Flash memory for each image. A whole number of Flash sectors are reserved for an image, even if the image will not completely fill this space.

Example storage in JN516x internal Flash memory is shown in Figure 17.

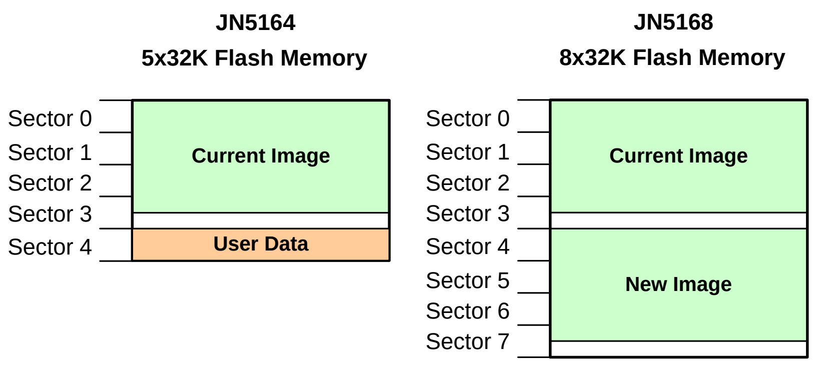


Figure 17: Example Image Storage in JN516x Flash Memory

Note: If OND upgrade images are to be stored in external Flash memory, this must be configured at compile-time - refer to Section 10.10.

10.5 Multi-Image Bootloader

During the boot process at JN516x start-up, the bootloader provided in on-chip ROM searches through internal NVM (Flash memory), looking at the start of each sector for the image header that identifies the current application image.

If OND is implemented, the bootloader may need to choose between multiple images stored in Flash memory. On finding/choosing an image, the JN516x bootloader may re-map the image to other Flash sectors. If the bootloader cannot find a valid image, it will search external Flash memory and copy the relevant image to internal Flash memory.

On the JN516x devices, an application image can be stored in any (reserved) sector of Flash memory.

10.6 OND Restrictions for JN5164

On the JN5164 device, it is only possible to update the application image using OND if the device is equipped with external Flash memory. However, even without external Flash memory, this device is still able to forward OND requests and blocks for other nodes, and to report its own OND image version information (for example via the OND MIB). Therefore, if a JN5164 device is to be deployed in a node of a network in which OND is used by other nodes, OND should be also enabled on the JN5164 device.

On a JN5164 device without external Flash memory but on which OND is enabled, the OND MIB (see Section 10.8) contains sensible read values but should not be written to. The image list will have just one entry, the current application image.

10.7 OND Process

This section outlines the process of performing an OND.

Note that the JN5164 device can only act as an intermediary in the OND process and cannot update its own application image via OND - for details, refer to Section 10.6.

10.7.1 Initiating an OND

The OND process is normally a broadcast initiated by an OND server in the LAN/WAN domain, which can be any one of:

- **LAN/WAN device:** This device may be a PC, tablet or mobile phone with an IP connection to the target WPAN.
- **Border-Router:** The OND can be initiated using a command line or web interface on the Border-Router, as provided in the Linksys router supplied in the JN516x-EK001 Evaluation Kit.

The Co-ordinator of the WPAN acts as the origin of the OND within the network and is therefore also referred to as a server.

There can be more than one OND server for a JenNet-IP system. The Co-ordinator can hold a list of the other servers (in the LAN/WAN domain) for the system. Then, if the Co-ordinator cannot fulfil an OND request from within the WPAN, it may pass on the request to another server in its list.

10.7.2 Downloading an Image

A wireless Router will receive an image, store it in NVM and pass it on within the WPAN. However, this Router may pass on blocks of the image before it has received all blocks of the image. This image download is illustrated below in Figure 18.

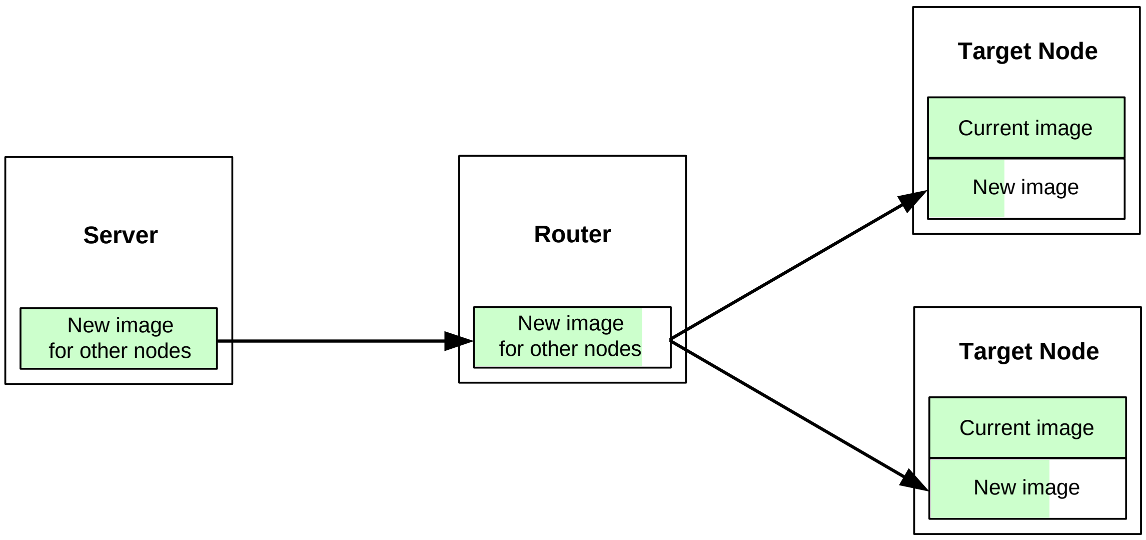


Figure 18: Downloading an OND Image

Note: An End Device which is capable of sleeping can receive upgrade images via OND but image blocks are only sent to it on request. The frequency at which the blocks are requested can be specified through the function **vOND_SleepConfigure()** - see Section 10.9.

An OND image arrives over-the-air at a wireless node one block at a time, but the blocks may not arrive in sequential order. The process of receiving image blocks is described below (also refer to Section 10.7.3 which provides additional infomation on the recovery of missing or erroneous blocks).

1. The node waits for a block to arrive. A timeout is applied to the arrival of a new block. Once this amount of time has passed since receiving the previous block, the node will request the block from the server (note that for a broadcast OND, the timeout is extended by a factor of five).
2. A block arrives in an IEEE 802.15.4 frame, which is verified using a 16-bit check sequence to detect errors introduced during transmission. If the frame is erroneous then it is rejected.
3. A successfully received block is written to NVM and then read back from NVM. If the read block does not match the received block, this indicates an NVM write-error. Since Flash memory can only be erased by sector, it is not possible to simply re-write the erroneous block - the whole image is discarded and the image is requested again from the server.
4. The last block of an image contains a checksum on the application binary that was calculated when the image was created. When all blocks of an image have been received and written to NVM, another checksum is calculated on the entire image, including the checksum included in the image. The final checksum should be 0 - if it is non-zero, the whole image is erased from NVM and the image is requested again from the server.
5. Once the new image has been successfully verified, its header is updated to indicate that the image is valid (and that the current image can be erased from NVM and the device reset). The device may automatically reset and run the new image or may wait until instructed to reset, depending on the ‘auto-reset’ field of the image footer (see Section 10.10.2).

10.7.3 Recovering Image Blocks

As stated in Section 10.7.2, the target node of an OND may need to request individual blocks that failed to arrive or were corrupted during transmission, or request the whole image again if an NVM write-error occurred.

Note the following:

- The image in NVM has a footer containing a block-map which is updated on the target node to keep track of which blocks have been successfully received during the download.
- If a block needs to be requested, the request will not be passed all the way to the OND server if a Router or the Co-ordinator (along the download path) can provide the relevant block.
- If a node does not receive a response to a block request within the timeout period, it will slightly increase the delay before sending another request. Eventually, after a large number of unanswered requests, the node will abandon the OND and erase the image in NVM.

10.8 Incorporating OND into an Application

This section describes how to integrate the OND facility into a JenNet-IP application for a wireless node (building the application is described in Section 10.10).

10.8.1 Configuration in Application

OND can be configured within the application on a wireless node by means of the three JenNet-IP stack parameters (see Section 9.3) listed and described below.

Parameter	Type	Values	Description
u8OND_SectorsAvailable	uint8	Default: 4 Max: 255	Number of NVM sectors available for the storage of application images.
u8OND_SectorSize	uint8	Default: 64 Max: 255	Size, in KBytes, of each NVM sector (this value is dictated by the NVM device used).
u8OND_SrvMaxServers	uint8	Default: 2 Max: 16	Number of entries in the list of OND servers held on the Co-ordinator.

Table 10: OND-related Stack Parameters

10.8.2 Initialisation in Application

In order to implement OND in a JenNet-IP system, OND initialisation code must be added to the applications that run on the server node and other nodes of the WPAN. These code additions are described below. The referenced functions are detailed in Section 10.9.

In all JN516x-based nodes, the application must initialise the JenOS Persistent Data Manager (PDM) before initialising OND (since OND uses PDM).

Server Node (Co-ordinator)

The Co-ordinator of a WPAN acts as an OND server (and is usually located within the Border-Router at the interface of the WPAN and the LAN/WAN domain). To enable this node as a server, the function **eOND_SrvInit()** must be called once the JenNet-IP stack has been initialised and the stack event E_STACK_STARTED has occurred.

Other Nodes

In the applications that run on the non-server nodes, the following OND initialisation functions must be called once the JenNet-IP stack has been initialised and the stack event E_STACK_JOINED has occurred:

- **Router:** The function **eOND_DevInit()** must be called
- **Sleeping End Device:** The function **eOND_SleepingDevInit()** must be called, optionally followed by **vOND_SleepConfigure()**

The number of bytes of stack required for OND on the different JN51xx chips are:

- JN5164-001: 0
- JN5168-001: 560

10.8.3 Performing a Download

An application upgrade can be pushed from an OND server or pulled from a wireless node, as described below.

Pushed from Server

The download of a new application image can be initiated on an OND server in the LAN/WAN domain, which will lead to a broadcast from the Co-ordinator in the WPAN. A receiving node that has been initialised for OND will check the first (or any) block of the image to determine whether the new image is relevant to itself - the node does this using the Device ID, chipset and image revision from the block. If the image is relevant, the node saves all blocks of the image to local NVM. If the node is a Router, it will pass the image blocks to its children (irrespective of whether it saves the image for itself). All of this is automatic, requiring no application involvement. If the auto-reset option has been enabled in the image at build-time (see Section 10.10.2) then once the download has completed, the JN51xx device will be automatically reset and the new image will be run on boot-up.

Note: An example of this type of OND which uses the JenNet-IP Browser with the JN516x-EK001 Evaluation Kit is described in Appendix J.

Pulled from Node

The application running on a node can request a specific application image from an OND server. For a sleeping End Device, this 'Pull' method must be used to obtain an upgrade image. The request is configured and initiated by accessing the MIB variables of the OND module which is provided with JenNet-IP (and fully detailed in Appendix G.3.4). Table 11 below lists the OND MIB variables and describes the effects of reading/writing from/to these variables.

In order to request an image from an OND server, the current application must:

1. If required, write the Device ID for the new image to the *DeviceID* variable (this step is only required if the Device ID of the node is to be changed).
2. Write the revision of the new image to the *Revision* variable.
3. Request the new image by writing any value to the *Download* variable.
4. Periodically read the *Images* variable to check whether the download has completed. A table will be returned containing an entry for every image in NVM. One entry will have a Status field value of '2' (Loading) if the download is in progress or '3' (Valid) if the download has successfully completed (see Appendix G.3.4).
5. If the new image is to be run, write the index of the relevant entry from the *Images* table (see previous step) to the *LoadImage* variable in order to transfer the new image to RAM and switch execution to the new image. This step is not needed for a compatible image in which the auto-reset option has been set at build-time (see Section 10.10.2).

Variable	Read Action	Write Action
Images	Returns list of all images on the node (uses the undocumented function **eOND_DevGetImage()**)	-
DeviceID	Returns the Device ID for the current image in use by the local node	Sets the Device ID for the next image to be downloaded (see Caution below)
ChipSet	Returns the chipset for the current image in use by the local node	Sets the chipset for the next image to be downloaded (see Caution below)
Revision	Returns the revision of the current image in use by the local node	Sets the revision for the next image to be downloaded (see Caution below)

Table 11: OND MIB Variables

Variable	Read Action	Write Action
Download	-	Starts a download of an image with the image identifiers currently set in the *Devi-ceID*, *ChipSet* and *Revision* MIB variables. (uses the undocumented function **eOND_DevCommenceUpdate()** with these values and with the server address set to all zeros (which, in JenNet, refers to the Co-ordinator))
LoadImage	Returns the image identification number of the current image	Switches to the image with the supplied image index number obtained by reading *Images* (uses the undocumented function **eOND_DevSwitchToImage()**).

Table 11: OND MIB Variables

A JN5164 device with OND enabled will have sensible read values for the OND MIB variables but the OND MIB should not be written to - see Section 10.6.

Caution: *Reading any variable of the OND MIB, except Images, will reset the values of DeviceID, ChipSet and Revision to those of the application image that is currently running on the node.*

10.9 OND Initialisation Functions

This section details the C functions that are used to initialise the OND facility in an application. Use of these functions is described in Section 10.8.2.

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

Function	Page
eOND_SrvInit	220
eOND_DevInit	221
eOND_SleepingDevInit	222
vOND_SleepConfigure	223

eOND_SrvInit

```
teOND_Result eOND_SrvInit(uint16 u16Port);
```

Description

This function is used on an OND server node (the Co-ordinator) to initialise the OND facility. The function reads NVM to determine which images or partial images are currently stored (and expects to find at least the current application image for the node). The number of the UDP port through which OND IPv6 traffic will pass must be specified.

On JN516x devices, the JenOS Persistent Data Manager (PDM) must be initialised before calling this function (since OND uses PDM on these devices).

Parameters

u16Port	Number of UDP port to be used for IPv6 traffic (the default port number for OND is 1874)

Returns

E_OND_OUT_OF_RANGE (unable to find valid NVM location of current image)

E_OND_INIT_ERROR (unable to initialise NVM access or configure UDP socket)

E_OND_SUCCESS (successful initialisation)

eOND_DevInit

```
teOND_Result eOND_DevInit(void);
```

Description

This function is used to initialise the OND facilityon a non-server node. The function reads NVM to determine which images or partial images are currently stored (and expects to find at least the current application image for the node).

The JenOS Persistent Data Manager (PDM) must be initialised before calling this function (since OND uses PDM).

On a sleeping End Device, the function **eOND_SleepingDevInit()** must be used instead of this function.

Parameters

None

Returns

E_OND_OUT_OF_RANGE (unable to find valid NVM location of current image)

E_OND_INIT_ERROR (unable to initialise NVM access)

E_OND_SUCCESS (successful initialisation)

eOND_SleepingDevInit

```
teOND_Result eOND_SleepingDevInit(void);
```

Description

This function is used to initialise the OND facility on a non-server node which is a sleeping End Device. The function reads NVM to determine which images or partial images are currently stored (and expects to find at least the current application image for the node).

The configuration performed by this function allows upgrade images to be obtained only by the 'Pulled from Node' method (see Section 10.8.3). Image blocks can only be requested while the device is awake. Also, only one upgrade image can be stored on the device (in addition to the currently running image).

The JenOS Persistent Data Manager (PDM) must be initialised before calling this function (since OND uses PDM).

Parameters

None

Returns

E_OND_OUT_OF_RANGE (unable to find valid NVM location of current image)

E_OND_INIT_ERROR (unable to initialise NVM access)

E_OND_SUCCESS (successful initialisation)

vOND_SleepConfigure

```
void vOND_SleepConfigure(uint16 u16PollDelay,
                         uint16 u16WakesPerPoll,
                         uint16 u16PollsPerRefresh);
```

Description

This function is used to configure how the blocks of an OND image are received by a sleeping End Device, including:

- Time between requesting an image block and polling the parent for the block (this must allow time for messages to pass up and down the network).
- Frequency at which image blocks are requested, in terms of the number of times the device will wake from sleep between consective block requests.
- Frequency at which the device will enquire about a new image, in terms of the number of times the device will wake from sleep between consective block requests (since the sleeping device does not receive broadcast blocks, it will not be aware when a new image is available). If this polling for a new OND image is disabled, a new image can be indicated by writing to the OND MIB on the device.

The default values for the parameters of this function are set during the call to **eOND_SleepingDevInit()**.

Parameters

u16PollDelay	Time, in milliseconds, from requesting a block to sending out a data request to retrieve the block from the parent. If set to 0, the JenNet auto-poll mechanism will be used. The default value is 500
u16WakesPerPoll	Number of wake-ups between consecutive block requests during an image download. The default value is 1
u16PollsPerRefresh	Number of wake-ups between consecutive block requests to determine whether a new image is available. The value of 0 disables this polling feature. The default value is 8

Returns

None

10.10 Building an Application with OND

This section describes how to build an application that incorporates OND. The final output is the application image to be loaded into NVM of the target device.

10.10.1 Makefile Modifications

Modifications to the application makefile are only required for OND if the upgrade image is to be stored in external Flash memory rather than internal Flash memory of the JN516x device. In this case, the makefile must be changed to select a different version of the JenNet library. For example, in the makefiles for the various device types provided in the *JenNet-IP Smart Home Application Note (JN-AN-1162)*, the following line is included to select the JenNet library:

```
APPLIBS += JenNet
```

This line must be changed to:

```
APPLIBS += JenNet_ExternalOND
```

10.10.2 Post-Build Modifications (using Checksum Tool)

Once an application has been built with the makefile described in Section 10.10, the resulting binary file can be modified in the following ways to produce the final application image:

- **Checksum:** A checksum is added to the final block in the image, so that the OND engine can check that the image has been received completely (this modification is mandatory)
- **Image Identifiers:** These are placed in the space where the encryption vector would normally be and are read by the Border-Router when it is given an image to pass into the network (mandatory if intending to distribute the image using the JenNet-IP Browser)
- **Footer:** A footer can be added to the end of the image, containing the image identifiers. Also see Footer below

The OND Checksum Tool can be used to modify a binary file in any of the above ways to produce the final OND image. This tool is supplied with the JenNet-IP software and is described in Section 10.10.3.

Footer

The footer is used for an image that is to be loaded into NVM using the JN51xx Flash Programmer and not distributed via OND. However, it is used by the OND initialisation code to obtain the image identifiers and characteristics of the current running image. Without this, if an image is loaded using the JN51xx Flash Programmer, the OND code would not know the image identifiers and would not be able to automatically update itself via OND. On the JN516x device, the footer is positioned at the end of the image itself so that it is feasible to include the footer in all images, regardless of whether they are to be programmed from the JN51xx Flash Programmer or via OND.

10.10.3 OND Checksum Tool

This tool is accessed from the command line using the following command:

```
OND_Checksum (flags) <infile> <outfile>
```

The flags are are listed and described in Table 12 below, where

M: Mandatory

N: Not required

O: Optional

Flag	Purpose	Requirement
`-r`	Indicates that a device receiving this image should reset itself as soon as it has completed downloading the image. Requires `-f` or `-6` flag to be used too.	O
`-t <timeout>`	Time-base for transmission of blocks, in units of 1/62500 second. The recommended values are 1 second for a small network (less than 50 nodes), 3 seconds for a medium network (50 to 150 nodes), 5 seconds for a large network (more than 150 nodes). Value is used by the receiving nodes to determine the rate at which they should request blocks. Requires `-f` or `-6` flag to be used too.	O
`-v <device ID> <chipset> <revision>`	Use the specified image identity values in the footer or in the IEEE/MAC address area of the image. Required when `-f`, `-m` or `-6` flag is used.	M
`-m`	Puts the image identity in the IEEE/MAC address area of the image. Requires `-v` flag to be used too. Optional, but required if intending to distribute the image using the JenNet-IP Browser.	O
`-6`	Specifies that image is for a JN516x device and adds footer to end of image. By default, a 1-second timeout is set and auto-reset is disabled. Requires `-v` flag to be used too, and can be used with `-r` and `-t` to set different values for timeout and reset.	M
`-f`	Adds a footer to the end of the final NVM sector for the image. By default, a 1-second timeout is set and auto-reset is disabled. Requires `-v` flag to be used too, and can be used with `-r` and `-t` to set different values for timeout and reset. This option can be specified for the JN516x device but has no effect.	N
`-i <size>`	Specifies the total NVM space, in KBytes, needed for the image - this is used to position the footer at the end of the final NVM sector for the image (32 and 128 are sensible values). This option is not needed for a JN516x device as the tool defaults to 128, which is the correct value for the device.	N

Table 12: OND Checksum Flags

Other flags (`-s`, `-b`, `-p`, `-a`) are available but are not useful during application development, so are not described here.

Specified values can be decimal (no prefix) or hexadecimal (prefixed with "0x").

The following is an example command for creating an image for any JN516x device (it is assumed that the command is issued from the **Applications/<App name>/Build** folder):

```
../../../Tools/OND/OND_Checksum -6 -m -v <device ID>
<chipset> <revision> <original binary name> <new binary name>
```

11. Standalone WPAN

A full JenNet-IP system comprises one or more WPANs, each with an IP connection via a Border-Router (see Figure 5 on page 41). However, a WPAN can operate in standalone mode without an IP connection. In this case, IP connectivity is provided as an expansion option through the addition of a Border-Router (see Section 11.3).

A typical example of a standalone WPAN is a home lighting system consisting of a remote control unit and a number of (controlled) lamps. Extending this system with the addition of a Border-Router would allow the home lighting system to be controlled from an IP-based device such as a PC, tablet or mobile phone.

11.1 Architecture and Operation

A standalone WPAN consists only of Routers and a pseudo-Co-ordinator. The pseudo-Co-ordinator, normally a remote control unit, forms the network. The identity and properties of the network are configured through this node, which acts as the agent through which other nodes join the network (see Section 11.2). Unlike normal Co-ordinators and Routers, the pseudo-Co-ordinator is able to sleep.

Once the network has formed, the pseudo-Co-ordinator can sleep and only wakes when a button is pressed. As a remote control unit, it is then used to send control commands to one or more target devices. A command is broadcast and all Routers that are within radio range will rebroadcast it - the rebroadcasting continues as the command propagates through the network until all Routers have received it. In this way, the network has a mesh-like topology. A target device will recognise a command that is for itself and act upon it. Note that the pseudo-Co-ordinator only sends commands and does not receive them. Also note that pinging is disabled in this kind of network - therefore, when there is no ‘user’ traffic, the network is silent.

Even without an IP connection, a standalone WPAN still employs IPv6 packets, by transporting data between wireless nodes in compressed IPv6 packets embedded in IEEE 802.15.4 frames. This provides the capability to extend the system into the LAN/ WAN domain (see Section 11.3).

11.2 WPAN Formation

The formation of a standalone WPAN requires:

- A remote control unit programmed as a JenNet-IP pseudo-Co-ordinator
- One or more target devices programmed as JenNet-IP Routers

Two situations are described below: a cold start (starting the network from scratch) and a warm start (waking from sleep).

Cold Start

When starting the network from scratch, the pseudo-Co-ordinator (remote control unit) must be started first. The application that runs on this device must establish a JenNet-IP network as described in Section 4.1.1 for a Co-ordinator. The target devices (Routers) can then be started and the applications that run on these devices must initialise themselves as Routers, again as described in Section 4.1.1.

Once a Router has joined the remote control unit, it does not need to remain a child of this unit in order to be controlled. The pseudo-Co-ordinator (remote control unit) can have a maximum number of children, determined by the JenNet Parameter `u8MaxChildren` (see Section 9.2) which is set to 10 by default. Therefore, a child can be discarded to ensure that sufficient child places are available to allow further nodes to join the pseudo-Co-ordinator. The function **vApi_DeleteChild()** can be used on the pseudo-Co-ordinator to break this parent-child relationship.

Warm Start

The pseudo-Co-ordinator (remote control unit) sleeps when not in use and is woken by any button-press. In this case, the device re-starts as described in Section 4.1.2. During sleep, memory is held and so the details of the network are preserved in the device, allowing the device to resume its place in the network on waking.

11.3 IP Extension

A standalone WPAN can be extended into a full JenNet-IP system by adding an IP connection, allowing the network to be controlled and monitored from an IP-based device (such as a PC, tablet or mobile phone connected to the Internet). This connection is added by introducing a Border-Router, which interfaces the WPAN to a LAN, which may be connected to a WAN - for example, interfacing a wireless home lighting system to the Internet connection in the home.

In this case, the Border-Router also becomes the WPAN Co-ordinator and the remote control unit effectively becomes an End Device. Again, this device sleeps when not in use and is woken by a button-press. The device is then used to send control commands to one or more target devices within the WPAN by broadcasting to all Routers within radio range - propagation of the command continues as described in Section 11.1.

When a command is received from an IP-based device, the remote control unit is not woken and does not play a role in relaying the command to the target device(s). In this case, as the Co-ordinator, the Border-Router broadcasts the incoming command to the Routers within radio range - propagation of the command continues as described in Section 11.1.

Unlike in a standalone WPAN, the Router nodes in a WPAN with IP connection implement the auto-ping mechanism described in Section 2.9.1.

Note: The remote control unit effectively acts as an End Device in that it originates commands, has no routing role and can sleep when not needed. The device does not receive commands and therefore has no need to poll its parent for messages. In addition, the device does not ping its parent.

In extending a standalone system to a full JenNet-IP system, the new system can be started in one of two ways:

- The new system can be started from scratch, which assumes that the previous standalone system has been completely powered down. In this case, the Border-Router/Co-ordinator must be started first in order to establish the WPAN, as described in Section 4.1.1 for a Co-ordinator, after which the Routers and End Device can be started in any order.
- The new system can be spawned from the running standalone system. In this case, the Border-Router node first joins the network of the remote control unit from which it obtains network parameter values such as PAN ID, radio channel and network key. The Border-Router then restarts as a Co-ordinator in order to create a WPAN with the same network parameters and takes over the standalone network as a full JenNet-IP system.

Note: A WPAN node in a full JenNet-IP system can be programmed to switch to standalone mode if it loses contact with its parent and then to leave standalone mode if it regains contact with its parent.

Part IV: Appendices

A. Notes on JenNet Initialisation

This appendix describes the use of certain JenNet parameters in the operation and maintenance of a wireless network in a JenNet-IP system:

- Routing (Appendix A.1)
- Losing a parent node (Appendix A.2)
- Losing a child node (Appendix A.3)
- Auto-polling (Appendix A.4)

The JenNet parameters can be set in the functions **vJIP_InitStack()** and **vJIP_ConfigureNetwork()**. These parameters are contained in structures that are fully detailed in Chapter 9.

A.1 Routing

The Co-ordinator and Routers of a network can each play a role in routing messages and, in JenNet-IP systems, their routing capability is enabled automatically.

A routing node contains both a Neighbour table and a Routing table (see Section 2.7.1). The Neighbour table is small, since a node can have an absolute maximum of only 16 children. The Routing table, however, can potentially accommodate entries for a very large number of descendant nodes and therefore take up significant memory space. For this reason, the application is allowed some control over the Routing table, in order to limit the amount of memory space occupied by the table.

The Routing table is represented in memory as an array of structures, where each structure contains the routing information for one descendant node (these structures are automatically filled in by the stack when the network is formed and are not the concern of the application). Each array element contains 12 bytes. This array must be declared in the application and configured using two JenNet parameters:

- *u32RoutingTableEntries* determines the number of elements in the array and therefore the maximum number of descendant nodes (excluding immediate children). This value should be set realistically to the maximum expected number of nodes in the network (plus 10-20% to allow for movement of nodes), so not to reserve more memory space than needed for the Routing table.
- **pvRoutingTableSpace* is a pointer to the Routing table in memory - thus, the array will start at this point in memory.

Note: If a node attempts to join a network and this requires a new entry in a Routing or Neighbour table which is already full, the join request will be ignored.

A.2 Losing a Parent Node (Orphaning)

A node must be able to determine if it has lost its parent and become an orphan. Once orphaned, the node may then need to re-join the network.

A.2.1 Detecting Orphaning

There are two ways a child node can determine whether it has been orphaned:

- Lost packets
- Lost pings

These methods are described below.

Lost Packets

A node may decide that it has lost its parent when a certain number of consecutively sent packets have been lost. In JenNet, this number is determined by the parameter *u8MaxFailedPkts*. Each packet is sent four times (original attempt plus three retries) before it is considered to be 'failed'. Therefore, when a child loses its parent, the total number of lost packets will be 4 x *u8MaxFailedPkts*. Since the node has now lost its parent, the orphaned node will attempt to re-join the network (see Appendix A.2.2).

Lost Pings

In a quiet network with little traffic, Routers and End Devices generate pings to avoid the loss of a parent (auto-pings are described in Section 2.9.1). If there is no other traffic on the link:

- A Router will periodically ping its parent at an interval determined by the JenNet parameter *u16RouterPingPeriod* (in units of 10 ms).
- An End Device will periodically ping its parent at an interval determined by the JenNet parameter *u8EndDevicePingInterval* (expressed in terms of sleep cycles). For example, if this interval is set to 4 and the sleep period is 2 seconds, the node will ping its parent every 8 seconds.

Given no other network traffic, the number of failed pings before the node decides that it has lost its parent is determined by the JenNet parameter *u8MaxFailedPkts* (which is set to 5, by default). Thus, in this case, the orphaned node will attempt to re-join the network (see Appendix A.2.2) after a time given by *u8MaxFailedPkts* multiplied by the ping interval.

Note: Further information on pinging from an End Device is provided in Appendix B.2 for the JenNet-IP release v1107 or above.

A.2.2 Re-joining the Network

When a node considers its parent to be lost (see Appendix A.2.1), JenNet initiates a stack reset and begins a search for a new parent. The application is notified with E_STACK_RESET.

The recovery method depends on the node type, as follows:

- An orphaned Router will continuously scan for a new parent until a network is joined. JenNet then sends an E_STACK_JOINED event to the application.
- An orphaned End Device will scan for a new parent. If the device is successful in re-joining the network, JenNet sends an E_STACK_JOINED event to the application. Otherwise, the device goes to sleep for a period determined by the JenNet parameter *u32EndDeviceScanSleep*, then scans again, repeating the scan/sleep cycle until the network has been successfully re-joined.

A.3 Losing a Child Node

A parent node must be able to determine whether its children are still active. The detection methods for the loss of a child node are different for End Device and Router children.

A.3.1 End Device Children

Two mechanisms are employed by a parent to determine whether an End Device child has become inactive and should therefore be removed from its set of children:

- A timeout on communications coming from the End Device
- Restrictions on the locally buffered messages destined for the End Device

These are described in the sub-sections below.

***Caution:** In order to avoid being removed from the network, an active End Device must ensure that **both** the communication timeout and the buffered message restrictions are not violated.*

Communication Timeout

For an End Device child, the parent implements a timeout period on communications from the child. This timeout period, in units of 100 ms, is determined by the value of the JenNet parameter *u32EndDeviceActivityTimeout*.

- If the parent does not receive a communication from the End Device child within this timeout period, it considers the child to be lost and removes it from the Neighbour table (this change will also be propagated up the tree to the Routing tables of ascendant nodes).
- If the parent does receive a communication from the End Device child within this timeout period, the timeout is reset and starts again.

Note that data polling from the child does not count as communication for this purpose.

Automatic pings from an End Device to its parent can be used to prevent this timeout mechanism from deducing that the child is lost when it is simply sending data infrequently. A ping is generated on waking after a number of sleep cycles, where this number is configured using *u8EndDevicePingInterval*. For this mechanism to work, the End Device child must sleep/wake regularly enough for the time between pings not to exceed the value of *u32EndDeviceActivityTimeout*, otherwise the parent will assume the child is lost.

Note: An End Device that must stay awake for long periods may need to regularly send data to its parent, to avoid being considered lost by the parent.

Buffered Message Restrictions

Data messages sent to an End Device are buffered by the node's parent and collected by the End Device through data polling. This allows messages that arrive while the End Device is asleep to be retained and later collected when the End Device is awake.

Pending messages for an End Device are stored in buffers from the parent's application buffer pool and are passed to one of its 802.15.4 MAC buffers for collection by the End Device child (the messages are fed through the MAC buffer one at a time). However, the parent will not indefinitely store a message in the MAC buffer - once a message has been in the MAC buffer for 8 seconds, the message is discarded and considered to be a failed communication by the parent.

When the number of failed messages reaches the value of the JenNet parameter *u8MaxFailedPkts*, the parent considers the End Device to be a lost child and will remove this child from its Neighbour table (this change will also be propagated up the tree to the Routing tables of ascendant nodes).

This mechanism has implications for End Devices that sleep for long periods and which therefore cannot often poll for data. Such an End Device can cause routing congestion in its parent and could be mistakenly removed from the network, because its parent has buffered a sufficient number of 'failed messages' for the End Device while it has been sleeping.

To prevent these situations, follow the recommendations below:

- Avoid sending messages to an End Device that is known to be sleeping, particularly if the sleep duration is long (more than 7 seconds).
- Avoid sending messages to many End Devices at the same time.
- If an End Device periodically requests data from other nodes, ensure that it frequently polls its parent for the responses (to clear the MAC buffer as quickly as possible).

Further recommendations on data buffering for an End Device child are provided in Appendix B.1 for the JenNet-IP release v1107 or above.

A.3.2 Router Children

For a Router child, the parent counts the consecutive failed communications with the child (unreturned 802.15.4 MAC acknowledgements) and considers the child to be lost when this count exceeds the value of the JenNet parameter *u8MaxFailedPkts*. In this case, the child is removed from the parent's Neighbour table and all descendant of the Router child are removed from the parent's Routing table (these changes will also be propagated up the tree to the Routing tables of ascendant nodes).

Automatic pings from a Router to its parent can be used to prevent the parent from assuming the child is lost when it is simply sending data infrequently. Regular pings will be generated by the Router child with a ping period configured through the JenNet parameter *u16RouterPingPeriod* (on parent and child). The parent will consider the Router child to be lost if it does not receive a ping or data from the child within the period defined by the product:

u8MaxFailedPkts x *u16RouterPingPeriod* x 10 ms

A.4 Auto-polling

An End Device has the potential to sleep and may therefore not always be in a position to receive data sent to it. For this reason, messages destined for an End Device are buffered by its parent and the End Device must poll the parent for these messages. The data polling mechanism is described in more detail in Section 4.7.

In JenNet, auto-polling is enabled on an End Device by default. Auto-polling is the periodic polling of the parent, where the poll period is set using the JenNet parameter *u32EndDevicePollPeriod*. By default, this is set to 5 seconds.

Note 1: Auto-polling can also be disabled through *u32EndDevicePollPeriod* (by setting it to zero). If auto-polling is disabled, the End Device can explicitly poll the parent, when required, using the manual polling function **eJIP_Poll()**.

Note 2: An auto-poll may not retrieve all the pending data for an End Device. Therefore, even when auto-polling is enabled, the manual polling function **eJIP_Poll()** should be called (once) following each auto-poll to ensure that any remaining data is collected.

Provided that auto-polling has not been disabled, an End Device will automatically poll its parent on waking from sleep, irrespective of the poll period set. This polling is in addition to the configured auto-polling. Thus, if you set the sleep period in **vJIP_Sleep()** to be shorter than the poll period defined in *u32EndDevicePollPeriod*, the End Device will poll the parent more often than configured through this parameter.

Buffer recommendations related to auto-polling are provided in Appendix B.1 for the JenNet-IP release v1107 or above.

B. Buffer and Ping Recommendations

This appendix provides various recommendations in the use of data buffers and pinging on network nodes when using JenNet-IP release v1107 or above.

B.1 Stack Queues and Buffers

The number of buffers and queues within the stack is very important for optimum operation of a node. There are two buffer pools - MAC buffers and 6LoWPAN buffers.

The recommendations for the use of these buffers are as follows:

- On all device types, the 6LoWPAN buffer pool (configured in the structure `tsJIP_InitData`) should have more entries than the MAC buffer pool (configured using the stack parameter *gMAC_u8MaxBuffers*) to allow for received packets as well as packets for transmission. The number of extra entries required depends on the expected number of packets received by the device and the rate at which they can be processed, which in turn is related to the rate at which **vJIP_Tick()** is called. For more information, refer to the description of *gMAC_u8MaxBuffers* in Section 9.3.
- On Routers and the Co-ordinator, the number of MAC buffers that can be used to store pending data for sleeping End Device children (configured using the stack parameter *u8JNT_IndirectTxBuffers*) should be less than the total number in the MAC buffer pool. This is to ensure that one or more MAC buffers will always be available for communication between Routers. For more information, refer to the description of *u8JNT_IndirectTxBuffers* in Section 9.3.
- On Routers and the Co-ordinator, if there is expected to be application-related traffic to the End Device children, the number of MAC buffers that can be used as End Device buffers (configured using the stack parameter *u8JNT_IndirectTxBuffers*) should be similar to the number of sleeping children allowed for each parent (configured using the JenNet network profile parameter *u8MaxSleepingChildren*). This is to minimise the likelihood of application-related packets being discarded. For more information, refer to the description of *u8JNT_IndirectTxBuffers* in Section 9.3 and the description of *u8MaxSleepingChildren* in Section 9.2.

B.2 End Device Ping Interval

It is recommended that the interval at which an End Device pings its parent is set to at least 2 sleep cycles. This is so that the End Device does not send out any ping requests if it is sending or receiving application-related data every time it wakes up.

This interval is set through the JenNet network parameter *u8EndDevicePingInterval* (see Section 9.1). It can be configured from within the callback function **v6LP_ConfigureNetwork()** by setting:

```
psNetworkConfigData->u8EndDevicePingInterval = 2;
```

C. Handling ICMP Messages

ICMP (Internet Control Message Protocol) is introduced in Section 3.2.

Many ICMP messages are handled by the JenNet-IP stack. For example, if a remote device sends an ICMP ping message to a JenNet-IP node, the response to this message is automatically built and sent by the stack.

However, certain ICMP messages are passed to the application layer. For example, the ICMP ‘destination unreachable’ message is generated by a remote device when it cannot find the destination for a packet. When this ICMP message arrives back at the local node, the stack passes the message to the application using the callback function **v6LP_DataEvent()** with the following parameter values:

- *iSocket* is set to the special ICMP socket identity SIXLP_ERROR_SOCKET, defined in the header file **6lp.h**
- *eEvent* is set to E_6LP_ICMP_MESSAGE

The ICMP message can be read with the **i6LP_RecvFrom()** function. The *iSocket* parameter in this function is passed the value SIXLP_ERROR_SOCKET from the same parameter of **v6LP_DataEvent()**. The buffer (pointed to by the parameter **pu8RxData*) returned by **i6LP_RecvFrom()** contains the full ICMP message, including the ICMP header.

The ICMP header contains four bytes:

- The first byte is the type of the message
- The second byte is a code that provides type-specific information
- The third and fourth bytes together form a 16-bit checksum

There is normally additional type-specific information following the ICMP header.

The most common ICMP types of interest to an application are listed and described in the table below:

Type	ICMPv6 Error Messages	Description
1	Destination Unreachable	Generated by a Router when it cannot find a route to the given IP address. Also generated by an IP host when a packet refers to a UDP port that has no open socket on the host.
2	Packet Too Big	Generated by a Router when the outgoing link has an MTU that is smaller than the packet. This is unlikely to happen for 6LoWPAN packets since the maximum MTU size in 6LoWPAN is 1280 bytes, which is the minimum MTU size that is permissible for an IPv6 link.
3	Time Exceeded	The packet has exceeded the hop limit or a fragment of the packet has taken longer to arrive than the fragment timeout on the remote host.
4	Parameter Problem	Caused by an incorrect IP header.

Table 13: Common ICMPv6 Error Messages

The codes (second byte of ICMP header) for the above ICMPv6 error messages are detailed in the table below.

Type	ICMPv6 Error Message Codes
1	**Destination Unreachable** 0 - No route to destination 1 - Communication with destination administratively prohibited 2 - Beyond scope of source address 3 - Address unreachable 4 - Port unreachable 5 - Source address failed ingress/egress policy 6 - Reject route to destination
2	**Packet Too Big** 0 (Only one code)
3	**Time Exceeded** 0 - Hop limit exceeded in transit 1 - Fragment reassembly time exceeded
4	**Parameter Problem** 0 - Erroneous header field encountered 1 - Unrecognised Next Header type encountered 2 - Unrecognised IPv6 option encountered

Table 14: ICMPv6 Error Message Codes

Note: Details of the ICMPv6 error messages are provided in RFC 4443 available from the IETF (www.ietf.org).

D. Identifiers

This appendix described various identifiers used in JenNet-IP:

- Device ID - see Appendix D.1
- Device Type ID - see Appendix D.2
- MIB ID - see Appendix D.3

Network Application ID is described separately in Appendix E.

D.1 Device ID

The nodes of a WPAN in a JenNet-IP system are categorised according to their main functionality (as distinct from their networking role), such as a type of lamp. Each node has a 32-bit Device ID which identifies the kind of device it is. All nodes with the same Device ID have the same set of MIBs and MIB variables.

The Device ID is made up from two components:

- Manufacturer ID (16 bits)
- Product ID (16 bits)

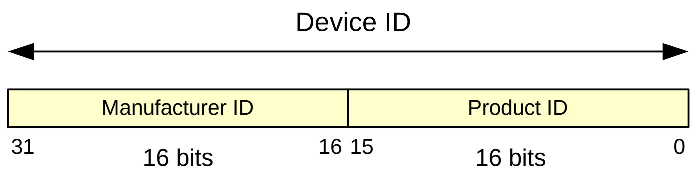


Figure 19: Device ID Format

Manufacturer ID

The Manufacturer ID is unique to the device manufacturer and is allocated by NXP - for NXP itself, this ID is 0x801. While this is a 16-bit value, the most siginificant bit is not part of the ID and is used to indicate whether the remaining 15 bits contain a valid Manufacturer ID - if this bit is set to ‘0’, the Manufacturer ID and Product ID are not valid values, and these fields can be used for other information.

During device development, it may be appropriate to use a general Manufacturer ID. The value 0x0001 should be used for this purpose.

Product ID

Device manufacturers are free to define the Product IDs of their devices.

Note: Device IDs are defined in the application makefile. For examples, refer to the Application Note *JenNet-IP Smart Home (JN-AN-1162)*.

D.2 Device Type ID

The Device Type ID is a 16-bit value indicating the role of the device in the JenNet-IP system - for example, 0x0001 represents a Border-Router.

There are two kinds of Device Type ID:

- **Standard:** The standard Device Type IDs are specified by NXP. The leading bit of a standard Device Type ID is ‘0’. A device of a standard type contains a set of standard MIBs (see MIB ID in Appendix D.3).
- **Manufacturer:** A manufacturer Device Type ID is specified by a manufacturer. The leading bit of a manufacturer Device Type ID is ‘1’. Since the same manufacturer Device Type ID may be used by multiple manufacturers, it is only meaningful when used in conjunction with the Manufacturer ID (see Appendix D.1).

Note: Device Type IDs are defined in the application makefile. For examples, refer to the Application Note *JenNet-IP Smart Home (JN-AN-1162)*.

D.3 MIB ID

The MIB ID is a 32-bit value identifying a particular Management Information Base (MIB) containing a particular set of MIB variables.

There are two kinds of MIB ID:

- **Standard:** The standard MIB IDs and the MIB contents are defined by NXP. The upper 16 bits of a standard MIB ID are 0xFFFF and the lower 16 bits identify the purpose of the MIB itself. These MIBs are intended for manufacturers who are designing products that will be interoperable with other JenNet-IP products (possibly from other manufacturers).
- **Manufacturer:** A manufacturer MIB ID and the MIB contents are defined by a manufacturer. The upper 16 bits of a manufacturer MIB ID contain the Manufacturer ID (see Appendix D.1) and the lower 16 bits identify the purpose of the MIB itself. A manufacturer-defined MIB is used when there is no standard MIB available to fulfil the required purpose.

E. Network Application ID

This appendix describes a method by which a JenNet-IP wireless network application can implement its own network identifier.

The JenNet-IP stack implements network identification by means of a 16-bit PAN ID, described in Section 2.5. A PAN ID can be pre-set on the network Co-ordinator or can be chosen by the Co-ordinator at start-up (such that the chosen value does not clash with the PAN ID of any other networks operating in the vicinity).

The applications that run on the nodes of a JenNet-IP system can also implement a 'Network Application ID' to ensure that new nodes join a network that is running the correct application. The Network Application ID is user data and can therefore be defined by the application developer - it would typically be a 32-bit value. If it is implemented, all product components (to be used in a JenNet-IP system) need to be programmed with the same Network Application ID value. To avoid clashes between different products, it should be a random value.

The Network Application ID is useful in the following stages of the node join process:

- **Channel Scan:** The Network Application ID can be included as user data in beacons that are transmitted by Routers. A joining node which receives these beacons during a channel scan can then choose to join a Router which is transmitting the appropriate Network Application ID. The implementation of this mechanism is described in Appendix E.1.
- **Route Establishment:** The Network Application ID can be included as user data in the Establish Route message that is sent up to the Co-ordinator by a joining node. Any intermediate Router that receives the message can then reject the joining node based on the value of the Network Application ID. The implementation of this mechanism is described in Appendix E.2.

If required, the Network Application ID can be implemented in an application using certain JenNet and JenNet-IP functions (both sets are included in JenNet-IP). These functions are described in Appendix E.3.

E.1 Channel Scan

The Network Application ID can be used in a JenNet-IP WPAN to ensure that a node joins a network that is running the appropriate application. During the channel scan of the join process (see Section 2.8), a joining node transmits a beacon request in each relevant channel and waits (up to 138.24 ms) for beacons from potential parents.

- A responding Router can include a Network Application ID as user data in its beacons. This beacon data can be pre-set by the application on the Router using the JenNet function **vApi_SetUserBeaconBits()**.
- If the joining node is to look for beacons containing a certain Network Application ID, this analysis must be incorporated in the user-defined callback function that is invoked when a beacon is received - this callback function can be registered using the JenNet function **vApi_RegBeaconNotifyCallback()**.

The above functions are described in Appendix E.3.

E.2 Route Establishment

During the node join process (Section 2.8), the network can accept or reject the joining node on the basis of the Network Application ID included as user data in an Establish Route message from the node.

- The application on the joining node can use the JenNet-IP function **v_6LP_SetUserData()** to pre-set the relevant Network Application ID as user data for inclusion in Establish Route messages. As part of the join process, this message is sent up to the Co-ordinator of the desired network via any intermediate Routers.
- Before entering the joining node into a Routing table, a Router that receives an Establish Route message can check the embedded Network Application ID. This analysis must be incorporated in the user-defined callback function that is invoked when an Establish Route message is received - this callback function can be registered using the JenNet-IP function **v_6LP_SetNwkCallback()**.

The above functions are described in Appendix E.3.

E.3 Functions

This section contains descriptions of the JenNet and JenNet-IP functions (and relevant callback functions) that support the passing of user data during the scanning and route establishment processes. These functions can be used by an application to implement a Network Application ID, as described in the preceding sections.

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

Function	**Page**
vApi_SetUserBeaconBits	245
vApi_RegBeaconNotifyCallback	246
v_6LP_SetUserData	247
v_6LP_SetNwkCallback	248

vApi_SetUserBeaconBits

void vApi_SetUserBeaconBits(uint8 ****pu8Bits*);**

Description

This JenNet function can be used to set the user data which is to be included in beacons transmitted by a Router or the Co-ordinator. This data could be used to represent a Network Application ID.

This beacon data is formatted as a 6-byte array. The function must provide a pointer to the first element of this array. The same data is carried in all beacons.

Note: Since the user data carried in a beacon cannot exceed 48 bits, this is the upper limit on the size of the Network Application ID (when the user data is used for this purpose).

Parameters

pu8Bits	Pointer to the first byte of the 6-byte array to be used as the user data in the beacons

Returns

None

vApi_RegBeaconNotifyCallback

```
void vApi_RegBeaconNotifyCallback(
                        trBeaconNotifyCallback prCallback);
```

Description

This JenNet function can be used to register a user-defined callback function that will be invoked when a beacon is received by the local node when it is attempting to join a network. The callback function **MibNwkConfig_bBeaconNotifyCallback()** is described below.

Parameters

prCallback	Pointer to callback function to be used to process beacons

Returns

None

MibNwkConfig_bBeaconNotifyCallback()

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

Description

This user-defined callback function is invoked when a beacon is received by a node that is attempting to join a network. The function must process the beacon, including the extraction of any user data contained in the beacon. This data could, for example, be the Network Application ID of the network from which the beacon comes.

The function returns a Boolean to indicate whether the beacon comes from a network Router which should be shortlisted for subsequent joining - for example, the Router may only be of interest if the beacon contains a certain Network Application ID (as user data).

This callback function is registered using the function **vApi_RegBeaconNotifyCallback()**, described above.

Parameters

psBeaconInfo	Pointer to first byte of 6-byte array of user data extracted from beacon
u16ProtocolVersion	Version of protocol used in beacon

Returns

TRUE if source Router to be shortlisted, FALSE if beacon to be discarded

v_6LP_SetUserData

void v_6LP_SetUserData(uint8 *u8DataLength*, **uint8 ***pu8Data***);**

Description

This JenNet-IP function can be used to set the user data which is to be included in the Establish Route message sent up to the Co-ordinator when the local node is attempting to join a network. This data could be used to represent a Network Application ID. The maximum data length is 16 bytes.

Parameters

u8DataLength	Number of bytes of user data (maximum of 16)
pu8Data	Pointer to first byte of user data

Returns

None

v_6LP_SetNwkCallback

```
void v_6LP_SetNwkCallback(tprNwkCallback prCallback);
```

Description

This JenNet-IP function can be used to register a user-defined callback function that will be invoked when an Establish Route message is received by the local (Router) node. The callback function **vHandleNwkCallback()** is described below.

Parameters

prCallback	Pointer to callback function to be used to process Establish Route message

Returns

None

vHandleNwkCallback()

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

Description

This user-defined callback function is invoked when an Establish Route message is received from a node that is attempting to join the network. The function must process the message, including the extraction of any user data contained in the message. This data could, for example, be the Network Application ID that is programmed into the node from which the message comes.

The function returns a Boolean to indicate whether the joining node should be entered into the local Routing table - for example, the joining node may only be accepted if the message contains a Network Application ID (as user data) which matches that of the network.

This callback function is registered using the function **v_6LP_SetNwkCallback()**, described above.

Parameters

psAddr	Pointer to structure (see Section 8.1.3) containing the IEEE/MAC address of the node from which the Establish Route message originates
u8DataLength	Number of bytes of user data in message (maximum of 16)
pu8Data	Pointer to first byte of user data extracted from message

Returns

TRUE if joining node to be entered in Routing table, FALSE if request to be discarded

F. JenNet-IP Data Packet Format

This appendix outlines the format of data packets in a WPAN of a JenNet-IP system. The data is contained in a series of embedded packets/frames, as described below from the top down.

IEEE 802.15.4 MAC Frame

Data is transported between the nodes of a JenNet-IP WPAN using the IEEE 802.15.4 wireless network protocol. The data is embedded in an IEEE 802.15.4 MAC frame, which has the basic format indicated in Figure 20 below.

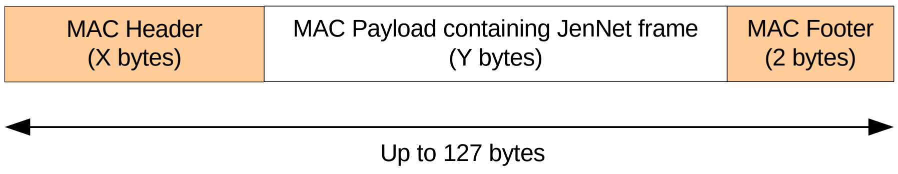


Figure 20: IEEE 802.15.4 MAC Frame

The MAC header and payload are each of variable length but typically the header is 21 bytes and the payload is 64 bytes in size (e.g. for a data-to-peer frame). The payload contains a JenNet frame (see below).

JenNet Frame

The MAC payload (above) contains a JenNet frame, which has the basic format indicated in Figure 21 below.

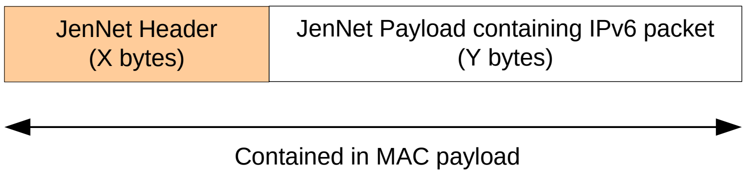


Figure 21: JenNet Frame

The JenNet header and payload are each of variable length but typically the header is 19 bytes and the payload is 45 bytes in size (e.g. for a data-to-peer frame, in which the header contains the source address for the frame). The payload contains an IPv6 packet (see below).

IPv6 Packet

The JenNet payload (above) contains an IPv6 packet, which has the basic format indicated in Figure 22 below.

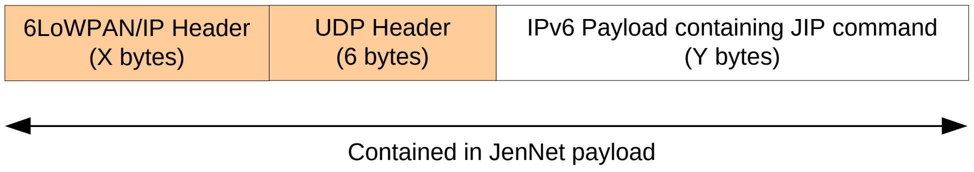

6LoWPAN/IP Header (X bytes)
UDP Header (6 bytes)
IPv6 Payload containing JIP command (Y bytes)
Contained in JenNet payload

Figure 22: IPv6 Packet

The 6LoWPAN/IP header and IPv6 payload are each of variable length but typically the header is 28 bytes and the payload is 11 bytes in size (e.g. for a data-to-peer frame). The UDP header is 6 bytes in size, compressed (see below).

The IPv6 packet is compressed (by the 6LoWPAN stack layer) for transportation in an IEEE 802.15.4 MAC frame. The typical field sizes quoted above are for a compressed IPv6 packet. When uncompressed, an IPv6 packet is typically 59 bytes in size, with the UDP header occupying 8 bytes.

The payload contains a JIP command. The standard commands and their formats are detailed in Appendix G.4.

Tip: If a JenNet-IP system uses battery-powered End Devices, frame/packet lengths should be minimised in order to conserve power during packet transmission.

G. JenNet-IP Principles

This appendix describes some of the low-level principles of JenNet-IP. This information is provided for developers who wish to implement the JIP layer directly in their software rather than through the supplied JenNet-IP APIs. For example, developers may devise their own API to implement the functionality of the JIP layer.

G.1 Introduction

The JIP layer sits above the UDP layer in the JenNet-IP stack. Relevant JenNet-IP concepts (including MIBs, variables and traps) are introduced in Section 3.4.

The layer enables management control of the JenNet-IP stack by allowing a remote device to set and retrieve the values of MIB variables, as well as a set of application variables.

G.1.1 JIP Modules

The JIP layer deals with software entities referred to as modules, where a module relates to a particular functional area (e.g. environment monitoring) and includes a MIB which contains the variables used to interface with the module. Standard (stack) modules are included in JenNet-IP. Application-specific modules can also be defined.

Standard Modules

The standard modules included in JenNet-IP are:

- Node module
- JenNet module
- Groups module
- Over-Network Download (OND) module
- DeviceID module

The above modules are detailed in Appendix G.3.

Module ID

A module is identified by a name and a 32-bit module ID:

- Standard modules have module IDs in the range 0xFFFFFF00 to 0xFFFFFFFE
- Application-specific modules have module IDs in the range 0xFFFFFE00 to 0xFFFFFEFF

The value 0xFFFFFFFF is not used as a module ID.

Module Index

A module is also dynamically allocated an index by the stack. Module index numbers are arbitrary and nothing should be inferred by their ordering. They start at zero and are incremented by one for each module.

G.1.2 JIP Variables

The operation of JenNet-IP is based on accessing variables - for example, to write configuration and control data, and read monitoring data. However, access to some variables will result in actions beyond simple read/write operations - for example, a write to change the RF channel variable will result in a physical change of the operating channel.

A variable is a value that is visible to the JIP layer. An individual variable can be disabled by the application if it does not currently hold a valid value or is not applicable to the device.

Variable Types

A variable is one of the following types (and cannot change type):

- 8-, 16-, 32- or 64-bit integer (signed or unsigned)
- Single- or double-precision float
- Variable-length string
- Blob
- Table of blobs

Variable Index

A variable is allocated an index within a module. These index numbers are arbitrary and nothing should be inferred by their ordering. They start at zero and are incremented by one for each variable.

G.1.3 JIP Commands

Access to a variable is performed as the result of a JIP command or request. A request is issued by a JIP client to the JIP server which hosts the relevant variable(s). The server may reply by sending a response to the requesting device. The standard requests and responses are summarised in the table below.

Requests from JIP Client	Responses from JIP Server
'Get' request	'Get' response
'Get by ID' request	
'Set' request	Set Response
'Set by ID' request	
'Query Modules' request	'Query Modules' response
'Query Variables' request	'Query Variables' response
'Trap' request	'Trap' response
'Untrap' request	
	'Trap' notification

Table 15: Standard JIP Commands

The above commands and the JIP command format are detailed in Appendix G.4.

Sockets and Ports

Requests and responses are passed between the JIP layer and UDP layer of the stack via a UDP socket, which is a logical entity associated with an IPv6 address and communications port on a device. The port number used for JIP layer communications is defined by the application, and the default port number is 1873. UDP sockets are introduced in Section 3.4.

Command Handles

Each request is assigned an 8-bit handle for identification purposes. The same handle is included in the corresponding response, allowing the requesting device to match a response with a request.

Multicasts

JenNet-IP allows a request to be multicast to multiple destinations using a pre-defined IPv6 multicast address associated with a group of nodes. However, only the ‘Set’ and ‘Set by ID’ requests are supported by multicasting - all other requests are ignored. Also, no response is generated for any request received on a multicast address, in order to avoid flooding the network with responses (this includes the suppression of trap notifications that would otherwise be generated by the resulting variable changes).

G.2 Discovery

In order to communicate, a node must initially discover the other nodes that exist within the network and the modules that are available on those nodes.

The nodes in the network can be discovered simply by retrieving the `NetworkTable` variable from the network Co-ordinator. This variable is a table containing the Host Interface ID and Device ID for all nodes currently in the network.

It is envisaged that a node may have access to an external database from which it can retrieve the module IDs of the modules supported on a node with a given Device ID, together with the list of variables for each module. Otherwise, the node will need to discover the modules and variables that each node offers by querying the nodes directly. To this end, the protocol allows for two types of query:

- Query Modules (see Appendix G.4.7 and Appendix G.4.8)
- Query Variables (see Appendix G.4.9 and Appendix G.4.10)

Note: In this case, it is worth considering caching the variables by module ID and the module IDs by Device ID, in order to speed up future queries and reduce network traffic.

The pseudocode fragment below illustrates the use of the above two query types to discover the modules and variables on a remote node. It is important to note that since each request will have only one response, it is necessary to make a series of requests in order to discover all of the remote modules.

```
ModuleList moduleDiscovery(void)
{
    ModuleList moduleList;
    int i = 0;
    int firstModule = 0;
    int numModulesOutstanding = 0xFF;

    do
    {
        sendQueryModuleRequest(firstModule,
                               numModulesOutstanding);

       QueryModuleResponse moduleResponse = waitForQueryModuleResponse();
        moduleList.add(moduleResponse);

        firstModule           = moduleList.count;
        numModulesOutstanding = moduleResponse.numModulesOutstanding;
    }
    while(numModulesOutstanding != 0)

    for(i = 0; i < moduleList.count; i++)
    {
        int firstVariable = 0;
        int numVariablesOutstanding = 0xFF;
        VariableList variableList = moduleList[i].variableList;

        do
        {
            sendQueryVariableRequest(i,
                                     firstVariable,
                                     numVariablesOutstanding);
            QueryVariableResponse variableResponse =
                waitForQueryVariableResponse();
            variableList.add(variableResponse);

            firstVariable           = variableList.count;
            numVariablesOutstanding =
                variableResponse.numVariablesOutstanding;
        }
        while(numVariablesOutstanding != 0)
    }

    return moduleList;
}
```

G.3 Standard Modules

Five standard modules are incorporated into JenNet-IP that are used by the stack. They are as follows:

Module Name	Module ID	Description
Node	0xFFFFFF00	Contains basic information about the host node
JenNet	0xFFFFFF01	Contains information about the nature and contents of the host network
Groups	0xFFFFFF02	Contains information about the multicast groups to which the host node belongs
OND	0xFFFFFF03	Contains information on the OND (Over-Network Download) images on the host node
DeviceID	0xFFFFFF04	Contains the Device ID of the host node and indicates the device types that the node can implement

Table 16: Standard (Stack) Modules

The above modules are detailed in the sub-sections below.

G.3.1 Node Module

The Node module has a module ID of 0xFFFFFF00 and contains certain information about the local node. The MIB variables for this module are detailed below.

Index	Variable Name	Type	Access	Description
0	MacAddr	uint64	Const	64-bit IEEE/MAC address of node
1	DescriptiveName	string	R/W	Human-readable name for the node - the name may be editable (depending on the application) and, if editable, should be saved to non-volatile memory by the application
2	Version	string	Const	String supplied by the application to indicate the application version running on the node
3	TxPower	uint8	R/W	Radio transmission power of the node - valid values are in the range 0-3, corresponding to the levels detailed in the description of the **bAHI_PhyRadioSetPower()** function in the *JN516x Integrated Peripherals API User Guide (JN-UG-3087)*

Table 17: Node Module Variables

G.3.2 JenNet Module

The JenNet module has a module ID of 0xFFFFFF01 and contains information about the network to which the local node belongs. The MIB variables for this module are detailed below.

Index	Variable Name	Type	Access	Description
0	DeviceType	uint32	Const	JenNet device type of node: 0 - Co-ordinator 1 - Router 2 - End Device
1	ParentInterface	uint64	R	Host Interface ID of node's parent
2	TreeVersion	uint32	R	Version of the network tree. Allows the monitoring of network changes - each time a node joins or leaves the network, this version number is incremented by one
3	SubTreeNodes	uint32	R	Total number of nodes in the tree below this node
4	NetworkTable	blob table	R	Table containing the addresses and Device IDs of all the nodes in the network - variable is only valid on the Co-ordinator. Each entry in the table is a blob of the format detailed in Figure 23 below
5	LastChange	blob	R	Blob in which the last network change is recorded (node added, removed or moved). This blob has the format detailed in Figure 24 below
6	NeighbourTable	blob table	R	Table containing the addresses of the node's immediate neighbours (parent and children) and the link properties to these nodes. Each entry in the table is a blob of the format detailed in Figure 25 below
7	Depth	uint32	R	Depth of the local node in the network tree (Co-ordinator is at a depth of zero)

Table 18: JenNet Module Variables

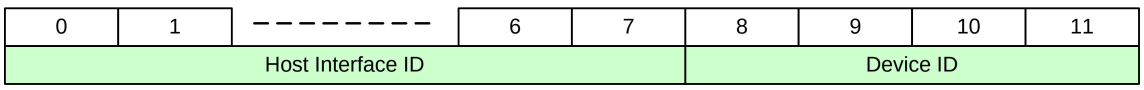


Figure 23: Format of NetworkTable Blob

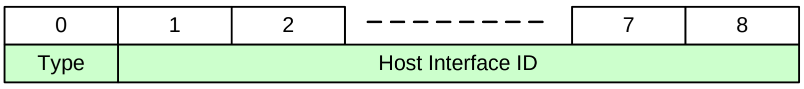


Figure 24: Format of LastChange Blob

Type: 0 - Join, 1 - Leave, 2 - Move

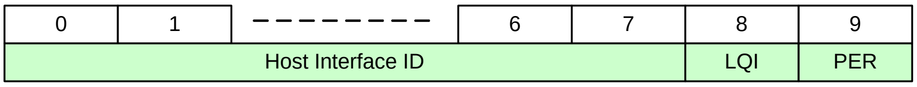


Figure 25: Format of NeighbourTable Blob

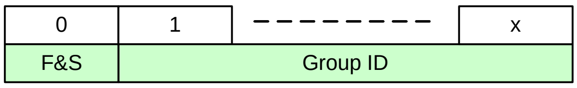
LQI = Link Quality Indicator, **PER** = Packet Error Rate
For definitions of these measures, refer to the Glossary in Appendix M.

G.3.3 Groups Module

The Groups module has a module ID of 0xFFFFFF02 and contains information about the multicast group(s) to which the local node belongs. The MIB variables for this module are detailed below.

Index	Variable Name	Type	Access	Description
0	Groups	blob table	R	Table indicating the multicast groups that the local node has joined. Each entry in the table is a blob of the format detailed in Figure 26 below
1	AddGroup	blob	R/W	Local node is added to a multicast group by writing the group details to this variable. The blob is of the format detailed in Figure 26 below. Reading this variable always gives 0
2	RemoveGroup	blob	R/W	Local node is removed from a multicast group by writing the group details to this variable. The blob is of the format detailed in Figure 26 below. Reading this variable always gives 0
3	ClearGroups	uint8	R/W	Local node can be removed from all multi-cast groups to which it belongs by writing any value to this variable. Reading this variable always gives 0

Table 19: Groups Module Variables



$1 \leq x \leq 14$

Figure 26: Format of Groups Blob

F&S = Flags and Scope (see below)

The blob shown above in Figure 26 is a variable-length compressed representation of the IPv6 multicast address of the group, shown below in Figure 27. The condensed Group ID field is obtained by removing the leading zero bytes of the multicast address.

0	1	2	--------	15
0xFF	F&S	Group ID		

Figure 27: IPv6 Multicast Address of Group

Flags and Scope

The F&S (Flags and Scope) field above comprises two 4-bit sub-fields, where the:

- 4 most significant bits contain the multicast address flags (see Table 20)
- 4 least significant bits represent the multicast address scope (see Table 21)

The four Flag bits (bits 7-4 of the F&S field) are used as follows:

Bit	Flag	Description
7	-	Reserved
6	R (Rendevous)	1: Rendezvous point included 0: Rendezvous point not included
5	P (Prefix)	1: Address has network prefix 0: Address has no prefix information
4	T (Transient)	1: Address is temporary (dynamically assigned) 0: Address is established (known throughout the network)

Table 20: Multicast Address Flags

The four Scope bits (bits 3-0 of the F&S field) are used as follows (also refer to the *JenNet-IP LAN/WAN Stack User Guide (JN-UG-3086)*, which introduces scope for unicast addresses):

Value	Scope	Description
0x1	Interface-local	Spans a single interface on a node and is only applicable to loopback transmissions
0x2	Link-local	Spans the same region as the link-local unicast scope, described in the *JenNet-IP LAN/WAN Stack User Guide (JN-UG-3086)*
0x4	Admin-local	Smallest scope that must be configured by administrator - that is, not automatically derived from physical connectivity or other settings
0x5	Site-local	Spans the same region as the site unicast scope, described in the *JenNet-IP LAN/WAN Stack User Guide (JN-UG-3086)*
0x8	Organisation-local	Spans multiple sites of a single organisation
0xE	Global	Unrestricted span, as described in the *JenNet-IP LAN/WAN Stack User Guide (JN-UG-3086)*

Table 21: Multicast Address Scopes

All other Scope values are reserved

G.3.4 OND Module

The OND (Over-Network Download) module has a module ID of 0xFFFFFF03 and contains information about over-network downloads to the local node. The MIB variables for this module are detailed below.

Index	Variable Name	Type	Access	Description
0	Images	blob table	R	Indicates the firmware images present on the node. Each entry in the table is a blob of the format detailed in Figure 28 below
1	DeviceID	uint32	R/W	Indicates the Device ID to request when a download is started using the Download variable (below). Defaults to the Device ID of the image currently running
2	ChipSet	uint16	R/W	Indicates the chipset to request when a download is started using the Download variable (below). Defaults to the chipset of the image currently running
3	Revision	uint16	R/W	Indicates the image revision to request when a download is started using the Download variable (below). Defaults to the revision of the image currently running
4	Download	uint8	R/W	An image download can be started by writing any value to this variable. The values of the DeviceID, ChipSet and Revision variables (above) will be used to request the appropriate image
5	LoadImage	uint8	R/W	The Images table index of the image to be selected and run - writing this index to the variable starts the transfer to RAM

Table 22: OND Module Variables

0	----	3	4	5	6	7	8	9	10	11	12
Device ID			Chipset		Revision		Blocks Remaining		Total Blocks		Status

Figure 28: Format of Images Blob

Status: 0 - Blank (no image currently present in this block)
1 - Current (image currently loaded and executing)
2 - Loading (image is in process of being loaded into device)
3 - Valid (image is complete and has passed checksum validation)

G.3.5 DeviceID Module

The DeviceID module has a module ID of 0xFFFFFF04, and contains the Device ID of the local node and the IDs of the device types that the node can implement. The MIB variables for this module are detailed below.

Index	Variable Name	Type	Access	Description
0	DeviceID	uint32	Const	32-bit identifier of the device (identifying the make and model). Device ID is described in Appendix D.1
1	DeviceTypes	blob	Const	Blob representing the device types that the node can implement. Each device type is represented by a 16-bit Device Type ID, as described in Appendix D.2. The blob is of the format detailed in Figure 29 below.

Table 23: DeviceID Module Variables

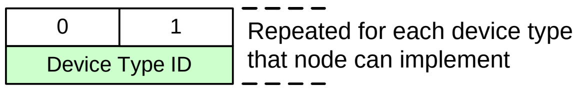


Figure 29: Format of DeviceTypes Blob

The possible standard device types are:

0x0001: Border-Router

G.4 Standard Commands

The JIP layer interacts with a module via the MIB variables associated with the module. This section details the standard commands for accessing these variables (the MIB variables for the standard modules are detailed in Appendix G.3).

All JIP commands have the general format illustrated below in Figure 30.

0	1	2	3	– – – – – – – –	x + 2
Version	Code	Handle	Payload (x bytes)		

Figure 30: Format of JIP Command

- **Version** is the JIP layer version number - it used to enable future upgrades to be backwards compatible and to provide an extra degree of validity checking (but is currently always set to 0)
- **Code** is the command code (from those listed in Table 24)
- **Handle** is the unique identifier for the command (request and response) and is user-defined, except bit 7 is reserved as a ‘stay awake’ flag (see Note 2 below)
- **Payload** is the variable-length payload (x bytes long) of the command

The standard JIP commands are listed and outlined in the table below.

Command	Code	Description
‘Get’ request	0x10	Request to obtain the value of a variable
‘Get by ID’ request	0x1C	Request to obtain the value of a variable in the module with specified ID
‘Get’ response	0x11	Response to a previous ‘Get’ or ‘Get by ID’ request
‘Set’ request	0x12	Request to set the value of a variable
‘Set by ID’ request	0x1D	Request to set the value of a variable in the module with specified ID
‘Set’ response	0x13	Response to a previous ‘Set’ or ‘Set by ID’ request
‘Query Modules’ request	0x14	Request to query the JenNet-IP database for a list of available modules
‘Query Modules’ response	0x15	Response to a previous ‘Query Modules’ request
‘Query Variables’ request	0x16	Request to query the JenNet-IP database for a list of available variables within a module
‘Query Variables’ response	0x17	Response to a previous ‘Query Variables’ request
‘Trap’ request	0x18	Request to enable the generation of a notification when a variable changes value
‘Untrap’ request	0x19	Request to disable the generation of a notification when a variable changes value

Table 24: Standard Commands

Command	Code	Description
‘Trap’ response	0x1A	Response to a previous ‘Trap’ request
‘Trap’ notification	0x1B	Notification of a change in a trapped variable

Table 24: Standard Commands

The formats of the above commands are detailed in the sub-sections below, which include descriptions of the payload fields. The enumeration values for some of the fields are detailed in Appendix G.6.

Note 1: A JIP command is embedded in an IEEE 802.15.4 MAC frame for transportation between wireless nodes, as described in Appendix F.

Note 2: Bit 7 of the handle is reserved as a ‘stay awake’ flag which is set by the sender of the packet to request a target End Device to stay awake in order to receive further packets. When a packet with this bit set is received by an End Device, the user-defined callback function **vJIP_StayAwakeRequest()** is invoked. For more information, refer to Section 4.6.3.

G.4.1 ‘Get’ Request

The ‘Get’ request can take one of the following two formats:

0	1	2	3	4	5
Version	Code	Handle	Payload (3 bytes)		
0x00	0x10	User-supplied	Module Index	First Variable Index	Variable Count

0	1	2	3	4	5	6	7
Version	Code	Handle	Payload (5 bytes)				
0x00	0x10	User-supplied	Module Index	Variable Index	First Table Entry		Entry Count

Figure 31: Formats of ‘Get’ Request

First Table Entry and Entry Count are optional and default to 0 and 255, respectively, if not present. This is to maintain backwards compatibility with non-table variables.

G.4.2 'Get by ID' Request

The 'Get by ID' request can take one of the following two formats:

0	1	2	3	4	5	6	7	8
Version	Code	Handle	Payload (6 bytes)					
0x00	0x1C	User-supplied	Module ID				First Variable Index	Variable Count

0	1	2	3	4	5	6	7	8	9	10
Version	Code	Handle	Payload (8 bytes)							
0x00	0x1C	User-supplied	Module ID				Variable Index	First Table Entry		Entry Count

Figure 32: Formats of 'Get by ID' Request

First Table Entry and Entry Count are optional and default to 0 and 255, respectively, if not present. This is to maintain backwards compatibility with non-table variables.

G.4.3 'Get' Response

The 'Get' response is always of the following format:

0	1	2	3	4	5		x + 2
Version	Code	Handle	Payload (x bytes)				
0x00	0x11	User-supplied	Module Index	First Variable Index	Variable Records		

Figure 33: Format of 'Get' Response

The Variable Records field of the payload contains a series of sub-fields relating to the variable to which the response applies (or the Variable Records field may be omitted):

0	1	2
Status	Type	Value
Success	(u)int8	

0	1	2	3
Status	Type	Value	
Success	(u)int16		

0	1	2	3	4	5
Status	Type	Value			
Success	(u)int32				

0	1	2	– – – – – – – – – – – – – –	9
Status	Type	Value		
Success	(u)int64			

0	1	2	3	4	5
Status	Type	Value			
Success	Float				

0	1	2	– – – – – – – – – – – – – –	9
Status	Type	Value		
Success	Double			

0	1	2	3	– – – – – – – – –	y + 2
Status	Type	String Length (y)	String (y bytes)		
Success	String				

0	1	2	3	– – – – – – – – –	y + 2
Status	Type	Blob Length (y)	Data (y bytes)		
Success	Blob				

0	1	2	3	4	5
Status	Type	Number of Remaining Entries		Table Version	
Success	Table				

Figure 34: Possible Formats Get Response of 'Variable Records' Field

In the case of a 'Get' response which relates to a Table variable (last response format in Figure 34), the first six bytes shown above may be followed by information on one or more table entries, where each entry is reported in the following format:

0	1	2	3	– – – – – – – – –	z + 2
Entry Index		Blob Length (z)	Data (z bytes)		

Figure 35: Table Entry Format in 'Get' Response

If a ‘Get’ response reports a failure to access the target variable, the Variable Records field contains only the Status sub-field, which is set to one of the following (for status enumerations, refer to Appendix G.6.3):

- Bad module index: Specified module index was out-of-range
- Bad variable index: Specified variable index was out-of-range for given module
- Disabled: Remote variable has been disabled, so cannot be read
- Error: Module index or variable index is not valid

G.4.4 ‘Set’ Request

The ‘Set’ request is always of the following format:

0	1	2	3	4	5	– – – – – – – –	x + 2
Version	Code	Handle	Payload (x bytes)				
0x00	0x12	User-supplied	Module Index	Variable Index	Variable Record		

Figure 36: Format of ‘Set’ Request

The Variable Record field of the payload contains a series of sub-fields relating to the variable to which the request applies. The possible formats for this field are shown in Figure 37 below.

0	1	2
Unused	Type	Value
	(u)int8	

0	1	2	3
Unused	Type	Value	
	(u)int16		

0	1	2	3	4	5
Unused	Type	Value			
	(u)int32				

0	1	2	– – – – – – – – – – – – – –	9
Unused	Type	Value		
	(u)int64			

0	1	2	3	4	5
Unused	Type	Value			
	Float				

0	1	2	– – – – – – – – – – – – – –	9
Unused	Type	Value		
	Double			

0	1	2	3	– – – – – – – – –	y + 2
Unused	Type	String Length (y)	String (y bytes)		
	String				

0	1	2	3	– – – – – – – – –	y + 2
Unused	Type	Blob Length (y)	Data (y bytes)		
	Blob				

Figure 37: Possible Formats of Set Request 'Variable Records' Field

G.4.5 'Set by ID' Request

The 'Set by ID' is always of the following format:

0	1	2	3	4	5	6	7	8		x + 2
Version	Code	Handle	Payload (x bytes)							
0x00	0x1D	User-supplied	Module ID				Variable Index	Variable Record		

Figure 38: Format of 'Set by ID' Request

The Variable Record field of the payload contains a series of sub-fields relating to the variable to which the request applies. The possible formats for this field are as shown for 'Set' request in Figure 37 above.

G.4.6 'Set' Response

The 'Set' response is always of the following format:

0	1	2	3	4	5
Version	Code	Handle	Payload (3 bytes)		
0x00	0x13	User-supplied	Module Index	Variable Index	Status Code

Figure 39: Format of 'Set' Response

The status code can be any of the following (for status enumerations, refer to Appendix G.6.3):

- Success: The request was successfully processed
- Bad module index: Specified module index was out-of-range
- Bad variable index: Specified variable index was out-of-range for given module
- Access not allowed: Variable is read-only or const
- Bad buffer size:
  - for string types, string is too long
  - for blob types, data is of incorrect length
- Wrong type: Specified variable type did not match that of variable
- Disabled: Specified variable has been disabled and so cannot be set
- Error: Unknown error

G.4.7 'Query Modules' Request

The 'Query Modules' request is always of the following format:

0	1	2	3	4
Version	Code	Handle	Payload (2 bytes)	
0x00	0x14	User-supplied	First Module Index	Required no. of Records

Figure 40: Format of 'Query Modules' Request

Note the following:

- 'First Module Index' can be set to 0x00 to indicate 'start at the first available value'
- 'Required Number of Records' can be set to 0xFF to indicate 'return as many records as possible'
- The number of records returned will be limited by the size of the UDP payload

G.4.8 ‘Query Modules’ Response

The ‘Query Modules’ response is always of the following format:

0	1	2	3	4	5	6	– – – – – – – –	x + 2
Version	Code	Handle	Payload (x bytes)					
0x00	0x15	User-supplied	Status Code	No. of Records Returned	No. of Records Left	List of Module Records		

Figure 41: Format of ‘Query Modules’ Response

The status code can be any of the following (for status enumerations, refer to Appendix G.6.3):

- Success: The request was successfully processed
- Bad module index: Specified ‘First Module Index’ is out-of-range
- Error: Unknown error

Each ‘Query Modules’ request results in a single ‘Query Modules’ response, in which several modules may be reported. The ‘List of Records’ field contains the returned records (if any) containing the names of the reported modules, with each record having the following format:

0	1	2	3	4	5	6	– – – – – – – – –	y + 5
Module Index	Module ID				String Length (y)	Module Name String (y bytes)		

Figure 42: Format of Returned ‘Module Record’

In order to discover all of the modules on a remote node, multiple ‘Query Modules’ requests may be needed. The requesting node will wait for each response then issue a further request until all modules have been reported (number of records left is zero).

G.4.9 ‘Query Variables’ Request

The ‘Query Variables’ request is always of the following format:

0	1	2	3	4	5
Version	Code	Handle	Payload (3 bytes)		
0x00	0x16	User-supplied	Module Index	First Variable Index	Required no. of Records

Figure 43: Format of ‘Query Variables’ Request

Note the following:

- ‘First Variable Index’ can be set to 0x00 to indicate 'start at the first available value'
- ‘Required Number of Records’ can be set to 0xFF to indicate 'return as many records as possible'
- The number of records returned will be limited by the size of the UDP payload

G.4.10 ‘Query Variables’ Response

The ‘Query Variables’ response is always of the following format:

0	1	2	3	4	5	6	7	---------	x + 2
Version	Code	Handle	Payload (x bytes)						
0x00	0x17	User-supplied	Status Code	Module Index	No. of Records Returned	No. of Records Left	List of Variable Records		

Figure 44: Format of ‘Query Variables’ Response

The status code can be any of the following (for status enumerations, refer to Appendix G.6.3):

- Success: The request was successfully processed
- Bad module index: Specified module index is out-of-range
- Bad variable index: Specified ‘First Variable Index’ is out-of-range for the specified module
- Error: Unknown error

Each ‘Query Variables’ request results in a single ‘Query Variables’ response, in which several variables may be reported. The ‘List of Records’ field contains the returned records (if any) containing the names of the reported variables, with each record having the following format:

0	1	2	---------	y + 2	y + 3	y + 4
Variable Index	String Length (y)	Variable Name String (y bytes)		Variable Type	Access Type	Security

Figure 45: Format of Returned ‘Variable Record’

In order to discover all of the variables of a module on a remote node, multiple ‘Query Variables’ requests may be needed. The requesting node will wait for each response then issue a further request until all of the module’s variables have been reported (number of records left is zero).

G.4.11 'Trap' Request

The 'Trap' request is always of the following format:

0	1	2	3	4	5
Version	Code	Handle	Payload (3 bytes)		
0x00	0x18	User-supplied	Notify Handle	Module Index	Variable Index

Figure 46: Format of 'Trap' Request

The originator of the request will be subsequently be notified of any change in the value of the trapped variable (specified by the Module Index and Variable Index values) via a 'Trap Notification' message - see Appendix G.4.14. The 'Notify Handle' will be passed as the user-supplied handle to the 'Trap Notification' message(s). The JenNet-IP stack keeps a record of the IP address and port number of all the nodes that have requested a trap.

G.4.12 'Untrap' Request

The 'Untrap' request is always of the following format:

0	1	2	3	4	5
Version	Code	Handle	Payload (3 bytes)		
0x00	0x19	User-supplied	Unused	Module Index	Variable Index

Figure 47: Format of 'Untrap' Request

The originator of the request will subsequently be no longer notified of a change in the value of the variable specified by the Module Index and Variable Index values.

G.4.13 'Trap' Response

The 'Trap' response is always of the following format:

0	1	2	3	4	5
Version	Code	Handle	Payload (3 bytes)		
0x00	0x20	User-supplied	Module Index	Variable Index	Status Code

Figure 48: Format of 'Trap' Response

This is a response to a 'Trap' or 'Untrap' request and is sent to the notification address specified in the request.

The status code can be any of the following (for status enumerations, refer to Appendix G.6.3):

- Success: The request was successfully processed
- Bad module index: Specified module index was out-of-range
- Bad variable index: Specified variable index is out-of-range for given module
- Disabled: Remote variable has been disabled (a notification will, however, be issued with a status of "Success" when the variable is enabled).
- Error: Remote node is unable to store IP address and port number

G.4.14 Trap Notifications

A Trap Notification is sent out each time a trapped variable is updated. The Trap Notification takes one of the following formats:

0	1	2	3	4	5	6	7
Version	Code	Handle	Payload (5 bytes)				
0x00	0x1B	User-supplied	Module Index	Variable Index	Status	Type	Value (1 byte)
					Success	(u)int8	

0	1	2	3	4	5	6	7	8
Version	Code	Handle	Payload (6 bytes)					
0x00	0x1B	User-supplied	Module Index	Variable Index	Status	Type	Value (2 bytes)	
					Success	(u)int16		

0	1	2	3	4	5	6	7	8	9	10
Version	Code	Handle	Payload (8 bytes)							
0x00	0x1B	User-supplied	Module Index	Variable Index	Status	Type	Value (4 bytes)			
					Success	(u)int32				

0	1	2	3	4	5	6	7	--------	14
Version	Code	Handle	Payload (12 bytes)						
0x00	0x1B	User-supplied	Module Index	Variable Index	Status	Type	Value (8 bytes)		
					Success	(u)int64			

0	1	2	3	4	5	6	7	8	9	10
Version	Code	Handle	Payload (8 bytes)							
0x00	0x1B	User-supplied	Module Index	Variable Index	Status	Type	Value (4 bytes)			
					Success	Float				

0	1	2	3	4	5	6	7	--------	14
Version	Code	Handle	Payload (12 bytes)						
0x00	0x1B	User-supplied	Module Index	Variable Index	Status	Type	Value (8 bytes)		
					Success	Double			

0	1	2	3	4	5	6	7	8	---	x + 7
Version	Code	Handle	Payload (x + 5 bytes)							
0x00	0x1B	User-supplied	Module Index	Variable Index	Status	Type	String Length (x)	String (x bytes)		
					Success	String				

0	1	2	3	4	5	6	7	8	---	x + 7
Version	Code	Handle	Payload (x + 5 bytes)							
0x00	0x1B	User-supplied	Module Index	Variable Index	Status	Type	Blob Length (x)	Data (x bytes)		
					Success	Blob				

0	1	2	3	4	5	6
Version	Code	Handle	Payload (4 bytes)			
0x00	0x1B	User-supplied	Module Index	Variable Index	Status	Type
					Success	Table Blob

Figure 49: Formats of a Trap Notification

A Trap Notification of the following format is issued when the state of a trapped variable is changed to disabled (if the variable is later enabled, a notification in a format from Figure 49 will then be issued with a status of "Success"):

0	1	2	3	4	5
Version	Code	Handle	Payload (3 bytes)		
0x00	0x1B	User-supplied	Module Index	Variable Index	Status Disabled

Figure 50: Format of Trap Notification for Disabling Variable

G.5 Low-Energy Frames

A low-energy device (introduced in Section 3.9) transmits basic IEEE 802.15.4 frames (rather than JenNet-IP frames) that carry the minimum payload necessary to be useful and secure. The IEEE 802.15.4 frame format is depicted in Figure 20 on page 249.

The only JenNet-IP command that a low-energy device can issue is the ‘Set by ID’ command, in order to remotely set the value of a variable in a specified MIB. The frame payload must specify the MIB, the variable and the new value of the variable to set.

The format of the frame payload is shown in Figure 51 and the payload fields are described in Table 25 below.

0	1	2	3	4	5	6	7	8	----	x + 7	x + 8	x + 9
Frame Counter			MIB ID				Variable Index	Variable Value (x bytes)			MIC	

Figure 51: Payload Format for Low-Energy Frame

Field	Description
Frame Counter *	Three most significant bytes of 4-byte frame counter
MIB ID **	Identifier of MIB (module) containing the variable - see Appendix D.3
Variable Index **	Index of MIB variable to set
Variable Value **	New value of MIB variable
MIC	Message Integrity Code (for security check)

Table 25: Payload Fields of Low-Energy Frame

* Least significant byte of frame counter is included as sequence number in the frame header
** The MIB ID, variable index and variable value fields are encrypted

The payload is specified as an array in the structure `tsMacFrame` (see Appendix L.4.1) which is passed into the MicroMAC function **vMMAC_StartMacTransmit()**. This structure also contains information for the MAC header of the frame, such as the destination PAN ID and address. Alternatively, a frame can be transmitted by directly accessing the PHY layer of the MicroMAC stack. For full details of the MicroMAC, refer to Appendix L.

G.6 Enumerations

This section details the enumerations that are used in the JIP commands detailed in Appendix G.4.

G.6.1 Variable Type Enumerations

The following enumerations are used to represent variable types:

Value	**Type**
0x00	8-bit signed integer
0x01	16-bit signed integer
0x02	32-bit signed integer
0x03	64-bit signed integer
0x04	8-bit unsigned integer
0x05	16-bit unsigned integer
0x06	32-bit unsigned integer
0x07	64-bit unsigned integer
0x08	32-bit IEEE 754 float
0x09	64-bit IEEE 754 double
0x0A	Text string
0x0B	Binary blob
0x4B	Blob table

Table 26: Variable Type Enumerations

G.6.2 Access Type Enumerations

The following enumerations are used to represent variable access types:

Value	Module
0x00	Const
0x01	Read only
0x02	Read and write

Table 27: Access Type Enumerations

G.6.3 Status Enumerations

The following enumerations are used to represent the reported status:

Value	Status
0x00	Success
0x7F	Timeout
0x8F	Bad module index
0x9F	Bad variable index
0xAF	Access not allowed
0xBF	Bad buffer size
0xCF	Wrong type
0xDF	Value rejected
0xEF	Disabled
0xFF	Error

Table 28: Status Enumerations

H. 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 web application version 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/Browser.html

You can write your own versions of these applications using the Java JIP API and C JIP API, detailed in the *JenNet-IP LAN/WAN Stack User Guide (JN-UG-3086)*.

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**.

H.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

H.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 H.2.1.

Alternatively, an IPv4 connection can be used between the PC/workstation and WPAN. Configuring an IPv4 connection is described in Appendix H.2.2.

H.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 wireless network 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/Browser.html**
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.

H.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**.

I. Memory Heap

The JenNet-IP stack uses a memory heap, of which you may need to be aware during application development. This appendix outlines the use of this heap by JenNet-IP.

I.1 Heap Organisation and Use

The heap is a block of RAM which is made available to the stack layers so that, when they are first initialised, they can acquire some of the storage that they need. This method is used in preference to statically-allocated memory, as it allows the amount of required memory to be varied by the application without needing to rebuild the stack library files. Examples of heap-allocated storage include IEEE 802.15.4 MAC frame buffers, JenNet Routing tables and OND image data.

In JenNet-IP the heap is kept very simple, for efficiency, and one limitation is that allocated memory space cannot be freed and returned to the heap - once space has been allocated, it can never be recovered. As such, the heap is only accessed during initialisation and is not used for short-lived data storage. Stack layers keep a record of the memory areas that they have obtained from the heap and will re-use the same areas following a warm start.

When using OND (Chapter 10), the function **eOND_DevInit()** allocates a 1-Kbyte memory buffer from the heap for use by OND.

The space available for the heap is determined during the application build process. The heap simply takes all of the available RAM above the application and beneath a fixed reserved space for the processor stack, although it is possible for the processor stack, at run-time, to extend beyond this reserved space.

I.2 Heap Error Conditions

There are two possible error conditions related to the heap:

- *Not enough heap space for all of the requests from the stack*

  This will immediately result in a trap exception on a JN516x device.

- *Processor stack extends into allocated heap*

  By default, this has no immediate effect other than corrupting some of the data, which may result in unpredictable behaviour. However, for a more deterministic response, the application can intercept the heap allocation function and enable a stack overflow exception. This is done in the Application Note *JenNet-IP Smart Home (JN-AN-1162)*. Doing this will result in a stack overflow exception as soon as the processor stack runs into the allocated part of the heap.

Both error conditions are fatal, as they indicate that there is not enough space to run both the stack and the application.

Exception handling is described in Appendix K.

J. Example Over-Network Download (OND)

This appendix provides instructions for performing an example Over-Network Download (OND) based on the JN516x-EK001 Evaluation Kit, described in the *JN516x-EK001 Evaluation Kit User Guide (JN-UG-3093)*. These instructions refer to the JenNet-IP Browser, introduced in Section 3.2.4 and provided in the Linksys router of the evaluation kit. OND is described in general terms in Chapter 10.

This OND example is based on the JenNet-IP system illustrated below.

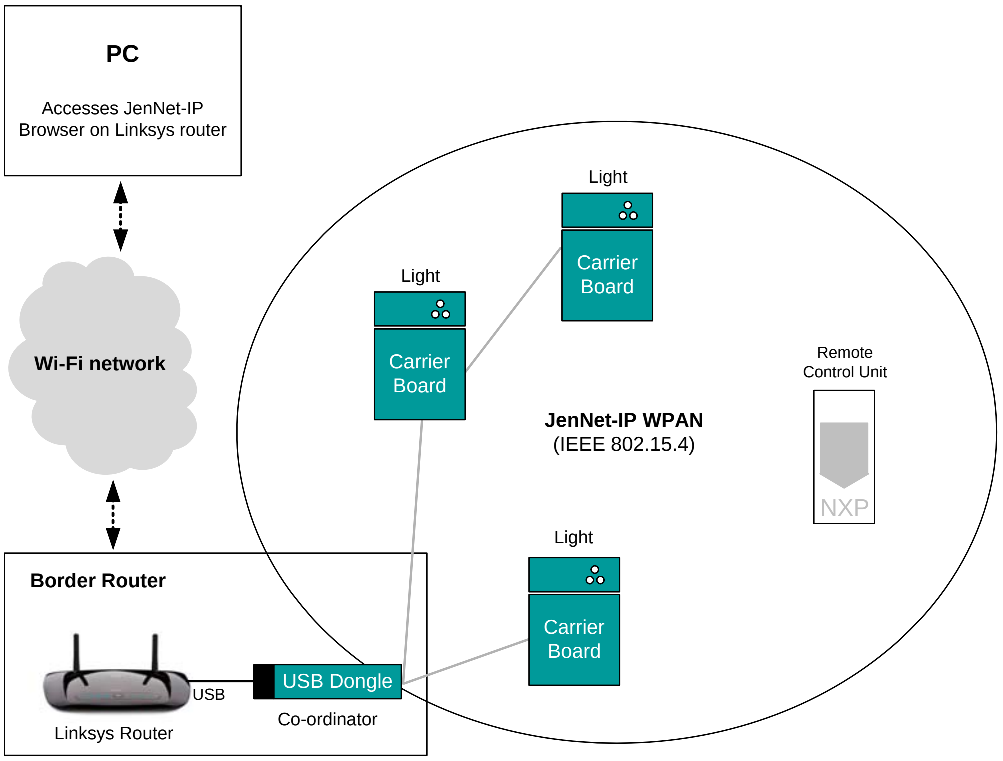


Figure 52: Example JenNet-IP System for OND

To download a new firmware image (for an upgrade or downgrade) from the PC to a node (light/bulb) of the WPAN:

1. Determine the version number of the firmware to be downloaded. The file should have the version number as the last field in the filename - for example, the version of the following firmware is 500:

0x11111111p_DeviceBulb_SSL2108_JN5168_v500.ond

2. Determine the version of the firmware which is currently present in the target device, as follows:

a) Access the OND MIB of the target device by directing the web browser on the PC to the following page of the JenNet-IP Browser:

http://192.168.11.1/Browser.html

b) Now click on the link to the target device.

c) Access the OND MIB and record the value of the “Revision” variable.

3. If the version number of the firmware to be downloaded is greater than the version number of the firmware already present in the device (an upgrade) then skip Step 4 and go straight to Step 5 - otherwise (for a downgrade), continue to Step 4.
4. For a downgrade, notify the target JenNet-IP device as to which version of firmware is to be downloaded by editing the device's OND MIB as follows:
   a) Enter the version number of the firmware to be downloaded (obtained in Step 1) in the "Revision" field of the OND MIB (read in Step 2) and click the **Update** button to the right of this field.
   b) Enter a '1' in the "Download" field and click the **Update** button to the right of this field.

> ***Caution:** Do not refresh this page after performing these steps, otherwise the download will fail.*

5. Initiate a firmware download from the Linksys router, as follows:
   a) Log into the Linksys router from a web browser on the PC using the following credentials:
   - **URL:** http://192.168.11.1
   - **Username:** root
   - **Password:** snap

   b) Navigate to the JenNet-IP page and then select the **Firmware** option.
   c) Choose a file to be uploaded using the **Choose File** button.
   d) Upload this file to the router by clicking the **Upload new firmware** button.
   e) Select the file to be downloaded to the target device using the **Distribute** radio button next to the appropriate image.
   f) Ensure that the **Automatic reset of Nodes** checkbox is:
   - ticked for a firmware upgrade
   - NOT ticked for a firmware downgrade

   g) Start the firmware distribution by clicking the **Begin Distribution** button.
   h) In the case of a downgrade, when the download is complete, check that all data was successfully received by the target device. This can be done be by viewing the OND MIB of the device. Under the "Images" section, two entries should be present - the current image and the newly downloaded image. The last two digits of each entry is the index number of the image.
   i) In the case of a downgrade, instruct the device to switch to the newly loaded image by setting the "LoadImage" variable in the OND MIB to the index number of the new image (determined in the previous step) and clicking the **Update** button to the right of this field.

   The device will now reboot using the new image. Success can be determined by re-examining the OND MIB "Revision" field, as described in Step 2c.

K. Exception Handling

This Appendix provides guidance on handling exceptions in JenNet-IP applications.

An exception is generally a situation in which the processor is unable to continue running the application. When this happens, the processor jumps to a specific memory address, which is in the boot sector of internal Flash memory of the JN516x device.

K.1 Exception Types

The most important exceptions are listed and outlined in Table 29 below:

Exception	Description
Bus error	The processor has tried to read from or write to a memory address that is not valid. Valid addresses are those for the Flash memory, RAM and peripherals. Trying to access other memory addresses, or peripherals that have not been enabled, will result in a ‘bus error’ exception.
Unaligned access	The processor can only access data if it is naturally aligned. This means that a 32-bit word must be on a 32-bit address boundary (least significant 2 bits of address are 0), a 16-bit half-word must be on a 16-bit address boundary (least significant bit of address is 0) and an 8-bit byte can be at any address boundary (all bits of address are used). Attempting to access data with a mis-aligned address will generate an ‘unaligned access’ exception. This may occur if the application attempts to use a byte array as a 32-bit word, for example.
Illegal instruction	The instruction that the processor has read from memory is not valid. The most common cause of this is jumping to the wrong location - for example, if a callback function is not registered correctly.
Stack overflow	The processor stack starts at the top (highest address) in RAM and moves down through the RAM as required by functions for temporary storage. It is possible to set a bottom limit on how far down the stack can extend - if this point is reached, a ‘stack overflow’ exception occurs. The limit is set to the optimum value by the heap manager in JenNet-IP applications so if this exception occurs, it means that the application has run out of RAM.
System Call	This can only be generated intentionally by hand-crafted assembler code and it should not be necessary for an application to handle it.
Trap	This can be generated by hand-crafted assembler code or debuggers. On the JN516x device, it is also generated by the heap manager if the allocated heap space is exhausted.

Table 29: Main Exceptions

Usually the above exceptions are unrecoverable errors, so the best response in a deployed system is to reset the device. During application development, however, it can be useful to process exceptions to help determine the cause of the problem.

K.2 Exception Handlers

All NXP JN51xx devices include a default handler for each of the exceptions listed in Appendix K.1. These default handlers store the current state of the device (the stack frame - see later) and implement hooks that allow the application to provide additional handlers, but otherwise they do nothing. As such, if the application does not provide an additional handler for a particular exception, the default handler will cause the device to hang when the exception is generated.

The applications in the Application Note *JenNet-IP Smart Home (JN-AN-1162)* provide additional handlers that store the exception information in Flash memory and then reset the device, which is the most effective response in a production component. Once the contents of the Application Note have been installed, the handler code can be found in the directory:

Applications/JN-AN-1162-JenNet-IP-Smart-Home/Common/Exception.c.

The process of registering a user-defined exception handler is described in Appendix K.3 below.

K.3 Handler Registration

For the JN516x device, a user-defined exception handler can be registered by the application simply by providing a function with a specific name - it will then be automatically used. The function prototypes are:

```
void vException_BusError(uint32 u32StackPointer, uint32 u32Vector);
void vException_UnalignedAccess (uint32 u32StackPointer, uint32 u32Vector);
void vException_IllegalInstruction (uint32 u32StackPointer, uint32
u32Vector);
void vException_SysCall (uint32 u32StackPointer, uint32 u32Vector);
void vException_Trap (uint32 u32StackPointer, uint32 u32Vector);
void vException_StackOverflow(uint32 u32StackPointer, uint32 u32Vector);
```

In all the above cases, the two parameters passed to the handler are:

- `u32StackPointer`: This is the stack pointer, which is the address of the base of the stack frame. The stack frame can be used to further analyse the cause of the exception, if required. Details of the stack frame are provided in Appendix K.4.
- `u32Vector`: The exception vector, which is the exception number. The exception vector is also available within the stack frame, and is provided to allow the application to use the same handler for several exceptions, if required.

K.4 Stack Frame

When an exception occurs, the default handler stores the stack frame on the processor stack in RAM. The contents of the stack frame are as indicated in Table 30.

Address Offset (from base address of stack frame)	**Value**	**Description**	**Comments**
0x00	r0	Contents of register 0	Always has value 0, but stored for consistency with debuggers
0x04	r1	Contents of register 1	Contains the stack pointer at the time that the exception occurred
0x08	r2	Contents of register 2	General-purpose register
: :	: :	: :	: :
0x20	r8	Contents of register 8	General-purpose register
0x24	r9	Contents of register 9	Contains the return address for the last function called prior to the exception
0x28	r10	Contents of register 10	General-purpose register
: :	: :	: :	: :
0x3C	r15	Contents of register 15	General-purpose register
0x40	Vector	Exception number	Values are: • bus error - 0x02 • unaligned access - 0x06 • illegal instruction - 0x07 • syscall - 0x0C • trap - 0x0E • stack overflow - 0x10
0x44	ESR	Status register	Contents of processor’s status register at time of exception
0x48	EPCR	Program counter	Value of program counter (execution address) at time of exception
0x4C	EEAR	Effective address	Effective address at time of exception. This value is valid if the processor was attempting to read or write data when the exception occurred, and is the address of the data that it was trying to access

Table 30: Stack Frame

The remaining RAM from offset 0x50 to the end of RAM is the processor call stack. With a disassembly of the application, it is possible to trace the execution all the way back to the most recent reset.

L. MicroMAC for Low-Energy Devices

A low-energy device (introduced in Section 3.9) does not require any JenNet-IP software components, as it transmits simple IEEE 802.15.4 frames (rather than JenNet-IP frames). Therefore, the software required by a low-energy device is an application and the IEEE 802.15.4 stack.

A special version of the IEEE 802.15.4 stack can be employed in which the MAC layer is replaced with an NXP-adapted ‘MicroMAC’ layer in order to minimise the energy required for frame transmissions (particularly useful for energy-harvesting nodes) and to reduce application code size. The MicroMAC stack is depicted in Figure 53 below.

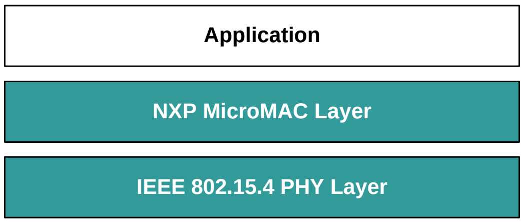


Figure 53: MicroMAC Stack

This appendix describes the NXP MicroMAC software, which comprises the MicroMAC stack and the MicroMAC Application Programming Interface (API) containing C functions for use in application development. This software is provided in the *JN516x JenNet-IP SDK (JN-SW-4165)*.

Note: The MicroMAC software is only supported on the JN516x range of microcontrollers and can be used on any device in the range: JN5168, JN5164 or JN5161.

L.1 Enabling the MicroMAC

In order to use the MicroMAC stack, it must be enabled for the application on the source node as follows:

- In the application’s makefile, add the MicroMAC library in the ‘Additional libraries’ section, as shown below:

```
# Application libraries
# Specify additional Component libraries
APPLIBS += MMAC
```

- Also in the makefile, set the stack parameter as follows:

```
JENNIC_STACK = None
```

- In the application code, reference the header file **MMAC.h**, as shown below:

```
#include "MMAC.h"
```

- In the application code, call **vMMAC_Enable()** as the first MicroMAC API function (see Section L.2.1 and Section L.3.1)

L.2 Application Coding for the MicroMAC

This section describes the function calls that are required in an application in order to use the MicroMAC to transmit and receive frames. The descriptions are organised in the following sub-sections:

- Initialisation - see Section L.2.1
- Transmitting frames - see Section L.2.2
- Receiving frames - see Section L.3.3

The referenced MicroMAC API functions are fully detailed in Section L.3.

Note: While receiving frames is supported by the MicroMAC, this feature is not currently used with a JenNet-IP system.

L.2.1 Initialisation

In order to initialise the MicroMAC, the first function that must be called is **vMMAC_Enable()**. This function enables the MAC hardware block on the JN516x device.

Next, MicroMAC interrupts should be enabled using the function **vMMAC_EnableInterrupts()**. This will allow interrupts to be generated to inform the application when frames have been transmitted and/or received. The above function requires a user-defined interrupt handler function to be specified, which will be automatically invoked when a MicroMAC interrupt occurs. For the prototype of this interrupt handler, refer to the description of **vMMAC_EnableInterrupts()** on page 290.

The radio transceiver of the JN516x device must then be set up by calling two functions:

- **vMMAC_ConfigureRadio()** must first be called to configure and calibrate the radio transceiver
- **vMMAC_SetChannel()** must then be called to select the IEEE 802.15.4 2.4-GHz channel on which the transceiver will operate (in the range 11-26)

The JN516x device is then ready to transmit and receive frames, as described in Appendix L.2.2 and Appendix L.2.3.

The above functions are fully detailed in Section L.3.1.

L.2.2 Transmitting Frames

A frame can be transmitted using the function **vMMAC_StartMacTransmit()**. When calling this function, a number of options are available and all these options require pre-configuration (before the above transmit function is called).

The transmit options and the necessary pre-configuration are as follows:

- **Delayed transmission**

  This option allows the transmission to be delayed until a certain 'time'. This time is represented by a value of the free-running 62500-Hz internal clock. Use of this feature requires the following pre-configuration:

  a) The timing function **u32MMAC_GetTime()** must first be called to obtain the current value of the internal clock.

  b) The function **vMMAC_SetTxStartTime()** must then be immediately called to specify the 'time' at which the next transmission should occur. This 'time' should be calculated by adding the 'current time' (obtained above) to the required delay (as a number of clock cycles).

- **Automatic acknowledgements**

  This option requests the transmitted frame to be acknowledged by the recipient. If no acknowledgement is received, the frame will be re-transmitted. Use of this feature requires pre-configuration by calling the **vMMAC_SetTxParameters()** function, in which the number of attempts to transmit a frame without an acknowledgement must be specified.

- **Clear Channel Assessment (CCA)**

  This option allows CCA to be implemented so that the transmission will only be performed when the relevant radio channel is clear of other traffic (for the details of CCA, refer to the IEEE 802.15.4 Specification). Use of this feature requires pre-configuration by calling the **vMMAC_SetTxParameters()** function, in which the following values must be specified:

  - Minimum and maximum values for the Back-off Exponent (BE)
  - Maximum number of back-offs (before the transmission is abandoned)

Once **vMMAC_StartMacTransmit()** has been called and the transmission has completed, an E_MMAC_INT_TX_COMPLETE interrupt is generated and the registered interrupt handler is invoked. This interrupt only indicates that the transmission attempt has completed and not that it has been successful. The function **u32MMAC_GetTxErrors()** can then be used to check for transmission errors.

Note: The function **vMMAC_StartPhyTransmit()** can be used as an alternative to the function **vMMAC_StartMacTransmit()**. The alternative function provides direct access to the PHY layer of the stack, if required. However, the 'automatic acknowledgements' option is not available with this function. MAC and PHY modes are described in Section L.6.

The above functions are fully detailed in Section L.3.2, except the timing function which is detailed in Section L.3.4.

The format of a frame that is transmitted from a low-energy device which uses the MicroMAC is described in Appendix G.5.

L.2.3 Receiving Frames

While receiving frames is supported by the MicroMAC, this feature is not currently used with a JenNet-IP system. The feature is described here for completeness.

A frame can be received using the function **vMMAC_StartMacReceive()**, which enables the radio receiver until a frame has arrived and been received. When calling this function, a number of options are available and some of these options require pre-configuration (before the above receive function is called).

The receive options and the necessary pre-configuration (if any) are as follows:

- **Delayed receive**

  This option allows enabling the radio receiver to be delayed until a certain 'time'. This time is represented by a value of the free-running 62500-Hz internal clock. Use of this feature requires the following pre-configuration:

  a) The timing function **u32MMAC_GetTime()** must first be called to obtain the current value of the internal clock.

  b) The function **vMMAC_SetRxStartTime()** must then be immediately called to specify the 'time' at which the receiver should be enabled. This 'time' should be calculated by adding the 'current time' (obtained above) to the required delay (as a number of clock cycles).

- **Automatic acknowledgements**

  This option allows automatic acknowledgements to be sent. If this option is enabled and an acknowledgement has been requested for a received frame, the stack will automatically return an acknowledgement to the sender of the frame.

- **Malformed frames**

  This option allows the rejection of received frames that appear to be malformed.

- **Frame Check Sequence (FCS) errors**

  This option allows the rejection of received frames that have FCS errors.

- **Address matching**

  This option allows the rejection of frames that do not contain the destination identifer values of the local node. These local node identifers must be pre-configured using the function **vMMAC_SetRxAddress()** and are as follows:

  - PAN ID of the network to which the local node belongs
  - 16-bit short address of the local node
  - 64-bit IEEE/MAC (extended) address of the local node

Once **vMMAC_StartMacReceive()** has been called and the receive has completed, the receiver is disabled and two interrupts are generated, with the registered interrupt handler invoked separately for each one:

- E_MMAC_INT_RX_HEADER signals the reception of the MAC header of the frame (but the interrupt is generated after receiving the whole frame)
- E_MMAC_INT_RX_COMPLETE signals the reception of the whole frame (but is generated after an acknowedgement has been sent, if requested/enabled)

These interrupts only indicate that the receive attempt has completed and not that it has been successful. The function **u32MMAC_GetRxErrors()** can then be used to check for receive errors.

Note: The function **vMMAC_StartPhyReceive()** can be used as an alternative to the function **vMMAC_StartMacReceive()**. The alternative function provides direct access to the PHY layer of the stack, if required. However, the ‘automatic acknowledgements’, ‘malformed frames’ and ‘address matching’ options are not available with this function. MAC and PHY modes are described in Section L.6.

The above functions are fully detailed in Section L.3.3, except the timing function which is detailed in Section L.3.4.

L.3 MicroMAC API

The MicroMAC library includes an API, comprising C functions for use by the application. These functions are divided into the following categories and detailed in the referenced sub-sections:

- Initialisation functions - see Section L.3.1
- Transmit functions - see Section L.3.2
- Receive functions - see Section L.3.3
- Timing function - see Section L.3.4
- Interrupt Control functions - see Section L.3.5
- Miscellaneous functions - see Section L.3.6

Note: The MicroMAC API is intentionally small and simple, in order to minimise the application size and also to minimise the run-time when used in energy-harvesting applications. Hence, the API functions do not carry out error checking or range checking on the values passed to them.

L.3.1 Initialisation Functions

The following Initialisation functions are provided in the MicroMAC API.

Function	Page
vMMAC_Enable	289
vMMAC_EnableInterrupts	290
vMMAC_ConfigureRadio	291
vMMAC_SetChannel	292

Note: All of the above functions must be called by the application, starting with the function **vMMAC_Enable()**. They should be called in the order in which they are listed above.

vMMAC_Enable

```
void vMMAC_Enable(void);
```

Description

This function enables the MAC hardware block and must be called before using any other MicroMAC functions.

After calling this function, the other MicroMAC Intialisation functions (described in this section) should be called.

Parameters

None

Returns

None

vMMAC_EnableInterrupts

```
void vMMAC_EnableInterrupts(void (*prHandler)(uint32));
```

Description

This function enables transmit and receive interrupts, and allows the application to register a user-defined callback function that will be invoked when a MicroMAC interrupt is generated.

The **uint32** value returned to the interrupt handler is a bitmap that indicates the nature of the MicroMAC interrupt. This value can be logical-ORed with the following enumerated values from `teIntStatus` to determine the type of interrupt:

- E_MMAC_INT_TX_COMPLETE (0x01)
- E_MMAC_INT_RX_HEADER (0x02)
- E_MMAC_INT_RX_COMPLETE (0x04)

For more information on these interrupt types, refer to Section L.5.5.

Parameters

prHandler	Pointer to the MicroMAC interrupt handler callback function

Returns

None

vMMAC_ConfigureRadio

```
void vMMAC_ConfigureRadio(void);
```

Description

This function configures and calibrates the radio transceiver on the JN516x device. It must be called before setting the channel (using **vMMAC_SetChannel()**) and before attempting to transmit or receive (using the functions detailed in Section L.3.2 and Section L.3.3).

Parameters

None

Returns

None

vMMAC_SetChannel

```
void vMMAC_SetChannel(uint8 u8Channel);
```

Description

This function sets the radio channel to be used by the radio transceiver. The required 2.4-GHz channel number in the range 11 to 26 must be specified.

The function must be called after the radio transceiver has been configured (using **vMMAC_ConfigureRadio()**).

Parameters

u8Channel	Required channel number in the range 11 to 26 (other values are not valid)

Returns

None

L.3.2 Transmit Functions

The following Transmit functions are provided in the MicroMAC API.

Function	Page
vMMAC_SetTxParameters	294
vMMAC_SetTxStartTime	295
vMMAC_StartMacTransmit	296
vMMAC_StartPhyTransmit	298
u32MMAC_GetTxErrors	299

vMMAC_SetTxParameters

```
void vMMAC_SetTxParameters(uint8 u8Attempts,
                           uint8 u8MinBE,
                           uint8 u8MaxBE,
                           uint8 u8MaxBackoffs);
```

Description

This function sets a number of transmit parameters in connection with 'automatic acknowledgements' and Clear Channel Assessment (CCA). These two features can be enabled for an individual 'MAC mode' transmission when the transmit function **vMMAC_StartMacTransmit()** is called. CCA can also be enabled for a 'PHY mode' transmission when the transmit function **vMMAC_StartPhyTransmit()** is called.

Note 1: The **vMMAC_SetTxParameters** function only needs to be called once on every cold or warm start **-** it does not need to be called for each transmit operation.

Note 2: The function does not need to be called if you are not going to use CCA or automatic acknowledgements (selected as options when calling the relevant transmit function).

When transmitting with automatic acknowledgements enabled, the transmitted frame must be acknowledged by the recipient. If no acknowledgement is received, the frame will be re-transmitted. The number of attempts to transmit a frame without an acknowledgement can be specified through the parameter *u8Attempts*.

The other three parameters are related to CCA (when enabled):

- Minimum and maximum values for the Back-off Exponent (BE) are specified through the parameters *u8MinBE* and *u8MaxBE*, respectively
- The maximum number of back-offs (before the transmission is abandoned) is specified through the parameter *u8MaxBackoffs*

For the details of CCA, refer to the IEEE 802.15.4 Specification. The above three function parameters correspond to the PIB attributes `macMinBE`, `macMaxBE` and `macMaxCSMABackoffs`, respectively, in the specification.

Parameters

u8Attempts	Maximum number of transmission attempts without receiving an acknowledgement
u8MinBE	Minimum value of Back-off Exponent to be used in CCA
u8MaxBE	Maximum value of Back-off Exponent to be used in CCA
u8MaxBackoffs	Maximum number of back-offs in CCA

Returns

None

vMMAC_SetTxStartTime

```
void vMMAC_SetTxStartTime(uint32 u32Time);
```

Description

This function sets the ‘time’ at which a transmission should begin. This time is specified as a value of the free-running 62500-Hz internal clock.

Before calling this function, the **u32MMAC_GetTime()** function should be called to obtain the current value of the clock. The application should then determine the required clock value to be specified in **vMMAC_SetTxStartTime()** in order to start the next transmission at the desired time.

If used, this function must be called before the relevant transmit function (**vMMAC_StartMacTransmit()** or **vMMAC_StartPhyTransmit()**), and a ‘delayed transmission’ must be enabled in the options specified in the transmit function. The transmitter will then be enabled and the transmission will be performed when the internal clock value matches the value specified in this function.

Note: This function only needs to be called if you are going to use the ‘delayed transmission’ feature (selected as an option when calling the relevant transmit function).

Parameters

u32Time	Internal clock value at which transmission should begin

Returns

None

vMMAC_StartMacTransmit

```
void vMMAC_StartMacTransmit(tsMacFrame *psFrame,
                            teTxOption eOptions);
```

Description

This function starts the transmitter in 'MAC mode', with the specified options, in order to transmit a frame. A pointer must be provided to the frame to be transmitted.

The MAC mode options relate to three features and are specified as enumerations:

Feature	Enumeration	Description
Delayed transmission	E_MMAC_TX_START_NOW	Start transmission as soon as this function is called
	E_MMAC_TX_DELAY_START	Start transmission at the time specified beforehand using **vMMAC_SetTxStartTime()**
Automatic acknowledgements and re-try	E_MMAC_TX_NO_AUTO_ACK	Do not enable automatic acknowledgements and re-try
	E_MMAC_TX_USE_AUTO_ACK	Enable automatic acknowl-edgements and re-try
Clear Channel Assessment (CCA)	E_MMAC_TX_NO_CCA	Do not enable CCA
	E_MMAC_TX_USE_CCA	Enable CCA
	E_MMAC_TX_USE_CCA_ ALIGNED	Enable CCA aligned to back-off clock (used in CAP period in beacon-enabled networks)

Enumerations for the three features must be combined in a logical-OR operation.

Note the following:

- If the 'delayed transmission' option is enabled, this feature should be pre-configured using the function **vMMAC_SetTxStartTime()**.
- If the automatic acknowledgements and/or CCA options are enabled, these features should be pre-configured using the function **vMMAC_SetTxParameters()**.

If interrupts have been enabled using **vMMAC_EnableInterrupts()**, an interrupt (E_MMAC_INT_TX_COMPLETE) will be generated once the transmission attempt has completed.

Parameters

psFrame	Pointer to a pre-filled structure containing the frame to be transmitted (see Section L.4.1)
eOptions	Value indicating the required features for this transmission (see above and Section L.5.1)

Returns

None

vMMAC_StartPhyTransmit

```
void vMMAC_StartPhyTransmit(tsPhyFrame *psFrame,
                            teTxOption eOptions);
```

Description

This function starts the transmitter in ‘PHY mode’, with the specified options, in order to transmit a frame. A pointer must be provided to the frame to be transmitted.

Note: This function provides direct access to the PHY layer of the stack. If you do not need this access, you should use the function **vMMAC_StartMacTransmit()** to transmit a frame.

The PHY mode options relate to two features and are specified as enumerations:

Feature	Enumeration	Description
Delayed transmission	E_MMAC_TX_START_NOW	Start transmission as soon as this function is called
	E_MMAC_TX_DELAY_START	Start transmission at the time specified beforehand using **vMMAC_SetTxStartTime()**
Clear Channel Assessment (CCA)	E_MMAC_TX_NO_CCA	Do not enable CCA
	E_MMAC_TX_USE_CCA	Enable CCA

Enumerations for the two features must be combined in a logical-OR operation.

Note the following:

- If the ‘delayed transmission’ option is enabled, this feature should be pre-configured using the function **vMMAC_SetTxStartTime()**.
- If the CCA option is enabled, this feature should be pre-configured using the function **vMMAC_SetTxParameters()**.

If interrupts have been enabled using **vMMAC_EnableInterrupts()**, an interrupt (E_MMAC_INT_TX_COMPLETE) will be generated once the transmission attempt has completed.

Parameters

psFrame	Pointer to a pre-filled structure containing the frame to be transmitted (see Section L.4.2)
eOptions	Value indicating the required features for this transmission (see above and Section L.5.1)

Returns

None

u32MMAC_GetTxErrors

```
uint32 u32MMAC_GetTxErrors(void);
```

Description

This function can be used to report any errors that have occurred during a frame transmission. It should only be called after the transmission has completed (indicated by an interrupt, if enabled).

The returned value is a bitmap that can be logical-ORed with the following enumerated values from `teTxStatus` to determine the error condition(s):

- E_MMAC_TXSTAT_CCA_BUSY (0x01)
- E_MMAC_TXSTAT_NO_ACK (0x02)
- E_MMAC_TXSTAT_ABORTED (0x04)

A returned value of 0 indicates no error.

For more information on the above error conditions, refer to Section L.5.2.

Parameters

None

Returns

32-bit bitmap indicating the errors that have occurred (see above)

L.3.3 Receive Functions

The following Receive functions are provided in the MicroMAC API.

Function	**Page**
vMMAC_SetRxAddress	301
vMMAC_SetRxStartTime	302
vMMAC_StartMacReceive	303
vMMAC_StartPhyReceive	305
bMMAC_RxDetected	306
u32MMAC_GetRxTime	307
u32MMAC_GetRxErrors	308

Note: While receiving frames is supported by the MicroMAC, this feature is not currently used with a JenNet-IP system. The Receive functions are described here for completeness.

vMMAC_SetRxAddress

```
void vMMAC_SetRxAddress(uint16 u16PanId,
                        uint16 u16Short,
                        MAC_ExtAddr_s *psMacAddr);
```

Description

This function configures settings for receiving frames when ‘address matching’ is enabled. Address matching can be enabled for ‘MAC mode’ when the receive function **vMMAC_StartMacReceive()** is called, but is not available for ‘PHY mode’.

The function specifies the following values for this purpose:

- PAN ID of the network to which the local node belongs
- 16-bit short address of the local node
- 64-bit IEEE/MAC (extended) address of the local node

Only received frames with destination parameters that match the values supplied to this function will be accepted.

Note 1: The **vMMAC_SetRxAddress()** function only needs to be called once on every cold or warm start - it does not need to be called for each receive operation.

Note 2: If receiving with address matching disabled or using ‘PHY mode’, the supplied values are ignored and so this function call is unnecessary.

Parameters

u16PanId	16-bit PAN ID of network to which local node belongs
u16Short	16-bit short address of local node
psMacAddr	Pointer to a structure containing 64-bit IEEE/MAC address of local node (see Section L.4.4)

Returns

None

vMMAC_SetRxStartTime

```
void vMMAC_SetRxStartTime(uint32 u32Time);
```

Description

This function sets the 'time' at which the receiver should be enabled. This time is specified as a value of the free-running 62500-Hz internal clock.

Before calling this function, the **u32MMAC_GetTime()** function should be called to obtain the current value of the clock. The application should then determine the required clock value to be specified in **vMMAC_SetRxStartTime()** in order to start the receiver at the desired time.

If used, this function must be called before the relevant receive function (**vMMAC_StartMacReceive()** or **vMMAC_StartPhyReceive()**), and a 'delayed receive' must be enabled in the options specified in the receive function. The receiver will then be enabled when the internal clock value matches the value specified in this function.

Note: This function only needs to be called if you are going to use the 'delayed receive' feature (selected as an option when calling the relevant receive function).

Parameters

u32Time	Internal clock value at which receiver should be enabled

Returns

None

vMMAC_StartMacReceive

**void vMMAC_StartMacReceive(tsMacFrame ****psFrame*,
teRxOption *eOptions***);**

Description

This function starts the receiver in ‘MAC mode’, with the specified options, in order to receive a frame. A pointer must be provided to a structure to which the received frame will be written.

The MAC mode options relate to six features and are specified as enumerations:

Feature	Enumeration	Description
Delayed receive	E_MMAC_RX_START_NOW	Start receiver as soon as this function is called
	E_MMAC_RX_DELAY_START	Start receiver at the time specified beforehand using **vMMAC_SetRxStartTime()**
Automatic acknowledgements	E_MMAC_RX_NO_AUTO_ACK	Do not enable automatic acknowledgements
	E_MMAC_RX_USE_AUTO_ACK	Enable automatic acknowl-edgements
Malformed frames	E_MMAC_RX_NO_MALFORMED	Reject frames that appear to be malformed
	E_MMAC_RX_ALLOW_MALFORMED	Accept frames that appear to be malformed
Frame Check Sequence (FCS) errors	E_MMAC_RX_NO_FCS_ERROR	Reject frames with FCS errors
	E_MMAC_RX_ALLOW_FCS_ERROR	Accept frames with FCS errors
Address matching	E_MMAC_RX_NO_ADDRESS_MATCH	Reject frames that do not match the node’s identifiers previously set with **vMMAC_SetRxAddress()**
	E_MMAC_RX_ADDRESS_MATCH	Accept frames that do not match the node’s identifiers previously set with **vMMAC_SetRxAddress()**

Feature	Enumeration	Description
Timing alignment	E_MMAC_RX_ALIGN_NORMAL	Send automatic acknowledgement 12 symbol periods after end of received frame
	E_MMAC_RX_ALIGNED	Send automatic acknowledgement on the next back-off clock (used in CAP period in beacon-enabled networks)

Enumerations for the six features must be combined in a logical-OR operation.

Note the following:

- If the 'delayed receive' option is enabled, this feature should be pre-configured using the function **vMMAC_SetRxStartTime()**.
- If the 'address matching' option is enabled, this feature should be pre-configured using the function **vMMAC_SetRxAddress()**.
- If the 'automatic acknowledgements' option is enabled, on receiving a frame the device will automatically send an acknowledgement frame.

Once a frame has been received, the receiver will be disabled and, if interrupts have been enabled using **vMMAC_EnableInterrupts()**, two successive interrupts (E_MMAC_INT_RX_HEADER and E_MMAC_INT_RX_COMPLETE) will be generated.

Parameters

psFrame	Pointer to a structure to receive the frame (see Section L.4.1)
eOptions	Value indicating the required receive features (see above and Section L.5.3)

Returns

None

vMMAC_StartPhyReceive

```
void vMMAC_StartPhyReceive(tsPhyFrame *psFrame,
                           teRxOption eOptions);
```

Description

This function starts the receiver in ‘PHY mode’, with the specified options, in order to receive a frame. A pointer must be provided to a structure to which the received frame will be written.

Note: This function provides direct access to the PHY layer of the stack. If you do not need this access, you should use the function **vMMAC_StartMacReceive()** to receive a frame.

The PHY mode options relate to two features and are specified as enumerations:

Feature	Enumeration	Description
Delayed receive	E_MMAC_RX_START_NOW	Start receiver as soon as this function is called
	E_MMAC_RX_DELAY_START	Start receiver at the time specified beforehand using **vMMAC_SetRxStartTime()**
Frame Check Sequence (FCS) errors	E_MMAC_RX_NO_FCS_ERROR	Reject frames with FCS errors
	E_MMAC_RX_ALLOW_FCS_ERROR	Accept frames with FCS errors

Enumerations for the two features must be combined in a logical-OR operation.

If the ‘delayed receive’ option is enabled, this feature should be pre-configured using the function **vMMAC_SetRxStartTime()**.

Once a frame has been received, the receiver will be disabled and, if interrupts have been enabled using **vMMAC_EnableInterrupts()**, two successive interrupts (E_MMAC_INT_RX_HEADER and E_MMAC_INT_RX_COMPLETE) will be generated.

Parameters

psFrame	Pointer to a structure to receive the frame (see Section L.4.1)
eOptions	Value indicating the required receive features (see above and Section L.5.3)

Returns

None

bMMAC_RxDetected

```
bool_t bMMAC_RxDetected(void);
```

Description

This function can be used to determine whether a frame is currently being received.

Note that the detection of a frame does not always result in the successful reception of the frame, since a frame can be discarded due to errors or poor reception quality.

Parameters

None

Returns

TRUE if receiving a frame, FALSE otherwise

u32MMAC_GetRxTime

```
uint32 u32MMAC_GetRxTime(void);
```

Description

This function can be used to obtain the time on the free-running 62500-Hz internal clock at which a frame was last received.

Note that the frame may not necessarily have been received succesfully, since a frame can be discarded due to errors or poor reception quality.

Parameters

None

Returns

Value of 62500-Hz internal clock when a frame was last received

u32MMAC_GetRxErrors

```
uint32 u32MMAC_GetRxErrors(void);
```

Description

This function can be used to report any errors that have occurred while receiving a frame. It should only be called after the frame has been received (indicated by an interrupt, if enabled).

The returned value is a bitmap that can be logical-ORed with the following enumerated values from `teRxStatus` to determine the error condition(s):

- E_MMAC_RXSTAT_ERROR (0x01)
- E_MMAC_RXSTAT_ABORTED (0x02)
- E_MMAC_RXSTAT_MALFORMED (0x20)

A returned value of 0 indicates no error.

For more information on the above error conditions, refer to Section L.5.4.

Parameters

None

Returns

32-bit bitmap indicating the errors that have occurred (see above)

L.3.4 Timing Function

The following Timing function is provided in the MicroMAC API.

Function	Page
u32MMAC_GetTime	310
vMMAC_SetCutOffTimer	311
vMMAC_SynchroniseBackoffClock	312

u32MMAC_GetTime

```
uint32 u32MMAC_GetTime(void);
```

Description

This function can be used to obtain the current 'time', based on the value of an internal clock which runs at 62500 Hz. The function is only useful when a 'delayed transmission' or 'delayed receive' is to be performed. The returned clock value can be used to determine the value to be specified in the function **vMMAC_SetTxStartTime()** or **vMMAC_SetRxStartTime()**, in order to start a transmission or receive at a certain time.

Parameters

None

Returns

Current value of 62500-Hz internal clock

vMMAC_SetCutOffTimer

```
void vMMAC_SetCutOffTimer(uint32 u32CutOffTime,
                          bool_t bEnable);
```

Description

This function can be used to set a cut-off time after which any pending transmit or receive operation will be automatically terminated. The cut-off time is specified as a value of the free-running 62500-Hz internal clock.

Parameters

u32CutOffTime	Value of 62500-Hz internal clock at which transmit/receive operation will be terminated
bEnable	Boolean allowing automatic Tx/Rx cut-off feature to be enabled/ disabled: TRUE - Enable FALSE - Disable

Returns

None

vMMAC_SynchroniseBackoffClock

void vMMAC_SynchroniseBackoffClock(bool_t *bEnable***);**

Description

This function can be used to configure the back-off clock to be synchronised to any subsequent received or transmitted frames. This feature is used in the CAP period in beacon-enabled networks (see the *IEEE 802.15.4 Stack User Guide (JN-UG-3024)*) and would normally be switched on for the transmission or reception of a regular beacon, then switched off for other transceiver operations.

Parameters

bEnable	Boolean allowing back-off clock synchronisation feature to be enabled/disabled: TRUE - Enable FALSE - Disable

Returns

None

L.3.5 Interrupt Control Functions

Interrupt Control functions are provided which allow fine control of the interrupt sources within the MicroMAC and also allow the MicroMAC to be used in polled mode (where interrupt processing has not been enabled) rather than in interrupt mode.

The following Interrupt Control functions are provided in the MicroMAC API.

Function	Page
vMMAC_ConfigureInterruptSources	314
u32MMAC_PollInterruptSource	315
u32MMAC_PollInterruptSourceUntilFired	316

vMMAC_ConfigureInterruptSources

```
void vMMAC_ConfigureInterruptSources(uint32 u32Mask);
```

Description

This function can be used to select transmit and receive interrupt sources within the JN516x Baseband Controller, to allow them to be used with CPU doze mode or in polled mode. The function allows the MicroMAC interrupts (from Tx complete, Rx complete and Rx header) to be selected by specifying the required sources as a logical-OR of the relevant `teIntStatus` enumerated values.

- If active interrupts are required, the function **vMMAC_EnableInterrupts()** must first be called to enable interrupt processing and then this function can optionally be called to select the sources that will generate interrupts (otherwise, interrupts will be generated by all sources).
- If interrupt polling is to be used then interrupt processing must not be enabled but this function must be called to select the interrupt sources of interest.

Parameters

u32Mask	Bitmap specifying the interrupts to be selected - can be specified as a logical-OR of: E_MMAC_INT_TX_COMPLETE (0x01) E_MMAC_INT_RX_HEADER (0x02) E_MMAC_INT_RX_COMPLETE (0x04)

Returns

None

u32MMAC_PollInterruptSource

uint32 u32MMAC_PollInterruptSource(uint32 *u32Mask***);**

Description

This function can be used to determine which JN516x Baseband Controller interrupts have been triggered in the case where interrupt processing has not been enabled - for example, **vMMAC_EnableInterrupts()** has not been called or CPU interrupts have been disabled. In this case, interrupt status is still maintained within the device but it is necessary to use this function to poll for this status.

The interrupt sources of interest must be specified as a logical-OR of the relevant `teIntStatus` enumerated values (see Appendix L.5.5). The function reads the interrupt source status and masks it to produce a bitmap of interrupts that are both of interest and have fired. This bitmap is returned by the function.

Once read by this function, the interrupt sources (of interest) are cleared.

Note: Alternatively, the function **u32MMAC_PollInterruptSourceUntilFired()** can be used, which is similar but only returns once there is at least one triggered interrupt (of interest) to report.

Parameters

u32Mask	Bitmap specifying the interrupt sources to be polled - can be specified as a logical-OR of: E_MMAC_INT_TX_COMPLETE (0x01) E_MMAC_INT_RX_HEADER (0x02) E_MMAC_INT_RX_COMPLETE (0x04)

Returns

Bitmap of the interrupt sources (of interest) that have fired (enumerated as above)

u32MMAC_PollInterruptSourceUntilFired

```
uint32 u32MMAC_PollInterruptSourceUntilFired(
                                        uint32 u32Mask);
```

Description

This function can be used to determine which JN516x Baseband Controller interrupts have been triggered in the case where interrupt processing has not been enabled - for example, **vMMAC_EnableInterrupts()** has not been called or CPU interrupts have been disabled. In this case, interrupt status is still maintained within the device but it is necessary to use this function to poll for this status.

Note: This function is similar to the function **u32MMAC_PollInterruptSource()** but only returns once there is at least one triggered interrupt (of interest) to report.

The interrupt sources of interest must be specified as a logical-OR of the relevant `teIntStatus` enumerated values (see Appendix L.5.5). The function reads the interrupt source status and masks it to produce a bitmap of interrupts that are both of interest and have fired. This bitmap is returned by the function once the bitmap is non-zero.

Once read by this function, the interrupt sources (of interest) are cleared.

Parameters

u32Mask	Bitmap specifying the interrupt sources to be polled - can be specified as a logical-OR of: E_MMAC_INT_TX_COMPLETE (0x01) E_MMAC_INT_RX_HEADER (0x02) E_MMAC_INT_RX_COMPLETE (0x04)

Returns

Bitmap of the interrupt sources (of interest) that have fired (enumerated as above)

L.3.6 Miscellaneous Functions

The following miscellaneous functions are provided in the MicroMAC API.

Function	Page
vMMAC_RadioOff	318
vMMAC_GetMacAddress	319

vMMAC_RadioOff

```
void vMMAC_RadioOff(void);
```

Description

This function can be used to switch off the radio receiver on the JN516x microcontroller of a low-energy device.

Parameters

None

Returns

None

vMMAC_GetMacAddress

```
void vMMAC_GetMacAddress(tsExtAddr *psMacAddr);
```

Description

This function can be used to retrieve the 64-bit IEEE/MAC address of the (local) low-energy device.

Parameters

psMacAddr	Pointer to location to receive MAC address

Returns

None

L.4 Structures

L.4.1 tsMacFrame

The `tsMacFrame` structure contains the frame for a ‘MAC mode’ operation.

```
typedef struct
{
    uint8          u8PayloadLength;
    uint8          u8SequenceNum;
    uint16         u16FCF;
    uint16         u16DestPAN;
    uint16         u16SrcPAN;
    MAC_Addr_u     uDestAddr;
    MAC_Addr_u     uSrcAddr;
    uint16         u16FCS;
    uint16         u16Unused;
    union
    {
       uint8    au8Byte[127];
       uint32   au32Word[32];
    } uPayload;
} tsMacFrame;
```

where:

- `u8PayloadLength` is the payload data length, in bytes
- `u8SequenceNum` is the sequence number for the frame (this is the least significant byte of the frame counter)
- `u16FCF` is the value of the Frame Control Field (FCF)
- `u16DestPAN` is the PAN ID of the destination network
- `u16SrcPAN` is the PAN ID of the source network
- `uDestAddr` is the address of the destination node (see Section L.4.3)
- `uSrcAddr` is the address of the source node (see Section L.4.3)
- `u16FCS` is the value of the Frame Check Sequence (FCS), filled in by the stack for a transmitted frame and provided as information for a received frame
- `u16Unused` is the number of bytes of padding to be added to the payload data to make the frame payload 32-bit word-aligned
- `uPayload` is a union containing the payload data as either a byte-array or word-array (for details of the payload, refer to Appendix G.5):
  - `au8Byte[127]` is the payload data as an array of bytes
  - `au32Word[32]` is the payload data as an array of words

For details of the FCF and FCS values, refer to the IEEE 802.15.4 Specification.

L.4.2 tsPhyFrame

The `tsPhyFrame` structure contains the frame for a ‘PHY mode’ operation.

```
typedef struct
{
    uint8           u8PayloadLength;
    uint8           au8Padding[3];
    union
    {
        uint8     au8Byte[127];
        uint32    au32Word[32];
    } uPayload;
} tsPhyFrame;
```

where:

- `u8PayloadLength` is the payload data length, in bytes
- `au8Padding[3]` is an array containing the bytes of padding to be added to the payload data to make the frame payload 32-bit word-aligned
- `uPayload` is a union containing the payload data as either a byte-array or word-array (for details of the payload, refer to Appendix G.5):
  - `au8Byte[127]` is the payload data as an array of bytes
  - `au32Word[32]` is the payload data as an array of words

L.4.3 MAC_Addr_u

The `MAC_Addr_u` union structure contains a node address as a 16-bit short address or a 64-bit extended address (IEEE/MAC address).

```
typedef union
{
   uint16           u16Short;
   MAC_ExtAddr_s    sExt;
} MAC_Addr_u;
```

where:

- `u16Short` is a 16-bit short address
- `sExt` is a structure containing a 64-bit extended address (see Section L.4.4)

L.4.4 MAC_ExtAddr_s

The MAC_ExtAddr_s structure contains a 64-bit extended (IEEE/MAC) address.

```
typedef struct
{
   uint32 u32L;
   uint32 u32H;
} MAC_ExtAddr_s;
```

where:

- u32L is the 'low word' (least significant 32-bit word) of the address
- u32H is the 'high word' (most significant 32-bit word) of the address

L.5 Enumerations

L.5.1 'Transmit Options' Enumerations

The teTxOption structure contains the enumerations used to specify the required options for transmitting a frame.

```
typedef enum
{
    /* Transmit start time: now or delayed */
    E_MMAC_TX_START_NOW    = 0x02,
    E_MMAC_TX_DELAY_START  = 0x03,

    /* Wait for auto ack and retry: don't use or use */
    E_MMAC_TX_NO_AUTO_ACK  = 0x00,
    E_MMAC_TX_USE_AUTO_ACK = 0x08,

    /* Clear channel assessment: don't use or use */
    E_MMAC_TX_NO_CCA       = 0x00,
    E_MMAC_TX_USE_CCA      = 0x10

} teTxOption;
```

The above enumerations are described in Table 31 below.

Feature	Enumeration	Description
Delayed transmission	E_MMAC_TX_START_NOW	Start transmission as soon as this function is called
	E_MMAC_TX_DELAY_START	Start transmission at the time specified beforehand using **vMMAC_SetTxStartTime()**
Automatic acknowledgements and re-try	E_MMAC_TX_NO_AUTO_ACK	Do not enable automatic acknowledgements and re-try
	E_MMAC_TX_USE_AUTO_ACK	Enable automatic acknowl-edgements and re-try
Clear Channel Assessment (CCA)	E_MMAC_TX_NO_CCA	Do not enable CCA
	E_MMAC_TX_USE_CCA	Enable CCA

Table 31: 'Transmit Options' Enumerations

L.5.2 'Transmit Status' Enumerations

The `teTxStatus` structure contains the enumerations used to indicate the status on transmitting a frame.

```
typedef enum
{
    E_MMAC_TXSTAT_CCA_BUSY = 0x01,
    E_MMAC_TXSTAT_NO_ACK = 0x02,
    E_MMAC_TXSTAT_ABORTED = 0x04
} teTxStatus;
```

The above enumerations are described in Table 32 below.

Enumeration	Description
E_MMAC_TXSTAT_CCA_BUSY	Radio channel was not free
E_MMAC_TXSTAT_NO_ACK	Acknowledgement was requested but not received
E_MMAC_TXSTAT_ABORTED	Transmission was aborted by the user

Table 32: 'Transmit Status' Enumerations

L.5.3 ‘Receive Options’ Enumerations

The `teRxOption` structure contains the enumerations used to specify the required options for receiving a frame (this feature is not currently used with a JenNet-IP system).

```
typedef enum
{
    /* Receive start time: now or delayed */
    E_MMAC_RX_START_NOW          = 0x0002,
    E_MMAC_RX_DELAY_START        = 0x0003,

    /* Wait for auto ack and retry: don't use or use */
    E_MMAC_RX_NO_AUTO_ACK        = 0x0000,
    E_MMAC_RX_USE_AUTO_ACK       = 0x0008,

    /* Malformed packets: reject or accept */
    E_MMAC_RX_NO_MALFORMED       = 0x0000,
    E_MMAC_RX_ALLOW_MALFORMED    = 0x0400,

    /* Frame Check Sequence errors: reject or accept */
    E_MMAC_RX_NO_FCS_ERROR       = 0x0000,
    E_MMAC_RX_ALLOW_FCS_ERROR    = 0x0200,

    /* Address matching: enable or disable */
    E_MMAC_RX_NO_ADDRESS_MATCH = 0x0000,
    E_MMAC_RX_ADDRESS_MATCH      = 0x0100

} teRxOption;
```

The above enumerations are described in Table 33 below.

Feature	Enumeration	Description
Delayed receive	E_MMAC_RX_START_NOW	Start receiver as soon as this function is called
	E_MMAC_RX_DELAY_START	Start receiver at the time specified beforehand using **vMMAC_SetRxStartTime()**
Automatic acknowledgements	E_MMAC_RX_NO_AUTO_ACK	Do not enable automatic acknowledgements
	E_MMAC_RX_USE_AUTO_ACK	Enable automatic acknowl-edgements
Malformed frames	E_MMAC_RX_NO_MALFORMED	Reject frames that appear to be malformed
	E_MMAC_RX_ALLOW_MALFORMED	Accept frames that appear to be malformed
Frame Check Sequence (FCS) errors	E_MMAC_RX_NO_FCS_ERROR	Reject frames with FCS errors
	E_MMAC_RX_ALLOW_FCS_ERROR	Accept frames with FCS errors
Address matching	E_MMAC_RX_NO_ADDRESS_MATCH	Reject frames that do not match the node's identifiers previously set with **vMMAC_SetRxAddress()**
	E_MMAC_RX_ADDRESS_MATCH	Accept frames that do not match the node's identifiers previously set with **vMMAC_SetRxAddress()**

Table 33: 'Receive Options' Enumerations

L.5.4 'Receive Status' Enumerations

The `teRxStatus` structure contains the enumerations used to indicate the status on receiving a frame (this feature is not currently used with a JenNet-IP system).

```
typedef enum
{
    E_MMAC_RXSTAT_ERROR = 0x01,
    E_MMAC_RXSTAT_ABORTED = 0x02,
    E_MMAC_RXSTAT_MALFORMED = 0x20
} teRxStatus;
```

The above enumerations are described in Table 34 below.

Enumeration	Description
E_MMAC_RXSTAT_ERROR	Frame Check Sequence (FCS) error occurred
E_MMAC_RXSTAT_ABORTED	Reception was aborted by the user
E_MMAC_RXSTAT_MALFORMED	Frame was malformed

Table 34: 'Receive Status' Enumerations

L.5.5 'Interrupt Status' Enumerations

The `teIntStatus` structure contains the enumerations used to indicate the nature of a MicroMAC interrupt.

```
typedef enum
{
    E_MMAC_INT_TX_COMPLETE = 0x01,
    E_MMAC_INT_RX_HEADER   = 0x02,
    E_MMAC_INT_RX_COMPLETE = 0x04
} teIntStatus;
```

The above enumerations are described in Table 35 below.

Enumeration	Description
E_MMAC_INT_TX_COMPLETE	Transmission attempt has finished
E_MMAC_INT_RX_HEADER	MAC header has been received (interrupt generated after the whole frame has been received)
E_MMAC_INT_RX_COMPLETE	Complete frame has been received (interrupt generated after an acknowledgement has been sent, if requested/enabled)

Table 35: 'Interrupt Status' Enumerations

L.6 MAC and PHY Transceiver Modes

Functions are provided in the MicroMAC API to transmit and receive IEEE 802.15.4 frames in ‘MAC mode’ and in ‘PHY mode’. This section describes these two modes.

Note: Developers should normally use MAC mode unless access to the PHY layer of the stack is specifically required - for example, to support non-standard frame formats.

L.6.1 MAC Mode

The following MicroMAC API functions allow IEEE 802.15.4 frames to be transmitted and received in MAC mode:

- **vMMAC_StartMacTransmit()** described in Section L.3.2
- **vMMAC_StartMacReceive()** described in Section L.3.3

The JN516x MAC hardware is able to assemble IEEE 802.15.4 frame headers automatically. This avoids the need for the software to concatenate the addressing fields and payload data into a continuous block of memory for transmission, which would require numerous byte-by-byte copy operations. Instead, the `tsMacFrame` structure (see Section L.4.1) allows the parts of the frame header to be stored in naturally-aligned elements, and the MAC hardware then assembles the continuous block of bytes for transmission itself based on the setting in the Frame Control Field (FCF). Similarly, for received frames, the MAC hardware interprets the FCF value and places each part of the frame header into the appropriate place in the `tsMacFrame` structure. Since the hardware is able to interpret the FCF and address fields, it is also able to perform actions such as automatic acknowledgements in both the transmit and receive directions, as well as address matching for received frames.

L.6.2 PHY Mode

The following MicroMAC API functions allow IEEE 802.15.4 frames to be transmitted and received in PHY mode:

- **vMMAC_StartPhyTransmit()** described in Section L.3.2
- **vMMAC_StartPhyReceive()** described in Section L.3.3

In PHY mode, the MAC hardware does not attempt to interpret the Frame Control Field and treats the entire frame as a stream of bytes. This has the disadvantage that address matching and automatic acknowledgement are disabled, but this mode is of value if non-standard frame formats are desired. Note that in this mode, the Frame Check Sequence is not calculated by the hardware - if required, it must be calculated and included in the payload by the application.

M. 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 6LoWPAN 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 6LoWPAN system is a wireless network 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 and Product ID.
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.
Fast Commissioning	Accelerated method of adding a node to a WPAN by using a pre-configured fixed channel for commissioning, which is known by all potential nodes (and is different from the channel used for normal operation).
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) employing a 128-bit addressing scheme.

Term	Description
IEEE 802.15.4	A standard wireless network protocol that is used as the lowest level of the 6LoWPAN software stack. Among other functionality, it provides the physical interface to the wireless network’s transmission medium (radio).
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.
Low-Energy Device	A wireless device with limited energy resources (possibly powered by ‘energy harvesting’) which is not formally a part of the WPAN but which is used to transmit commands to nodes in the WPAN. The device minimises power consumption by using the MicroMAC software stack.
LQI	Link Quality Indicator: A measure in the range 0-255 of the signal strength of a packet received over a radio link, where 0 and 255 are the minimum and maximum measured strengths respectively. In a Neighbour table entry, the LQI value indicates the signal strength of the last packet received from the relevant neighbouring node.
MIB	Management Information Base: A database comprising a table of local variables, held in memory on a wireless network node.
MicroMAC	NXP-adapted version of the IEEE 802.15.4 MAC stack layer that minimises memory usage and power consumption. Together with the IEEE 802.15.4 PHY layer, it forms the MicroMAC stack. Used in low-energy devices.
MiniMAC	NXP-adapted version of the IEEE 802.15.4 MAC stack layer that helps to reduce code-size. The MiniMAC layer replaces the standard IEEE 802.15.4 MAC layer in the JenNet-IP stack (release v1107 and later) and they are functionally the same.
Network Application ID	An application-level network identifier comprising 32 bits unique to the application, defined by the application developer
OND	Over-Network Download: Allows application software upgrades on WPAN nodes by distributing the replacement software through the WPAN from a ‘server’ device in the LAN/WAN domain and updating the software in a node with minimal interruption to node operation.
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.
PER	Packet Error Rate: A measure of the number of packets that successfully reached their destination as a percentage of the total packets sent. Thus, the PER is in the range 0-100, where 0 indicates that all packets were successful and 100 indicates that no packets were successful.
Router	A wireless network node which provides routing functionality (in addition to input/output/processing), if used as a parent node. Also see Routing.

Term	Description
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 6LoWPAN 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	Various additions and corrections made
1.2	10-Jan-2013	Updated for the JN516x devices
1.3	13-Feb-2013	Various updates and corrections made
1.4	15-Aug-2013	Updated for JenNet-IP v1.1 - added fast commissioning mode, low-energy devices and the MicroMAC stack, and made other minor modifications/corrections
1.5	11-Sept-2014	Updated for JenNet-IP v1.2- introduced the MiniMAC stack layer, added new functions for the MicroMAC, added buffering and pinging recommendations, replaced toolchain with ‘BeyondStudio for NXP’ and made other minor modifications/corrections

Important Notice

NXP Semiconductors

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