JenNet Stack User Guide

JN-UG-3041
Revision 2.0
28 September 2010

Contents

Part II: Reference Information

About this Manual

This manual provides a single point of reference for information relating to the JenNet wireless network protocol which can be implemented on the JN5148 and JN5139 devices. Little or no previous knowledge of wireless networks is assumed - all relevant concepts are covered by this manual. Guidance is provided on use of the Application Programming Interfaces (APIs) for JenNet and the API resources (functions, network parameters, enumerations, data types, events, etc) are fully detailed. The manual should be used as a reference resource throughout JenNet application development.

Note: This manual incorporates the former *Jenie API User Guide (JN-UG-3042)* and *Jenie API Reference Manual (JN-RM-2035)*.

Organisation

The manual is divided into three parts:

- **Part I: Concept and Operational Information** comprises 6 chapters:
  - Chapter 1 introduces the wireless network concepts and features relating to the JenNet protocol.
  - Chapter 2 further details the operational features of JenNet.
  - Chapter 3 introduces the JenNet software, including the Jenie API which is used by the application to interact with JenNet.
  - Chapter 4 describes how to use the API functions to perform commonly required tasks in setting up and operating a JenNet wireless network.
  - Chapter 5 outlines the structure of a JenNet application.
  - Chapter 6 addresses a number of advanced issues relating to JenNet application design - this chapter therefore supplements Chapter 4.
- **Part II: Reference Information** comprises 5 chapters:
  - Chapter 7 details the C functions of the Jenie API.
  - Chapter 8 details a number of supplementary JenNet functions for advanced users.
  - Chapter 9 details the Jenie and JenNet global network parameters.
  - Chapter 10 lists the enumerated types and data types.
  - Chapter 11 lists the stack events.
- **Part III: Appendices** comprises 4 appendices that provide various ancillary information, including the JN5148/JN5139 hardware and memory requirements of JenNet, and a glossary of the main terms used in JenNet wireless 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

AES	Advanced Encryption Standard
API	Application Programming Interface
CA	Collision Avoidance
CSMA	Carrier Sense, Multiple Access
JenNet	Jennic Network
LQI	Link Quality Indication
MAC	Media Access Control
PAN	Personal Area Network
QPSK	Quadrature Phase-Shift Keying
RF	Radio Frequency

Related Documents

JN-AN-1059 Wireless Network Deployment Guidelines

JN-AN-1061 JenNet Application Templates Application Note

JN-AN-1085 JenNet Tutorial Application Note

JN-UG-3024 IEEE 802.15.4 Wireless Networks User Guide

Feedback Address

If you wish to comment on this manual, please provide your feedback by writing to us (quoting the manual reference number and version) at the following postal address or e-mail address:

Applications
NXP Laboratories UK Ltd
Furnival Street
Sheffield S1 4QT
United Kingdom

doc@jennic.com

Part I:
Concept and Operational Information

1. Introduction to JenNet

The JenNet wireless network protocol has been developed to provide low-power, wireless connectivity for a wide range of applications that perform monitoring or control functions. It provides a simpler alternative to the ZigBee PRO protocol. JenNet simplifies and streamlines application development, therefore reducing development costs and time-to-market.

JenNet overcomes the traditional limitations of low-power, wireless network solutions - short range and restricted coverage, as well as vulnerability to node and radio link failures. It achieves this by building on the established IEEE 802.15.4 standard for packet-based, wireless transport. JenNet enhances the functionality of IEEE 802.15.4 with integrated set-up intelligence, facilitating easy installation, as well as routing intelligence and self-healing. JenNet incorporates listen-before-talk and can co-exist with other wireless technologies (such as Bluetooth and Wi-Fi) in the same operating environment.

Wireless connectivity means that a JenNet network can be installed easily and cheaply, and JenNet's built-in intelligence and flexibility allow networks to be easily adapted to changing needs by adding, removing or moving network devices. The protocol is designed to allow devices to appear and disappear from the network, so devices can be put into a power-saving mode when not active. This means that many devices in a JenNet network can be battery-powered, making them self-contained and, again, reducing installation costs.

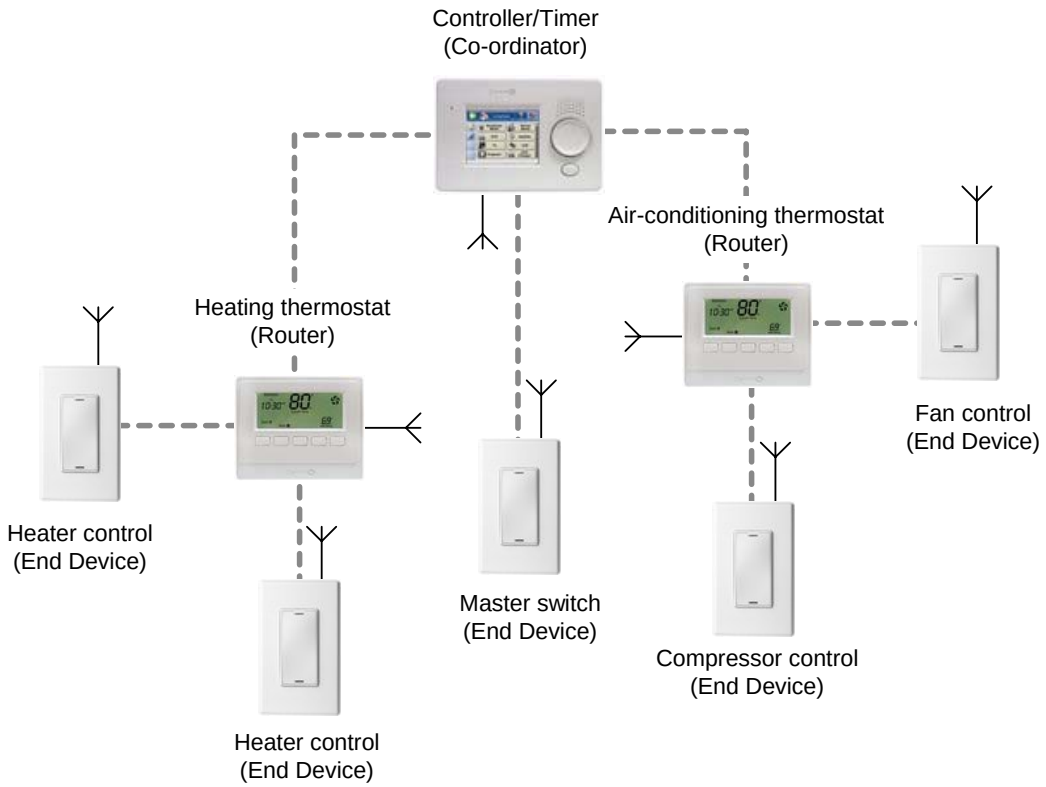

Figure 1: Example JenNet Network (Home Heating and Air-conditioning)

1.1 Ideal Applications for JenNet

JenNet is suitable for a wide range of applications, covering both commercial and domestic use, which include:

- Point-to-point cable replacement (e.g. wireless mouse, remote controls, toys)
- Security systems (e.g. fire and intruder)
- Environmental control (e.g. heating and air-conditioning)
- Lighting control
- Home automation (e.g. home entertainment, doors, gates, curtains and blinds)
- Automated meter reading (AMR)
- Industrial automation (e.g. plant monitoring and control)

JenNet's wireless communications also enable some applications to be developed that currently cannot be implemented with cabled systems. Examples are applications that involve mobility, which must be free of cabling (e.g. asset tracking in warehouses). Existing applications (such as lighting control and industrial plant monitoring) that currently rely on cable-based systems can be implemented more cheaply, as JenNet reduces or removes cable installation costs. It can also be beneficial in environments where cable-based solutions can be difficult and expensive to install - for example, in home security systems, sensors need to be easy to install (no cables or power supply wiring), small and self-contained (battery-powered).

1.2 Radio Frequency Operation

JenNet provides wireless, radio-based network connectivity 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 are as follows:

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

Thus, JenNet offers a high data-rate and a large selection of channels. It also offers the possibility of automatically selecting the best frequency channel at initialisation (the channel with least detected activity) - this is achieved by setting the desired channel number to 0.

The range of a radio transmission is dependent on the operating environment - for example, inside a building or outside. With a standard module (around 0 dBm output power), a range of over 200 metres can typically be achieved in an open area (ranges in excess of 450 metres have been measured), but inside a building this can be reduced due to absorption, reflection, diffraction and standing wave effects caused by walls and other solid objects. High-power modules (greater than 15 dBm output

power) can achieve a factor of five greater than this. In addition, the range between devices can be extended in a JenNet network, since the Tree topology (see Section 2.1.2) can use intermediate nodes (Routers) as stepping stones when passing data to destinations.

Tip: For guidance on the deployment of radio devices, refer to the Application Note *"Wireless Network Deployment Guidelines" (JN-AN-1059)*.

1.3 Battery-Powered Components

There are many wireless applications that are battery-powered, e.g. light-switches, active tags and security detectors. The JenNet and IEEE 802.15.4 protocols are specifically designed for battery-powered applications. From a user perspective, battery power has certain advantages:

- **Easy and low-cost installation of devices**: No need to connect to separate power supply
- **Flexible location of devices**: Can be installed in difficult places where there is no power supply, and can even be used as mobile devices
- **Easily modified network**: Devices can easily be added or removed, on a temporary or permanent basis

Since these devices are generally small, they use low-capacity batteries and therefore battery use must be optimised. This is achieved by restricting the amount of time for which energy is required by the device - since the major power drain in the system is when the radio transceiver is operating, data may be sent infrequently (perhaps once per hour or even per week) which results in a low duty cycle (transmission time as proportion of time interval between transmissions). When data is not being sent, the device reverts to a low-power sleep mode to minimise power consumption.

A network device can also potentially use "energy harvesting" to absorb and store energy from its surroundings - for example, the use of a solar cell panel on a device in a well-lit environment.

Note: In practice, not all devices in a network can be battery-powered, particularly those that need to be switched on all the time (and cannot sleep), such as Routers and Co-ordinators. Such devices can often be installed in a mains-powered appliance that is permanently connected to the mains supply (even if not switched on) - for example, a ceiling lamp or an electric radiator. This avoids the need to install a dedicated mains power connection for the network device.

1.4 Easy Installation and Configuration

One of the great advantages of a JenNet network is the ease with which it can be installed and configured.

As already mentioned, the installation is simplified and streamlined by the use of certain battery-powered devices with no need for power cabling. In addition, since the whole system is radio-based, there is no need for control wiring to any of the network devices. Therefore, JenNet avoids much of the wiring and associated construction work required when installing cable-based networks.

The configuration of the network depends on how the installed system has been developed. There are three system possibilities - pre-configured, self-configuring and custom:

- **Pre-configured system:** A system in which all parameters are configured by the manufacturer. The system is used as delivered and cannot readily be modified or extended. Example: vending machine.
- **Self-configuring system:** A system that is installed and configured by the end-user. The network is initially configured by sending "discovery" messages between devices. Some initial user intervention is required to set up the devices - for example, by setting switches on the devices. Once installed, the system can be easily modified or extended without any re-configuration by the user - the system detects when a device has been added, removed or simply moved, and automatically adjusts the system settings. Examples: A DIY home security or home lighting system in which extra devices can be added at a later date.
- **Custom system:** A system that is tailored for a specific application/location. It is designed and installed by a system integrator using custom network devices.

1.5 Reliable Radio Communication

JenNet employs a range of techniques to ensure reliable communications - that is, to ensure communications reach their destinations uncorrupted. Corruption could result, for example, from radio interference or poor transmission/reception conditions. These techniques are provided by the underlying IEEE 802.15.4 protocol and are as follows:

- **Coding:** JenNet applies a coding mechanism to radio transmissions. The coding method employed in the 2400-MHz band uses QPSK (Quadrature Phase-Shift Keying) modulation with conversion of 4-bit data symbols to 32-bit chip sequences. Due to this coding, there is a high probability that a message will get through to its destination intact, even if there are conflicting transmissions (more than one device transmitting in the same frequency channel at the same time).
- **Listen before send:** The transmission scheme also avoids transmitting data when there is activity on its chosen channel - this is known as Carrier Sense, Multiple Access with Collision Avoidance (CSMA-CA). This means that before beginning a transmission, a node will listen on the channel to check whether it is clear. If activity is detected on the channel, the node delays the transmission for a random amount of time and listens again. If the channel is now clear, the transmission can begin, otherwise the 'delay and listen' cycle is repeated.
- **Acknowledgements:** Message acknowledgements are used to ensure that messages reach their destinations. When a message arrives at its destination, the receiving device sends an acknowledgement to indicate that the message has been received. If the sending device does not receive an acknowledgement within a certain time interval, it re-sends the original message (it can re-send the message several times until the message has been acknowledged).

The above reliability measures allow a JenNet network to operate in an insular, protected environment, even when there are other networks nearby operating in the same frequency band. Therefore, adjacent JenNet networks will not interfere with each other. In addition, JenNet networks can operate in the neighbourhood of networks based on other standards, such as Wi-Fi and Bluetooth, without any interference.

1.6 Routing

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.

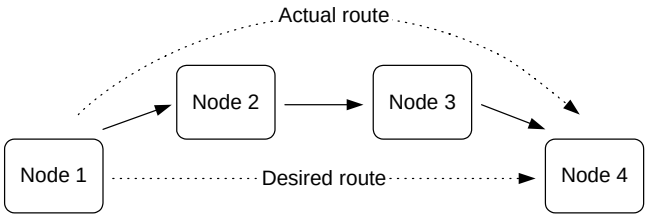

Figure 2: Routing between Network Nodes

The process of receiving data destined for another node and passing it on is known as routing. 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.

The use of routing enables more nodes to communicate and greater distances to be covered via intermediate nodes, whilst also maintaining the low-power operation of individual nodes. Routing is described further in Section 2.5.

1.7 Network Topologies

A JenNet network can adopt either of the two topologies illustrated below: Star or Tree.

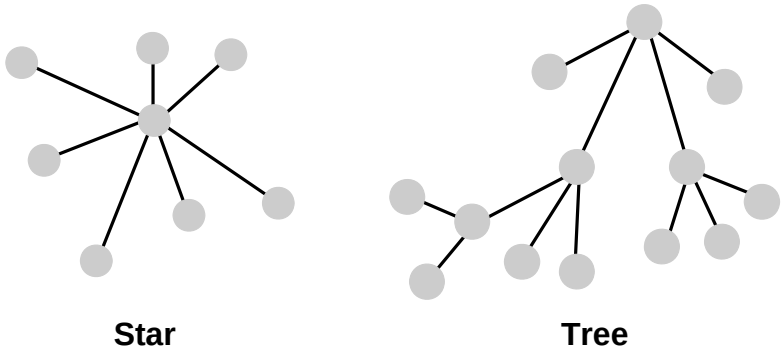

Figure 3: JenNet Network Topologies

The way that a message is routed from one node to another depends on the topology:

- **Star:** The network has a central node, which is linked to all other nodes in the network. All messages travel via the central node.
- **Tree:** The network has a top node with a branch/leaf structure below. To reach its destination, a message travels up the tree (as far as necessary) and then down the tree.

There is always one node that takes a co-ordinating role in a network - the central node in a Star topology, the top node in a Tree topology. There must also be nodes with the role of relaying messages from one neighbouring node to another. JenNet node types and network topologies are described in more detail in Section 2.1.

1.8 Security

JenNet incorporates security measures to prevent intrusion from potentially hostile agents and from neighbouring networks:

- **Encryption:** To prevent external agents from interpreting JenNet data messages, the data is encrypted at the source and decrypted at the destination using the same key - only devices with the correct key can decrypt the encrypted data. This feature is based on the AES (Advanced Encryption Standard) CCM* algorithm, which is a very high-security encryption system implemented at the IEEE 802.15.4 level (and built into the JN5148/JN5139 device as a hardware function).
- **Integrity:** This service adds a 128-bit Message Integrity Code (MIC) to a message, which allows the detection of any message tampering by devices without the correct encryption/decryption key.
- **Replay Attack Prevention:** A nonce is used to protect against replay attacks in which old messages are later re-sent to a device. An example of a replay attack is someone recording the open command for a garage door opener and then replaying it to gain unauthorised entry into the garage.

1.9 Co-existence

The JenNet standard ensures “co-existence” - that is, network devices built to the JenNet standard (possibly from different manufacturers) can exist in the same network without interfering with each others’ operation.

1.10 Basic Software Architecture

The software that runs on each node of a wireless network is organised into three basic levels forming the software stack illustrated and described below.

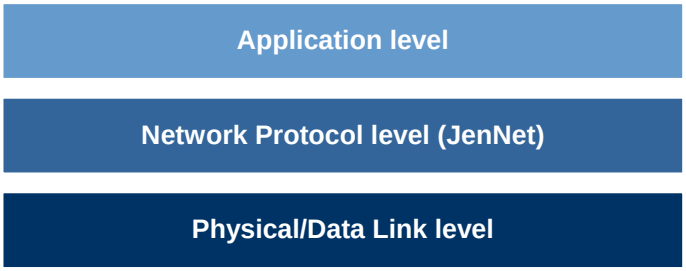

Figure 4: Basic Software Architecture

These basic levels are described below (from top to bottom):

- **Application level:** Contains the user-developed application that runs on the node. This software gives the device its functionality - the application is mainly concerned with converting input into digital data and/or digital data into output.
- **Network Protocol level:** Provides the network functionality, as well as the glue between the application and the Physical/Data Link level (below). It consists of stack layers concerned with network structure, routing and security. This level is provided by JenNet.
- **Physical/Data Link level:** This level consists of two separate layers - the Physical layer and the Data Link layer:
  - The Data Link layer is responsible for assembling, delivering and decomposing messages.
  - The Physical layer is concerned with the interface to the physical transmission medium (radio, in this case).

  In the JenNet software stack, this level is provided by the IEEE 802.15.4 standard.

The above software architecture is described in more detail in Section 3.1.

2. Operational Features

This chapter details some of the important operational features of JenNet:

- Node types and network topologies - see Section 2.1
- Node addresses - see Section 2.2
- Network identifiers - see Section 2.3
- Network formation process - see Section 2.4
- Message routing - see Section 2.5
- Services - see Section 2.6
- Binding (of services) - see Section 2.7
- Data transfer - see Section 2.8
- Auto-ping - see Section 2.9

Note: Incorporating these features in your application code by means of API functions is covered in Chapter ?.

2.1 Node Types and Network Topologies

A wireless network can be made up from nodes of three types, introduced in Table 1.

Node Type	Role
Co-ordinator	The Co-ordinator is an essential node and plays a fundamental role at system initialisation, during which its tasks are: • Selects the radio channel to be used by the network • Starts the network • Allows 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: • Relays messages from one node to another (routing) • Allows 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 have children. It can often be battery-powered and, when not transmitting or receiving, can sleep in order to conserve power.

Table 1: Node Types and Their Roles

Note that every wireless network must have a Co-ordinator.

The application on each node configures the host 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).

A wireless network that uses JenNet can have either of two topologies, which determine how the nodes are linked and how messages propagate through the network. These topologies are Star and Tree, and are presented in the sub-sections below (in fact, the Star topology is a special case of the Tree topology).

2.1.1 Star Topology

This is the simplest and most limited of the possible topologies.

A Star topology consists of a Co-ordinator and a set of End Devices. Each End Device can communicate only with the Co-ordinator. Therefore, to send a message from one End Device to another, the message must be sent via the Co-ordinator, which relays the message to the destination.

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

The Star topology is illustrated in the figure below.

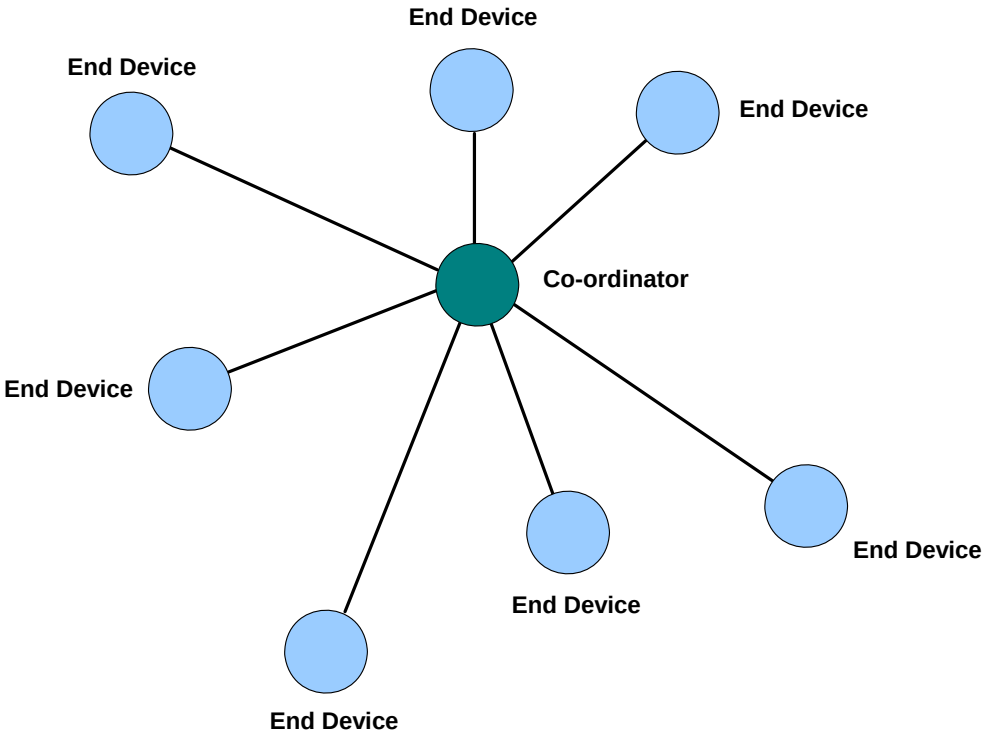

Figure 5: Star Topology

A disadvantage of this topology is that there is no alternative route if the RF link fails between the Co-ordinator and the source or target device. In addition, the Co-ordinator can be a bottleneck and cause congestion.

2.1.2 Tree Topology

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.

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

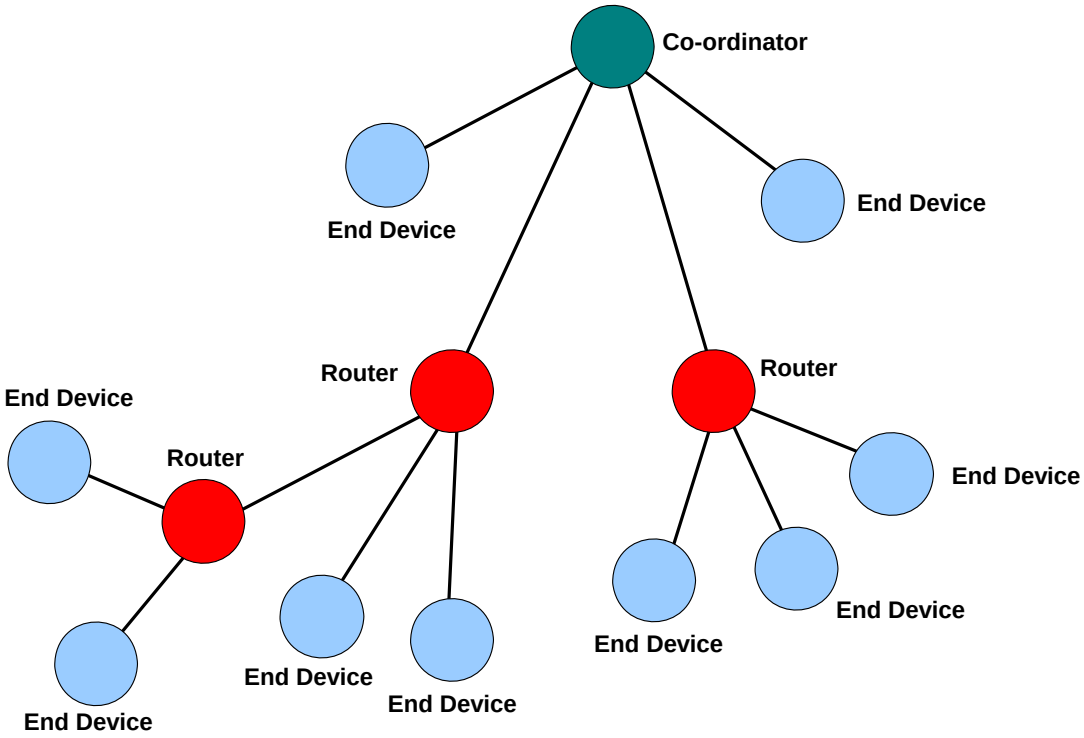

Figure 6: Tree Topology

Note that:

- The Co-ordinator and Routers can have children, and can therefore be parents (they may each have up to 16 children).
- End Devices cannot have children, and therefore cannot be parents.

The communication rules in a Tree topology are:

- A node can only directly communicate with its parent and its 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.

A disadvantage of this topology is that there is no alternative route if a necessary link fails. However, the JenNet protocol provides the facility to automatically repair failed routes.

Note: It is important when designing and deploying a Tree network that all nodes are within range of Routers, so that reliable communication can occur.

In a JenNet network, there is a maximum permissible number of children that a Router or the Co-ordinator can have. The default maximum is 10 but this limit can be reduced on individual nodes. These children may be End Devices or Routers, and the maximum number of End Device children can also be configured, leaving the remaining child places reserved for Routers.

2.2 Node Addressing

The basic way of referring to a node in a network is by means of a numeric address. In JenNet, the 64-bit IEEE or MAC address is used. This is a unique 64-bit value assigned to a device at the time of manufacture and is fixed for the lifetime of the device. It therefore provides a unique ID for the device. It is also sometimes called the extended address. JenNet uses it as the network address of the node.

2.3 Network Identification and Isolation

This section describes how a JenNet wireless network can be uniquely identified and isolated from other wireless networks operating in the same space, thus allowing networks to function without interfering with each other.

2.3.1 Network Identification

Wireless networks must be uniquely identified so that there is no confusion between neighbouring networks. JenNet networks are individually identified using two values:

- **Network Application ID:** This is a 32-bit value which is pre-determined by the system developer. It is the value used throughout the application to identify the network. It may correspond to a particular product from a manufacturer, such as an intruder alarm system. Therefore, the Network Application ID is common to all networks based on the same product and, in this sense, is not truly unique.
- **PAN (Personal Area Network) ID:** This is a 16-bit value which must be unique to the network. It is pre-set by the system developer, but the Co-ordinator "listens" for the PAN IDs of any neighbouring networks to check that the specified PAN ID is unique. If it is not unique, the Co-ordinator automatically increments the PAN ID until a unique value is found. Once set, the PAN ID is used at a low level in network messages, but is not used in the application.

The detailed implementation of these identifiers is described in Section 6.1. Information on operating multiple networks with duplicate identifiers is provided below.

Duplicate Network Application IDs

The Network Application ID provides the only fixed way of identifying your JenNet network in your application. It should be assigned a random value. However, there is no mechanism to ensure that the Network Application ID is unique. While it is improbable that two independent JenNet networks deployed in the same space will have the same Network Application ID, this remains a possibility, particularly if the networks are based on the same product (e.g. intruder alarms from the same manufacturer) - see Section 6.1 for more information.

For a large commissioned system, it may be possible to set the Network Application ID manually during deployment, to avoid the Network Application IDs of other JenNet networks operating in the neighbourhood, where these IDs are obtained using a site survey tool.

Networks with duplicate Network Application IDs operating in the same space should not be a problem, provided that their PAN IDs are unique (see below) or the networks are adequately isolated (see Section 2.3.2).

Duplicate PAN IDs

The default PAN ID that is pre-set by the system developer cannot be guaranteed to uniquely identify a network and may be dynamically changed by the Co-ordinator at start-up in order to avoid the PAN IDs of other networks. Even with this dynamic setting, it is still possible to obtain separate networks with the same PAN ID operating in the same radio space, particularly if the networks run the same application (in which case, the networks will have the same default PAN ID and Network Application ID). This may occur in the following circumstances:

- The Co-ordinators of these networks were powered up simultaneously and selected the same PAN ID.
- Branches of separate networks with the same PAN ID (initially operating in different radio spaces) grow and eventually meet.

If this occurs, the radio traffic in one network may be received and propagated through the other network sharing the PAN ID, resulting in network instability.

A useful way of avoiding PAN ID clashes between networks based on the same product (running the same application) is to generate the default PAN ID using part of the Co-ordinator's MAC address (see Section 2.2). Since MAC addresses are globally unique, this reduces the likelihood of conflicting PAN IDs.

Networks with duplicate PAN IDs operating in the same space should not be a problem if the networks are adequately isolated, as described in Section 2.3.2.

2.3.2 Network Isolation

It is normally practicable for a JenNet wireless network to be uniquely identified within its operating environment using its Network Application ID and PAN ID (described in Section 2.3.1). However, it is possible to operate networks with the same Network Application ID and PAN ID in the same neighbourhood without conflict. This is achieved by carefully managing radio channels and/or using encryption, as described below.

Radio Channels

Networks can be operated in separate radio channels to avoid contention. However, using this method to isolate networks means that moving channels to avoid a busy, congested channel may prove more difficult.

Encryption

For systems that extend over large areas (for example, street lighting), the use of encryption can be used to ensure that a network is isolated from third party networks. With this security feature enabled, nodes without the correct key will be unable to join a network, even if configured with a matching Network Application ID.

2.4 Network Formation

This section outlines the processes of starting a network through the Co-ordinator and then growing the network by allowing other devices to join it.

Note: A JenNet network uses the Network Application ID (see Section 2.3) to bring nodes together to form the network. Therefore, the user applications of all nodes of the network must be programmed with the same Network Application ID.

2.4.1 Starting a Network

The Co-ordinator is responsible for starting a network according to the following process:

1. **Radio Channel Selected:** The Co-ordinator selects a specified radio channel or searches for a suitable channel (usually the one which has least activity). This search can be limited to those channels 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 unique 16-bit PAN ID to the network. A PAN ID is pre-set by the system developer, but the Co-ordinator "listens" for the PAN IDs of any neighbouring networks to check that the specified PAN ID is unique - if it is not, the Co-ordinator increments the PAN ID until a unique value is found.
3. **Network Application ID Obtained:** The Co-ordinator obtains the 32-bit Network Application ID from the local application.
4. **Network Ready for Joining:** The Co-ordinator now 'listens' for requests from other nodes (Routers and End Devices) to join the network.

2.4.2 Joining a Network

Routers and End Devices can join an existing network already created by a Co-ordinator. Both Routers and the Co-ordinator have the capability to allow other nodes to join the network, but this feature of the node can be enabled or disabled (the node also has a maximum child capacity - see Section 2.1.2). The join process is as follows:

1. **Required Network Found:** A node (Router or End Device) wishing to join the network first scans the available channels to find operating networks. To identify which network it should join, the node uses the Network Application ID specified in its application.
2. **Best Parent Selected:** Initially, the Co-ordinator will be the only potential parent of a new node. However, once the network has partially formed, the device may be able to 'see' the Co-ordinator and one or more Routers from the network. In this case, it uses the following criteria, in the given order of precedence, to choose its parent:

a) Depth in tree (preference given to parent highest up the tree)

b) Number of children (preference given to parent with fewest children)

c) Signal strength (preference given to parent with strongest signal)

3. **Join Request Sent:** The node then sends a message to the selected parent (Co-ordinator or Router), asking to join the network. The selected parent determines whether it is can allow the device to join. If this is the case, it accepts the join request. If no parent is found, the joining node searches again (although an End Device will sleep before restarting the search).

The handshaking between parent and child when a new node joins the network is known as association. Typically, associations will be enabled by the application for a limited duration by pressing a button on the potential parent node.

Note: A Router or Co-ordinator can be configured to have a time-period during which joins are allowed. The join period may be initiated by a user action, such as pressing a button. An infinite join period can be set, so that child nodes can join the parent node at any time.

2.5 Message Routing

Communication between network nodes is implemented as messages, where a message is organised as a “frame” comprising a set of fields. A frame may contain payload data. A number of frame types are available, the required type depending on the purpose of the communication - frame types are detailed in Appendix B.

2.5.1 Message Propagation and Routes

The way that a message propagates through a JenNet network depends on the network topology. However, in all topologies, the message usually needs to pass through one or more intermediate nodes before reaching its final destination. The message therefore contains two destination addresses (as well as the source address):

- Address of the final destination (this must be specified at the application level)
- Address of the node which is the next "hop" (this is automatically inserted by the JenNet stack)

Both are IEEE/MAC addresses.

The way these addresses are used in message propagation depends on the network topology, as follows:

- **Star topology:** Both addresses are needed and the "next hop" address is that of the Co-ordinator.
- **Tree topology:** Both addresses are needed and the "next hop" address is that of the parent of the sending node. The parent node then re-sends the message to the next relevant node - if this is the target node itself, the "final destination"

address is used. The last step is then repeated and message propagation continues in this way until the target node is reached.

Note: Application programs in intermediate nodes are not aware of the relayed message or its contents - the relaying mechanism is handled by the JenNet stack.

2.5.2 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 and 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 and give the Co-ordinator knowledge of all nodes in the network. These tables are assembled automatically by the stack as the network is initialised and formed.

2.5.3 Establishing Routes

Routes are established when a new node joins the network and also when a node moves to a new parent. This involves updating the Routing tables of certain Routers and the Co-ordinator, to make them aware of the new node. The process is outlined below:

1. An "Establish Route" message is sent from the new node to the Co-ordinator, at the top of the tree.

   As the message progresses up the branch to the top of the tree, each node through which it passes adds the address of the new node to its Routing table, along with the address of the next-hop neighbour which has just passed the packet upwards.
2. When the message reaches the Co-ordinator, the latter sends:
   - a response back to the new node to confirm that the new route has been established.
   - a response to the parent of the new node to indicate whether the new node has been accepted or rejected by the Co-ordinator.
3. Once a route is established, a 'Network Up' notification is generated locally to indicate to the application on the device that it has fully joined the network. The event also provides the address of the parent node to the application.

2.5.4 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 up the tree.

2.5.5 Routing Example

This section describes the routing involved in sending a message from an End Device to another node in the network by means of a "Data-to-Peer" frame (see Appendix B.2.2) or a "Data-to-Service" frame (see Appendix B.2.4).

1. The message is first passed from the End Device to its Router parent, after which it is treated identically to a message generated by the Router itself.
2. The Router checks its Neighbour table to determine whether the destination node is one of its own children.

   If this is the case, it relays the message to the relevant child.

   If this is not the case, the Router consults its Routing table to check whether the destination node is listed as a descendant.

   - If it is listed, the next-hop address is retrieved and the message is forwarded via the corresponding intermediate child node.
   - Otherwise, the message is passed up the branch to the Router's parent. Eventually, the branch may join another branch down which the destination node is located (in which case, the Router at the intersection of the two branches has the destination node listed in its Neighbour or Routing table). Otherwise, the message eventually reaches the Co-ordinator, which has a Neighbour or Routing table entry for every node in the network.

A broadcast message can be sent to all nodes in the network using a "Data-to-Network" frame (see Appendix B.2.3). This message is sent to all nodes within radio range, which then re-broadcast the message. The frame has an in-built 'Time To Live' parameter that determines how many times it can be re-broadcast - by default, this is set to 5 broadcasts.

2.5.6 Message Acknowledgements

When a message is sent from one node to another node, the destination node can be requested to send an acknowledgement back to the source node to indicate that the message has been successfully received. Thus, if no acknowledgement is received, the source node can assume that the original message did not reach its destination and can attempt to re-send the message.

Note: These acknowledgements are end-to-end, meaning that they are sent by the final destination node to the source node and not by intermediate nodes along the route.

These end-to-end acknowledgements are implemented at the application level and can be enabled or disabled for an individual message transmission. In addition, the IEEE 802.15.4 level implements acknowledgements for each hop of a message to its final destination, but these are transparent to the application and cannot be disabled.

2.5.7 Sequence Number History

Each message is given a sequence number (the first byte of the frame) by the sending node, which allows the order in which messages were sent from the node to be determined.

Nodes maintain a history of sequence numbers of the last messages received. In this history, the sequence number of a message is stored with the address of the originating node.

- On End Devices, the sequence number of only the last received message is stored until the next message is received. This is possible since an End Device only ever communicates directly with its parent.
- The Co-ordinator and Routers maintain a sequence number history for the last ten messages received. If a message is received and its sequence number/ source address combination is found to be already in the sequence number history, the message is silently discarded. This avoids passing messages to the application that have been received multiple times.

The oldest item in the sequence number history is over-written by the sequence number of the latest message received.

2.5.8 Route Repair

'Route repair' involves removing a previously established route and establishing a new route to replace it. This occurs when a node concludes that its parent or child is no longer able to receive or reply to messages sent to it. There are two ways in which this conclusion may be reached:

- The first is the absence of IEEE 802.15.4 MAC acknowledgements, either for outgoing messages or for poll requests to a parent. When communication is lost with a parent, the local application is informed with a "stack reset" notification. The node then attempts to re-join the network.
- The second involves a node receiving an "Unknown-Node" message from its parent or a child. This occurs in either of the following situations:
  - A child has re-joined the network through another parent. The "Unknown-Node" message will be received by the former parent when it attempts to communicate with the child. This parent will then delete the child from its Neighbour table.
  - A parent has dis-owned a child, because it considers the child to be unreachable. The "Unknown-Node" message will be received by the child when it attempts to communicate with the parent. Upon receipt of this message, the child will reset its stack and re-join the network, possibly through a new parent.

The network joining process is described in Section 2.4.2.

2.6 Services

"Service" is a term used in Jenie to refer to a node property that can provide and/or receive data - it can correspond to a feature, function or capability of the node. Examples of services are:

- Temperature sensor
- Light level sensor
- Keypad data entry
- LCD output

An individual node can support up to 32 separate services. Each service available in a network is identified by an ID number, between 1 and 32 (inclusive). The Service IDs are represented by bit positions in the network's Service Profile - see Section 2.6.1.

Two services must be compatible in order to communicate with each other - that is, one service must provide meaningful data for the other service to interpret. For example:

- A temperature sensor and a heating controller are compatible services
- A temperature sensor and a garage door controller are not compatible services

The concept of compatible services is illustrated in the lighting control example in Figure 7 on page 40. Here, a number of services provide data to a "lighting controller" service, which is connected to a lamp. These services are:

- A "light on/off" service on a light switch node
- A "light on/off" service on a dimmer switch node
- A "light level" service on the same dimmer switch node

Note: It is the responsibility of the user application to determine whether services are compatible.

2.6.1 Service Profile

A network has a Service Profile which summarises all the services available in the network. This is a 32-bit value that is pre-determined by the system developer.

The Service Profile incorporates a list of all the available services in the system and their corresponding Service IDs. It is a 32-bit number in which each bit position corresponds to a specific service, where '1' signifies supported and '0' signifies unsupported. The bit positions correspond to the Service IDs as follows: bit 0 represents Service 1, bit 1 represents Service 2..... bit 31 represents Service 32.

The concept of the Service Profile is illustrated in Figure 7 on page 40, where the Service Profile is expressed as the hexadecimal value 0x00000007.

2.6.2 Service Discovery

Services allow a node to determine with which other nodes it could possibly communicate. For example, a heating control node may be interested in nodes with a temperature sensor (one service) or a switch (another service).

The application on a node can specify to JenNet which services it supports. An application can also request all nodes that support a particular service. JenNet will then reply with the address of each appropriate node, without additional effort by the application. This process is called “service discovery” and is described in more detail in Section 4.5.

Note: Service discovery is an essential step as the only way for a node to obtain the addresses of the remote nodes that provide the services it requires.

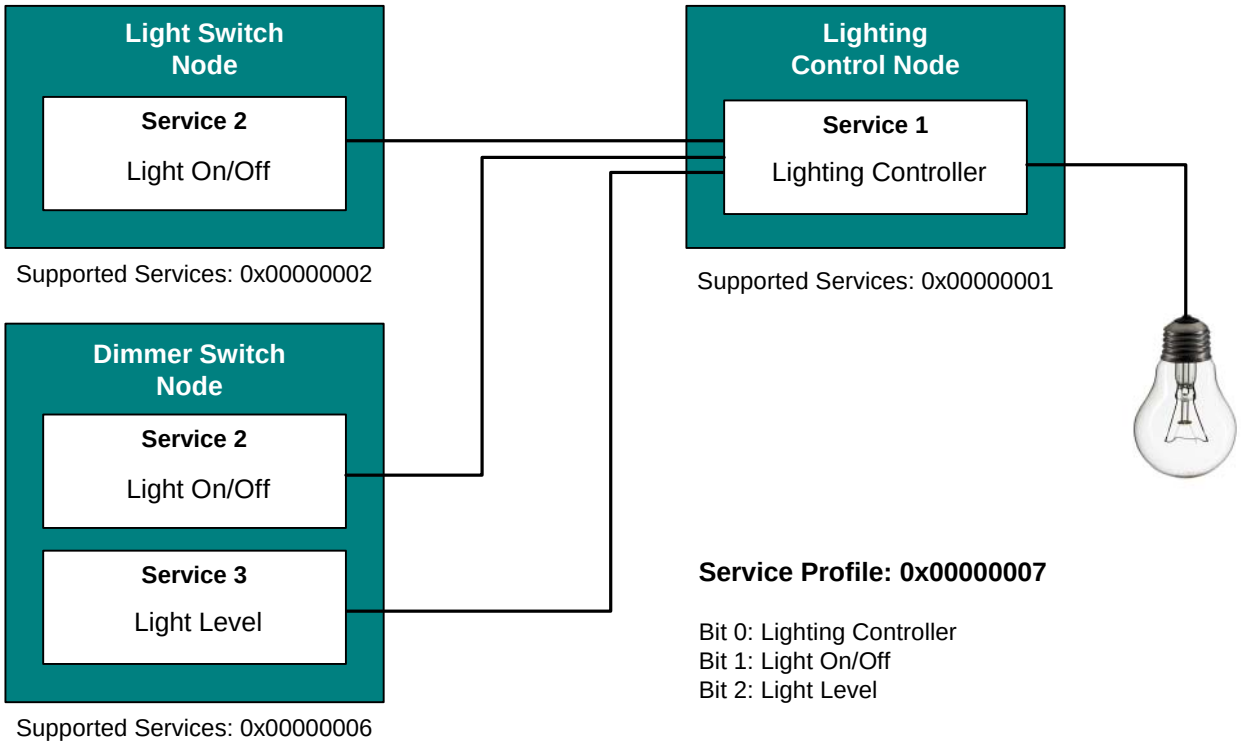

Figure 7: Example Lighting Control System

2.7 Binding

As described in Section 2.6, a "service" on one node may need to communicate with a particular service on another node. For example, a heating controller may need to take its temperature input from a temperature sensor on a remote node.

Normally for each communication, the address of the target node must be specified. Alternatively, service "binding" can be used which, once set up, allows communication between two services to be performed without the need to specify an address.

Binding associates a service on one node with a service on another node. It is analogous to wiring a cable between a sensor and an input on a control unit. Thus, sending data from a service on the local node will automatically route the message to the associated service on the remote node (see example of data transfer using binding in Section 2.8).

The binding of services is illustrated in the lighting control system of Figure 7 on page 40.

JenNet maintains a table of bindings on each node, containing the following information:

- **Source service:** The service from which data is sent on the local node
- **Destination service:** The service to receive the data on the remote node
- **Destination node:** The address of the remote node

This information is held in a Binding table on the source node for the binding.

2.7.1 Types of Binding

It is possible to have complex bindings for an individual node/service - the possible binding types are one-to-one, one-to-many and many-to-one:

- **One-to-one:** A binding in which a node/service is bound to one (and only one) other node/service
- **One-to-many:** A binding in which a source node/service is bound to more than one destination node/service
- **Many-to-one:** A binding in which more than one source node/service is bound to a single destination node/service

These are illustrated in Figure 8 below.

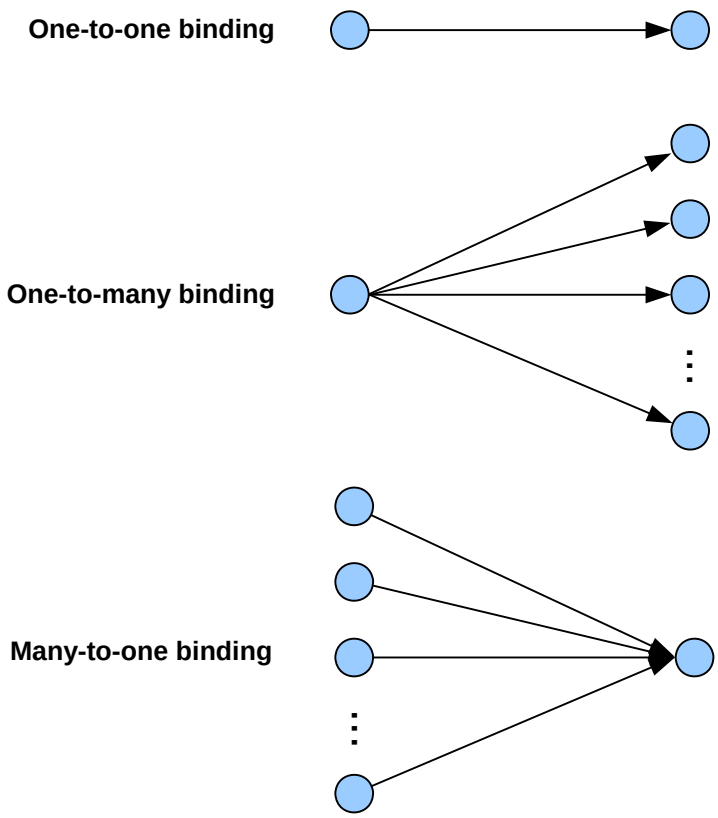

Figure 8: Binding Types

As an example of these bindings, consider a lighting control system:

- In the one-to-one case, a single switch controls a single light
- In the one-to-many case, a single switch controls several lights, perhaps in the same room
- In the many-to-one case, several switches control a single light, such as a light on a staircase, where there are switches at the top and bottom of the stairs, either of which can be used to turn on the light

It is also possible to envisage many-to-many bindings where in the last scenario there are several lights on the staircase, all of which are controlled by either switch.

2.7.2 Example Bindings

As a further example, consider the case of an intruder alarm consisting of four nodes - a control unit, two motion sensors and an alarm box (featuring a siren and light). Seven services are defined in this example system, as described in the table below.

Service	Name	Description
1	Zone 1 Trigger	This service receives indications of sensors being triggered in Zone 1 and acts on this to sound the alarm, after a delay (Zone 1 being the entry/exit zone, so requiring a delay to allow the user to disable the alarm before it sounds)
2	Zone 2 Trigger	This service receives indications of sensors being triggered in Zone 2 and acts on this to sound the alarm immediately.
3	Tamper Trigger	This service receives indications of the tamper indication being triggered on any connected node, and notifies the user.
4	Alarm Control	This service is used to control the alarm box, starting or stopping the siren and light.
5	Tamper Output	This service sends an indication if the node has been tampered with.
6	Trigger Output	This service sends an indication if the sensor detects an intruder.
7	Alarm Control	This service receives commands and uses them to control the siren and light.

Table 2: Services in Example Intruder Alarm System

The particular services on each node are shown in Figure 9 below, which also shows the bindings between services on different nodes.

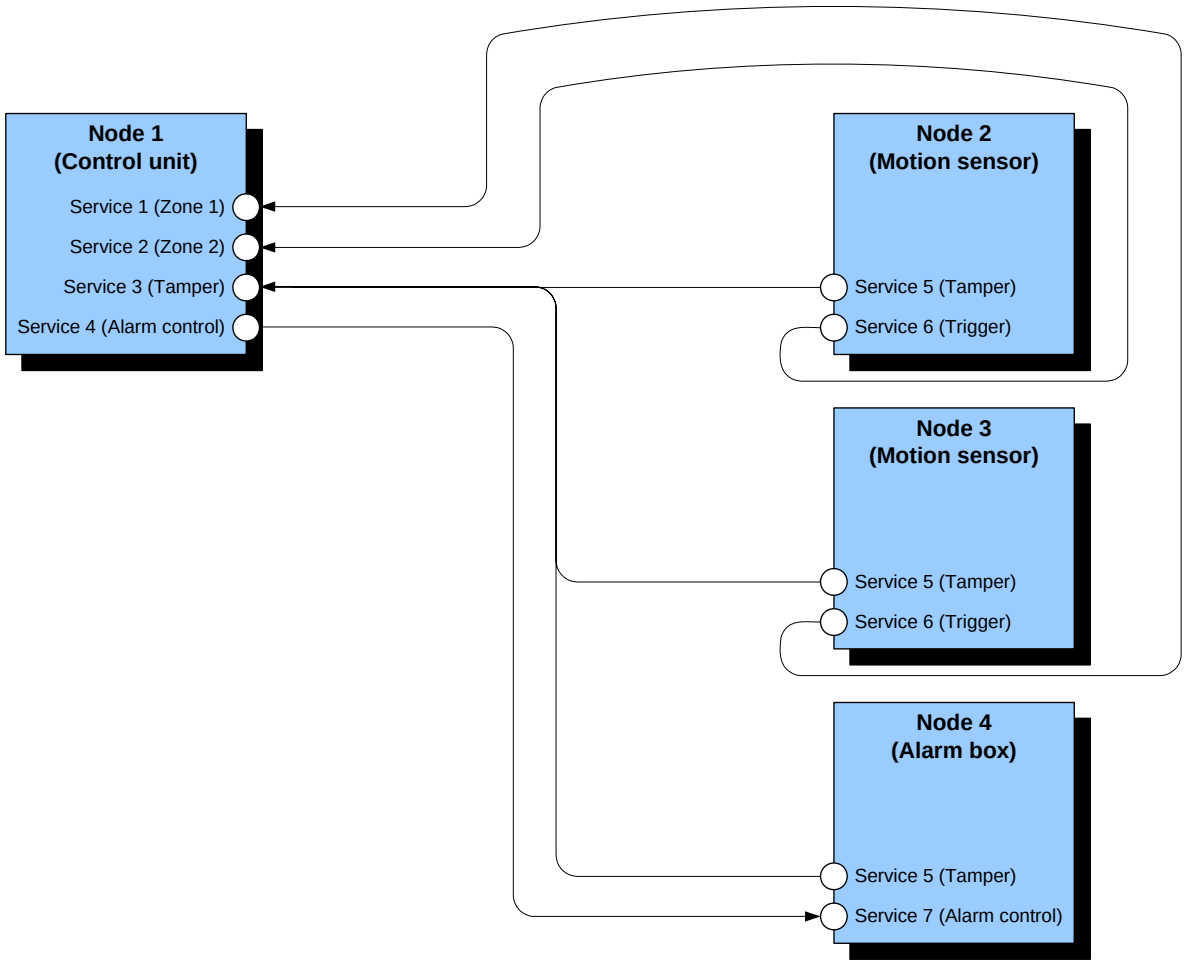

Figure 9: Bindings in Example System

The bindings in the above system are summarised in the table below.

Source Node	Source Service	Destination Node	Destination Service
1	4	4	7
2	5	1	3
2	6	1	2
3	5	1	3
3	6	1	1
4	5	1	3

Table 3: Binding Relationships in Example System

2.8 Data Transfer

During the normal operation of a node, it will need to send data to one or more remote nodes, and/or receive data from remote nodes by means of messages.

2.8.1 Data Transfer Methods

JenNet supports the following methods for transferring data from one node to another:

- **Using Addresses:** Data is sent to the destination node using the address of that node (the address obtained from the discovery stage - see Section 2.6.2).
- **Using Binding:** Data is sent from a service on the local node to one or more bound services on remote nodes. The destinations are determined by the previously defined binding - no addresses are needed (except when setting up the binding). For example, if Service 2 on the local node is bound to Service 4 on a remote node and Service 5 on another remote node, specifying Service 2 as the source service will automatically assume destination Services 4 and 5 on the relevant nodes - see Figure 10 below.

  The available services are summarised in the network's Service Profile - refer to Section 1.8. For information on binding, refer to Section 2.7.

Note: It is also possible to perform a data broadcast to all nodes in the network, or perform a multi-cast to selected nodes (using their addresses).

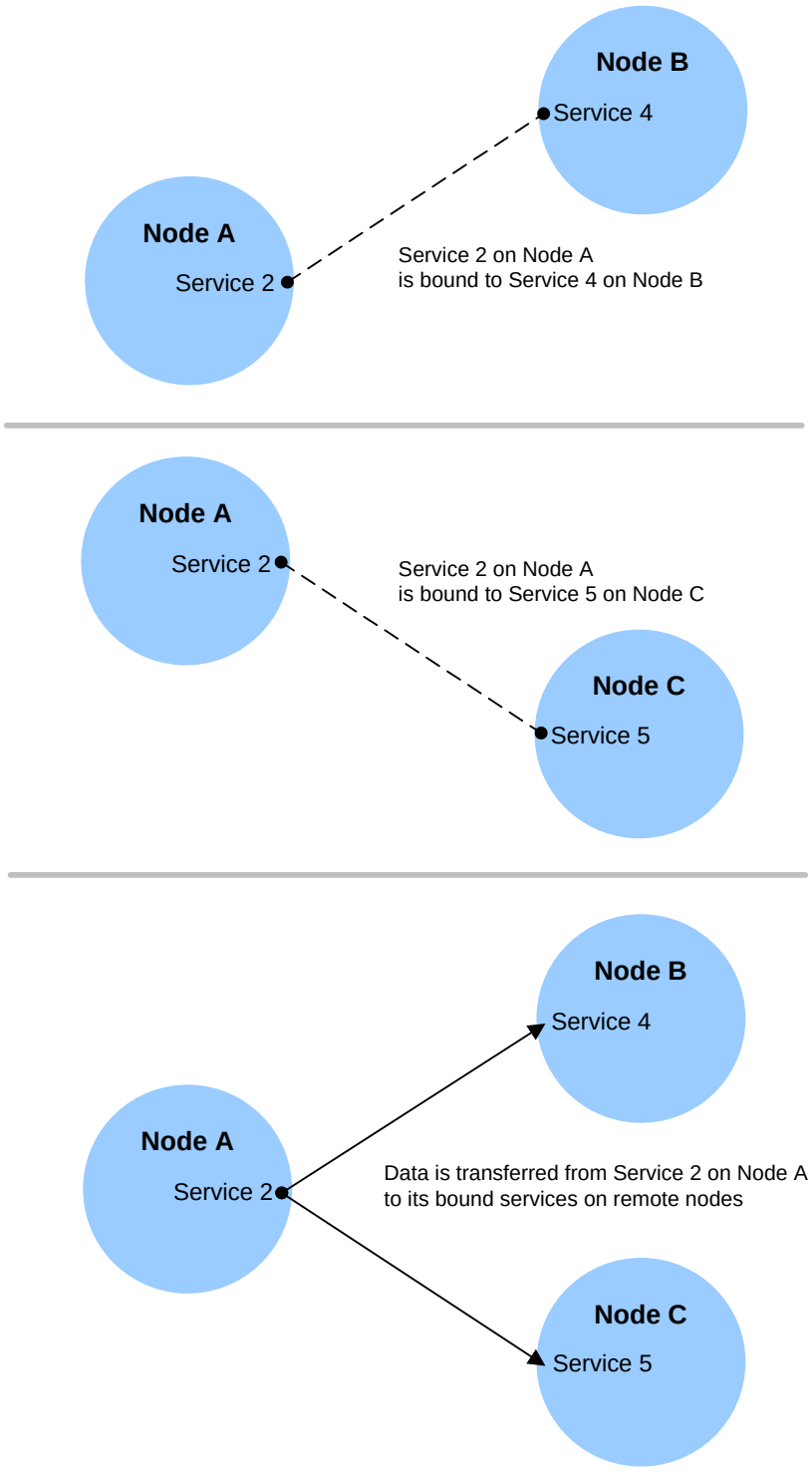

Figure 10: Data Transfer using Binding

2.8.2 Data Polling (End Device Only)

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 target node may be in sleep mode. In this case, the parent of the target 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 data waiting to be delivered.

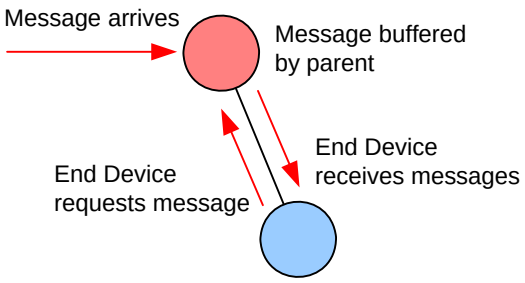

Figure 11: End Device Polling

Caution: *Pending data is buffered in the parent for a maximum of 7 seconds and then, if uncollected, is discarded. Failure by an End Device to poll for pending data within this time limit can lead to orphaning (rejection by its parent).*

2.9 Auto-ping

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. On each ping, the node sends a message to its parent:

- If the parent is still present and accepts the node as its child, it sends a response.
- Otherwise, one of two error situations may exist:
  - If the parent is not present, no response is sent. If a certain number (five, by default - see Section 6.4.1) of consecutive pings are unacknowledged in this way, the child considers its parent to be lost and attempts to re-join the network.
  - If the parent is present but has dis-owned the child, an "Unknown-Node" message is sent back. In this case, the child will attempt to re-join the network.

Note: 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 countdown to the next ping is cancelled.

An End Device has additional auto-ping requirements, described below.

End Device Pinging

An End Device can sleep, which must be taken into account when it pings its parent. A ping can be sent from the End Device to the parent just before the End Device enters sleep (for more details of this timing, see Communication Timeout in Section 6.5.1). The response to this ping will be buffered by the parent for later collection by the End Device (as described in Section 2.8.2). Therefore, to ensure that the auto-ping feature works correctly, an End Device must operate as follows:

1. The End Device wakes from sleep and then performs any processing that is necessary before it can return to sleep. If no data packets are transmitted to its parent during this time, an auto-ping packet may be generated just before the device re-enters sleep mode (depending on the ping interval - again, see Section 6.5.1).
2. In order to obtain the response to a ping, the End Device must wake again and then poll its parent for any pending data within 7 seconds of sending the ping (see Section 2.8.2). Failure to poll the parent within this time will cause the ping response to be discarded and may lead to the eventual orphaning of the End Device (depending on the presence of other traffic between the two devices).

3. JenNet Stack and APIs

This chapter describes the JenNet software, which comprises:

- JenNet stack software
- Application Programming Interfaces (APIs):
  - **Jenie API:** This is the main function library used by an application to interact with the JenNet stack
  - **JenNet API:** This is a secondary function library for advanced users who require low-level functions to interact with the JenNet stack software

The above software is provided as part of the JN5148 and JN5139 Software Developer's Kits (SDKs).

3.1 JenNet Stack

The JenNet software stack is illustrated in Figure 12 below (this provides a more detailed picture than the diagram in Section 1.10).

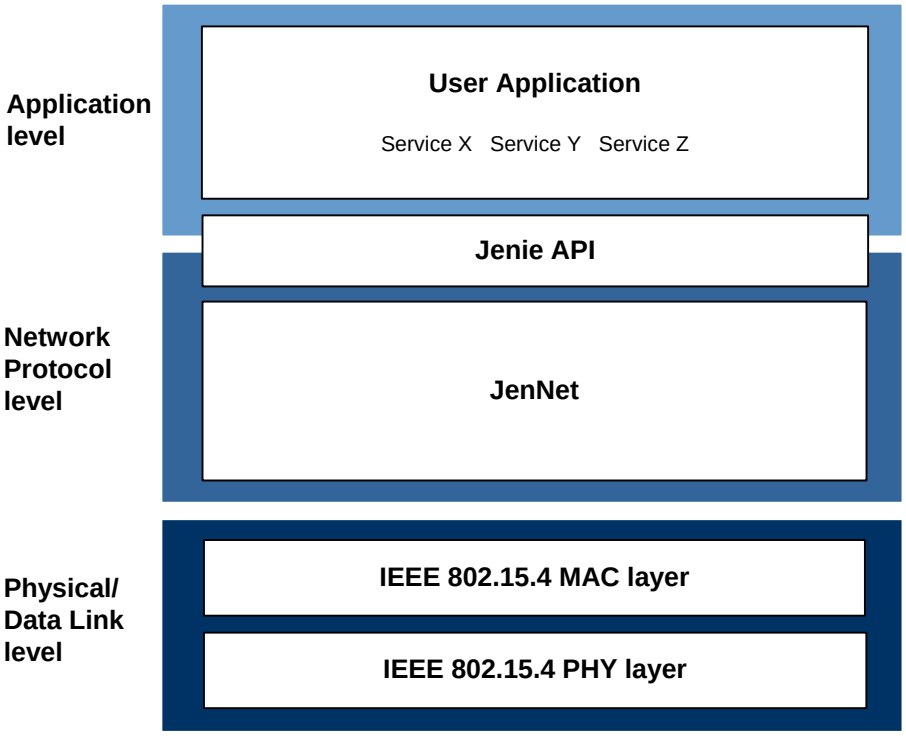

Figure 12: Detailed JenNet Software Architecture

Figure 12 shows (from top to bottom):

Application Level

This includes the user application that makes use of services provided by the node.

The user-defined application interacts with the network principally through the Jenie API. For more information on the Jenie API, refer to Section 3.2.

Network Protocol Level

This is the JenNet network layer that handles network addressing and routing by invoking actions in the IEEE 802.15.4 MAC layer (see below). Its tasks include:

- Starting the network
- Adding devices to and removing them from the network
- Routing messages to their intended destinations
- Applying security to outgoing messages

Physical/Data Link Level

This is provided by the IEEE 802.15.4 standard and consists of two separate layers - the Physical layer and the Data Link layer:

- **Data Link layer:** This is provided by the IEEE 802.15.4 MAC (Media Access Control) layer. It is responsible for message delivery, as well as for assembling data frames to be transmitted and for decomposing received frames (all are MAC frames).
- **Physical layer:** This is provided by the IEEE 802.15.4 PHY 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).

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

3.2 Jenie API

The Jenie API provides the principal mechanism by which the user application interacts with the JenNet software stack. The API comprises C functions and associated resources (data types, enumerations, etc), providing a simple, easy-to-use interface designed to streamline application development for wireless networks.

3.2.1 Function Types

The Jenie API includes two types of function:

- **'Application to stack' functions:** These functions are called in the application to interact with the JenNet software stack. They are defined in the Jenie API.
- **'Stack to application' or 'Callback' functions:** These functions are called by the JenNet software stack to interact with the application. Their prototypes are included in the Jenie API but they are user-defined, so you must define their content in your application code.

3.2.2 Functionality

The Jenie API provides functionality (through the 'application to stack' functions) for implementing network management, data transfer and system tasks, as follows.

Network Management Tasks

The network management functionality is largely concerned with starting and forming the wireless network. These management tasks include:

- configure and initialise the network
- start a device as a Co-ordinator, Router or End Device
- determine whether a Router or Co-ordinator is accepting join requests
- advertise local node services and seek remote node services
- establish bindings between local and remote node services
- handle stack management events

Data Transfer Tasks

The data transfer functionality is concerned with sending and receiving data. These tasks include:

- send data to a remote node or broadcast data to all Router nodes
- send data to a bound service on a remote node
- handle stack data events

System Tasks

The system functionality is largely concerned with implementing sleep mode, controlling the radio transmitter and dealing with hardware events. These tasks include:

- configure and start sleep mode
- configure, start and stop the radio transmitter
- handle hardware events

Note that JN5148/JN5139 'Doze mode' is not supported by the Jenie API.

3.3 JenNet API

The JenNet API may be used in conjunction with the Jenie API to access features of the underlying JenNet stack layer. The JenNet API comprises C functions that provide additional control over how nodes join a network, inter-network communication and the operation of the JenNet stack.

Note: The JenNet API is intended for advanced users who require more control over the network than is available through the Jenie API. The JenNet API is not normally needed in a JenNet wireless network application.

3.4 Software Installation

JenNet is provided as part of the JN5148 and JN5139 Software Developer's Kits (SDKs), available from the Support area of the Jennic web site (www.jennic.com/support).

The SDK Libraries installer (JN-SW-4040 for JN5148, JN-SW-4030 for JN5139) includes the following software components:

- Jenie API and JenNet API
- JenNet protocol software
- IEEE 802.15.4 protocol software
- Integrated Peripherals API
- Board API

In addition, a set of development tools is provided in the SDK Toolchain installer (JN-SW-4041 for JN5148, JN-SW-4031 for JN5139), which includes:

- Cygwin CLI
- Eclipse IDE (JN-SW-4041 only) or Code::Blocks IDE (JN-SW-4031 only)
- JN51xx compiler
- JN51xx Flash programmer

You will need the JN51xx compiler and JN51xx Flash programmer, and either the Cygwin CLI or the relevant IDE (Integrated Development Environment), depending on your chosen development environment.

Caution: *You must install the SDK Toolchain before installing the SDK Libraries. Full installation instructions for the SDK are provided in the relevant SDK Installation Guide (JN-UG-3064 for the JN5148 SDK, JN-UG-3035 for the JN5139 SDK).*

Note: It is possible to install the SDKs for the JN5148 and JN5139 devices on the same PC (JN-SW-4040 and JN-SW-4041 for JN5148, JN-SW-4030 and JN-SW-4031 for JN5139).

4. Application Tasks

This chapter describes the main tasks that you may perform using the Jenie API in your applications.

You must create a separate application for each node type in your wireless network: Co-ordinator, Router, End Device. The tasks required depend on the node type.

Note: Low-level tasks for a particular node type are handled automatically by the network level software (JenNet). Therefore, once you have specified the type of node in the application code, you need not be concerned with the detailed tasks for that node.

The tasks that you must program are presented here in approximately the order they are likely to occur in the application code, and are as follows (where a task is specific to a particular node type, this is indicated in the task description in this chapter):

- Starting the network (by creating a Co-ordinator)
- Starting other nodes and allowing devices to join the network
- Configuring the radio transmitter on a node
- Configuring security for data transfer
- Registering and requesting services (service discovery)
- Binding services
- Sending and receiving data
- Entering and leaving sleep mode (for an End Device)
- Saving and restoring context data
- Leaving the network

Throughout the task descriptions, references are made to the relevant functions from the Jenie API. Full details of the Jenie API functions are provided in Chapter 7 of this manual.

The Jenie API functions are divided into “application to stack” functions and “stack to application” (or “callback”) functions. For further information, refer to Section 3.2.1.

Tip: Further guidance to application development using the Jenie API is provided in the Application Note *JenNet Tutorial (JN-AN-1085)*. An application template is also available in the Application Note *JenNet Application Template (JN-AN-1061)*, which provides a useful starting point for your application development.

4.1 Starting the Network (Co-ordinator only)

The first step in creating a wireless network is to start and initialise the device that is to act as the network Co-ordinator. Thus, this task is only performed in the application that runs on the device which has been chosen as the Co-ordinator.

The network is first configured using the **vJenie_CbConfigureNetwork()** callback function, which acts as the entry point for the application code. This function allows network parameters to be set, including those listed in the table below (for full network parameter definitions, refer to Chapter 9).

Network Parameter	Description
gJenie_PanID	**PAN ID**: 16-bit value used to identify network - should not not clash with PAN IDs of neighbouring networks, but will be modified by the Co-ordinator if it does.
gJenie_NetworkApplicationID	**Network Application ID**: 32-bit value used to identify network.
gJenie_Channel	**Channel**: 2.4-GHz radio channel to use, or auto-channel selection (default: auto-channel selection).
gJenie_ScanChannels	**Scan Channels**: Bitmap of set of 2.4-GHz channels to scan (bit *x* represents channel *x*), if auto-channel selection enabled (default: all channels).
gJenie_MaxChildren	**Maximum Children**: Maximum number of children that the Co-ordinator can have (default: 10).
gJenie_MaxSleepingChildren	**Maximum Sleeping Children**: Maximum number of children that can be End Devices (default: 8). The remaining child slots are then reserved exclusively for Routers, although any number of child slots can be used for Routers.
gJenie_RoutingEnabled	**Routing Capability**: Must be used to enable the routing capability of the Co-ordinator.
gJenie_RoutingTableSize	**Routing Table Size**: Maximum number of entries in Routing table on Co-ordinator.
gJenie_RoutingTableSpace	**Routing Table**: Pointer to Routing table.
gJenie_RouterPingPeriod	**Router Ping Period**: Period for auto-pings generated by any Router children (default: 5 seconds).
gJenie_EndDeviceChildActivity Timeout	**End Device Child Activity Timeout**: Timeout for communications (data polling excluded) from End Device child, used to determine whether child has been lost..
gJenie_RecoverFromJpdm	**Recover Network Context**: Option to recover network context data from external non-volatile memory during a cold start following power loss to on-chip memory (data previously saved).
gJenie_RecoverChildren FromJpdm	**Recover Child/Neighbour Table**: Option to recover Child/ Neighbour table when context data is recovered from non-volatile memory (see *gJenie_RecoverFromJpdm*).

Parameter settings that are not relevant to the Co-ordinator will be ignored.

Further guidance on using some of the global parameters is provided in Chapter 6.

The Co-ordinator, and therefore the network, is then started by calling the function **eJenie_Start()**. Within this function, you must specify that the device to be started is the Co-ordinator.

Note: The function **eJenie_Start()** is normally called within the callback function **vJenie_CbInit()**, which must be defined in your application code.

Once the Co-ordinator has been started as described above, it is ready to accept join requests from other devices (see Section 4.2) and the network will then grow.

Note: The Co-ordinator is configured, by default, to permit other nodes to join it. If at any time you wish to disable joinings, use the **eJenie_SetPermitJoin()** function.

4.2 Starting Other Nodes (Routers and End Devices)

Once the network has been started through the Co-ordinator, as described in Section 4.1, other devices can join the network. The tasks described in this section can be performed in applications to be run on Routers and End Devices.

The device (Router or End Device) is first configured using the callback function **vJenie_CbConfigureNetwork()**, which acts as the entry point for the application code. This function allows network parameters to be set, including those listed in the table below (for full network parameter definitions, refer to Chapter 9).

Network Parameter	Description
gJenie_NetworkApplicationID	**Network Application ID**: Identifies the network to join.
gJenie_ScanChannels	**Scan Channels**: Bitmap of set of 2.4-GHz channels to scan when searching for a parent (bit *x* represents channel *x*).
gJenie_MaxChildren	**Maximum Children**: Maximum number of children that a Router can have (default: 10).
gJenie_MaxSleepingChildren	**Maximum Sleeping Children**: Maximum number of children that can be End Devices (default: 8). The remaining child slots are then reserved exclusively for Routers, although any number of child slots can be used for Routers.
gJenie_MaxFailedPkts	**Failed Communications**: Number of failed communications before node considers its parent or child to be lost (default: 5).
gJenie_RoutingEnabled	**Routing Capability**: Used to enable the routing capability of a Router (must be disabled for an End Device).

Network Parameter	Description
gJenie_RoutingTableSize	**Routing Table Size**: Maximum number of entries in Routing table on Router.
gJenie_RoutingTableSpace	**Routing Table**: Pointer to Routing table for Router.
gJenie_RouterPingPeriod	**Router Ping Period**: Period for auto-pings generated by a Router (default: 5 seconds).
gJenie_EndDevicePingInterval	**End Device Ping Interval**: Number of sleep cycles between auto-pings of an End Device to its parent (default: 1).
gJenie_EndDeviceScanSleep	**End Device Scan Sleep**: Amount of time following a failed scan that an End Device waits (sleeps) before starting another scan (default: 10 seconds). Avoid settings less than 1 second for large networks.
gJenie_EndDevicePollPeriod	**End Device Poll Period**: Time between auto-poll data requests sent from an End Device (while awake) to its parent (default: 5 seconds).
gJenie_EndDeviceChildActivity Timeout	**End Device Child Activity Timeout**: Timeout for communications (data polling excluded) from End Device child, used by Router to determine whether child has been lost.
gJenie_RecoverFromJpdm	**Recover Network Context**: Option to recover network context data from external non-volatile memory during a cold start following power loss to on-chip memory (data previously saved).
gJenie_RecoverChidren FromJpdm	**Recover Child/Neighbour Table**: Option on a Router to recover Child/Neighbour table when context data is recovered from non-volatile memory (see *gJenie_RecoverFromJpdm*).

Parameter settings that are not relevant to Routers or End Devices will be ignored.

Further guidance on using some of the global parameters is provided in Chapter 6. The device is then started by calling the function **eJenie_Start()**. Within this function, you must specify that the device is to be started as a Router or an End Device.

Note: The function **eJenie_Start()** is normally called within the callback function **vJenie_CbInit()**, which must be defined in your application code.

Once the device has been started, it will transmit beacon requests to search for a parent in the network with a particular Network Application ID. All potential parent nodes (Routers and the Co-ordinator), which are in range, receive this request and respond with beacons describing their ability to accept children. Given two or more responses from different potential parents, a joining device will select the parent according to the set of criteria described in Section 2.4.2. If the device fails to find a parent, it will search again. After nine failed attempts, it will generate a stack reset event (E_JENIE_STACK_RESET) before repeating the scan process once again (this event provides the application with an opportunity to undertake any outstanding actions). Also note that after each failed attempt to find a parent, an End Device will sleep (for the period *gJenie_EndDeviceScanSleep*) before the next attempt.

Note: A Router is configured, by default, to permit other nodes to join it. If at any time you wish to disable joinings, use the **eJenie_SetPermitJoin()** function.

4.3 Configuring the Radio Transmitter

The radio transmission power of the JN5148/JN5139 device can be set using the function **eJenie_RadioPower()**. The power levels for JN5148/JN5139-based modules can be set in the following ranges:

- A standard module has a transmission power range of:
  - -30 to +1.5 dBm if JN5139-based
  - -32 to +2.5 dBm if JN5148-based
- A high-power module has a transmission power range of:
  - -7 to +17.5 dBm if JN5139-based
  - -16.5 to +18 dBm if JN5148-based

The function allows you to set the power to one of six (JN5139) or four (JN5148) possible levels in the power range - for details of these levels, refer to the function description in Section 7.1.3.

Note 1: The power level can be set in the above ranges but should normally be left at the default value.

Note 2: 'Boost mode' of the JN5139 device is not supported by JenNet.

eJenie_RadioPower() can also be used to switch the radio transmitter off and on.

4.4 Configuring Security

Data sent between network nodes can be optionally encrypted and decrypted for secure communications using the AES (Advanced Encryption Standard) CCM* algorithm. This encryption/decryption is based on a security key (a value) that can be defined by the user. Thus, when data is sent from one node to another, it is encrypted by the originating node using a security key and the destination node decrypts the data using this same key. The security measures also include data integrity using a MIC (Message Integrity Code) and replay attack prevention using a nonce. For more information on security, refer to Section 1.8.

Security is enabled and the security key is specified using the function **eJenie_SetSecurityKey()**. This function is called separately for each destination node - on each call, the security key and 64-bit IEEE/MAC address of the remote node are specified.

> **Note:** In the current software release, the same security key is used for communication with all nodes. It is not possible to use different keys for different node pairs. Therefore, **eJenie_SetSecurityKey()** only needs to be called once for communication with the whole network.

Security in communications with a particular node can also be disabled using the function **eJenie_SetSecurityKey()**.

4.5 Discovering Services

A node of a JenNet network can support up to 32 services, where a service is a feature, function or capability of the node (for example, the support of keypad input). In setting up a JenNet network, “service discovery” must be implemented to find the services available and which nodes provide them. Service discovery is implemented in two stages:

1. Each node must make the rest of the network aware of the services that it has to offer by “registering” these services.
2. Each node must find out which other nodes provide services that are compatible with its own (services that can communicate, such as temperature sensor and heating control) - it does this by “requesting” services.

The above two stages are described in more detail below.

> **Note:** Service discovery is a useful technique in allowing the discovery of node addresses as well as node capabilities.

4.5.1 Registering Services

Each node must first register its services with the network - that is, advertise the services it has to offer.

The services of an individual node are defined in a 32-bit value based on the Service Profile of the network (see Section 2.6.1). Each bit position represents a specific service, ‘1’ indicating that the service is supported and ‘0’ indicating that it is not supported by the node. This 32-bit value is defined in the header of the application.

Registering the services of a node makes them available to other nodes. In the case of a Router and the Co-ordinator, this list of registered services is held locally.

However, for an End Device, the list is registered with its parent node. Thus, a Router or the Co-ordinator holds lists of services supported by all its child nodes.

Services are registered using the function **eJenie_RegisterServices()**. The behaviour following this call is dependent on the node type:

- **Co-ordinator or Router**: In this case, the services are registered locally and the function call is able to return immediately with success or failure.
- **End Device**: In this case, the services must be registered with the parent node and the function call returns with deferred status, since this takes time. Once the services have been registered with the parent, this is indicated by means of an E_JENIE_REG_SVC_RSP response (management stack event) received using the callback function **vJenie_CbStackMgmtEvent()**.

4.5.2 Requesting Services

A node must determine with which other nodes it can potentially communicate - to allow communication, the remote node must provide one or more services compatible with the service(s) of the local node.

To determine the compatible nodes, the local node sends out a service request containing a list of those services which are of interest. This is done using the Jenie API function **eJenie_RequestServices()**. The requested services are specified through a 32-bit value (based on the network's Service Profile) in which the 1s indicate the required services. This function call returns immediately and the results from individual nodes are returned later as E_JENIE_SVC_REQ_RSP responses (management stack events), received via the callback function **vJenie_CbStackMgmtEvent()**.

These responses contain the 64-bit IEEE/MAC address of the relevant remote node and a 32-bit value detailing the services supported by the node (where 1s indicate the supported services). The application can then determine with which node(s) it should communicate.

When an End Device is added to the network, it will take time to register the new node's services with its parent, following a call to **eJenie_RegisterServices()**. If a remote node requests services using **eJenie_RequestServices()** before this registration has completed, no results will be returned for the services of the new End Device. Therefore, if the remote node is particularly interested in the services of this End Device, it may be necessary to re-request services until an E_JENIE_SVC_REQ_RSP response is received containing the relevant IEEE/MAC address. One approach is to implement a timeout on the requesting node from the moment that **eJenie_RequestServices()** was called - if no response from the relevant End Device has been received within the timeout period then **eJenie_RequestServices()** should be called again.

4.6 Binding Services

In JenNet applications, communication between nodes can be simplified by binding services. Thus, a service on one node can be bound to a compatible service on another node to facilitate easy communication - for bound services, all future communications between the services will not need to specify node addresses.

Note: Service binding is not a requirement for nodes to communicate. You can implement communication between nodes without service binding, in which case you will need to use node addresses.

The function **eJenie_BindService()** is used to bind a service to another service on a remote node. The following information must be specified:

- local service
- remote node's address
- remote service

The last two items could have been obtained from an E_JENIE_SVC_REQ_RSP event received as the result of a service request (see Section 4.5.2). Once a service binding has been created, messages can be sent from the local service to the remote service as described in Section 4.7.2.

You can bind a service to multiple remote services - this requires separate calls to **eJenie_BindService()**.

If you later wish to unbind two services, use the function **eJenie_UnBindService()**.

4.7 Transferring Data

Once the network has been set up, messages can be exchanged between nodes. Data should be sent between two nodes only if the application on the destination node is capable of interpreting the received data (for example, for temperature data, the target node contains a heating controller).

Note: "Service discovery" (described in Section 4.5) can be used to establish which nodes are capable of communicating with each other. Service discovery will also give you the node addresses.

There are two ways of sending data from one node to another - the basic method uses node addresses and the alternative method uses bound services, as described in the sub-sections below.

In all cases, data sent to an End Device will be buffered on its parent node until the End Device polls its parent for data - for more details, refer to Section 4.7.3. Also note

that when an End Device wakes from sleep without memory held, data must not be transmitted by the End Device until the node is back in the network - see Section 4.9.2.

4.7.1 Sending and Receiving Data using Addresses

Data can be sent to a remote node using the function **eJenie_SendData()**. This method requires you to specify the 64-bit IEEE/MAC address of the target node.

Tip: A node can send data to the Co-ordinator by specifying a target address of zero.

Tip: It is also possible to implement data broadcasts to all Router nodes using **eJenie_SendData()**.

The sent data arrives at the target node through an E_JENIE_DATA event, received via the callback function **vJenie_CbStackDataEvent()**.

4.7.2 Sending and Receiving Data using Bound Services

Data can be sent from a service to one or more bound services using the function **eJenie_SendDataToBoundService()**. This method assumes the source and destination services have been bound as described in Section 4.6. It is not necessary to use the target node address. The local service (from which the data originates) is specified and the destination is then the remote service(s) to which the local service has been previously bound.

The sent data arrives at the target node through an E_JENIE_DATA_TO_SERVICE event, received via the callback function **vJenie_CbStackDataEvent()**.

4.7.3 Receiving Data for an End Device

Data sent to an End Device is buffered on its parent, in case the End Device is sleeping when the data arrives. It is the responsibility of the End Device to collect any pending data from its parent. It should do this regularly and always on waking from sleep when data is expected, since a build-up of unclaimed data for the End Device on its parent will eventually cause the End Device to be orphaned by its parent (see Section 6.5).

***Caution:** Pending data is buffered on the parent for up to 7 seconds before the data is discarded. Therefore, polling should be performed at least once every 7 seconds, otherwise data may be lost and the End Device may eventually be orphaned.*

Polling of the parent can be conducted manually or automatically, as described below.

Manual Polling

The End Device can manually poll its parent for data using **eJenie_PollParent()** (in which case, auto-polling should be disabled - see Auto-Polling below). Following this function call, an E_JENIE_POLL_CMPLT event is generated on the End Device.

If there is pending data for the End Device, this event contains a status value of E_JENIE_POLL_DATA_READY and is followed by an E_JENIE_DATA event containing the data. However, this data event will only contain one data message. If there are multiple pending data messages for the End Device, they must be collected by repeated calls to **eJenie_PollParent()** (see Caution below) until there is no further pending data, indicated when the event E_JENIE_POLL_CMPLT contains a status value of E_JENIE_POLL_NO_DATA.

Tip: In your End Device code, you should call **eJenie_PollParent()** repeatedly until the E_JENIE_POLL_NO_DATA status is obtained, indicating that there is no more data for the End Device.

Note: The E_JENIE_POLL_CMPLT event is also generated if no response is received from the parent. In this case, the event also contains a status value of E_JENIE_POLL_NO_DATA.

Auto-Polling

By default, an End Device is configured to automatically poll its parent on a periodic basis. The default polling period is 5 seconds, but this can be changed on the End Device through the global parameter *gJenie_EndDevicePollPeriod*, which can also be used to disable auto-polling (by setting a polling period of 0).

Note that with auto-polling enabled, an End Device will automatically poll its parent on waking from sleep, irrespective of the polling period set.

If there is pending data for the End Device, data will be received by the End Device immediately following the auto-poll - the response from the parent will result in an E_JENIE_POLL_DATA_READY event on the End Device, followed by an E_JENIE_DATA event containing the data. However, only one data message will be delivered on each auto-poll. In order to collect any other pending data messages (particularly before going to sleep), the application could then perform repeated manual polls using the **eJenie_PollParent()** function until there is no more pending data (see Manual Polling above).

Auto-polling and *gJenie_EndDevicePollPeriod* are also described in Section 6.6.

***Caution:** Do not **call eJenie_PollParent()** repeatedly with an interval of less than 100 ms between calls, otherwise the stack may freeze.*

4.8 Obtaining Signal Strength Measurements

The apparent radio signal strength of a received data packet is measured by the receiving node and this information can be accessed by the application. The signal strength is measured in terms of a Link Quality Indication (LQI) value, which is an integer in the range 0-255 where 255 represents the strongest signal.

This information can be obtained from the stack in one of two ways:

- **From Neighbour tables:** Details of every direct descendant of a routing node (Router or Co-ordinator) are stored in the Neighbour table on the node. These details include the strength (LQI value) of the last received packet from the neighbour. Jenie API functions are provided to access the contents of a Neighbour table on the local node:
  - **u8Jenie_GetNeighbourTableSize()** can first be used to obtain the number of entries in the Neighbour table.
  - **eJenie_GetNeighbourTableEntry()** can then be used to obtain the information from an individual table entry - this information is placed in a structure of type **tsJenie_NeighbourEntry**, which includes an element `u8LinkQuality` containing an LQI value.
- **From last packet received:** You can use the JenNet API function **u8Api_GetLastPktLqi()** to obtain the LQI value of the last packet received by the local node. A description of this function is provided in Chapter 8.

The relationships between the LQI value and the detected power, P, in dBm for the JN5139 and JN5148 devices are approximately given by the formulae below.

For the JN5139 device:

$$P = (LQI - 305)/3$$

For the JN5148 device:

$$P = (7 \times LQI - 1970)/20$$

The above formulae are valid for $0 \leq LQI \leq 255$.

Caution: *The relationships saturate at the LQI values of 0 and 255, and so power measurements obtained from these extreme LQI values are not reliable (an LQI value of 255 indicates that the power is at or above the maximum detectable level and an LQI value of 0 indicates that the power is at or below the minimum detectable level).*

4.9 Entering and Leaving Sleep Mode (End Devices Only)

When using battery-powered nodes (or nodes with other autonomous power sources, such as solar power), it is desirable to conserve power as much as possible. This maximises battery life and consequently reduces maintenance work involving battery replacement. One way of doing this is to put the node into a low-power sleep mode during periods when the node does not need to be active (for example, between data transmissions). Since Routers and the Co-ordinator need to be constantly active for routing and joining purposes, only End Devices can be put into sleep mode.

JenNet provides the functionality to put an End Device into sleep mode and bring it out again. Sleep mode is entered using the function **eJenie_Sleep()**. There may be a delay between calling this function and the start of the sleep period, since the node must first finish performing any tasks that remain to be completed. The device can be put to sleep for a fixed time-period which is pre-configured using the function **eJenie_SetSleepPeriod()** - this function only needs to be called once, since the configured period applies to all subsequent calls to **eJenie_Sleep()**. As an example, if the End Device is expected to transmit data every 30 seconds, the sleep duration should be set to a value less than 30 seconds. This method uses a wake timer to wake the device from sleep and requires the on-chip 32-kHz oscillator to be running during sleep - this is configured through the call to **eJenie_Sleep()**. Alternatively, the device can be woken by a hardware event originating from the on-chip comparators or DIOs, and this method does not require the oscillator to be running.

Caution: *If you set a sleep duration greater than 7 seconds using **eJenie_SetSleepPeriod()**, avoid sending data to this End Device while it is asleep (while it is not polling its parent for data). This will prevent the End Device from being orphaned by its parent.*

Sleep mode can be entered with or without preserving the contents of on-chip RAM (maintaining this volatile memory during sleep will consume more power). Again, the required option is configured through the function **eJenie_Sleep()**. The cases of sleep with memory held and sleep without memory held are described in the sub-sections below.

Note 1: The function **eJenie_Sleep()** must only be called from within the main application task, represented by the callback function **vJenie_CbMain()**. It must not be called from any other callback function.

Note 2: The function **eJenie_Sleep()** should not be called while the node is attempting to join a network, as the stack controls sleep during this time - that is, between starting/resetting the stack and the event E_JENIE_NETWORK_UP.

4.9.1 Sleep Mode with Memory Held

Sleep mode with memory held is specified when the function **eJenie_Sleep()** is called to enter sleep mode. On-chip RAM will remain powered during sleep and therefore context data will be preserved. This allows the node to easily resume network operation when it exits sleep mode.

When the node wakes from sleep with memory held, the stack calls the user-defined callback function **vJenie_CbInit()** which should initiate a 'warm restart'. The device does not re-join the network immediately but remains in the idle state until **eJenie_Start()** is called. The device then restarts as a network node using the context data held in on-chip RAM.

4.9.2 Sleep Mode without Memory Held

Sleep mode without memory held is specified when the function **eJenie_Sleep()** is called to enter sleep mode. In this case, on-chip RAM is powered down during sleep and context data held in this volatile memory must be saved to external non-volatile memory (e.g. Flash) before calling **eJenie_Sleep()**. This data can be saved using the function **vJPDM_SaveContext()**.

When the node wakes from sleep without memory held, the stack calls the user-defined callback function **vJenie_CbInit()** which should initiate a 'cold restart'. This callback function must call the function **eJPDM_RestoreContext()** to retrieve the application context data stored in non-volatile memory before entering sleep. The network context data will be retrieved automatically, provided the global parameter *gJenie_RecoverFromJpdm* has been set. The device does not re-join the network immediately but remains in the idle state until **eJenie_Start()** is called. The device then restarts as a network node using the context data that has been re-loaded into on-chip RAM.

Note: Before using **vJPDM_SaveContext()** and **eJPDM_RestoreContext()**, you should refer to Section 4.10 on saving and restoring context data.

Caution: *After waking from sleep without memory held, you must wait for the E_JENIE_NETWORK_UP event before attempting to transmit data. Failure to do this will result in the 'send data' function returning the error code E_JENIE_ERR_STACK_BUSY.*

4.10 Saving and Restoring Context Data

Context data, which describes the current state of the network and application, is held in on-chip memory. If the chip enters a period when its memory is not powered (such as a power failure or sleep mode without memory held), this data will be lost and the node must re-start from scratch when power is resumed. However, JenNet provides the facility to save a copy of this context data to external non-volatile memory (e.g. Flash) so that after power loss, node operation can resume from where it left off. This section describes the steps to take in your code in order to use this feature.

Two Jenie API functions are provided for this purpose:

- **vJPDM_SaveContext()**: This function saves both network and application context to non-volatile memory. It must only be called in the main loop callback function, **vJenie_CbMain()**, and must not be called in event handling callback functions.
- **eJPDM_RestoreContext()**: This function is used to recover application context from non-volatile memory (network context can be recovered automatically). The first time this function is called (after a cold start), it is used to set up a memory buffer in which application context data will subsequently be stored.

The cases of saving/restoring network and application context data are dealt with separately in the sub-sections below.

In addition, the function **vJPDM_EraseAllContext()** is provided, which erases all context data stored in non-volatile memory. This function is used in reverting back to the default context data. You should immediately follow this function call with a software reset, by calling **vJPI_SwReset()**, to ensure that the current context data is lost (and not re-saved) and the default context data is restored to non-volatile memory.

4.10.1 Network Context

In order to save network context data to external non-volatile memory, it is first necessary to set the global parameter *gJenie_RecoverFromJpdm* to TRUE when the callback function **vJenie_CbConfigureNetwork()** is called. The network context can then be saved at any time using the function **vJPDM_SaveContext()** in the main loop callback function, **vJenie_CbMain()** (but not in event handling callback functions).

Subsequently, whenever the application starts the stack using the function **eJenie_Start()**, the saved network context will automatically be copied back into memory and the stack will be returned to its state from when **vJPDM_SaveContext()** was last called.

Note: If this feature is not enabled using the parameter *gJenie_RecoverFromJpdm*, the stack will always re-start from scratch. In this case, the application must then re-establish any service bindings that existed. However, you will still be able to save and restore application context, as described in Section 4.10.2.

In addition to *gJenie_RecoverFromJpdm*, the following global parameters are used in conjunction with this feature and can be set from **vJenie_CbConfigureNetwork()**:

- *gJpdmSector*: Sector of Flash memory to use (default: Sector 3)
- *gJpdmSectorSize*: Size of sector to use (default: 32 Kbytes)
- *gJpdmFlashType*: Type of Flash memory used (default: auto-detect)
- *gJpdmFlashFuncTable*: Pointer to function table for custom Flash device (default: NULL)

The last two parameters above allow you to use a range of Flash devices as external non-volatile memory.

A further global parameter, *gJenie_RecoverChildrenFromJpdm*, can be used to enable/disable the recovery of a Router's or Co-ordinator's Child/Neighbour table among its context data (this option is enabled by default).

- If enabled, the parent node will be able to remember its child nodes and quickly resume its role in the network following a power loss. However, problems will occur if any of its children have in the meantime re-joined the network via other parent nodes.
- If disabled, the parent will lose all knowledge of its previous children and will dis-own them when it re-joins the network. Therefore, the children will all need to re-join the network and it does not matter if some of them have already re-joined via new parents during the power loss.

4.10.2 Application Context

In order to save application context to external non-volatile memory, you must include a call to the function **eJPDM_RestoreContext()** within the initialisation callback function **vJenie_CbInit()**:

- The first time that the application is run, there is no saved application data to restore and the **eJPDM_RestoreContext()** function registers a buffer in on-chip memory in which to store application data. The buffer is set up using the macro **JPDM_DECLARE_BUFFER_DESCRIPTION**.
- When the application is subsequently re-started, **eJPDM_RestoreContext()** will recover application context data from external non-volatile memory, previously stored using the function **vJPDM_SaveContext()**. The recovered data is stored in the buffer that was set up using the macro **JPDM_DECLARE_BUFFER_DESCRIPTION**.

The function **eJPDM_RestoreContext()** must always be called for a cold start. The use of this function is illustrated in the code fragment below.

```
struct sMyAppData
{
   //... data here
};

PRIVATE sMyAppData sData;

PRIVATE tsJPDM_BufferDescription sMyBufferDescriptor =
JPDM_DECLARE_BUFFER_DESCRIPTION("MyAppData", &sData, sizeof(sData));

PUBLIC void vJenie_CbInit(bool_t bWarmStart)
{
   //...

   if(!bWarmStart)
   {
      eJPDM_RestoreContext(&sMyBufferDescriptor);
   }

   //...
}
```

Note: You can save/restore application context irrespective of whether you save/restore network context (described in Section 4.10.1). If both save/restore operations are enabled, a single call to the function **vJPDM_SaveContext()** will save both network and application context to external non-volatile memory.

4.11 Leaving the Network

A node may leave the network under the control of the application (e.g. when an End Device is temporarily removed to replace its batteries) or under the control of the stack (e.g. when the parent suffers a power interruption). This section describes leaving the network from the points-of-view of the leaving node and its parent.

On the Leaving Node

A node can leave the network by calling the function **eJenie_Leave()** in its application code (this function call could, for example, be linked to a button press on the node). This dis-associates the node from its parent and stops the stack on the node. The node will then remain out of the network until the function **eJenie_Start()** is called, when the stack will be re-started and the node will attempt to find another parent.

Alternatively, a node may leave the network automatically under the control of the stack (normally in situations where the node considers its parent to be lost). In this case, the node will automatically try to re-join the network without calling **eJenie_Start()**. This case is linked to the global parameters described in Section 6.4 - refer to this section for more information.

In either of the above cases, when a node leaves the network, the event E_JENIE_STACK_RESET is generated on the node.

On the Parent Node

The way a parent node detects the loss of a child node depends on whether the child is an End Device or a Router, and is linked to the global parameters described in Section 6.5 - refer to this section for more information.

When a child node leaves the network, the event E_JENIE_CHILD_LEAVE is generated on the parent node.

5. Application Structure

This chapter outlines the code structure of a JenNet application. The code described corresponds to the application template which is provided and detailed in the Application Note *JenNet Application Template (JN-AN-1061)*.

Caution: *The Jenie API functions must not be called from interrupt context (for example, from within a user-defined callback function). Instead, the application should set a flag to indicate that the call should be made later, outside of interrupt context.*

5.1 JenNet Application Templates

The Application Note *JenNet Application Template (JN-AN-1061)* provides a good starting point for developing your own JenNet applications. Separate skeleton code is provided for each node type: Co-ordinator, Router, End Device. You can modify the supplied code to adapt it to your own application needs.

Tip: You will also find the Application Note *Jenie Tutorial (JN-AN-1085)* very useful. This takes a step-by-step approach to developing a wireless network application using the Jenie API and JenNet networking protocol.

The supplied application templates assume the following:

- The network topology will be a Tree.
- You have one device which will act as the Co-ordinator.
- You have at least one other device (each to act as a Router or an End Device).
- You will use pre-determined values for the PAN ID and Network Application ID.

5.2 Code Descriptions

This section describes JenNet application source code at the function level - this is for a cold start. The code includes two types of function (introduced in Section 3.2.1):

- 'Application to stack' functions are called in the application to interact with the software stack.
- 'Stack to application' or 'callback' functions are called by the software stack to interact with the application.

The general structure of the application code is illustrated in Figure 13. The sub-sections which follow describe the code for the Co-ordinator, Router and End Device.

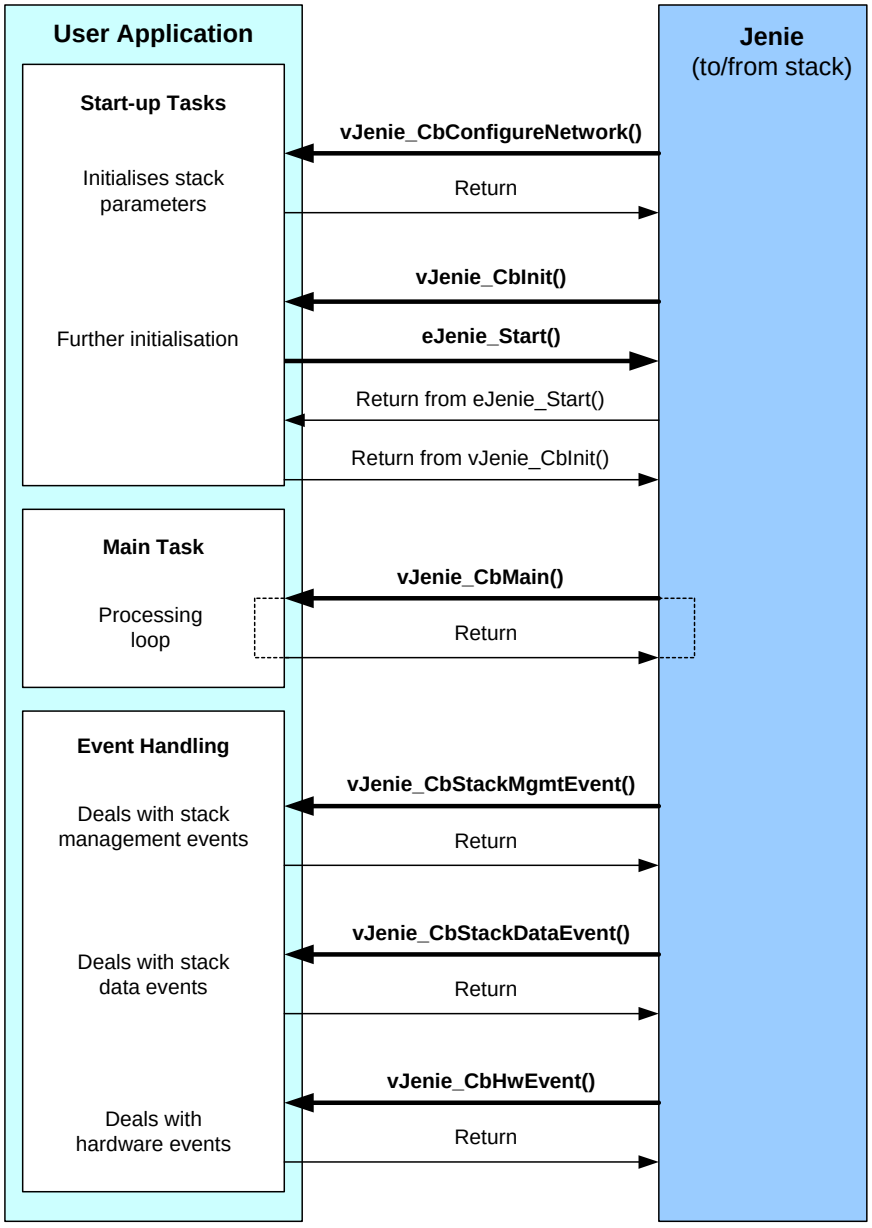

Figure 13: Application Code Overview

Note: The code for a warm start is similar to the above code (for a cold start) except the network configuration callback function **vJenie_CbConfigureNetwork()** is not called.

5.2.1 Co-ordinator Code

The Co-ordinator application is structured as illustrated in Figure 13 and described below:

1. The entry point from JenNet into the Co-ordinator application is the callback function **vJenie_CbConfigureNetwork()**, which is used for a cold start (at system start-up or reset). This function can be used to initialise stack parameters, including:
   - PAN ID (16-bit value)
   - Network Application ID (32-bit value)
   - Radio frequency channel for network
   - Maximum number of children (for the Co-ordinator)
   - Routing functionality (enable for Co-ordinator)
   - Routing table size
   - Array for Routing table
2. JenNet then calls the callback function **vJenie_CbInit()**, specifying a cold start. This function performs any further initialisation and then calls the function **eJenie_Start()**, which starts the Co-ordinator (and therefore the network).
3. Once the Co-ordinator has been initialised and started, JenNet calls the callback function **vJenie_CbMain()**, which is the main application task. This function must define any processing that is to be performed by the application.

   vJenie_CbMain() is called repeatedly by JenNet, but between calls JenNet may generate events which are sent to the application. The application must define callback functions that can be invoked by JenNet to deal with these events:
   - **vJenie_CbStackMgmtEvent()** - this function deals with stack management events (for example, a service request response received from a remote node).
   - **vJenie_CbStackDataEvent()** - this function deals with stack data events (for example, a message containing data received from a remote node or a response to one of the local node's own messages).
   - **vJenie_CbHwEvent()** - this function deals with hardware events from the JN5148/JN5139 device or carrier board.

   Once the appropriate function has dealt with the event, control is returned to JenNet which continues to call **vJenie_CbMain()**.

5.2.2 Router Code

The Router application is structured as illustrated in Figure 13 and described below (the overall structure is very similar to that of the Co-ordinator code).

1. The entry point from JenNet into the Router application is the callback function **vJenie_CbConfigureNetwork()**, which is used for a cold start (at system start-up or reset). This function can be used to initialise stack parameters, including:
   - Network Application ID of network to join
   - Maximum number of children (for the Router)
   - Routing functionality (enable for Router)
   - Routing table size
   - Array for Routing table
2. JenNet then calls the callback function **vJenie_CbInit()**, specifying a cold start. This function performs any further initialisation and then calls the function **eJenie_Start()**, which starts the Router (which will then attempt to join the network).
3. Once the Router has been initialised and started, JenNet calls the callback function **vJenie_CbMain()**, which is the main application task. This function must define any processing that is to be performed by the application.

   vJenie_CbMain() is called repeatedly by JenNet, but between calls JenNet may generate events which are sent to the application. The application must define callback functions that can be invoked by JenNet to deal with these events:
   - **vJenie_CbStackMgmtEvent()** - this function deals with stack management events (for example, a service request response received from a remote node).
   - **vJenie_CbStackDataEvent()** - this function deals with stack data events (for example, a message containing data received from a remote node or a response to one of the local node's own messages).
   - **vJenie_CbHwEvent()** - this function deals with hardware events from the JN5148/JN5139 device or carrier board.

   Once the appropriate function has dealt with the event, control is returned to JenNet which continues to call **vJenie_CbMain()**.

5.2.3 End Device Code

The End Device application is structured as illustrated in Figure 13 and described below (the overall structure is very similar to that of the Co-ordinator and Router code):

1. The entry point from JenNet into the End Device application is the callback function **vJenie_CbConfigureNetwork()**, which is used for a cold start (at system start-up or reset). This function can be used to initialise stack parameters, including:
   - Network Application ID of the network to join
   - Routing functionality (disable for End Device)
2. JenNet then calls the callback function **vJenie_CbInit()**, specifying a cold start. This function performs any further initialisation and then calls the function **eJenie_Start()**, which starts the End Device (which will then attempt to join the network).

   While attempting to join the network, an End Device may sleep between scans and therefore go through a number of warm re-starts following the sleep periods.
3. Once the End Device has been initialised and started, JenNet calls the callback function **vJenie_CbMain()**, which is the main application task. This function must define any processing that is to be performed by the application. This includes putting the node into sleep mode, if required, using the function **eJenie_Sleep()**.

   vJenie_CbMain() is called repeatedly by JenNet, but between calls JenNet may generate events which are sent to the application. The application must define callback functions that can be invoked by JenNet to deal with these events:
   - **vJenie_CbStackMgmtEvent()** - this function deals with stack management events.
   - **vJenie_CbStackDataEvent()** - this function deals with stack data events (for example, a message containing data received from a remote node or a response to one of the local node's own messages).
   - **vJenie_CbHwEvent()** - this function deals with hardware events from the JN5148/JN5139 device or carrier board.

   Once the appropriate function has dealt with the event, control is returned to JenNet which continues to call **vJenie_CbMain()**.

6. Advanced Issues in Network Operation

This chapter deals with a range of JenNet network features and issues that are not covered in the descriptions of application tasks in Chapter 4. These areas include:

- Identifying the network (Section 6.1)
- Sending messages (Section 6.2)
- Routing (Section 6.3)
- Losing a parent node (Section 6.4)
- Losing a child node (Section 6.5)
- Auto-polling (Section 6.6)

Many of these descriptions refer to the use of global parameters. These global parameters can be set in the function **vJenie_CbConfigureNetwork()**, and are fully listed and described in Chapter 9.

6.1 Identifying the Network

As described in Section 2.3, JenNet uses two identifiers to distinguish a network from other JenNet networks operating in the same space - the Network Application ID and PAN ID. Two global parameters must be set to initialise these identifiers:

- *gJenie_NetworkApplicationID* represents the Network Application ID. This is a 32-bit fixed value used throughout the application to identify the network. It will usually be set at the time of manufacture and take the same value in all networks based on a particular product. However, it should be unique within a given operating environment - that is, it should not clash with the Network Application IDs of neighbouring networks. Such a clash is unlikely if the Network Application ID assigned during design/manufacture is a **random** value. However, this may become an issue when using multiple networks based on the same product (see Section 2.3 and Joining Networks with Duplicate Network Application IDs below).
- *gJenie_PanID* represents the PAN ID of the network. This is a 16-bit value which is used by the lower stack levels to identify the network and must be unique within the operating environment - that is, it must not clash with the PAN IDs of neighbouring networks. To this effect, the network Co-ordinator will determine the uniqueness of the specified PAN ID at system start-up by 'listening' to neighbouring networks - if the specified PAN ID is found elsewhere, the value of this global parameter will be automatically adjusted until a unique value is obtained. In this respect, it does not matter which value you assign to this global parameter (except 0xFFFF, which is forbidden), as it may be changed by the system. However, the chances of the PAN ID being changed in this way can be minimised by deriving the value of this global parameter from part of the Co-ordinator's MAC address (which is globally unique).

Joining Networks with Duplicate Network Application IDs

It is theoretically possible for two or more JenNet networks with the same Network Application ID to operate concurrently, even in the same frequency channel, since at the network level the PAN ID is used to differentiate between the networks and the PAN ID is always unique. In practice, problems may occur when forming one of these networks. When a Router or End Device attempts to join the network, it will only be able to identify the required network through the Network Application ID, since this value is hard-coded in the application which runs on the joining node. This node does not know the PAN ID of the desired network, since this value may have been re-configured dynamically by the Co-ordinator (and will not be known by the joining node until it has successfully joined the network). Therefore, it is possible that the joining node will join another network with the same Network Application ID, i.e. the wrong network. You may, however, be able to prevent a node from joining the wrong network by using the function **eJenie_SetPermitJoin()** to control the 'permit joining' status of potential parents. This is a useful feature to build into a wireless network product, particularly if you expect multiple networks based on the product to be deployed in the same operating space.

Tip: For more information on handling neighbouring networks with the same Network Application ID, refer to the Application Note *Jenie Controlled Network Membership (JN-AN-1116)*.

6.2 Sending Messages

6.2.1 Timing Issues in Data Sends

There are two timing phenomena to take into consideration when sending data messages - simultaneous packets and hetrodyning, which may lead to packet loss. These effects are described below.

Caution: *Packet loss can have serious consequences and may lead to network disruption such as the loss of a parent or child node - see Section 6.4 and Section 6.5.*

Simultaneous Packets

If several child nodes all send packets at exactly the same time to a parent then packets may be lost - for example, if the children respond at the same time to a broadcast requesting data. The solution is to stagger the responses to the broadcast request in the application by using a short random delay, perhaps seeded from the MAC address of the sending node.

The effect of simultaneous sends can also be observed if all Routers send periodic data to the Co-ordinator. If the Routers are started simultaneously (for example,

following a power outage), their timers will be approximately synchronised and they will perform their periodic sends at roughly the same time. This may result in packet loss at the Co-ordinator. A better approach is to start a node's timer when it joins to the network, allowing the Router timers and therefore periodic sends to be staggered. However, even in this case, you may also see the effect of heterodyning (see below).

A further technique to reduce packet collisions is to add a small random delay before sending each packet (see below).

Hetrodyning

If several child nodes send packets to a parent asynchronously (say, every 500 ms), over time the transmissions may slowly drift into and out of synchronisation. This is because the crystals used to time the transmissions on the child nodes have slightly different frequencies. The effect is called heterodyning and is similar to beat frequencies in sound.

Thus, the children may start by sending data at different times but, over a long period of time, the transmissions will become synchronised, packet collisions will occur and packets may be lost. Therefore, the system will initially run well but, after a period of time, there will be an increase in the rate at which packets are lost, followed by a decrease in this rate (as the transmissions move out of synchronisation again).

To reduce this effect, add a small random delay to the time between data transmissions. For example, use **rand()** seeded with the MAC address of the sending node to ensure that nodes are not using the same pseudo-random numbers.

6.2.2 Re-tries in Data Sends

When a message is sent using the function **eJenie_SendData()** or **eJenie_SendDataToBoundService()** with the *u8TxFlags* option `TXOPTION_SILENT` cleared, JenNet submits the packet to the IEEE 802.15.4 MAC layer of the protocol stack and returns E_JENIE_DEFERRED. If a buffer is free, the MAC layer will attempt to send the packet. If the send fails, three further attempts will be made, making four tries in total.

Depending on the outcome of the send, JenNet will (eventually) generate one of the following stack events:

- E_JENIE_PACKET_SENT: A MAC acknowledgement has been received from the next hop node, confirming the send
- E_JENIE_PACKET_FAILED: There was no MAC layer buffer free for the send or no MAC acknowledgement has been received to confirm the send

Note: In **eJenie_SendData()**, if the *u8TxFlags* option `TXOPTION_SILENT` is enabled, the above stack events will not be generated.

6.2.3 End-to-End Acknowledgements for Data Sends

When sending data using the function **eJenie_SendData()** or **eJenie_SendDataToBoundService()**, an end-to-end acknowledgement can be requested by enabling the *u8TxFlags* option `TXOPTION_ACKREQ`. In this case, the final destination node should return an acknowledgement to the source node, once the data has been received (note that these acknowledgements are different from the IEEE 802.15.4 MAC acknowledgements mentioned in Section 6.2.2, which simply indicate that a data packet has reached the next hop towards its destination).

It should be noted that the use of end-to-end acknowledgements will double the packet overhead of the network. Therefore, you should only request an end-to-end acknowledgement when it is essential that a packet reaches its destination. The following guidelines should be useful:

- Do request acknowledgements when sending commands that will change the operation of the network.
- Do not request acknowledgements when sending regular sensor readings.

Also be aware that all of the original packet data is returned in an end-to end acknowledgement. Therefore, if you are sending large data packets, this will impact heavily on network performance.

6.3 Routing

The Co-ordinator and Routers of a network can each play a routing role, but their routing capability must be explicity enabled in the application using the global parameter *gJenie_RoutingEnabled* (when a Router is to act as an End Device, this variable must be used to disable routing for the node).

6.3.1 Neighbour Tables and Routing Tables

A routing node contains both a Neighbour table and a Routing table (see Section 2.5.2). 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 is of the type **tsJenieRoutingTable** and 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). This array must be declared in the application and configured using two global parameters:

- *gJenie_RoutingTableSize* determines the size of 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 descendants, so not to reserve more memory space than needed for the Routing table.
- *gJenie_RoutingTableSpace* is a pointer to the Routing table in memory - thus, the array will start at this point in memory.

Note that for the Co-ordinator, the value of *gJenie_RoutingTableSize* will determine (but will not be equal to) the maximum permissible number of nodes in the network.

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 rejected and the joining node's potential parent will receive a notification event of type E_JENIE_CHILD_REJECTED.

6.3.2 Stale Route Purging

Routing tables can retain stale routes as nodes join and leave the network. Stale routes will normally be removed by traffic exercising the Routing tables, but some stale entries may persist in quiet networks. An automatic ‘route purge’ mechanism can be run in the background, which checks the validity of every entry in the Routing table.

If the application is continuously generating traffic from all nodes then the Routing tables will be kept up-to-date by the application's traffic. Therefore, in this case, automatic purging is not required. However, if the application sends data infrequently then the tables could be out-of-date following a recovery activity and the automatic purging becomes essential.

In very long thin networks, the purging can add excessive traffic following a network recovery (e.g. following a power outage), with all the nodes issuing 'purge route' packets at the same time. The excessive traffic can result in collisions and possible packet loss.

It is recommended that for very large networks, which may be long and thin with regular traffic, purging is disabled on Router nodes and enabled on the Co-ordinator with the purging interval increased from the default value of 1 second (per entry) - a function for setting this interval is outlined below. The ideal level is dependent upon the level of application network traffic and the number of nodes on the network - the value can be increased until the number of route purge messages are not significantly contributing to packet losses caused by network contention.

Two JenNet API functions are provided for route purging:

- **vApi_SetPurgeRoute():** Allows route purging to be enabled/disabled.
- **vApi_SetPurgeInterval():** Allows interval between route purging activities (one entry per activity) to be set in units of 100 ms (the default interval is 1 s).

The above functions need to be called after the 'network up' event (E_JENIE_NETWORK_UP), when the default network operation is fully established.

6.3.3 Automatic Route Importation

JenNet provides a mechanism which allows a whole network branch to move within the network - this speeds up network recovery (e.g. following a power outage). This route importation feature is used when a Router node has moved in the network and has descendant children. Initially, the Routing tables of all ascendant nodes, up to and including the Co-ordinator, will contain either no routing or stale routing for this branch of the tree.

If we rely solely on the ‘purge route’ mechanism (which has the primary purpose of removing fragments of stale routing on all Routers) to clean up the Routing tables (see Section 6.3.2), it is highly likely that many packets will be lost due to traffic flowing down the old stale routes. This is because the purge route mechanism is a very slow process and does not repair a route but simply deletes stale fragments.

Another alternative is to rely on demand-driven route repair, which would be used for every packet mis-routed. This is quite a heavy process, as each route repair would result in a 'find node' broadcast followed by an 'establish route' message being sent from every node involved.

The route importation process tries to minimise traffic by performing a route repair between the newly joined Router and the Co-ordinator, rather than from leaf nodes all the way up to the Co-ordinator (as would be the case if a 'find node' message were generated).

A Boolean network parameter, *gRouteImport*, is provided in JenNet to enable/disable route importation (it is enabled by default). Thus, to disable route importation, the following line of code is required:

```
gRouteImport=FALSE; // to disable the route import mechanism
```

The feature can be disabled at any time, including prior to starting the stack.

Note: The ‘route importation’ and ‘purge route’ mechanisms can both be disabled, leaving only the demand-driven repair process, if this suits the application or network layout.

6.4 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.

6.4.1 Detecting Orphaning

There are three ways a child node can determine whether it has been orphaned:

- Lost packets
- Lost pings
- 'Unknown Node' message

Lost Packets

A node may decide that it has lost its parent when a certain number of consecutively sent packets have been lost (including unacknowledged poll packets - see Section 6.6). This number is determined by the global parameter *gJenie_MaxFailedPkts*. Due to the retries (see Section 6.2.2), when this happens the total number of lost packets will be 4 x *gJenie_MaxFailedPkts*. Since the node has now lost its parent, it will attempt to re-join the network (see Section 6.4.2).

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). If there is no other traffic on the link:

- A Router will periodically ping its parent at an interval determined by the global parameter *gJenie_RouterPingPeriod* (in units of 100 ms).
- An End Device will periodically ping its parent at an interval determined by the global parameter *gJenie_EndDevicePingInterval* (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 global parameter *gJenie_MaxFailedPkts* (which is set to 5, by default). In this case, the node will attempt to re-join the network (see Section 6.4.2) after a time given by *gJenie_MaxFailedPkts* multiplied by the ping interval.

Note that the chance of a failed (ping) packet increases as the ping-rate increases. You are therefore advised to keep the ping period as long as possible but short enough to detect a failed link within reasonable time.

Note: Following a failed ping, the ping will be re-sent after a random back-off time - this helps multiple nodes to avoid becoming synchronised in their ping attempts to their parent.

Unknown Node

A node can detect that it has been orphaned if it receives a JenNet UNKNOWN_NODE message in response to a message previously sent to its parent. This may occur if the parent has lost the child from its Neighbour table because the parent has been reset without context saving of neighbour information (that is, the global parameter *gJenie_RecoverChildrenFromJpdm* has been set to zero). On receiving this response, a stack reset will automatically be generated on the child and the node will attempt to re-join the network (see Section 6.4.2).

6.4.2 Re-joining the Network

When a node considers its parent to be lost (see Section 6.4), JenNet initiates a stack reset and begins a search for a new parent. The application is notified with E_JENIE_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_JENIE_NETWORK_UP 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_JENIE_NETWORK_UP event to the application. Otherwise, the device goes to sleep for a period determined by the global parameter *gJenie_EndDeviceScanSleep*, then scans again, repeating the scan/sleep cycle until the network has been successfully re-joined.

6.5 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.

6.5.1 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 is determined by the value of the global parameter *gJenie_EndDeviceChildActivityTimeout*.

- 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 just before going to sleep, with a ping interval defined in terms of a number of sleep cycles configured using the global parameter *gJenie_EndDevicePingInterval* (therefore the ping is not necessarily sent before every sleep period). 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 *gJenie_EndDeviceChildActivityTimeout*, otherwise the parent will assume that 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 using the function **eJenie_PollParent()**. This allows messages that arrive while the End Device is asleep to be retained and later collected when the End Device is awake.

A total of 12 message buffers in the parent are used for this purpose - 4 of these are 802.15.4 MAC buffers and 8 are JenNet buffers. The MAC buffers are filled first and when these become full, the JenNet buffers are used, forming a FIFO queue which feeds into the MAC buffers. An End Device child collects its messages from the MAC buffers, but the parent will not indefinitely store a message in one of these buffers - once a message has been in a 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 global parameter *gJenie_MaxFailedPkts*, 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 buffers as quickly as possible).

In addition, an End Device with a sleep duration of longer than 7 seconds should not use auto-pinging of its parent, since the ping responses will not be retrieved from the parent quickly enough and therefore count as failed packets. Instead, while awake, the End Device should:

1. Send a message to its parent - if there is no data to send, it should send an empty message
2. Poll its parent to clear any pending messages

6.5.2 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 global parameter *gJenie_MaxFailedPkts*. 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 global parameter *gJenie_RouterPingPeriod* (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:

gJenie_MaxFailedPkts x *gJenie_RouterPingPeriod* x 100 ms

Note: The global parameter *gJenie_RouterPingPeriod* must be set to the same value on the parent and child Routers. It must also be set to this same value on the Co-ordinator, which uses this parameter setting for detecting the loss of Router children (but does not need it for generating pings itself).

6.6 Auto-polling (End Device Only)

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.

In Jenie, 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 global parameter *gJenie_EndDevicePollPeriod*. By default, this is set to 5 seconds.

Note: Auto-polling can also be disabled through *gJenie_EndDevicePollPeriod* (by setting it to zero). If auto-polling is disabled, the End Device can explicitly poll the parent using the function **eJenie_PollParent()**.

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 means that if you set the sleep period using **eJenie_SetSleepPeriod()** to be shorter than the polling period defined in *gJenie_EndDevicePollPeriod*, the End Device will poll the parent more often than configured through this global parameter.

Note that any lost (unacknowledged) poll packets will count as failed packets and will therefore contribute to causing a stack reset if this count reaches the value of the global parameter *gJenie_MaxFailedPkts* (lost packets are described in Section 6.4). Decreasing the polling period set through *gJenie_EndDevicePollPeriod* has the effect of increasing the chances of a failed packet and a stack reset. You are therefore advised not to poll more often than is necessary.

Receiving End Device data using auto-polling is described in Section 4.7.3.

6.7 Beacon Calming

If other networks are scanning the operating channel of your network, this can affect your network's performance, since all the nodes in your network may be responding to the beacon requests (by sending beacons). A mechanism is available to manage repeated beacon request activity and reduce the beacon activity over air. This 'beacon calming' feature executes an algorithm that limits the sending of beacons in relation to the level of beacon activity and the number of available children.

For large dense networks, you should enable the beacon calming feature using the JenNet API function **Nwk_SetBeaconCalming()**. This function sets a time-window during which a node will respond to beacon requests:

1. A node with no children will always respond.
2. As a node acquires children, the time window is reduced.
3. A node that has reached the maximum number of children will not respond at all.

This feature is disabled by default.

6.8 Packet Loss

Various circumstances in which packets may be lost, and the possible consequences, have already been mentioned in the preceding sections of this chapter. This section summarises these scenarios, and provides information and advice on packet loss.

Lost packets may include unacknowledged data packets, pings and polling requests. The loss of packets can be monitored from the viewpoints of an End Device and a parent as follows:

- **End Device:** By default in JenNet, consecutive lost packets are counted on an End Device and this count is used to assess whether the link to the parent node has failed. If this count exceeds the value of the global parameter *gJenie_MaxFailedPkts* (or 4 x *gJenie_MaxFailedPkts*, if re-tries are included) then the End Device will reset its stack and try to find another parent.
- **Parent:** A parent node (Router or Co-ordinator) can also monitor packet loss in the application. Counters for successful and failed transmission attempts to each of the node's children and to its own parent (if relevant) are maintained in the Neighbour table on the node, which can be accessed using the function **eJenie_GetNeighbourTableEntry()**. These counters can be used by the application to monitor the level of packet loss and if excessive packet loss is occurring for a particular child, the parent can remove the child from the network using the JenNet API function **vNwk_DeleteChild()**.

Therefore, excessive packet loss can lead to network self-healing and a changing network shape. Under normal circumstances, this works well to find the best radio path to a parent, but high traffic rates can also result in lost packets and subsequent re-forming of the network.

6.8.1 Packet Collisions

Packet collisions can occur in areas of traffic congestion in the network. The following scenarios may lead to packet loss in this way:

Simultaneous Packets

Packet loss can occur when packets are sent simultaneously from multiple child nodes to a common parent. This scenario is described in Section 6.2.1.

Heterodyning

When multiple nodes transmit periodically with approximately the same transmission interval, the transmissions may drift into and out of synchronisation, causing packet loss during the synchronised phases. This phenemenon of heterodyning is described in more detail in Section 6.2.1.

Unsolicited Packets

A large number of unsolicited packets travelling up the network (towards the Co-ordinator) can lead to collisions and lost packets - for example, periodic data packets containing sensor readings. The solution is to ‘pull’ the packets up the network, as

described in Section 6.8.2, allowing over-air transmissions of data packets to be scheduled.

Clashes of Periodic Data and Ping Transmissions

Collisions can occur between a Router's periodic data packets to the Co-ordinator (e.g. containing sensor readings) and the Router's ping packets to its own parent.

This effect depends on the selected timings. For example, if a Router passes data to the Co-ordinator every 20 seconds and the ping-rate is 10 seconds then data packets and ping packets may be sent at the same time, with data packets colliding with ping responses coming back from the parent. However, this is not likely to be a problem if the data slightly precedes the scheduled ping, since there will be no need for the ping and it will be postponed by the stack.

You should configure your timings to avoid such clashes. For example, if your Routers send data every 20 seconds then a ping period of 13 seconds would be a sensible choice. However, the best way of avoiding these clashes is to add a degree of randomisation to the timings of the data transmissions - that is, offset each transmission by a random number of milliseconds from its scheduled time.

Increased Collisions with Network Depth

If packets are passing down the network at the same time as other packets are passing up the network, this contributes to the risk of packet collisions and associated packet losses. This problem becomes more acute in deeper networks. It is therefore advisable to use high values of *gJenie_MaxFailedPkts* for deep networks or control the packet direction using a pull system from the Co-ordinator.

6.8.2 Minimising Packet Loss

You can take steps in your application and your network design to make processing time available for handling packets and therefore minimise packet loss. These measures are described below.

Application Deployment

If the application makes intensive use of interrupts and dominates use of the processor in the main loop, giving very little processing resource to the stack, then the outcome will be that buffers will fill and packets will be lost. Therefore, you should not deploy such applications on nodes that need to process a high throughput of packets.

No End Device Children for Co-ordinator

If possible, do not allow End Devices to directly join the Co-ordinator node. This can be done by setting the global parameter *gMaxSleepingChildren* to 0 on the Co-ordinator. Adopting this strategy will increase the efficiency of the Co-ordinator for processing network traffic.

Start-up Delays

Substantial traffic is generated when a network starts up and nodes begin to join. This can cause congestion, collisions and lost packets, particularly at the Co-ordinator. The problem can be overcome by staggering the network join requests submitted by potential nodes. This effect can be achieved by introducing different start delays before calling **eJenie_Start()** in the joining nodes.

'Node-up' Messages

The Co-ordinator can create a list of all the nodes that have joined the network. This list can be assembled by the Co-ordinator from application-level 'node-up' messages that can be sent by the nodes as they join the network. However, these packets do not form a reliable basis for creating a node list, as they may be lost in the sudden, instense activity of a network recovery. The most reliable approach is to contruct the node list from the regular data packets received from the nodes. However, nodes that do not often send data packets to the Co-ordinator should send regular 'node-up' messages to indicate their presence. All of these packets can also be used to detect the loss of nodes from the network - a node may be considered to be lost if a number of expected packets from the node have failed to arrive.

Pushing Packets vs Pulling Packets

Sending packets up the network (for example, to the Co-ordinator) is referred to as 'pushing' packets. This can be undesirable, as it may lead to congestion, collisions and lost packets if many nodes send packets up the network at the same time. If a 'push' approach to sending data is to be adopted, it is advisable to introduce some degree of randomisation (delays) and/or beaconing to control the traffic flow. A synchronisation message can be broadcast from the Co-ordinator to all the nodes, prompting them to restart their timers. Each node can then transmit in its own timeslots, reducing the amount of simultaneous network traffic.

An alternative method of transferring packets up the network, which avoids the congestion problems of pushing packets, is to 'pull' the packets up the network. In this case, the destination node requests the packets from the source nodes by sending messages using the function **eJenie_SendData()** - for example, the Co-ordinator may request sensor readings from various nodes. This allows a node which is high in the tree, such as the Co-ordinator, to control the flow of packets up the network.

6.8.3 Route Updates

If a Router node and all of its children are moved within a network, the Routing tables for this branch of the network must be updated as quickly as possible, since packets may be lost as they are passed down stale routes. JenNet provides an automatic 'route importation' mechanism to handle these updates - this feature is described in Section 6.3.3.

6.9 Network Self-Healing

6.9.1 Automatic Recovery

The 'automatic recovery' mechanism of JenNet can be summarised as the following collection of features (previously mentioned in this chapter):

- Auto-polling feature, which prevents the accumulation of packets for an End Device in the buffers of its parent and therefore prevents the End Device from being orphaned
- End Device Child Activity Timeout feature, which detects when an End Device child is no longer active in the network (and should therefore be orphaned)
- Auto-ping feature, which allows an End Device or Router to check that its parent is still active in the network
- Maximum Failed Packets feature, which detects when a node has lost its parent

Automatic recovery can be disabled by disabling all of these features in addition to route purging, which can be disabled using the JenNet API function **vApi_SetPurgeRoute()**. It is then the responsibility of the application to detect whether communications have been lost and to take the appropriate action by calling **eJenie_Leave()** - this call first forces the local node to leave the network (if connected), then invokes a stack reset and finally forces the node to re-join the network.

To disable the automatic recovery mechanism, set the following global parameters to zero:

- *gJenie_EndDevicePollPeriod* (End Devices only)
- *gEndDeviceChildActivityTimeout* (Routers and Co-ordinator only)
- *gJenie_RouterPingPeriod* (Routers only)
- *gJenie_MaxFailedPkts*

6.9.2 Network Recovery

If the whole or part of a network suffers from a failure, such as a power outage on one or more routing nodes, the network will attempt to recover from this situation.

Normal Recovery

The most extreme case is when only the Co-ordinator is reset and the rest of the network tries to continue to function without it. When the Co-ordinator restarts, it will detect that the default PAN ID is in use by the old network and will select a new PAN ID - in this way, the Co-ordinator loses contact with its old network. In this situation, all the Co-ordinator's previous child nodes will hold all of the tree below them as a functioning network, until the maximum number of failed packets is reached for communications to the parent Co-ordinator. The child node should then attempt to re-join the Co-ordinator with the new PAN ID. So the whole network will slowly disconnect down the tree - the Co-ordinator must wait for the previous network to collapse and then re-build the whole network (with the new PAN ID). This process is slow, so it will take some time for the network to fully recover.

Recovery with Context Data

Network recovery can be speeded up by using context saving on the Co-ordinator (see Section 4.10). This requires the Co-ordinator to save context data (including the PAN ID) during normal operation. On a Co-ordinator reset, the saved data is retrieved, allowing the Co-ordinator to restart with the existing PAN ID and with the Co-ordinator's children able to just re-connect to it (thus, the normal network disassembly/reassembly process is by-passed and the network is instantly re-started).

If a node goes through a reset, it may be desirable for the application to be restored to the state that it was in before the reset - for example, in the case of a streetlight node, if the lamp was illuminated before the reset then the node should be restarted with the lamp illuminated (and not in a default 'off' state). Again, this can be achieved by storing key variables through context saving:

- If the application changes state infrequently, the state could be stored in non-volatile memory using the save context data feature.
- If the application changes state on a very regular basis then saving to non-volatile memory should be avoided, as the memory's maximum write limit may be exceeded.

The wake timer register can be used to store small quantities of data.

6.10 Key Performance Parameters

This section describes how certain key network parameters affect the performance of the network. The full set of network parameters are listed and described in Chapter 9.

6.10.1 Broadcast TTL (Time To Live)

gJenie_MaxBcastTTL

The broadcast TTL (Time To Live) is represented by the global parameter *gJenie_MaxBcastTTL* and defines the maximum number of hops for which a broadcast message will stay alive in the network. Each time the broadcast message is re-transmitted, the TTL counter of the message is decremented. When this counter reaches zero, the broadcast packet is discarded.

If a network is likely to be very long and thin, the TTL value needs to reflect the depth of the network - for example, if the network is 20 nodes deep then the TTL value should be much greater than 20 (twice the depth is a good guide, giving 40).

If you need to adjust the size of the TTL value for different broadcast packets (i.e. to vary the network penetration of the packets), you can use the JenNet API function **vApi_SetBcastTTL()** to set the required value before you send the broadcast using **eJenie_SendData()**.

The TTL count is the 'last resort' mechanism to stop circulating broadcast packets. The normal mechanism is a small history buffer of packet sequence numbers. If the sequence number has been seen before (broadcast sequence numbers are not modified by the network) then the packet is quietly discarded. Therefore, the TTL mechanism is not used under normal circumstances.

Caution: *Setting a very large TTL value to fit all possible networks is fine provided that the network is quiet. Otherwise, the high traffic level will erase the broadcast from the sequence history buffer and the packet will keep travelling through the network until the TTL count has expired. This can add to the traffic load for a short period of time.*

6.10.2 Automatic Recovery Threshold

gJenie_MaxFailedPkts

The automatic recovery threshold is represented by the global parameter *gJenie_MaxFailedPkts* and defines the maximum number of consecutive failed packets before the node will consider its connection with the network to be lost. The node will then reset the stack (and leave the network).

For large networks that are either very deep or have high traffic levels, this value should be set to 10 or higher, so that the network can tolerate intermittent packet loss or interferers.

If this value is too low then your network will occasionally change shape for no apparent reason.

Setting the value to 0 disables the failed packet detection and automatic recovery mechanisms, i.e. stops the node stack from resetting in order to leave the network and find a new parent.

6.10.3 Ping Period

gJenie_RouterPingPeriod
Jenie_EndDevicePingInterval

The ping mechanism is used by a node to test the link to its parent when there is no other application traffic. If there is regular network traffic, this traffic will allow the loss of the link to be detected and the ping mechanism can remain inactive. In a quiet network, the ping mechanism should be active and the ping period should be made as long as possible to stop unnecessary ping traffic from blocking up the network.

For a Router, the interval between consecutive pings is set through the global parameter *gJenie_RouterPingPeriod*, which must be set to the same value on child and parent Routers (including the Co-ordinator). If there is no other network traffic, the time for a Router to detect the loss of its parent or a parent to detect the loss of a Router child is given by:

gJenie_MaxFailedPkts x *gJenie_RouterPingPeriod* x 100 ms

The value of *gJenie_RouterPingPeriod* needs to be large enough not to flood the network with ping packets, but small enough to provide a reasonable detection period.

For an End Device, the global parameter *gJenie_EndDevicePingInterval* sets the interval between pings in terms of a number of sleep-wake cycles. If there is no other network traffic, the time for an End Device to detect the loss of its parent is given by:

gJenie_MaxFailedPkts x *gJenie_EndDevicePingInterval* x sleep-wake period

Since the parent has no knowledge of the sleep-wake periods of its End Device children, it applies a fixed timeout to pings from its children, where this timeout is set through the global parameter *gJenie_EndDeviceChildActivityTimeout*.

Setting *gJenie_EndDevicePingInterval* to 0 disables the automatic recovery mechanism when there is no other traffic, i.e. stops the End Device stack from resetting in order to leave the network and find a new parent. Therefore, in this case, the application will be responsible for detecting the node loss.

6.10.4 End Device Poll Period

gJenie_EndDevicePollPeriod

The rate at which an End Device polls its parent for any buffered packets is set in terms of a poll period via the global parameter *gJenie_EndDevicePollPeriod*.

Very frequent polling (a short poll period) may impact the performance of the parent Router and should be avoided. In the Router buffers, there is an 8-second packet persistence time of queued messages, so the poll period should be less than 8 seconds. The optimum poll period depends on the expected rate at which messages for the End Device will be received by the parent - you should poll frequently enough not to allow too many messages to accumulate in the Router buffers.

The End Device will automatically poll its parent when it wakes from sleep (provided that polling is not disabled - see below). Therefore, the poll period set through *gJenie_EndDevicePollPeriod* is only important when the node is awake for long periods (otherwise, polling on waking will suffice).

Automatic polling can be disabled by setting *gJenie_EndDevicePollPeriod* to 0. The application must then poll manually using **eJenie_PollParent()**.

6.10.5 End Device Scan Sleep Period

Jenie_EndDeviceScanSleepPeriod

If an End Device is not connected to a network, it will sleep between scans for a parent. The sleep period between scans is set via the global parameter *Jenie_EndDeviceScanSleepPeriod*. If a network has a large number of End Devices, this setting affects the speed of network recovery - a very long sleep period between scans means that the network will take longer to start up, but reduces the amount of beacon traffic and preserves battery life. Therefore, longer periods are recommended if there is a high density of End Devices in the same radio sphere.

Following a failed scan, if a different sleep period (than the period set through *Jenie_EndDeviceScanSleepPeriod*) is required before starting another scan, the joining functionality of the stack must first be aborted. This is achieved by calling the function **eJenie_Leave()** after the E_JENIE_STACK_RESET event which follows the failed scan. The application can then force the device to sleep for the desired duration by calling **eJenie_SetSleepPeriod()** to set the sleep duration followed by **eJenie_Sleep()** to put the device into sleep mode. This approach allows the sleep period to be altered between scan attempts - for example, to introduce extended sleep periods in order to conserve battery life while the device is failing to join a network.

Note: The ‘sleep between scans’ period can also be set at run-time using the JenNet API function **vApi_SetScanSleep()**. This setting over-rides the *Jenie_EndDeviceScanSleepPeriod* global parameter setting but does not replace it.

Part II:
Reference Information

7. Jenie API Functions

This chapter details the core functions of the Jenie API. These functions are divided into two categories, according to how they are called:

- "Application to Stack" functions, described in Section 7.1.
- "Stack to Application" functions, described in Section 7.2.

For a full introduction to the Jenie API, refer to Chapter 3.

Note: In addition to the functions of the Jenie API, functions of the JenNet API are also available and are described in Chapter 8. The JenNet API functions are intended for advanced users who require more control over the network than is available using the Jenie API.

7.1 "Application to Stack" Functions

This section details the "Application to Stack" functions of the Jenie API. These functions are called in the application to invoke tasks in the underlying stack. They are pre-defined in the header file **Jenie.h**.

The function descriptions are divided by sub-section into functions that deal with management tasks, data transfer tasks and operating system tasks.

Caution: *The "Application to Stack" functions described in this section must not be called from interrupt context (for example, from within a user-defined callback function). Instead, the application should set a flag to indicate that the call should be made later, outside of interrupt context.*

7.1.1 Network Management Functions

The network management functions are largely concerned with tasks to start and form the wireless network. These tasks include:

- Configure and initialise network
- Start a device as a Co-ordinator, Router or End Device
- Determine whether a Router or Co-ordinator is accepting join requests
- Advertise local node services and seek remote node services
- Establish bindings between local and remote node services
- Configure security used for message encryption/decryption

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

eJenie_Start

```
teJenieStatusCode eJenie_Start(
                                    teJenieDeviceType eDevType);
```

Description

This function is normally (but not always) called from **eJenie_CbInit()** and starts the stack on the device.

- On the Co-ordinator, this will start a network.
- On an End Device or Router, starting the stack causes the node to find and join a network.
- On a sleeping End Device, once the device has woken from sleep with memory contents held, this function will cause the stack to resume without the device needing to re-associate with its parent.

The appropriate behaviour of this function for a given node type requires the application to be linked with the relevant library file - **Jenie_TreeCRLib.a** for the Co-ordinator or a Router, **Jenie_TreeEDLib.a** for an End Device.

Parameters

eDevType	Indicates the role of the device in the network - one of Co-ordinator, Router, End Device: E_JENIE_COORDINATOR E_JENIE_ROUTER E_JENIE_END_DEVICE

Returns

One of:

E_JENIE_SUCCESS

E_JENIE_ERR_INVLD_PARAM

For explanations, refer to Chapter 10.

eJenie_Leave

```
teJenieStatusCode eJenie_Leave(void);
```

Description

This function disassociates the node from its parent and therefore causes the device to leave the network.

On leaving the network, the device enters the idle state in which **vJenie_CbMain()** is called regularly, but the device does not necessarily attempt to establish or join a network. The device will remain in the idle state until **eJenie_Start()** is called.

Parameters

None

Returns

One of:

E_JENIE_SUCCESS
E_JENIE_ERR_UNKNOWN
E_JENIE_ERR_STACK_BUSY

For explanations, refer to Chapter 10.

eJenie_RegisterServices

```
teJenieStatusCode eJenie_RegisterServices(
                                    uint32 u32Services);
```

Description

This function is used to register the services available on the local node.

To do this, a list of supported services is submitted to the network as a 32-bit value based on the network's Service Profile, in which each bit position corresponds to a particular service in the network. Here, a bit value of '1' indicates the corresponding service is supported, while '0' indicates the service is not supported.

- If the local node is the Co-ordinator or a Router, the registered services are held locally and this function can return immediately with status code success or failure.
- If the local node is an End Device, the registered services are submitted to its parent and the status code is deferred - it is eventually received as the stack management event E_JENIE_REG_SVC_RSP via a call to the callback function **vJenie_CbStackMgmtEvent()**.

This function will not successfully return until the network is up and running. Until the network is up, the function will return the error code E_JENIE_ERR_STACK_BUSY.

Parameters

u32Services	32-bit value detailing the services to be registered (see above)

Returns

One of:

E_JENIE_SUCCESS
E_JENIE_DEFERRED
E_JENIE_ERR_UNKNOWN
E_JENIE_ERR_STACK_BUSY

For explanations, refer to Chapter 10.

eJenie_RequestServices

```
teJenieStatusCode eJenie_RequestServices(
                                    uint32 u32Services
                                    bool_t bMatchAll);
```

Description

This function is used to find remote nodes that have the specified services. The remote nodes will respond individually.

The requested services are specified as a 32-bit value based on the network's Service Profile, in which each bit position corresponds to a particular service in the network. Here, a bit value of '1' indicates the corresponding service is requested, while '0' indicates the service is not requested.

You must also specify whether all of the requested services or at least one of the requested services must be present on the remote node for the latter to generate a response.

The function returns almost immediately but will not be successful until the network is up and running. Until the network is up, the function will return the error code E_JENIE_ERR_STACK_BUSY.

Responses from the remote nodes will be received as a series of E_JENIE_SVC_REQ_RSP stack events via the callback function **vJenie_CbStackMgmtEvent()**.

Parameters

u32Services	32-bit value detailing the requested services (see above)
bMatchAll	Indicates whether ALL or ANY of the requested services should be present on the remote node to warrant a response: TRUE: All the requested services should be present FALSE: Any of the requested services should be present

Returns

One of:

E_JENIE_ERR_INVLD_PARAM
E_JENIE_DEFERRED
E_JENIE_ERR_UNKNOWN
E_JENIE_ERR_STACK_BUSY

For explanations, refer to Chapter 10.

eJenie_BindService

```
teJenieStatusCode eJenie_BindService(
                                    uint8 u8SrcService,
                                    uint64 u64DestAddr,
                                    uint8 u8DestService);
```

Description

This function is used to bind a local service to the specified service on the specified remote node. Among the parameters of this function, you must specify the address of the remote node (*u64DestAddr*) and the remote service (*u8DestService*). These two pieces of information will have been obtained from the event E_JENIE_SVC_REQ_RSP received as the result of a service request submitted using the function **eJenie_RequestServices()**.

Once a service binding has been created, messages can be sent to the remote service(s) using the function **eJenie_SendDataToBoundService()**.

If you wish to subsequently unbind two services, use the **eJenie_UnBindService()** function.

Note: You can call **eJenie_BindService()** more than once to bind a local source service to several destination services. However, if using the Jenie API v1.4 or lower, you are advised not to bind to more than four destination services.

Parameters

u8SrcService	Service ID of local service to be bound
u64DestAddr	Address of the remote node which contains the bound service
u8DestService	Service ID of the service to be bound to on the remote node

Returns

One of:

E_JENIE_SUCCESS
E_JENIE_ERR_INVLD_PARAM
E_JENIE_ERR_STACK_RSRC

For explanations, refer to Chapter 10.

eJenie_UnBindService

```
teJenieStatusCode eJenie_UnBindService(
                                    uint8 u8SrcService,
                                    uint64 u64DestAddr,
                                    uint8 u8DestService);
```

Description

This function is used to unbind a local service from the specified service on the specified remote node. The services must have been previously bound using the function **eJenie_BindService()**. Among the parameters, you must specify the local service (*u8SrcService*), the address of the remote node (*u64DestAddr*) and the remote service (*u8DestService*) to be unbound. These parameters can be used as described in the table below to remove one or more bindings in one function call:

Source Service ID	Remote Node Address	Remote Service ID	Action
1-32	Valid address	1-32	Remove the specific entry described by the three parameters
0xFF	Not used	1-32	Remove all bindings where the destination service matches the parameter passed
1-32	Valid address	0xFF	Remove all bindings where the source service matches the parameter passed
0xFF	Not used	0xFF	Remove all bindings

Once the services have been unbound, messages can no longer be sent to the remote service(s) using the function **eJenie_SendDataToBoundService()**.

Parameters

u8SrcService	Service ID of local service to be unbound
u64DestAddr	Address of the remote node which contains the bound service
u8DestService	Service ID of the service to be unbound on the remote node

Returns

One of:

E_JENIE_SUCCESS
E_JENIE_ERR_INVLD_PARAM
E_JENIE_ERR_STACK_RSRC

For explanations, refer to Chapter 10.

eJenie_SetPermitJoin

```
teJenieStatusCode eJenie_SetPermitJoin(
                                    bool_t bAssociate);
```

Description

This function is used to enable or disable "permit joining" on the Co-ordinator or a Router - that is, it configures the device to allow or forbid other devices (End Devices or Routers) to associate with it, and therefore to join the network.

Parameters

bAssociate	"Permit joining" status to set: TRUE - allow joinings FALSE - forbid joinings

Returns

E_JENIE_SUCCESS

bJenie_GetPermitJoin

```
bool_t bJenie_GetPermitJoin(void);
```

Description

This function is used to obtain the current "permit joining" state of the Co-ordinator or Router - that is, whether the device is currently allowing other devices (End Devices or Routers) to associate with it, and therefore to join the network.

Parameters

None

Returns

One of:

TRUE - joinings allowed

FALSE - joinings forbidden

eJenie_SetSecurityKey

```
teJenieStatusCode eJenie_SetSecurityKey(
                                    tsJenieSecKey *pKey,
                                    uint64 u64Addr);
```

Description

This function is used to enable security and set a key value for encrypting/decrypting data during communications between the local node and the specified remote node - that is, the local node will encode the data with the specified key and the remote will decode the data with the same key. Note that this function must therefore also be called on the remote node to set the same key value.

The function should be called from within the callback function **vJenie_CbInit()** and should not be called from within **vJenie_CbConfigureNetwork()**.

When security is enabled, the data that is encrypted is the payload of the IEEE 802.15.4 MAC frame.

Caution: *In the current software release, the specified security key is used for communication with all nodes (the specified address is ignored). All nodes must use the same key. Therefore, this function only needs to be called once for communication with the whole network.*

This function can also be used to disable security in communications with the specified remote node by specifying a NULL security key pointer.

Parameters

***pKey**	Pointer to a security key. A NULL pointer disables security.
u64Addr	Address of remote node associated with specified key - ignored in current release (see Caution above).

Returns

One of:

E_JENIE_SUCCESS

E_JENIE_ERR_INVLD_PARAM

For explanations, refer to Chapter 10.

7.1.2 Data Transfer Functions

The data transfer functions are concerned with sending and receiving data. These tasks include:

- Send data to a remote node or broadcast data to all Router nodes
- Send data to a bound service on a remote node

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

eJenie_SendData

```
teJenieStatusCode eJenie_SendData(uint64 u64DestAddr,
                                  uint8 *pu8Payload,
                                  uint16 u16Length,
                                  uint8 u8TxFlags);
```

Description

This function is used to send data to the specified remote node. This type of send requires the address of the destination node - this is the 64-bit IEEE/MAC address of the device. This address will have been previously obtained as the result of a Service Discovery implemented using **eJenie_RequestServices()**.

A data broadcast to all Router nodes can also be performed using this function - in this case, the destination address must be set to zero and `TXOPTION_BDCAST` must be selected in the transmission options (*u8TxFlags*).

The maximum payload data size depends on the type of transmission (and therefore the JenNet frame type) and whether security has been enabled (using the function **eJenie_SetSecurityKey()**), as follows:

Type of Transmission	Security Disabled	Security Enabled
Broadcast to all nodes	89 bytes	68 bytes
Unicast to Co-ordinator	90 bytes	69 bytes
Unicast to any other node	82 bytes	61 bytes

This function will not successfully return until the network is up and running. Until the network is up, the function will return the error code E_JENIE_ERR_STACK_BUSY.

A call to this function will (eventually) result in an E_JENIE_PACKET_SENT or E_JENIE_PACKET_FAILED event to indicate the success or failure of the sent message reaching the first hop to the destination, unless the transmission option `TXOPTION_SILENT` or `TXOPTION_BDCAST` has been set in which case these events are not generated.

Parameters

u64DestAddr	Address of the destination node. For a broadcast or to send data to the Co-ordinator, this parameter must be set to zero (for a broadcast, *u8TxFlags* must also be set appropriately)
***pu8Payload**	Pointer to the data to be sent
u16Length	Length of data to be sent, in bytes (for limits, see above)
u8TxFlags	Sets the transmission options. These options are detailed in Table 6 on page 164. The values can be logical ORed to simultaneously specify more than one option

Returns

One of:

E_JENIE_ERR_INVLD_PARAM
E_JENIE_DEFERRED
E_JENIE_ERR_UNKNOWN
E_JENIE_ERR_STACK_BUSY

For explanations, refer to Chapter 10.

eJenie_SendDataToBoundService

```
teJenieStatusCode eJenie_SendDataToBoundService(
                                  uint8 u8Service,
                                  uint8 *pu8Payload,
                                  uint16 u16Length,
                                  uint8 u8TxFlags);
```

Description

This function is used to send data from a local service to a remote service, where these services have previously been bound using **eJenie_BindService()**. Only the local service needs to be specified.

The maximum payload data size depends on whether security has been enabled (using the function **eJenie_SetSecurityKey()**), as follows:

- If security is disabled, the maximum data size is 74 bytes.
- If security is enabled, the maximum data size is 53 bytes.

This function will not successfully return until the network is up and running. Until the network is up, the function will return the error code E_JENIE_ERR_STACK_BUSY.

A call to this function will (eventually) result in an E_JENIE_PACKET_SENT or E_JENIE_PACKET_FAILED event to indicate the success or failure of the sent message reaching the first hop to the destination.

Parameters

u8Service	Service ID of local service from which data is to be sent
***pu8Payload**	Pointer to the data to be sent
u16Length	Length of data to be sent, in bytes (for limits, see above)
u8TxFlags	Sets the transmission options. These options are detailed in Table 6 on page 164. The values can be logical ORed to simultaneously specify more than one option

Returns

One of:

E_JENIE_ERR_INVLD_PARAM
E_JENIE_DEFERRED
E_JENIE_ERR_UNKNOWN
E_JENIE_ERR_STACK_BUSY

For explanations, refer to Chapter 10.

eJenie_PollParent

```
teJenieStatusCode eJenie_PollParent(void);
```

Description

This function is used by an End Device to check if its parent is holding pending data for it. If data is pending, a data event will be received after a short delay via the callback function **vJenie_CbStackDataEvent()**. The function **eJenie_PollParent()** can, for example, be called after the End Device has come out of sleep mode.

The function will not successfully return until the network is up and running. Until the network is up, the function will return the error code E_JENIE_ERR_STACK_BUSY. Therefore, the function should not be called until an E_JENIE_NETWORK_UP event has been generated (and must not be called immediately after a E_JENIE_STACK_RESET event).

If a call to this function is successful (i.e. it returns a status of E_JENIE_DEFERRED) then an E_JENIE_POLL_CMPLT event will be generated. If this event contains a status value of E_JENIE_POLL_DATA_READY, this indicates that data is available which will follow immediately in a E_JENIE_DATA event. However, this data event may not deliver all the pending data for the node. You are therefore advised to call **eJenie_PollParent()** repeatedly until there is no further pending data, indicated when the event E_JENIE_POLL_CMPLT contains a status value of E_JENIE_POLL_NO_DATA.

The E_JENIE_POLL_CMPLT event will also be generated (and the status E_JENIE_DEFERRED returned) if no response is received from the parent. In this case, the event also contains a status value of E_JENIE_POLL_NO_DATA.

Caution 1: *Call this function regularly if auto-polling is disabled (through the global parameter gJenie_EndDevicePollPeriod), since a build-up of unclaimed data for the End Device on its parent will eventually cause the End Device to be orphaned by its parent.*

Caution 2: *Do not call this function repeatedly with an interval of less than 100 ms between calls, otherwise the stack may freeze.*

Parameters

None

Returns

One of:

E_JENIE_DEFERRED
E_JENIE_ERR_UNKNOWN
E_JENIE_ERR_STACK_BUSY

For explanations, refer to Chapter 10.

7.1.3 System Functions

The system functions are largely concerned with implementing sleep mode and controlling the radio transmitter. These tasks include:

- Save and restore context data
- Configure and start sleep mode
- Configure, start and stop the radio transmitter
- Obtain the version number of a component on the node

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

vJPDM_SaveContext

```
void vJPDM_SaveContext(void);
```

Description

This function is used to save both network and application context data to external non-volatile memory. This allows the data to be recovered and the node to resume normal operation following power loss to the on-chip memory (e.g. power failure or sleep without memory held). Network and application context save/restore can be individually enabled but if both are enabled, a single call to this function will save both sets of context data.

Caution: *The function* ***vJPDM_SaveContext()*** *must only be called in the main loop callback function,* ***vJenie_CbMain()****. It must not be called in event handling callback functions.*

To enable save/restore of network context, the global parameter *gJenie_RecoverFromJpdm* must be set to TRUE within the callback function **vJenie_CbConfigureNetwork()**. The saved data will then be automatically recovered when the stack is re-started using **eJenie_Start()**.

To enable save/restore of application context, the **eJPDM_RestoreContext()** function must be called within the callback function **vJenie_CbInit()**.

Note that for a Router, Child/Neighbour tables and Routing tables will not be saved.

Parameters

None

Returns

None

eJPDM_RestoreContext

```
teJenieStatusCode eJPDM_RestoreContext(
                    tsJPDM_BufferDescription *psDescription);
```

Description

This function is used to retrieve application context data stored in external non-volatile memory, where this data was previously saved using the function **vJPDM_SaveContext()**. This allows application data to be recovered so that the node can resume normal operation following power loss to the on-chip memory (e.g. power failure or sleep without memory held).

The **eJPDM_RestoreContext()** function must be included in the callback function **vJenie_CbInit()** for a cold start:

- The first time the application is run, the function registers a buffer in on-chip memory where the application context data will be stored - this buffer is set up using the macro **JPDM_DECLARE_BUFFER_DESCRIPTION** (see below).
- When the application is subsequently re-started, the function will recover saved application context data from external non-volatile memory. The recovered data is stored in the buffer set up using **JPDM_DECLARE_BUFFER_DESCRIPTION**.

The above macro is defined as follows:

JPDM_DECLARE_BUFFER_DESCRIPTION(*name*, *ptr*, *size***)**

where:

name is a label for the buffer as an ASCII string in quotes

ptr is a pointer to the start of the buffer in on-chip memory

size is the number of bytes in the buffer

Parameters

***psDescription**	Pointer to descriptor of on-chip memory buffer in which application context data will be stored.

Returns

One of:

E_JENIE_SUCCESS

E_JENIE_ERR_INVLD_PARAM

For explanations, refer to Chapter 10.

The invalid parameter code is returned, for example, if the specified memory buffer size is too large (it must not be greater than the external memory sector size determined by the global variable *gJpdmSectorSize*).

vJPDM_EraseAllContext

```
void vJPDM_EraseAllContext(void);
```

Description

This function can be used to erase all context data (application and network) in non-volatile memory, previously stored using the function **vJPDM_SaveContext(void)**.

You are likely to want to do this in order to revert back to the default context data. To prevent the current context data from automatically being re-saved in non-volatile memory, you should immediately follow this function call with a software reset, by calling **vJPI_SwReset()**. This will ensure that the current context data is lost and the default context data is restored to RAM.

Parameters

None

Returns

None

eJenie_SetSleepPeriod

```
teJenieStatusCode eJenie_SetSleepPeriod(
                                uint32 u32SleepPeriodMs);
```

Description

This function can be used on an End Device to set the duration for which the device will sleep when put into sleep mode using the function **eJenie_Sleep()**.

This wake method uses an on-chip wake timer, for which the 32-kHz oscillator must be running during sleep. Therefore, a sleep mode with oscillator running must be specified through **eJenie_Sleep()**.

Note: The sleep duration must be set to at least 100 ms. The stack imposes a 100-ms minimum sleep period since a shorter period could result in a wake timer firing before the pre-sleep housekeeping tasks have been completed.

Caution: *If you set a long sleep duration, greater than 7 s (7000 ms), avoid sending data to this End Device while it is asleep (while it is not polling its parent for data). This will prevent the End Device from being orphaned by its parent.*

Parameters

u32SleepPeriodMs	Sleep duration, in milliseconds (at least 100)

Returns

E_JENIE_SUCCESS

eJenie_Sleep

```
teJenieStatusCode eJenie_Sleep(
                    teJenNetSleepMode eSleepMode);
```

Description

This function can be used to put an End Device into sleep mode. The function informs the stack that the application will be ready to sleep once it has performed any tasks that remain to be completed. It must be called from **vJenie_CbMain()** only (and from no other callback function).

The following sleep options can be specified:

- with or without the 32-kHz on-chip oscillator running
- with or without preserving the contents of on-chip RAM (memory held)

Note that 'doze mode' of the JN5148/JN5139 device is not supported by JenNet.

The device can be pre-configured to sleep for a fixed duration, set using the function **eJenie_SetSleepPeriod()**. This wake method uses the on-chip wake timers, for which the 32-kHz oscillator must be running during sleep. The device can alternatively be woken from sleep by an event deriving from the on-chip comparators or DIOs - this method does not require the oscillator to be running during sleep.

Holding memory during sleep enables the device to retain context data which will allow the device to quickly resume its network operation on waking. However, "sleep with memory held" consumes more power than "sleep without memory held". If you have selected "sleep without memory held", you can save context data (externally) before sleeping using the function **vJPDM_SaveContext()**.

On waking from sleep, the network stack calls the function **vJenie_CbInit()**, and the device remains in the idle state and does not rejoin the network until this function calls **eJenie_Start()**. While in the idle state, **vJenie_CbMain()** is regularly called by the network stack, so that other necessary tasks can be performed. If you have selected "sleep without memory held", you will need to perform a cold restart and retrieve the stored application context data by calling the function **eJPDM_RestoreContext()** before calling **eJenie_Start()** in **vJenie_CbInit()**.

Note that if an on-chip wake timer is used to wake the device from "sleep with memory held", no event is generated via the **vJenie_CbHwEvent()** function or the registered system controller callback function (although a wake-up initiated by a DIO or comparator will generate a hardware event).

In 'deep sleep' mode, all switchable power domains are powered off and the 32-kHz oscillator is stopped. This mode can only be exited by power cycling (switching off then on) or resetting the chip (a DIO event can be used to trigger a reset).

Parameters

eSleepMode Specifies the required sleep mode, one of:
- E_JENIE_SLEEP_OSCON_RAMON
- E_JENIE_SLEEP_OSCON_RAMOFF
- E_JENIE_SLEEP_OSCOFF_RAMON
- E_JENIE_SLEEP_OSCOFF_RAMOFF
- E_JENIE_SLEEP_DEEP

Returns

E_JENIE_SUCCESS

E_JENIE_ERR_UNKNOWN

eJenie_RadioPower

```
teJenieStatusCode eJenie_RadioPower(int8 iPowerLevel,
                                    bool_t bHighPower);
```

Description

This function can be used to set the transmit power level of the radio transceiver, or to switch the radio transceiver on or off.

The transmit power level can be set to values which depend on the module type. The possible values are listed in the table below.

Power Level (dBm)			
JN5139 Modules		**JN5148 Modules**	
Standard	**High-Power**	**Standard**	**High-Power**
-30	-7	-32	-16.5
-24	-1	-20.5	-5
-18	+5	-9	+6.5
-12	+11	+2.5	+18
-6	+15	-	-
+1.5	+17.5	-	-

The default power setting is the highest power level for the module type.

Set the parameter *iPowerLevel* to the nearest integer to the desired power level - for example, to achieve a power level of +6.5 dBm on a JN5148 high-power module, set the parameter to +6 or +7. In addition to the above values, 20 and 21 are used to switch the radio transmitter on and off, respectively (enumerations are available to do this). An 'invalid parameter' error will be returned if an out-of-range power level is specified.

To set the power level for a high-power module, you must enable high-power mode using the parameter *bHighPower*.

Caution: *This function should be called only after* ***eJenie_Start()*** *has been called, otherwise it will have no effect. It can be called immediately after* ***eJenie_Start()*** *to configure the radio power at the earliest opportunity.*

Note: 'Boost mode' on the JN5139 device, detailed in the *JN5139 Datasheet (JN-DS-JN5139)*, is not supported by JenNet and cannot be configured using this function.

Parameters

iPowerLevel Desired power level (nearest integer), or one of:

E_JENIE_RADIO_OFF - switch radio transceiver off
E_JENIE_RADIO_ON - switch radio transceiver on

bHighPower Enables high-power mode for a high-power module:

TRUE - high-power mode enabled
FALSE - high-power mode disabled

Returns

One of:

E_JENIE_SUCCESS

E_JENIE_ERR_INVLD_PARAM

For explanations, refer to Chapter 10.

u32Jenie_GetVersion

```
uint32 u32Jenie_GetVersion(
                    teJenieComponent eComponent);
```

Description

This function is used obtain the version number of the specified component of the system. The function provides a means of checking that the host device is operating.

Parameters

eComponent Component for which version number is needed, one of:

E_JENIE_COMPONENT_JENIE (Jenie API)
E_JENIE_COMPONENT_NETWORK (JenNet)
E_JENIE_COMPONENT_MAC (IEEE 802.15.4)
E_JENIE_COMPONENT_CHIP (JN51xx chip)

Returns

Version number of component, as described below:

Component	Bits	Description
E_JENIE_COMPONENT_JENIE	31-0	Jenie API version number
E_JENIE_COMPONENT_NETWORK	31-16	Network stack protocol (JenNet) revision
	15-0	Network stack software revision
E_JENIE_COMPONENT_MAC (IEEE 802.15.4)	31-24	Non-zero value identifying special or custom build
	23-16	Really major revision
	15-8	Minor (patch) revision
	7-0	Major revision (only changes with new ROM version)
E_JENIE_COMPONENT_CHIP	31-28	Revision number: 0x0 for R0, 0x1 for R1, etc
	27-22	Metal mask version ID
	21-12	Part number: 0x000 for JN5121 0x002 for JN5139 0x004 for JN5148
	11-0	Manufacturer's identification

7.1.4 Statistics Functions

The statistics functions are concerned with interrogating the Routing and Neighbour tables of the local node:

- A Neighbour table contains routing information for all immediate children as well as the node's parent (which is the first entry in the table).
- A Routing table contains routing information for all descendant nodes (lower in the tree) that are not immediate children.

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

u16Jenie_GetRoutingTableSize

```
uint16 u16Jenie_GetRoutingTableSize(void);
```

Description

This function obtains the number of valid entries in the local node's Routing table. Since the table is likely to become fragmented, the value returned is the number of valid entries in the table and not the size of the table.

This function is only applicable to routing nodes (Routers and Co-ordinator).

Parameters

None

Returns

Number of valid entries in the local Routing table

eJenie_GetRoutingTableEntry

```
teJenieStatusCode eJenie_GetRoutingTableEntry(
                  uint16 u16EntryNum,
                  tsJenie_RoutingEntry *psRoutingEntry);
```

Description

This function is used to obtain the specified entry of the local node's Routing table.

- If the entry exists, the function returns E_JENIE_SUCCESS and populates the structure of type **tsJenie_RoutingEntry** pointed to by *psRoutingEntry* - for details of this structure, see Section 10.2.
- If the entry does not exist, the function returns E_JENIE_ERR_INVLD_PARAM (referring to the *u16EntryNum* parameter) but fills in the `u16TotalEntries` field of the structure pointed to by *psRoutingEntry*.

Parameters

u16EntryNum	Index of the required Routing table entry (this value is a 'logical index' and not the physical location of the entry in the table).
***psRoutingEntry**	Pointer to the data structure of type **tsJenie_RoutingEntry** to be automatically filled in - see Section 10.2.

Returns

One of:

E_JENIE_SUCCESS

E_JENIE_ERR_INVLD_PARAM

For explanations, refer to Chapter 10.

u8Jenie_GetNeighbourTableSize

```
uint8 u8Jenie_GetNeighbourTableSize(void);
```

Description

This function is used to obtain the size (number of entries) of the local node's Neighbour table. The result includes the node's parent as well as its children.

This function is only applicable to routing nodes (Routers and Co-ordinator).

Parameters

None

Returns

Number of entries in the Neighbour table

eJenie_GetNeighbourTableEntry

```
teJenieStatusCode eJenie_GetNeighbourTableEntry(
                uint8 u8EntryNum,
                tsJenie_NeighbourEntry *psNeighbourEntry);
```

Description

This function is used to obtain the specified entry of the local node's Neighbour table.

- If the entry exists, the function returns E_JENIE_SUCCESS and populates the structure of type **tsJenie_NeighbourEntry** pointed to by *psNeighbourEntry* - for details of this structure, see Section 10.2.
- If the entry does not exist, the function returns E_JENIE_ERR_INVLD_PARAM (referring to the *u8EntryNum* parameter) but fills in the `u8TotalEntries` field of the structure pointed to by *psNeighbourEntry*.

This function is only applicable to routing nodes (Routers and Co-ordinator).

Note that entry zero of a Neighbour table is always for the node's parent. Since the Co-ordinator has no parent, entry zero should never be specified in this function for the Co-ordinator.

Parameters

u8EntryNum	Index of the required Neighbour table entry (this value is a 'logical index' and not the physical location of the entry in the table).
***psNeighbourEntry**	Pointer to the data structure of type **tsJenie_NeighbourEntry** to be automatically filled in - see Section 10.2.

Returns

One of:

E_JENIE_SUCCESS

E_JENIE_ERR_INVLD_PARAM

For explanations, refer to Chapter 10.

eJenie_ResetNeighbourStats

```
teJenieStatusCode eJenie_ResetNeighbourStats(
                                    uint16 u16EntryNum);
```

Description

This function is used to reset the statistics components of the specified Neighbour table entry on the local node. The components that are reset are:

- `u8LinkQuality` - quality of link with relevant neighbouring node
- `u16PktsLost` - number of unacknowledged packets sent to the node
- `u16PktsSent` - number of acknowledged packets sent to the node
- `u16PktsRcvd` - number of packets received from the node

If the entry is not found, the function returns E_JENIE_ERR_INVLD_PARAM.

Parameters

u16EntryNum	Index of the relevant Neighbour table entry (this value is a 'logical index' and not the physical location of the entry in the table).

Returns

One of:

E_JENIE_SUCCESS

E_JENIE_ERR_INVLD_PARAM

For explanations, refer to Chapter 10.

7.2 "Stack to Application" Functions

This section details the "Stack to Application" functions of the Jenie API. These are callback functions triggered by events from the underlying stack. They provide the opportunity for the application software to receive information and respond at defined points during program execution, such as at stack initialisation, or at regular intervals.

Note: You must define these callback functions in your application code, even those functions that are not used in your code (and are therefore empty).

The callback functions handle:

- stack management events
- data events
- hardware events

Stack management and data events are described in Chapter 11. Hardware events are interrupts generated by the on-chip peripherals and are described in the *Integrated Peripherals API User Guide (JN-UG-3066)*.

Note: None of these functions except **vJenie_CbInit()** is allowed to block.

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

vJenie_CbConfigureNetwork

```
void vJenie_CbConfigureNetwork(void);
```

Description

This function is the first callback of an application and is called before the stack initialises itself, providing the application with the opportunity to initialise/override default stack parameters - for full details of these parameters, refer to Chapter 9. The function is only called during a cold start.

Parameters

None

Returns

None

vJenie_CbInit

```
void vJenie_CbInit(bool_t bWarmStart);
```

Description

This function is called after the stack has initialised itself. It provides the application with the opportunity to perform any additional hardware or software initialisation that may be required.

This callback function should normally include a call to the function **eJenie_Start()**.

Parameters

bWarmStart Specifies whether the device has undergone a cold or warm start:

TRUE - warm start
FALSE - cold start

Returns

None

vJenie_CbMain

```
void vJenie_CbMain(void);
```

Description

This function is the main application task. It is called many times per second by the stack and provides the opportunity for the application to perform any processing that is required. This function should be non-blocking.

Parameters

None

Returns

None

vJenie_CbStackMgmtEvent

```
void vJenie_CbStackMgmtEvent(
                        teJenieEventType eEventType,
                        void *pvEventPrim);
```

Description

This function is called by the stack to inform the application that one of a number of stack management events has occurred. For example, the node may have received a service request response from a remote node.

For further details of the stack management events, refer to Section 11.1.

Parameters

eEventType — The type of stack management event received, one of:

- E_JENIE_REG_SVC_RSP
- E_JENIE_SVC_REQ_RSP
- E_JENIE_POLL_CMPLT
- E_JENIE_PACKET_SENT
- E_JENIE_PACKET_FAILED
- E_JENIE_NETWORK_UP
- E_JENIE_STACK_RESET
- E_JENIE_CHILD_JOINED
- E_JENIE_CHILD_LEAVE
- E_JENIE_CHILD_REJECTED

**pvEventPrim* — Pointer to event primitive (if relevant, or NULL if not)

Returns

None

vJenie_CbStackDataEvent

```
void vJenie_CbStackDataEvent(
                                teJenieEventType eEventType,
                                void *pvEventPrim);
```

Description

This function is called by the stack to inform the application that one of a number of stack data events has occurred. For example, the node may have received a message from a remote node or a response to one of its own messages.

For further details of the data events, refer to Section 11.2.

Parameters

eEventType	The type of data event received, one of: E_JENIE_DATA E_JENIE_DATA_TO_SERVICE E_JENIE_DATA_ACK E_JENIE_DATA_TO_SERVICE_ACK
**pvEventPrim*	Pointer to event primitive (if relevant, or NULL if not)

Returns

None

vJenie_CbHwEvent

```
void vJenie_CbHwEvent(uint32 u32DeviceId,
                      uint32 u32ItemBitmap);
```

Description

This function is called by the stack to inform the application that a hardware event has occurred - that is, an event has been generated by an on-chip peripheral of the JN5148 or JN5139 device.

Caution: *A hardware event is provided by an on-chip tick timer every 10 ms. This tick timer cannot be controlled by the application and is not guaranteed to always run on the JN5139 device. Therefore, your application must not use this tick timer directly.*

The on-chip peripherals can be controlled using the Integrated Peripherals API, described in the *Integrated Peripherals User Guide (JN-UG-3066)*. The parameters *u32DeviceId* and *u32ItemBitmap* of this function are identical to the parameters of the callback functions in the Integrated Peripherals API.

Parameters

u32DeviceId	Indicates the on-chip peripheral that generated the event
u32ItemBitmap	Indicates the source within the peripheral that caused the event Bits 15-8 of this bitmap parameter are also used to deliver a received data byte to the application when a UART ‘received data’ or ‘timeout’ interrupt occurs.

Returns

None

8. JenNet API Functions

This chapter details the core functions of the JenNet API. The JenNet API functions are intended for advanced users who require more control over the network than is available using the Jenie API (detailed in Chapter 7).

If using the JenNet API, your project must include the JenNet header file **SDK\Jenie\Include\JenNetApi.h**, as well as **SDK\Common\Include\mac_sap.h** for the declarations of the structures `MAC_Addr_s` and `MAC_ExtAddr_s`.

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

eApi_SendDataToExtNwk

```
teJenNetStatusCode eApi_SendDataToExtNwk(
                                MAC_Addr_s *psDestAddr,
                                uint8 *pu8Payload,
                                uint8 u8Length);
```

Description

This function is used to request the transmission of a data frame to another node that is not necessarily in the same network (not necessarily having the same PAN ID).

The destination address is specified using the `MAC_Addr_s` structure (shown in Section 10.2.5), which allows the application to specify the destination PAN ID and either a 64-bit extended address (IEEE/MAC address) or a 16-bit short address (as used in IEEE 802.15.4).

If a broadcast short address and a broadcast PAN ID are used, the packet will be sent to all nodes within radio range, irrespective of which network they are in.

The JenNet parameter *bPermitExtNwkPkts* is set to FALSE by default. Setting this to TRUE for the local node enables the reception of external network packets (packets for which the source PAN ID is not the same as the local PAN ID).

Parameters

**psDestAddr*	Pointer to address of the destination node Note that the `MAC_Addr_s` structure contains the PAN ID and either a 16-bit short or 64-bit extended address
**pu8Payload*	Pointer to the data to be sent
u8Length	Length of the data to be sent, in bytes

Returns

E_JENNET_DEFERRED
: The node successfully passed the packet to the IEEE 802.15.4 MAC layer

E_JENNET_ERROR
: The node was not able to pass the request into the IEEE 802.15.4 MAC layer

vNwk_DeleteChild

void vNwk_DeleteChild(MAC_ExtAddr_s *_psNodeAddr_);

Description

This function is used on a parent node to force an immediate child to leave the network by deleting its entry in the local Neighbour table. The node to be removed is specified using its 64-bit IEEE/MAC address in the `MAC_ExtAddr_s` structure (shown in Section 10.2.6).

There will be a delay before the child node attempts to rejoin a network, as its 'failed packet threshold' must first be exceeded.

Note that **vNwk_DeleteChild()** is called on the parent node. In contrast, the Jenie API function **eJenie_Leave()** can be called on a child node to remove itself from the network.

Parameters

*_psNodeAddr_	Pointer to the IEEE/MAC address of the node to remove

Returns

None

vApi_SetScanSleep

void vApi_SetScanSleep(uint32 *u32ScanSleepDuration***);**

Description

This function allows the application to set the scan sleep duration at run-time. It only applies to End Devices since Routers/Co-ordinators are not able to sleep.

The scan sleep period is the amount of time for which the End Device sleeps between channel scans when trying to join the network - that is, if the device fails to join the network after one scan, it will sleep for this period before scanning again. Increasing this period will help to preserve battery life in the End Device.

Obtaining no results in the scan sort callback function (registered using **vApi_RegScanSortCallback()**) or a STACK_RESET event are useful points at which to change the scan sleep period.

Parameters

u32ScanSleepDuration	Time, in milliseconds, to sleep after scan timeout

Returns

None

vApi_SetBcastTTL

void vApi_SetBcastTTL(uint8 *u8MaxTTL*);

Description

This function allows the application to modify the TTL (Time To Live) of broadcast packets that originate from the local node. The TTL value is defined as the maximum number of hops of a broadcast message. To allow broadcast packets to propagate all the way through the network, this value should be set to at least the expected depth of the network. In fact, the parameter *u8MaxTTL* should be set as follows:

u8MaxTTL = Desired maximum number of broadcast hops - 1

Parameters

u8MaxTTL	Maximum number of hops - 1 Therefore, for a single hop, set this value to 0

Returns

None

vApi_SetPurgeRoute

void vApi_SetPurgeRoute(bool_t *bPurge***);**

Description

This function is used to tailor the route maintenance behaviour by allowing route purging to be enabled/disabled.

By default, all Routers and the Co-ordinator will periodically check all entries in their Routing tables for possible stale routes. A stale route is one that has not carried any traffic in a given period of time. In long thin network topologies, this policy may be inefficient, as the same routes will be purged by each Router. It may be more efficient and less traffic intensive to disable this feature on Routers and just leave it enabled on the Co-ordinator.

Route maintenance is also configured using the function **vApi_SetPurgeInterval()**.

Parameters

bPurge	Enable/disable route purging: TRUE - enable (default) FALSE - disable

Returns

None

vApi_SetPurgeInterval

void vApi_SetPurgeInterval(uint32 *u32Interval***);**

Description

This function is used together with **vApi_SetPurgeRoute()** to tailor the automatic route maintenance. The function can be used to adjust the route maintenance cycle - it sets the period of time between each route maintenance activity.

The default period is one second, which means that a Routing table entry is examined every second (even if the entry is not used). The length of time taken to process the whole Routing table is determined by the table size, which is user-defined at build time - for example, a Routing table comprising 100 entries will take 100 seconds to process (even if only one of the entries is actually used). Routes will be interrogated if they have not been used in two cycles, e.g. 200 seconds.

Setting a smaller period will improve clean-up time after network reconfiguration due to node failure, but will generate more traffic travelling down the tree which could cause contention with user data flowing up the tree. Setting a larger value will extend the time taken to clean-up.

This feature may not be required if there is regular traffic generated from all the network nodes.

Parameters

u32Interval	Route maintenance period in units of 100 ms

Returns

None

vNwk_SetBeaconCalming

void vNwk_SetBeaconCalming(bool *bState***);**

Description

This function enables/disables ‘beacon calming’.

When a large, dense network attempts to recover from a major failure, large numbers of beacons are generated which can slow the flow of essential network management messages. Enabling beacon calming suppresses beacons generated by Routers that are statistically less able to accept associations. Hence, the speed of network recovery increases.

Parameters

bState	Enable/disable beacon calming: TRUE - enable FALSE - disable (default)

Returns

None

vApi_SetUserBeaconBits

void vApi_SetUserBeaconBits(uint16 *u16Bits***);**

Description

This function is used to set the user-defined part of the beacon payload. This can then be used to control network formation.

The function must be called after the network has started, otherwise the bits will be cleared.

Parameters

u16Bits	16 bits of user data to be inserted in the beacon payload

Returns

None

u16Api_GetUserBeaconBits

```
uint16 u16Api_GetUserBeaconBits(void);
```

Description

This function is used to read the 16-bit user-defined part of a beacon payload. These user-defined bits can be used for any application functionality, such as to control network formation.

The contents of beacons received using **vApi_RegBeaconNotifyCallback()** can be inspected for the user bits, and beacons accepted or discarded on the basis of these bits.

Parameters

None

Returns

16 bits of user data read from the beacon payload

u8Api_GetLastPktLqi

```
uint8 u8Api_GetLastPktLqi(void);
```

Description

This function returns the LQI value (detected radio signal strength) of the last packet received, and must be called in the data event handler **vJenie_CbStackDataEvent()** in response to a data event. This guarantees that the returned LQI value applies to the packet which is going to be processed. Calling the function at any other time will return the LQI value of the last packet processed, which may be one that was routed or may be a network management packet.

This is the LQI value of the last hop to its destination node.

For further information on the LQI value, including an approximate relationship between the LQI value and the detected power in dBm, refer to Section 4.8.

Parameters

None

Returns

The LQI value of the last packet received

u16Api_GetDepth

```
uint16 u16Api_GetDepth(void);
```

Description

This function is used to return the number of hops of the local node from the Co-ordinator. Since a JenNet network employs a Tree topology, the result is the depth of the local node in the network.

Parameters

None

Returns

Number of hops from Co-ordinator

u8Api_GetStackState

```
uint8 u8Api_GetStackState(void);
```

Description

This function returns the current state of the JenNet stack and provides a mechanism for determining the current operation of the stack.

Parameters

None

Returns

The state of the JenNet stack, one of:

E_JENNET_IDLE (0x00)
E_JENNET_ENERGY_SCAN (0x01)
E_JENNET_WAITING_FOR_ENERGY_SCAN (0x02)
E_JENNET_ACTIVE_SCAN (0x03)
E_JENNET_WAITING_FOR_ACTIVE_SCAN (0x04)
E_JENNET_ASSOCIATE (0x05)
E_JENNET_ASSOCIATE_SKIP_ESTABLISH_ROUTE (0x06)
E_JENNET_WAITING_FOR_ASSOCIATE (0x07)
E_JENNET_WAITING_FOR_ASSOCIATE_SKIP_ESTABLISH_ROUTE (0x08)
E_JENNET_START_COORD (0x09)
E_JENNET_START_COORD_SKIP_ESTABLISH_ROUTE (0x0A)
E_JENNET_ESTABLISH_ROUTE (0x0B)
E_JENNET_WAITING_FOR_ESTABLISH_ROUTE (0x0C)
E_JENNET_RUNNING (0x0D)
E_JENNET_WAITING_FOR_BACKOFF (0x0E)
E_JENNET_SLEEP (0x0F)

u32Api_GetVersion

uint32 u32Api_GetVersion(teJenNetComponent *eComponent*,
tsVersionInfo* *psVersionInfo*);

Description

This allows a stack version text string to be obtained.

The Jenie API function **u32Jenie_GetVersion()** is used to gather information on the stack versions. An extra text string of the version is available through **u32Api_GetVersion()**.

Parameters

eComponent	Set to NETWORK_VERSION to return version data
psVersionInfo	Pointer to structure which, if allocated, will hold the supplementary version string

Returns

None

vApi_RegBeaconNotifyCallback

```
void vApi_RegBeaconNotifyCallback(
                    trBeaconNotifyCallback prCallback);
```

Description

This function registers a user-defined callback function that will be invoked when a beacon is received. This provides an opportunity for the application to either collect information about other nodes in the vicinity or prevent the stack from joining particular parents (by ignoring selected beacons).

The prototype for the callback function is detailed below.

Parameters

prCallback	Pointer to callback function

Returns

None

Callback Function

```
typedef bool_t (*trBeaconNotifyCallback)(
                    tsScanElement *psBeaconInfo,
                    uint32 u32NetworkID,
                    uint16 u16ProtocolVersion);
```

Description

This user-defined callback function is invoked on receipt of a beacon. It can delete the beacon and extract data from it. If forcing the shape of the network, only beacons from target parents should be accepted. The beacons can also be saved for possible load balancing activity later.

The execution time of this function should be kept to a minimum.

Parameters

**psBeaconInfo*	Pointer to the received beacon - for `tsScanElement` structure, see Section 10.2.4
u32NetworkID	Network Application ID from beacon
u16ProtocolVersion	Stack version from beacon

Returns

TRUE	Accept the beacon for sorting
FALSE	Delete the beacon

vApi_RegLocalAuthoriseCallback

```
void vApi_RegLocalAuthoriseCallback(
                    trAuthoriseCallback prCallback);
```

Description

This function registers a user-defined callback function that will be invoked when a node attempts to join the Co-ordinator or a Router. The function provides an opportunity for the application to prevent potential child nodes from accessing the network and can be used to force nodes onto other adjacent parents or networks.

Parameters

prCallback	Pointer to callback function

Returns

None

Callback Function

```
typedef bool_t (*trAuthoriseCallback)(MAC_ExtAddr_s *psAddr);
```

Description

This user-defined callback function provides the opportunity to block nodes with specific IEEE/MAC addresses from joining as children. The passed IEEE/MAC address can be compared with a list of permitted or forbidden addresses, and then accepted or rejected accordingly. A rejected node will then attempt to join the network again, until it finds a parent node which accepts its join request.

Parameters

**psAddr*	Pointer to IEEE/MAC address

Returns

TRUE	Joining process continues
FALSE	Joining is denied

vApi_RegNwkAuthoriseCallback

```
void vApi_RegNwkAuthoriseCallback(
                    trAuthoriseCallback prCallback);
```

Description

This function registers a user-defined callback function that will be invoked when a node attempts to join the network. This event only occurs on the Co-ordinator and provides an opportunity for the application to prevent the joining node from accessing the network. This mechanism can be used to force nodes onto other adjacent networks.

Parameters

prCallback	Pointer to callback function

Returns

None

Callback Function

```
typedef bool_t (*trAuthoriseCallback)(MAC_ExtAddr_s *psAddr);
```

Description

This user-defined callback function provides the opportunity to block nodes with specific IEEE/MAC addresses from joining the network. The Co-ordinator receives the passed IEEE/MAC address which can be compared with a list of permitted or forbidden addresses, and then accepted or rejected accordingly. The rejected node will then attempt to join a network again, until it finds a network which accepts its join request.

Parameters

**psAddr*	Pointer to the IEEE/MAC address

Returns

TRUE	Joining process continues
FALSE	Joining is denied

vApi_RegScanSortCallback

```
void vApi_RegScanSortCallback(
                    trSortScanCallback prCallback);
```

Description

This function registers a user-defined callback function that will be invoked when a network scan completes. Access to the scan list is provided so that the application can change the order in which the stack attempts to associate with potential parents.

Parameters

prCallback	Pointer to callback function

Returns

None

Callback Function

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

Description

This user-defined callback function provides the opportunity to over-ride the default operation of the stack and customise the beacon sort algorithm to obain a preferred order of association attempts. The function is called on completion of an active scan. The stack attempts to associate with the first entry in the list then steps through the list until an association is successful. If none are successful, the active scan is re-started.

To delete beacons, use the **vApi_RegBeaconNotifyCallback()** function and return FALSE to ignore specific beacons.

The execution time of this function should be kept to a minimum.

Parameters

**pasScanResult*	Pointer to (input) array of scan results containing suitable parents - for `tsScanElement` structure, see Section 10.2.4
u8ScanListSize	Number of suitable parents in the scan results array
**pau8ScanListOrder*	Pointer to (output) array of **uint8** indicating the sorted order of potential parents from most desirable to least desirable parent (e.g. 3, 4, 1, 6, 0, 2, 5, 7) - the integers correspond to the positions of the parents in the initial scan results (**pasScanResult*)

Returns

TRUE	Control returned to application and scanning process stopped
FALSE	Control returned to stack and scan process resumed

The function should normally return FALSE unless the scan process is to be aborted.

9. Global Network Parameters

This chapter details the global network parameters that can be set on each JenNet node. Two sets of parameters are presented:

- Global parameters which are part of the Jenie API (Section 9.1)
- Global parameters which are part of the JenNet stack (Section 9.2)

The values of these parameters can be set in the **vJenie_CbConfigureNetwork()** callback function at the start of the application. If a parameter is not set by the application, its default value is used.

Note: Normally, it is only necessary to use the Jenie parameters in your application code. The JenNet parameters are intended for advanced users who require more control over the network than is available through the Jenie parameters.

Some of the JenNet parameters are duplicated in the Jenie parameters. The Jenie values are loaded into the JenNet parameters by **vJenie_CbConfigureNetwork()**, which occurs once at the program start.

Important: Setting a duplicate Jenie parameter through **vJenie_CbConfigureNetwork()** automatically sets the equivalent JenNet parameter, but directly setting the JenNet parameter does not automatically set the equivalent Jenie parameter. Therefore, where a parameter is duplicated, you are strongly advised to set the Jenie version rather than the JenNet version.

9.1 Jenie Parameters

The Jenie global network parameters are listed and described in the table below.

Parameter Name	Description	Default Value	Range
gJenie_PanID	16-bit PAN ID to identify network (if no existing network with same PAN ID). **Co-ordinator only**	0xAAAA	0-0xFFFE
gJenie_NetworkApplicationID	32-bit Network Application ID used to identify and form network.	0xAAAA AAAA	0-0xFFFFFFFF
gJenie_Channel	The 2.4-GHz channel to be used by the network, or auto-scan (see *gJenie_ScanChannels* below). **Co-ordinator only**	0	0: Auto-scan 11-26: Channel
gJenie_ScanChannels	Bitmap (32 bits) of the set of channels to consider when performing an auto-scan of the 2.4-GHz band for a suitable channel to use. The Co-ordinator will select the quietest channel from those available (auto-scan must have been enabled via *gJenie_Channel*). Other node types will scan the possible channels to search for network.	0x07FFF800 (all channels)	0x00000800 - 0x07FFF800 (Bit 11 set ⇒ Ch 11, Bit 12 set ⇒ Ch 12,...)
gJenie_MaxChildren	Maximum number of children the node can have. **Co-ordinator and Routers only**	10	0-16
gJenie_MaxSleepingChildren	Maximum number of children that can be End Devices (nodes capable of sleeping). This value must be less than or equal to *gJenie_MaxChildren*. The remaining child nodes are reserved exclusively for Routers, although any number of children can be Routers. **Co-ordinator and Routers only**	8	0-*gJenie_MaxChildren*
gJenie_MaxBcastTTL	Determines the maximum number of hops that a broadcast message sent from the local node can make. Set this value to one less than the desired maximum (so the value 0 corresponds to one hop).	5	0-255
gJenie_MaxFailedPkts	Number of missed communications (MAC acknowledgments) before parent considered to be lost (and node must try to find a new parent).	5	0-255 Zero value disables the feature

Table 1: Global Network Parameters (Jenie)

gJenie_RoutingEnabled	Enables/disables routing capability of the node (must be disabled for End Devices).	0	0: Disable routing 1: Enable routing For End Devices, always set to 0
gJenie_RoutingTableSize	Number of elements in array used to store the Routing table. Should be set to a value slightly larger than the maximum number of network nodes, to allow for nodes leaving and joining. **Co-ordinator and Routers only**	-	0-1000 Note that the upper limit may be restricted by the amount of available RAM. Each Routing table entry uses 12 bytes.
gJenie_RoutingTableSpace	Pointer to the Routing table array in memory. The Routing table is an array of structures of type **tsJenieRoutingTable,** where this array is declared in the application. **Co-ordinator and Routers only**	NULL	-
gJenie_RouterPingPeriod	Time between auto-pings generated by a Router (to its parent). Set in units of 100 ms. The same value should be set in all routing nodes in the network. **Co-ordinator and Routers only**	50 (5 seconds)	0-6553 Zero value disables pings. Non-zero values below 50 are not recommended
gJenie_EndDevicePingInterval	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
gJenie_EndDeviceScanSleep	Amount of time following a failed scan that an End Device waits (sleeps) before starting another scan. Set in milliseconds. **End Devices only**	10000 or 0x2710 (10 seconds)	0xC8-0xFFFFFFEB Values below 0x3E8 (1 second) are not recommended for large networks
gJenie_EndDevicePollPeriod	Time between auto-poll data requests sent from an End Device (while awake) to its parent. Set in units of 100 ms. **End Devices only**	50 or 0x32 (5 seconds)	0-0xFFFFFFFF Zero value disables auto-polling
gJenie_EndDeviceChildActivity Timeout	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). **Co-ordinator and Routers only**	0 (Timeout disabled)	0-0xFFFFFFFF Timeout is value set multiplied by 100 ms 0 disables the timeout but this is not advised, as child slots may fill with inactive End Devices, preventing other devices from joining.

Table 1: Global Network Parameters (Jenie)

gJenie_RecoverFromJpdm	Indicates whether network context data is to be recovered from external non-volatile memory during a cold start following power loss to the on-chip memory. Data must have been previously saved to external memory using **vJPDM_SaveContext()**.	0	0: Disable recovery 1: Enable recovery
gJenie_RecoverChildren FromJpdm	Enables the recovery of child/neighbour table when restoring context data from non-volatile memory. Context recovery must also be enabled using *gJenie_RecoverFromJpdm*. **Co-ordinator and Routers only**	1	0: Disable recovery 1: Enable recovery
gJpdmSector	Number of sector where context will be saved in external non-volatile memory.	3*	Positive integer
gJpdmSectorSize	Size of sector where context data will be saved in external non-volatile memory.	32768 (32K)*	Size in bytes
gJpdmFlashType	Type of Flash memory device used as external non-volatile memory.	Auto-detect	E_FL_CHIP_ST_M25P10_A E_FL_CHIP_SST_25VF010 E_FL_CHIP_ATMEL_AT25F512 E_FL_CHIP_CUSTOM E_FL_CHIP_AUTO
gJpdmFlashFuncTable	Pointer to function table for custom Flash memory device.	NULL	-

Table 1: Global Network Parameters (Jenie)

* For the 25P40 Flash memory device (used with the JN5148 device), the default sector is 7 and the default sector size is 65536 (64K) bytes.

9.2 JenNet Parameters

The JenNet parameters are detailed in the tables below, according to the node type(s) to which they apply.

Note: The JenNet parameters are intended for advanced users who require more control over the network than is available through the Jenie parameters. Normally, it is only necessary to use the Jenie parameters, detailed in Section 9.1.

Co-ordinator Parameters

Parameter Name	Description	Default Value	Range
gChannel	The 2.4-GHz channel to be used by the network, or an auto-scan (stack will automatically select a channel).	0	0: Auto-scan 11-26: Channel
gPanID	PAN ID used to form the network, if no pre-existing network found with the same PAN ID.	0xAAAA	0-0xFFFE

Table 2: Co-ordinator Parameters (JenNet)

General Parameters

Parameter Name	Description	Default Value	Range
gInternalTimer	The timer to be used as an internal timer: Timer 0, Timer 1 or the Tick Timer. The valid values are shown to the right and defined in the header file **AppHardwareApi.h**.	E_AHI_DEVICE_TICK_TIMER	E_AHI_DEVICE_TICK_TIMER E_AHI_DEVICE_TIMER0 E_AHI_DEVICE_TIMER1
gMaxBcastTTL	Determines the maximum number of hops that a broadcast message sent from the local node can make. Set this value to one less than the desired maximum (so the value 0 corresponds to one hop).	5	0-255
gMaxFailedPkts	Number of missed communications (MAC acknowledgments) before parent considered to be lost (and node must try to find a new parent).	5	1-255
gMinBeaconLQI	Minimum valid radio signal strength (as an LQI value) of a beacon - the stack rejects beacons with signal strength less than this value.	0	0-255 For information on LQI values, refer to Section 4.8

Table 3: General Parameters (JenNet)

Parameter Name	Description	Default Value	Range
gNetworkID	32-bit Network Application ID used to identify an individual application/ network.	0xAAAAAAAA	0-0xFFFFFFFF
gScanChannels	Bitmap (32 bits) of the set of channels to consider when performing an auto-scan of the 2.4-GHz band for a suitable channel to use. The Co-ordinator will select the quietest channel from those available (auto-scan must have been enabled via *gChannel*.). Other node types will scan the possible channels to search for network.	0x07FFF800 (All Channels)	0x00000800 -0x07FFF800 (Bit 11 set ⇒ Ch 11, Bit 12 set ⇒ Ch 12,...)

Table 3: General Parameters (JenNet)

Co-ordinator/Router Parameters

Parameter Name	Description	Default Value	Range
gEDChildActivityTimeout *	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), provided End Device purging has been enabled through *gRouterPurgeInactiveED*.	0	0-0xFFFFFFFF Timeout is value set multiplied by 100 ms
gMaxChildren	Maximum number of children that the node can have.	10	0-16
gMaxSleepingChildren	Maximum number of children that can be End Devices (nodes capable of sleeping). This value must be less than or equal to *gMaxChildren*. The remaining child nodes are reserved exclusively for Routers, although any number of children can be Routers.	8	1-*gMaxChildren*
bPermitExtNwkPkts	Enables/disables reception of packets from external networks. Do not configure in **vJenie_CbConfigureNetwork()**.	FALSE (disabled)	TRUE - enable FALSE - disable
gRouteImport	Enables/disables the ability of routing nodes to import routes from child nodes that have children.	TRUE (enabled)	TRUE - enable FALSE - disable
gRouterEnableAutoPurge	Enables/disables the auto-purge facility which removes inactive nodes from the network.	TRUE (enabled)	TRUE - enable FALSE - disable

Table 4: Co-ordinator/Router Parameters (JenNet)

Parameter Name	Description	Default Value	Range
gRoutingTableSize	Number of elements in array used to store the Routing table. Should be set to a value slightly larger than the maximum number of network nodes, to allow for nodes leaving and joining. Set 0 for End Devices.	0	0-1000 Note that the upper limit may be restricted by the amount of available RAM. Each Routing table entry uses 12 bytes.
gRouterPingPeriod **	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.	500 (5 seconds)	500-65535
gRouterPurgeInactiveED *	Enable/disable the timeout on End Device activity - see the parameter *gEDChildActivityTimeout*.	FALSE (disabled)	TRUE - enable FALSE - disable
gpvRoutingTableSpace	Pointer to space allocated for the Routing table. This space must be equal to [sizeof(tsJenieRoutingTable) x *gRoutingTableSize*], or NULL in the case of an End Device.	NULL	Pointer

Table 4: Co-ordinator/Router Parameters (JenNet)

* The JenNet parameters *gRouterPurgeInactiveED* and *gEDChildActivityTimeout* are combined into a single Jenie parameter, *gJenie_EndDeviceChildActivityTimeout,* detailed in Section 9.1.

** The JenNet parameter *gRouterPingPeriod* and the Jenie parameter *gJenie_RouterPingPeriod* control the same feature but have different units (10 ms and 100 ms, respectively).

End Device Parameters

Parameter Name	Description	Default Value	Range
gEndDevicePingInterval	Number of sleep cycles between auto-pings generated by an End Device (to its parent).	1	0-255 Zero value disables pings
gEndDevicePollPeriod	Time between auto-poll data requests sent from an End Device (while awake) to its parent. Set in units of 100 ms.	50 or 0x32 (5 seconds)	0-0x FFFFFFFF Zero value disables auto-polling.
gEndDeviceScanSleep	Amount of time following a failed scan that an End Device waits (sleeps) before starting another scan. Set in milliseconds.	10000 or 0x2710 (10 seconds)	0xC8-0xFFFFFFEB Values below 0x3E8 (1 second) are not recommended for large networks

Table 5: End Device Parameters (JenNet)

10. Enumerations and Data Types

This chapter lists the enumerations and data types used by the Jenie API.

10.1 Enumerations and Defines

The following enumerated types and defines are used by the Jenie API functions and are included in the header file **Jenie.h**.

10.1.1 teJenieStatusCode (Return Status)

These status responses are returned by most Jenie API function calls.

```
typedef enum
{
   E_JENIE_SUCCESS,          /*0 Function successfully completed*/
   E_JENIE_DEFERRED,       /*1 Stack response deferred*/
   E_JENIE_ERR_UNKNOWN,    /*2 Unknown error*/
   E_JENIE_ERR_INVLD_PARAM, /*3 Error - invalid parameter*/
   E_JENIE_ERR_STACK_RSRC,  /*4 Error - insufficient resources*/
   E_JENIE_ERR_STACK_BUSY  /*5 Error - stack too busy*/
} teJenieStatusCode;
```

10.1.2 teJenieDeviceType (Node Type)

```
typedef enum
{
    E_JENIE_COORDINATOR,
    E_JENIE_ROUTER,
    E_JENIE_END_DEVICE
} teJenieDeviceType;
```

10.1.3 teJenieComponent (Component)

```
typedef enum
{
   E_JENIE_COMPONENT_JENIE,  /*Jenie*/
   E_JENIE_COMPONENT_NETWORK, /*Network level - JenNet*/
   E_JENIE_COMPONENT_MAC,    /*IEEE 802.15.4*/
   E_JENIE_COMPONENT_CHIP    /*JN513x chip*/
} teJenieComponent;
```

10.1.4 teJenieRadioPower (Radio Transceiver)

```
typedef enum
{
   E_JENIE_RADIO_ON  = 20,
   E_JENIE_RADIO_OFF = 21
} teJenieRadioPower;
```

10.1.5 teJeniePollStatus (Poll Status)

```
typedef enum
{
   E_JENIE_POLL_NO_DATA, /* No data available */
   E_JENIE_POLL_DATA_READY, /* Data pending */
   E_JENIE_POLL_TIMEOUT /* Poll failed since no response */
}teJeniePollStatus;
```

Note that E_JENIE_POLL_TIMEOUT is returned if the poll-cycle fails to complete within 200 ms (due to no acknowledgement from the poll target within this time).

10.1.6 TXOPTION #defines

Code	Value	Description
TXOPTION_ACKREQ	0x01	Requests an acknowledgement from the destination node
TXOPTION_BDCAST	0x04	Sends a broadcast message to all Routers in the network
TXOPTION_SILENT	0x08	Sends without packet sent/failed notification

Table 6: TXOPTION #defines

10.2 Data Types

The following data types are used by the Jenie API and are included in the header file **Jenie.h**, unless stated otherwise.

10.2.1 tsJenieSecKey (Security Key)

```
typedef struct
{
    uint32 u32register0;
    uint32 u32register1;
    uint32 u32register2;
    uint32 u32register3;
} tsJenieSecKey;
```

10.2.2 tsJenie_RoutingEntry (Routing Table Entry)

```
typedef struct
{
    uint16  u16EntryNum;     // Entry number
    uint16  u16TotalEntries; // Total number of entries in table
    uint64  u64DestAddr;     // Destination address
    uint64  u64NextHopAddr;  // Next hop address
}tsJenie_RoutingEntry;
```

10.2.3 tsJenie_NeighbourEntry (Neighbour Table Entry)

```
typedef struct
{
  uint8   u8EntryNum;      // Entry number
  uint8   u8TotalEntries;  // Total number of entries in table
  uint64  u64Addr;         // Address of neighbouring node
  bool_t  bSleepingED;     // If device is a sleeping node
  uint32  u32Services;     // Services provided by the node
  uint8   u8LinkQuality;   // Last received link quality info
  uint16  u16PktsLost;     // Sent packets not acknowledged
  uint16  u16PktsSent;     // Sent packets acknowledged
  uint16  u16PktsRcvd;     // Packets received from node
}tsJenie_NeighbourEntry;
```

Note: `u8LinkQuality` is a Link Quality Indication (LQI) value in the range 0-255. For more information on the LQI value, including an approximate relationship between the LQI value and detected power in dBm, see Section 4.8.

10.2.4 tsScanElement (Scan Results)

This structure is used by the JenNet API, described in Chapter 8. It contains information about a remote node - for example, properties reported as the result of an energy scan.

```
typedef struct
{
    MAC_ExtAddr_s   sExtAddr; // MAC address of remote node
    uint16          u16PanId; // PAN ID of host network
    uint16          u16Depth; // Depth of node in network
    uint8           u8Channel; // Channel number (11-26)
    uint8           u8LinkQuality; // Link quality to node
    uint8           u8NumChildren; // Number of child nodes
    uint16          u16UserDefined; // User-defined value
}tsScanElement;
```

For more information on `u8LinkQuality`, refer to the Note on page 165.

10.2.5 MAC_Addr_s

This structure is contained in the header file **mac_sap.h** and is needed when using the JenNet API.

```
typedef struct
{
  uint8 u8AddrMode; /* Address mode: 2 for short, 3 for extended */
  uint16 u16PanId; /* PAN ID */
  MAC_Addr_u uAddr; /* Address */
} MAC_Addr_s;
```


10.2.6 MAC_ExtAddr_s

This structure is contained in the header file **mac_sap.h** and is needed when using the JenNet API.

```
typedef struct
{
  uint32 u32L; /* Low word */
  uint32 u32H; /* High word */
} MAC_ExtAddr_s;
```


11. Stack Events

The chapter details the stack events (management and data) that can be handled by the Jenie API callback functions described in Section 7.2.

11.1 Management Events and Structures

The table below lists and describes the stack management events that can be handled by the callback function **vJenie_CbStackMgmtEvent()**.

Stack Event	Description	Structure Type
E_JENIE_REG_SVC_RSP	To register the services of an End Device requires communication with the parent. In this case, the return value of the call to **eJenie_RegisterServices()** indicates a deferred response. This event is generated when the registration is complete.	NULL
E_JENIE_SVC_REQ_RSP	Indicates a response to a service request has been received from a remote node.	tsSvcReqRsp
E_JENIE_POLL_CMPLT	Indicates that the End Device has finished polling the parent node for data. **End Devices only**	tsPollCmplt
E_JENIE_PACKET_SENT	Indicates that a packet has been successfully sent (to the next node).	NULL
E_JENIE_PACKET_FAILED	Indicates that a packet send (to the next node) has failed.	NULL
E_JENIE_NETWORK_UP	Indicates that the network is up and running.	tsNwkStartUp
E_JENIE_STACK_RESET	Indicates that the stack is going to reset, normally because the node has left the network or lost its parent.	NULL
E_JENIE_CHILD_JOINED	Indicates that a child has joined a Router/Co-ordinator.	tsChildJoined
E_JENIE_CHILD_LEAVE	Indicates that a child has left a Router/Co-ordinator.	tsChildLeave
E_JENIE_CHILD_REJECTED	Indicates that a request by a node to join a network has been rejected by the Co-ordinator (normally due to lack of space in the Co-ordinator's routing table).	tsChildRejected

Table 7: Stack Management Events

vJenie_CbStackMgmtEvent() has two parameters, the first being the stack event as described above. The second parameter is a pointer to a data structure that contains additional information fields. If the additional data is not necessary, the second parameter is simply a NULL pointer. If a pointer to the primitive is sent, it must be cast to the appropriate type described below.

Note: In the descriptions below, each of `u64SrcAddress` and `u64ParentAddress` is a 64-bit IEEE/MAC address.

11.1.1 tsSvcReqRsp

```
typedef struct
{
    uint64      u64SrcAddress; /* Address of responding node */
    uint32      u32Services; /* Services available on node */
} tsSvcReqRsp;
```

`u32Services` is a 32-bit value in which each bit position represents a network service - the bit representations are as in the network's Service Profile, defined in the header file **Jenie.h**. In `u32services`, '1' indicates that the responding node supports the corresponding service and '0' indicates that the service is not supported by the node.

11.1.2 tsPollCmplt

```
typedef struct
{
    teJeniePollStatus ePollStatus;
}tsPollCmplt;
```

For details of the enumerated type `teJeniePollStatus`, refer to Section 10.1.5.

11.1.3 tsChildJoined

```
typedef struct
{
    uint64 u64SrcAddress; /* Address of node that has joined */
} tsChildJoined;
```

11.1.4 tsChildLeave

```
typedef struct
{
    uint64 u64SrcAddress; /* Address of node that has left */
} tsChildLeave;
```

11.1.5 tsChildRejected

```
typedef struct
{
    uint64 u64SrcAddress; /* Address of rejected node */
} tsChildRejected;
```

11.1.6 tsNwkStartUp

```
typedef struct{
    uint64 u64ParentAddress;  /*Address of parent node*/
    uint64 u64LocalAddress;   /*Address of local node*/
    uint16 u16Depth;          /*Depth of node in the network*/
    uint16 u16PanID;          /*PAN ID of the network*/
    uint8  u8Channel;         /*Operating channel */
}tsNwkStartUp
```

11.2 Data Events and Structures

The table below lists and describes the data events that can be handled by the callback function **vJenie_CbStackDataEvent()**.

Stack Event	Description	Structure Type
E_JENIE_DATA	Indicates that data has been received from another node. Event contains the data.	tsData
E_JENIE_DATA_TO_SERVICE	Indicates that data has been received from another node, destined for a particular service on the local node. Event contains the data.	tsDataToService
E_JENIE_DATA_ACK	Indicates that a response has been received from a remote node, acknowledging receipt of data previously sent from the local node.	tsDataAck
E_JENIE_DATA_TO_SERVICE_ACK	Indicates that a response has been received from a remote node, acknowledging receipt of data previously sent from the local node to a particular service on the remote node.	tsDataToServiceAck

Table 8: Data Events

vJenie_CbStackDataEvent() has two parameters, the first being the stack event as described above. The second parameter is a pointer to a data structure that contains additional information fields. The pointer must be cast to an appropriate type described below.

Note: In the descriptions below, `u64SrcAddress` is a 64-bit IEEE/MAC address.

11.2.1 tsData

```
typedef struct
{
  uint64    u64SrcAddress; /* Address of message source */
  uint8     u8MsgFlags;  /* Flags reserved for future use */
  uint16    u16Length;   /* Length of data payload, in bytes */
  uint8     *pau8Data;   /* Pointer to data payload */
}tsData;
```

11.2.2 tsDataToService

```
typedef struct
{
   uint64    u64SrcAddress; /* Address of message source */
   uint8     u8SrcService; /* Service on sending node */
   uint8     u8DestService; /* Service on receiving node */
   uint8     u8MsgFlags; /* Flags reserved for future use */
   uint16    u16Length; /* Length of data payload, in bytes */
   uint8     *pau8Data; /* Pointer to data payload */
}tsDataToService;
```

11.2.3 tsDataAck

```
typedef struct
{
    uint64  u64SrcAddress; /* Address of acknowledgement source */
}tsDataAck;
```

11.2.4 tsDataToServiceAck

```
typedef struct
{
    uint64 u64SrcAddress; /* Address of sending node */
}tsDataToServiceAck;
```

Part III: Appendices

A. Hardware and Memory Use

This appendix details the JN5148/JN5139 hardware required by JenNet, and the memory resources required by JenNet, with and without the Jenie API.

A.1 Hardware Resources

The JN5148/JN5139 hardware required by the JenNet stack is as follows:

- **For End Devices only:** Wake Timer 0 for sleep mode.
- **For all devices:** Tick timer for scheduling - this timer fires every 10 ms and a tick is passed up to the application as a hardware event via the callback function **vJenie_CbHwEvent()**.

A.2 Memory Resources (JenNet Only)

This section details the memory resources required by the JenNet stack (without the Jenie API) on the JN5148 and JN5139 devices.

Note: The Routing table size is configurable at application compile-time.

JN5148 Memory Resources

From the 128 KB of RAM on the JN5148 device, the exact memory resources required by the JenNet stack depend on the size of the Routing table, as indicated in Table 1 below.

Routing Table Size	Memory Required		
	Co-ordinator	Router	End Device
25	27 KB	27 KB	17 KB
100	28 KB	28 KB	17 KB
250	30 KB	30 KB	17 KB
500	33 KB	33 KB	17 KB
1000	39 KB	39 KB	17 KB

Table 1: JN5148 Memory Required by JenNet Stack

Note: The above figures do not include 6 KB for the 802.15.4 stack layers, 4 KB for the machine stack and 2 KB for the heap

JN5139 Memory Resources

From the 96 KB of RAM on the JN5139 device, the exact memory resources required by the JenNet stack depend on the size of the Routing table, as indicated in Table 2 below.

Routing Table Size	Memory Required		
	Co-ordinator	Router	End Device
25	41 KB	41 KB	29 KB
100	42 KB	42 KB	29 KB
250	44 KB	44 KB	29 KB
500	47 KB	47 KB	29 KB
1000	53 KB	53 KB	29 KB

Table 2: JN5139 Memory Required by JenNet Stack

Note: The above figures do not include 6 KB for the 802.15.4 stack layers, 4 KB for the machine stack and 2 KB for the heap.

A.3 Memory Resources (JenNet and Jenie API)

This section details the memory resources required by the JenNet stack with the Jenie API on the JN5148 and JN5139 devices.

Note: The Routing table size is configurable at application compile-time.

JN5148 Memory Resources

From the 128 KB of RAM on the JN5148 device, the exact memory resources required by the JenNet stack (with Jenie API) depend on the size of the Routing table, as indicated in Table 3 below.

Routing Table Size	Memory Required		
	Co-ordinator	Router	End Device
25	31 KB	31 KB	20 KB
100	32 KB	32 KB	20 KB
250	33 KB	33 KB	20 KB
500	36 KB	36 KB	20 KB
1000	42 KB	42 KB	20 KB

Table 3: JN5148 Memory Required by JenNet Stack with Jenie API

Note: The above figures do not include 6 KB for the 802.15.4 stack layers, 4 KB for the machine stack and 2 KB for the heap.

JN5139 Memory Resources

From the 96 KB of RAM on the JN5139 device, the exact memory resources required by the JenNet stack (with Jenie API) depend on the size of the Routing table, as indicated in Table 2 below.

Routing Table Size	Memory Required		
	Co-ordinator	Router	End Device
25	51 KB	51 KB	37 KB
100	52 KB	52 KB	37 KB
250	54 KB	54 KB	37 KB
500	57 KB	57 KB	37 KB
1000	63 KB	63 KB	37 KB

Table 4: JN5139 Memory Required by JenNet Stack with Jenie API

Note: The above figures do not include 6 KB for the 802.15.4 stack layers, 4 KB for the machine stack and 2 KB for the heap.

B. Frames

This appendix details the different types of JenNet frame that can be exchanged between nodes.

A JenNet frame is carried as the payload of an IEEE 802.15.4 MAC frame. A MAC frame can contain a maximum of 127 bytes and comprises a header, payload and footer, as shown below.

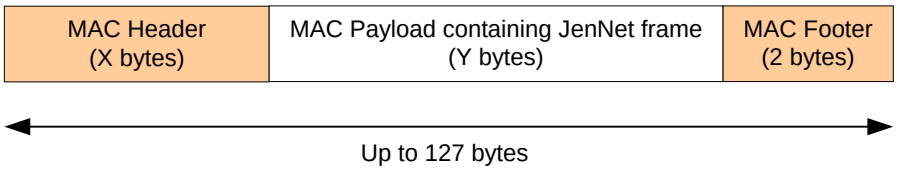

Figure 1: MAC Frame Structure

All JenNet frames follow the same basic structure, shown below.

Figure 2: JenNet Frame Structure

There are several types of JenNet frame. The header is the same for all frame types and is described in Appendix B.1. The body of the JenNet frame depends on the frame type, as described in Appendix B.2.

B.1 Frame Header

All JenNet frames have a header, consisting of 12 bytes and structured as shown below.

Figure 3: Frame Header

As shown above, the header contain fields for frame flags, sequence number, frame type, source address and CRC (Cyclic Redundancy Check) value. The Source Address field contains the 64-bit IEEE/MAC address of the originating node. The Frame Flags and Frame Type fields are detailed below.

Frame Flags Field

The Frame Flags field of the header consists of a single byte and is structured as shown below.

Reserved Bit 7	Reserved Bit 6	Reserved Bit 5	Reserved Bit 4	End Device Bit 3	Response Bit 2	Response Required Bit 1	Success Bit 0

Figure 4: Frame Flags Field

The sub-fields are as follows:

- **End Device flag:** Set to '1' to specify that the node which initiated a transaction is a sleeping End Device. This bit only has meaning in outgoing request frames - it has no meaning in response frames.
- **Response flag:** Set to '1' to indicate that the frame is a response.
- **Response-Required flag:** Set to '1' to indicate that the sender requires a response to be sent back from the destination node.
- **Success flag:** Set to '1' in a response to indicate that the request was a success. In many transactions, this has no real meaning - success is implicit in the fact that a response has been sent

Frame Type Field

The Frame Type field of the header consists of a single byte. The Frame Type values are listed in the table below.

Value	Frame Type
0	DATA_TO_COORD
1	DATA_TO_PEER
2	ESTABLISH_ROUTE
3	DELETE_ROUTE
4	REPAIR_ROUTE
5	ACTIVATE_SERVICES
6	SERVICE_REQUEST
7	PING
8	UNKNOWN_NODE
9	DATA_TO_NETWORK
10	Not used
11	DATA_TO_SERVICE
12	PURGE_ROUTE

Table 5: Frame Type Values

Value	Frame Type
13	FIND_NODE
14	ROUTE_REQUEST
15	ROUTE_IMPORT
16	SET_DEPTH

Table 5: Frame Type Values

B.2 Frame Body

The frame body depends on the frame type. There are sixteen frame types, listed below:

- Data-to-Coordinator
- Data-to-Peer
- Data-to-Network
- Data-to-Service
- Establish-Route
- Delete-Route
- Repair-Route
- Purge-Route
- Find-Node
- Route-Request
- Route-Import
- Activate-Services
- Service-Request/Response
- Ping
- Unknown-Node
- Set-Depth

These are described in the sub-sections below.

B.2.1 Data-to-Coordinator Frame

A Data-to-Coordinator frame is used to send data from an End Device or Router to the Co-ordinator.

No destination address is required, since the Co-ordinator will always be the device at the top of the tree (or at the centre of the star). Only the address of the source device is included in the frame (in the header).

The Data-to-Coordinator frame structure is shown below.

Figure 5: Data-to-Coordinator Frame Structure

If the Response-Required flag is set in the Frame Type field (in the header), the Co-ordinator sends a response back to the source node. The response frame is identical to the request frame except that the Response-Required flag is cleared and the Response flag is set.

B.2.2 Data-to-Peer Frame

A Data-to-Peer frame is used to send data to a remote device, for which the IEEE/MAC address is known. Both the source address and destination address are included in the frame (the source address is in the header).

The Data-to-Peer frame structure is shown below.

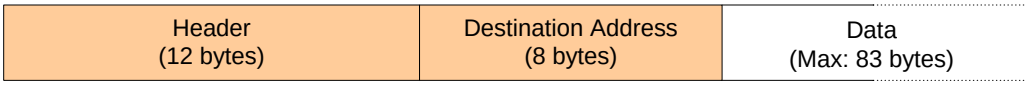

Figure 6: Data-to-Peer Frame Structure

If the Response-Required flag is set in the Frame Type field (in the header), the remote node sends a response back to the local node. The response frame is identical to the request frame except that the Response-Required flag is cleared and the Response flag is set.

B.2.3 Data-to-Network Frame

A Data-to-Network frame is used to send data to all nodes in the network.

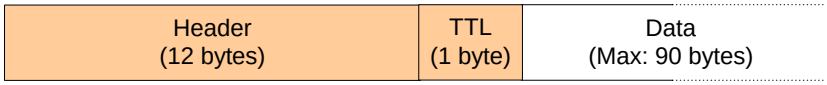

Figure 7: Data-to-Network Frame Structure

The Response-Required flag (in the header) is ignored when this frame is received.

This frame is broadcast and the value of the TTL (Time-to-Live) field determines the number of times the frame can be forwarded during the broadcast.

B.2.4 Data-to-Service Frame

A Data-to-Service frame is used to send data to a specific service on a remote node, for which the IEEE/MAC address is known. Both the source address and destination address are included in the frame (the source address is in the header).

The Data-to-Service frame structure is shown below.

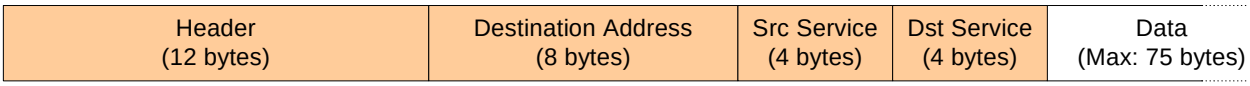

Figure 8: Data-to-Service Frame Structure

If the Response-Required flag is set in the Frame Type field (in the header), the remote node sends a response back to the local node. The response frame is identical to the request frame except that the Response-Required flag is cleared and the Response flag is set.

B.2.5 Establish-Route Frame

The Establish-Route frame is sent by a device to establish a route from itself all the way up the branch that it has joined, ending at the Co-ordinator. Routing table entries are created in all ascendant nodes, with the sole exception of the node's parent.

The frame is initially passed to the node's parent. It is then passed upwards to the next node in the tree, which adds a Routing table entry for the local node, before passing the frame up the branch. Each time a Routing table entry is created, the next hop address is the address of the node from which the frame has been received.

The Establish-Route frame structure is shown below.

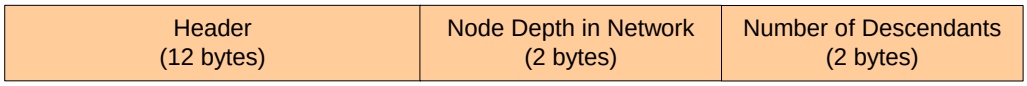

Figure 9: Establish-Route Frame Structure

The source address of the local node is included in the frame (in the header), along with the number of descendant nodes (including immediate children) of which the node is aware. If the Response-Required flag is set in the Frame Type field, the Co-ordinator node sends a response back to the local node. The response frame is identical to the request frame except:

- the Depth field is set to the depth of the local node in the network
- the sequence number is not the same
- the Response-Required flag is cleared
- the Response flag is set

B.2.6 Delete-Route Frame

A Delete-Route frame is sent by a node to purge the entire network of all Routing table entries that relate to the node. Neighbour table entries are also removed.

This frame is generated whenever a node attaches or re-attaches to the network, or when a node receives a Repair-Route frame from a remote node. The frame also initiates network-wide garbage collection. The frame is initially sent in unicast mode from the local node to its parent. The parent then broadcasts the frame to the network.

The frame consists only of the header, as shown below.

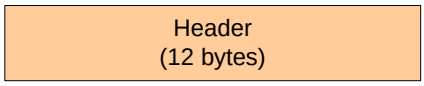

Figure 10: Delete-Route Frame Structure

Only the source address of the local node is included in the frame. If the Response-Required flag is set in the Frame Type field, the parent node sends a response back to the local node. The response frame is identical to the request frame except that the Response-Required flag is cleared and the Response flag is set.

B.2.7 Repair-Route Frame

A Repair-Route frame is sent by a node which has lost communication with a remote node. The assumption is that the remote device has left the network and then re-joined, but without successfully removing all Routing table entries for its old route.

Repair-Route is a request for the remote node to repeat the "Delete-Route, Establish-Route" procedure. The frame is first sent from the local node to its parent. The parent then passes the frame to the next node down the route. Any Router which receives the frame checks whether the destination node is one of its children. If this is the case, the frame is sent to the child only. Any device which receives such a directed frame initiates the "Delete-Route, Establish-Route" procedure.

The Repair-Route frame structure is shown below.

Figure 11: Repair-Route Frame Structure

The source address of the local node is included in the frame (in the header), along with the destination address of the remote node. If the Response-Required flag is set in the Frame Type field (in the header), the remote node sends a response back to the local node. The response frame is identical to the request frame except that the Response-Required flag is cleared and the Response flag is set.

B.2.8 Purge-Route Frame

A Purge-Route frame is used to check whether Routing table entry (that has not been used for a long time) is still valid - that is, whether the corresponding descendant node still exists in the network.

The Purge-Route frame structure is shown below.

Figure 12: Purge-Route Frame Structure

The source address of the local node is included in the frame (in the header), along with the destination address of the descendant node.

B.2.9 Find-Node Frame

A Find-Node frame is used by the Co-ordinator to check for the existence of a node for which it has no Routing table entry but for which it has a message to be routed.

The Find-Node frame structure is shown below.

Figure 13: Find-Node Frame Structure

The source address of the Co-ordinator is included in the frame (in the header), along with the destination address of the remote node. If it exists, the remote node sends an Establish-Route frame back to the Co-ordinator.

The frame is broadcast and the value of the TTL (Time-to-live) field determines the number of times the frame can be forwarded during the broadcast.

B.2.10 Route-Request Frame

The Route-Request frame is used in Routing table maintenance.

The Route-Request frame structure is shown below.

Figure 14: Route-Request Frame Structure

B.2.11 Route-Import Frame

The Route-Import frame is used in re-shaping the network.

The Route-Request frame structure is shown below.

Figure 15: Route-Import Frame Structure

B.2.12 Activate-Services Frame

The Activate-Services frame is sent by an End Device to register a set of services with its parent. The parent then uses the list of registered services whenever it receives a Request-Services frame from another node.

Note: Routers do not send Activate-Services frames, as they advertise their own services locally rather than through their parent.

The Activate-Services frame structure is shown below.

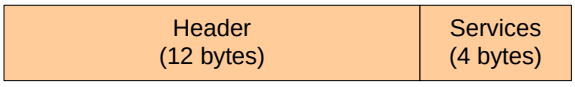

Figure 16: Activate-Services Frame Structure

The most significant bit of the Services field is ignored - only the bottom 31 bits specify services.

If the Response-Required flag is set in the Frame Type field (in the header), the parent sends a response back to the local node. The response frame is identical to the request frame except that the Response-Required flag is cleared, and the Response flag is set.

B.2.13 Service-Request Frame

The Service-Request frame is used by nodes to discover remote nodes that offer a specified set of services.

- If a Router generates this frame, it broadcasts the frame immediately.
- If an End Device generates the frame, it first sends the frame in unicast mode to its parent. The parent then broadcasts the frame to the network.

The Service-Request frame structure is shown below.

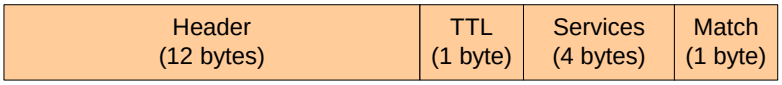

Figure 17: Service-Request Frame Structure

Only the source address of the local node is included in the frame (in the header). The least significant byte of the frame is set to:

- 1 if all the specified services must match
- 0 if any of the specified services can match

This frame is broadcast and the value of the TTL (Time-to-Live) field determines the number of times the frame can be forwarded during the broadcast.

When a Router receives a Service-Request frame, if either itself or one of its End Device children offers the specified services then it sends a response to the local node, regardless of the status of the Response-Required flag in the Frame Type field (in the header). The structure of the Response frame is shown in the figure below.

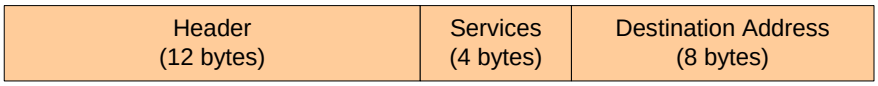

Figure 18: Service-Request Response Frame Structure

The Response frame is identical to the Service-Request frame, except that the address of the node which offers the specified services is appended, and the Response flag is set in the Frame Type field.

B.2.14 Ping Frame

A Ping frame is used by devices to check the integrity of its parent:

- If the parent is present and still accepts the local node as its child, a Ping response frame is sent to the local node.
- If the parent is not present, no response is sent.
- If the parent is present but does not accept the local node as its child (for example, if it has left and re-joined the network), an Unknown-Node frame is sent to the local node (see Appendix B.2.15).

The frame consists only of the header, as shown below.

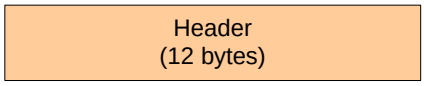

Figure 19: Ping Frame Structure

Neither source nor destination addresses are included, as this is a 1-hop transaction and the IEEE 802.15.4 MAC layer addressing is sufficient. If a response is to be sent, it is sent regardless of the state of the Response-Required flag in the Frame Type field. The response frame is identical to the Ping frame except that the Response-Required flag is cleared, and the Response flag is set.

B.2.15 Unknown-Node Frame

An Unknown-Node frame is generated by a Router which has received a frame from a device which is neither its child nor its parent.

The frame consists only of the header, as shown below.

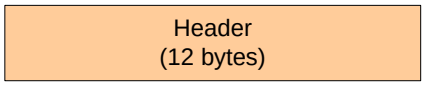

Figure 20: Unknown-Node Frame Structure

Neither source nor destination addresses are included, as this is a 1-hop transaction and the IEEE 802.15.4 MAC layer addressing is sufficient.

B.2.16 Set-Depth Frame

A Set-Depth frame is sent by a parent node to all of its descendants to indicate the depth of the node in the network. The frame may be sent after the parent and its descendants have been moved in the network, in order to inform the descendants of the parent's new depth in the network. The descendants can then update their own depths.

The Set-Depth frame structure is shown below.

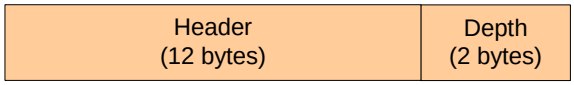

Figure 21: Set-Depth Frame Structure

C. Beacons

A JenNet beacon is a frame used by a routing node (Router or Co-ordinator) to describe itself to other nodes of the network during a network scan. In fact, these beacons are sent by the IEEE 802.15.4 MAC layer and not by the JenNet level, and so are distinct from JenNet frames (and do not follow their structure).

The structure of a JenNet beacon is shown below:

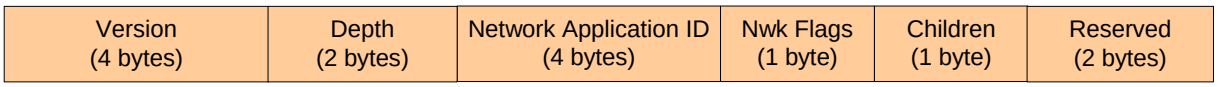

Figure 22: JenNet Beacon Frame Structure

The beacon includes the following information:

- JenNet version
- Depth of node in network (Co-ordinator is at depth zero)
- Network Application ID
- Network flags
- Number of child nodes

D. Glossary

Term	Description
Address	A numeric value that is used to identify a network node. In JenNet, the 64-bit IEEE/MAC address of the device is used.
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 node, as well as high-level interfacing to the network.
Binding	The process of associating a service on one node with a compatible service on another node so that communication between them can be performed without specifying addresses.
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 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.
Context Data	Data which reflects the current state of the node. The context data must be preserved during sleep mode (of an End Device).
Co-ordinator	The node through which a 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.
End Device	A 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.
IEEE 802.15.4	A standard network protocol that is used as the lowest level of the JenNet software stack. Among other functionality, it provides the physical interface to the network's transmission medium (radio).
Jenie API	Easy-to-use interface between the application and the JenNet software stack.
JenNet	Proprietary network protocol which is based on IEEE 802.15.4. An application interacts with the JenNet software stack through the Jenie API.
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. Note that the parent must have "permit joining" enabled.
Network Application ID	A 32-bit value that identifies the network application (e.g. a product). It is used in JenNet as the main way to identify a network (rather than using the PAN ID).
PAN ID	Personal Area Network Identifier - this is a 16-bit value that uniquely identifies the network in that all neighbouring networks must have different PAN IDs.
Parent	A 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.

Term	Description
Registering Services	The process by which a node provides a list of its services to the network. A parent node holds its own service list and those of its children.
Requesting Services	The process by which a node specifies the services that it requires from other nodes. The remote nodes send responses detailing which of these services they support.
Router	A node which provides routing functionality (in addition to input/output/processing) if used as a parent node. Also see Routing.
Routing	The ability of a 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.
Service	A JenNet concept corresponding to a feature, function or capability of a node (e.g. support of LCD display). A node can support up to 32 services.
Service Profile	The list of services supported in a network. It is represented as a 32-bit value in which each bit represents a service - ‘1’ indicating service supported, ‘0’ indicating service not supported.
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. Only End Devices can 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.
UART	Universal Asynchronous Receiver Transmitter - a standard interface used for cabled serial communications between two devices (each device must have a UART).

Revision History

Version	Date	Comments
1.0-1.4	2007-2010	Editions covering only JenNet stack information (Jenie API documented separately).
2.0	28-Sept-2010	Re-worked to incorporate former *Jenie API User Guide (JN-UG-3042)* and *Jenie API Reference Manual (JN-RM-2035)*. JPI functions removed from Jenie API description. Other minor modifications and corrections also made.

Important Notice

Jennic reserves the right to make corrections, modifications, enhancements, improvements and other changes to its products and services at any time, and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders, and should verify that such information is current and complete. All products are sold subject to Jennic's terms and conditions of sale, supplied at the time of order acknowledgment. Information relating to device applications, and the like, is intended as suggestion only and may be superseded by updates. It is the customer's responsibility to ensure that their application meets their own specifications. Jennic makes no representation and gives no warranty relating to advice, support or customer product design.

Jennic assumes no responsibility or liability for the use of any of its products, conveys no license or title under any patent, copyright or mask work rights to these products, and makes no representations or warranties that these products are free from patent, copyright or mask work infringement, unless otherwise specified.

Jennic products are not intended for use in life support systems/appliances or any systems where product malfunction can reasonably be expected to result in personal injury, death, severe property damage or environmental damage. Jennic customers using or selling Jennic products for use in such applications do so at their own risk and agree to fully indemnify Jennic for any damages resulting from such use.

All trademarks are the property of their respective owners.

NXP Laboratories UK Ltd
(Formerly Jennic Ltd)
Furnival Street
Sheffield
S1 4QT
United Kingdom

Tel: +44 (0)114 281 2655
Fax: +44 (0)114 281 2951
E-mail: info@jennic.com

For the contact details of your local Jennic office or distributor, refer to the Jennic web site:

www.Jennic.com
TECHNOLOGY FOR A CHANGING WORLD