Jenie API
User Guide

JN-UG-3042

Revision 1.8

17 March 2010

Contents

About this Manual 7

Organisation 7
Conventions 8
Acronyms and Abbreviations 8
Related Documents 8
Feedback Address 9

1. Fundamental Concepts 11

1.1 Wireless Operation 12
1.1.1 Radio Communication 12
1.1.2 Battery Power 13
1.2 Network Communications 13
1.3 Network and Node Types 14
1.3.1 Star Topology 15
1.3.2 Tree Topology 16
1.4 Network Identitification and Isolation 18
1.4.1 Identification 18
1.4.2 Isolation 19
1.5 Node Addressing 20
1.6 Software Architecture 20
1.7 Services 21
1.7.1 Service Profile 23
1.7.2 Service Discovery 23
1.8 Bindings 24
1.9 Network Formation 26
1.10 Data Transfer 27
1.11 Routing 29
1.11.1 Neighbour and Routing Tables 29
1.11.2 Routing Process on a Node 29
1.12 Configurable Protocol Operations 30
1.12.1 Message Acknowledgements 30
1.12.2 Data Polling (End Device only) 30
1.12.3 Auto-ping 31

2. What is Jenie? 33

2.1 Jenie Architecture 33
2.2 Jenie Functionality 35
2.2.1 Core Functionality 35
2.2.2 Hardware Functionality 36
2.3 Forms of Jenie 36
2.4 Jenie API 37
2.5 Installing Jenie 38

3. Application Tasks 39

3.1 Starting the Network (Co-ordinator only) 40
3.2 Starting Other Nodes (Routers and End Devices) 41
3.3 Configuring the Radio Transmitter 43
3.4 Configuring Security 43
3.5 Discovering Services 44
3.5.1 Registering Services 44
3.5.2 Requesting Services 45
3.6 Binding Services 46
3.7 Transferring Data 46
3.7.1 Sending and Receiving Data using Addresses 47
3.7.2 Sending and Receiving Data using Bound Services 47
3.7.3 Receiving Data for an End Device 47
3.8 Obtaining Signal Strength Measurements 49
3.9 Entering and Leaving Sleep Mode (End Devices only) 50
3.9.1 Sleep Mode with Memory Held 51
3.9.2 Sleep Mode without Memory Held 51
3.10 Saving and Restoring Context Data 52
3.10.1 Network Context 52
3.10.2 Application Context 53
3.11 Leaving the Network 55

4. Working with the Jenie API 57

4.1 Jenie Application Templates 58
4.1.1 Pre-requisites 58
4.1.2 Supplied Files 58
4.2 Code Descriptions 59
4.2.1 Co-ordinator Code 60
4.2.2 Router Code 61
4.2.3 End Device Code 62

4.3 Building Your Application 63
4.3.1 Building Code using Makefiles 63
4.3.2 Building Code using Eclipse (JN5148 only) 64
4.3.3 Building Code using Code::Blocks (JN5139 only) 65
4.4 Downloading Code to Nodes 68

5. Controlling Hardware Peripherals 69
5.1 ADC 70
5.2 DACs 72
5.3 Comparators 73
5.4 Digital I/O 75
5.5 Timers 76
5.5.1 Timer/PWM Mode 78
5.5.2 Delta-Sigma Mode (NRZ and RTZ) 79
5.5.3 Capture Mode 80
5.6 Wake Timers 82
5.7 Serial Peripheral Interface (SPI) 84
5.8 Serial Interface (2-wire) 86
5.9 Intelligent Peripheral (IP) Interface 87

6. Advanced Issues in Network Operation 89
6.1 Identifying the Network 89
6.2 Sending Messages 90
6.2.1 Timing Issues in Data Sends 90
6.2.2 Re-tries in Data Sends 91
6.2.3 End-to-End Acknowledgements for Data Sends 92
6.3 Routing 93
6.3.1 Neighbour Tables and Routing Tables 93
6.3.2 Stale Route Purging 94
6.3.3 Automatic Route Importation 95
6.4 Losing a Parent Node (Orphaning) 96
6.4.1 Detecting Orphaning 96
6.4.2 Re-joining the Network 97
6.5 Losing a Child Node 97
6.5.1 End Device Children 97
6.5.2 Router Children 99
6.6 Auto-polling (End Device only) 100
6.7 Beacon Calming 100
6.8 Packet Loss 101
6.8.1 Packet Collisions 101
6.8.2 Minimising Packet Loss 102
6.8.3 Route Updates 104

© Jennic 2010

6.9 Network Self-Healing 104
6.9.1 Automatic Recovery 104
6.9.2 Network Recovery 105
6.10 Key Performance Parameters 106
6.10.1 Broadcast TTL (Time To Live) 106
6.10.2 Automatic Recovery Threshold 106
6.10.3 Ping Period 107
6.10.4 End Device Poll Period 108
6.10.5 End Device Scan Sleep Period 108

Appendices 109

A. Hardware and Memory Usage 109
A.1 Hardware Resources 109
A.2 Memory Resources 109

B. Glossary 111

About this Manual

This manual provides an introduction to Jenie - Jennic's proprietary interface that enables simplified and streamlined development of wireless network applications for the Jennic JN5139 and JN5148 wireless microcontrollers. In particular, this manual describes the Jenie Application Programming Interface (API). Little or no previous knowledge of wireless networks is assumed - all relevant concepts are covered by this manual.

Note: This manual provides high-level descriptions of the Jenie API and related wireless network concepts. For detailed descriptions of the Jenie API functions that can be used in application code, refer to the *Jenie API Reference Manual (JN-RM-2035)*.

Tip: Reference is made in this manual to the software levels that underlie Jenie and the wireless network application: JenNet and IEEE 802.15.4. In order to develop applications using Jenie, you should not need knowledge of these lower software levels beyond the information provided in this manual. However, if you do require more information, refer to the Jennic User Guide for the relevant protocol (see Related Documents).

Organisation

This manual consists of 6 chapters and 2 appendices, as follows:

- Chapter 1 provides a background in essential wireless network concepts, as well as those specific to Jenie.
- Chapter 2 introduces Jenie in terms of its architecture, functionality and available forms.
- Chapter 3 describes how to use the Jenie API to incorporate the network tasks into your wireless network application.
- Chapter 4 describes the essential features of applications that use the Jenie API, as well as how to build and download these applications.
- Chapter 5 describes the control of JN5139/JN5148 hardware peripherals using dedicated Jenie API functions.
- Chapter 6 addresses a number of advanced issues relating to wireless network application design using Jenie - this chapter therefore supplements Chapter 3.
- The Appendices provide information on the JN5139/JN5148 hardware and memory needed by Jenie, and a glossary of the main terms used in Jenie-based 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

API	Application Programming Interface
JenNet	Jennic Network
LQI	Link Quality Indication
MAC	Media Access Control
PAN	Personal Area Network
UART	Universal Asynchronous Receiver Transmitter

Related Documents

[1] Wireless Network Deployment Guidelines (JN-AN-1059)

[2] Jenie Application Templates Application Note (JN-AN-1061)

[3] Jenie Tutorial Application Note (JN-AN-1085)

[4] Jenie API Reference Manual (JN-RM-2035)

[5] JenNet Stack User Guide (JN-UG-3041)

[6] IEEE 802.15.4 Wireless Networks User Guide (JN-UG-3024)

Feedback Address

If you wish to comment on this manual, or any other Jennic user documentation, 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
Jennic Ltd
Furnival Street
Sheffield S1 4QT
United Kingdom

doc@jennic.com

1. Fundamental Concepts

The Jenie software from Jennic provides an easy-to-use, high-level interface for developing wireless network applications for the Jennic JN5139 and JN5148 wireless microcontrollers. Jenie simplifies and streamlines application development, therefore reducing development costs and time-to-market.

Jenie provides a software interface used in application programs to interact easily and efficiently with the lower level software that implements network operations (such as network initialisation, formation and communication). This is illustrated in Figure 1 below.

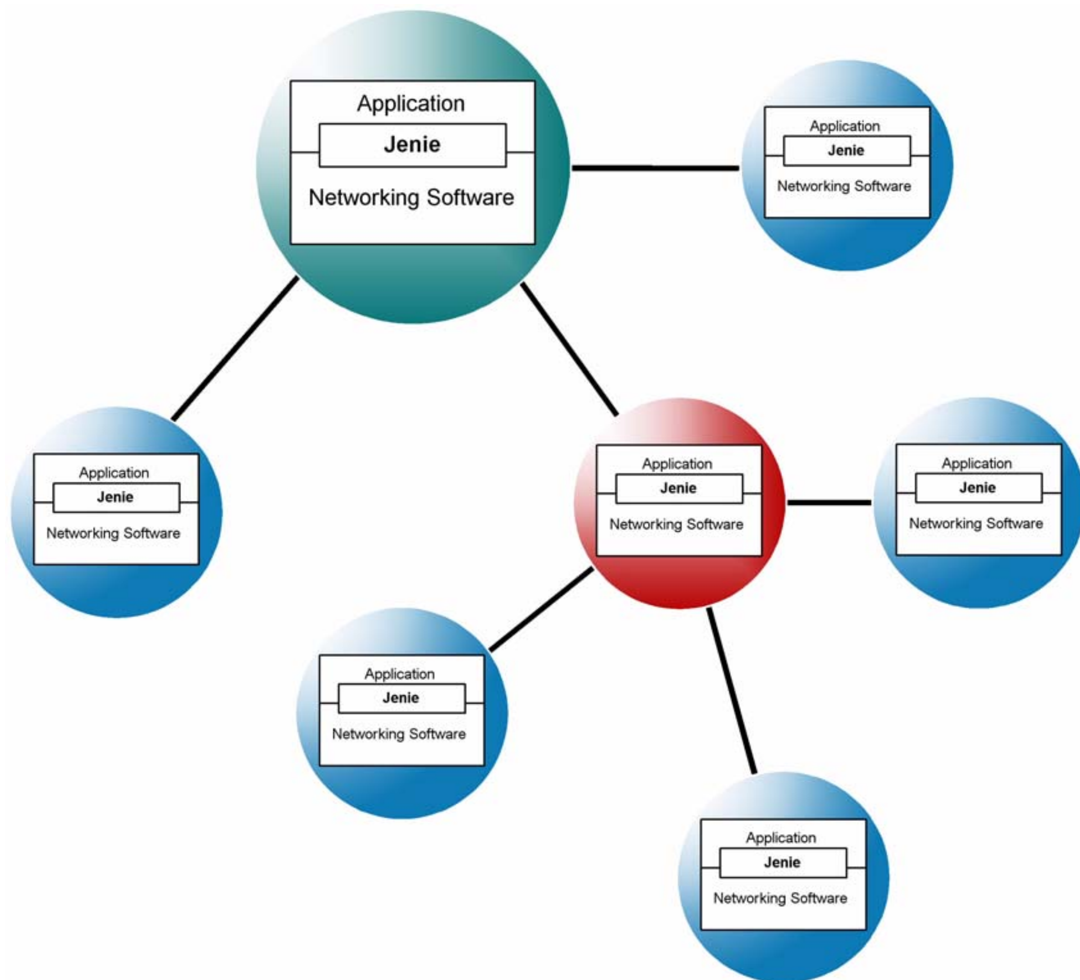


Figure 1: Jenie in Simple Wireless Network

This chapter describes essential wireless network concepts before Jenie is presented in more detail in the remainder of this manual. These concepts should provide a sufficient foundation for developing simple wireless network applications using Jenie.

Jenie allows applications to be built on top of Jennic's proprietary JenNet protocol. The software structure is further detailed in Chapter 2.

Tip: In order to develop applications using Jenie, you should not need knowledge of JenNet beyond the information provided in this manual. However, if you do require more information, refer to the *JenNet Stack User Guide (JN-UG-3041)*, available from the Support area of the Jennic web site (www.jennic.com/support).

1.1 Wireless Operation

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

1.1.1 Radio Communication

Jenie is designed to run on Jennic's wireless microcontrollers, featuring integrated radio transceivers which operate 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, this RF band is split into 16 channels. It is possible to automatically select the best channel (that with least detected activity) at system start-up.

The range of a radio transmission is dependent on the operating environment (inside or outside a building), the Jennic module (carrying the wireless microcontroller) and the type of antenna used. Using a JN5139 or JN5148 standard module fitted with an external dipole antenna, a range of 1 km can typically be achieved in an open area. Inside a building, this can be reduced due to absorption, reflection, diffraction and standing wave effects caused by walls and other solid objects. High-power modules can achieve a factor of five greater than this.

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

1.1.2 Battery Power

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

Low-capacity batteries are often used and their use is optimised by restricting the time for which energy is required. To this end, data is transmitted infrequently (perhaps once per hour or even per week), with the device reverting to low-power sleep mode the rest of the time. However, not all network devices can be battery-powered, since some node types must be switched on all the time (see Section 1.3).

1.2 Network Communications

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

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

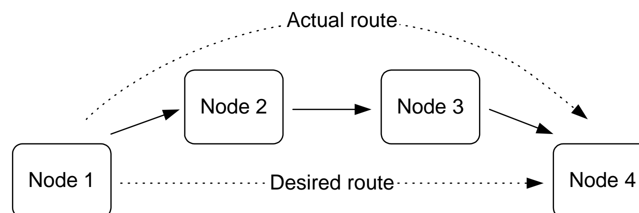


Figure 2: Routing between Network Nodes

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. Routing is described further in Section 1.11.

1.3 Network and Node Types

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

- Co-ordinator
- Router
- End Device

These node types and their roles are summarised in Table 1 below. Note that every wireless network must have a Co-ordinator.

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

The application on each node configures that 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 Jenie 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).

1.3.1 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.

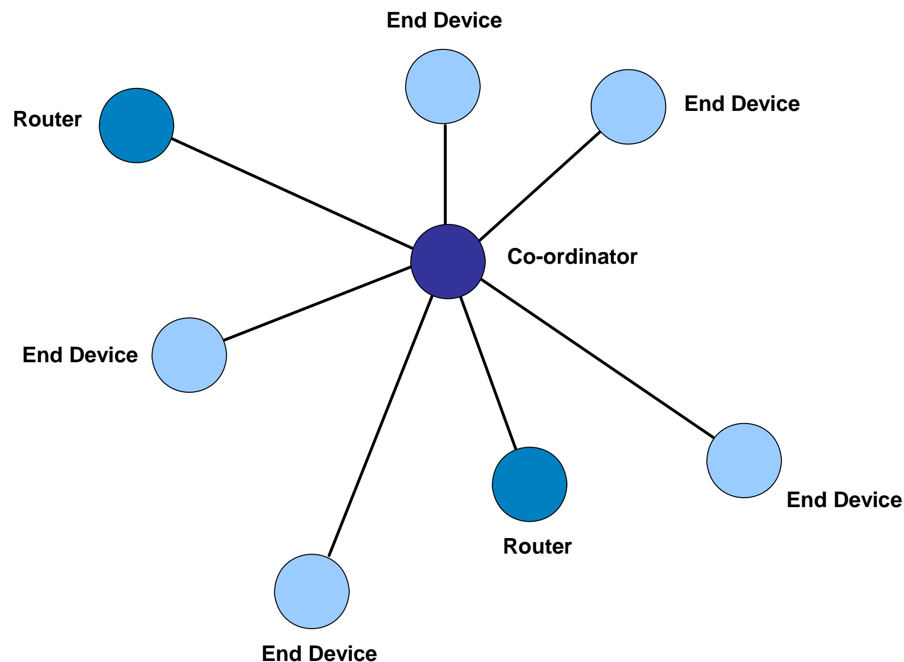


Figure 3: Star Topology

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.

 © Jennic 2010

1.3.2 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 - only its application will be relevant.

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

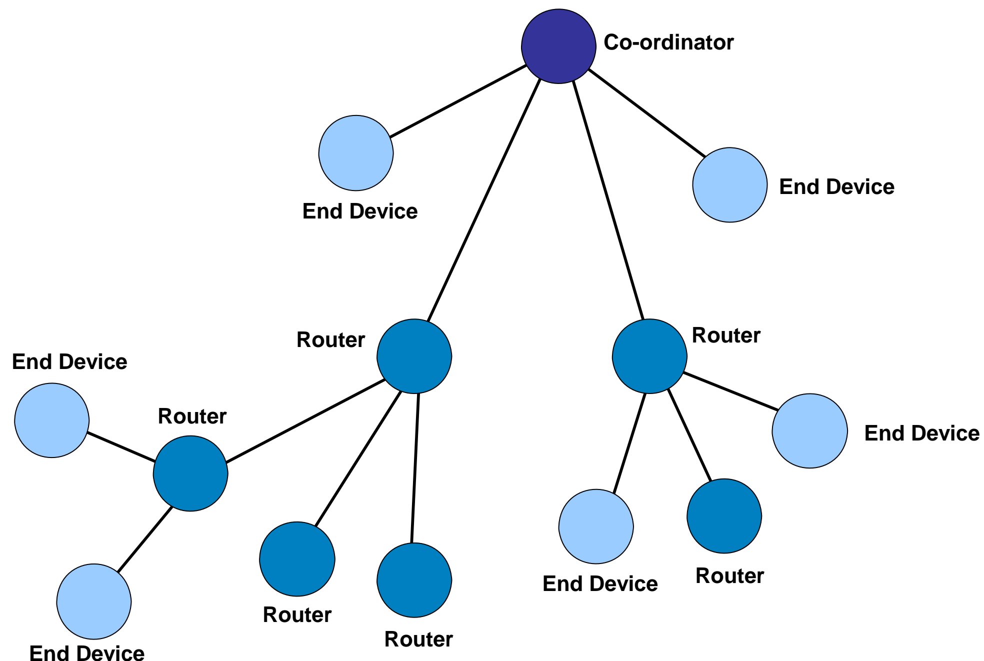


Figure 4: Tree Topology

Note that:

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

The communication rules in a Tree topology are:

- A child can only directly communicate with its parent and with its own children (if any).
- A parent can only directly communicate with its children and with its own parent.
- 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 network 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.

© Jennic 2010

1.4 Network Identitification and Isolation

This section describes how a Jenie 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.

1.4.1 Identification

Wireless networks must be uniquely identified so that there is no confusion between neighbouring networks. Jenie 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 truely 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 Jenie 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 Jenie 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 Jenie 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 1.4.2).

© Jennic 2010

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. 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 1.4.2.

1.4.2 Isolation

It is normally practicable for a Jenie wireless network to be uniquely identified within its operating environment using its Network Application ID and PAN ID (described in Section 1.4.1). 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.

© Jennic 2010

1.5 Node Addressing

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

1.6 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.

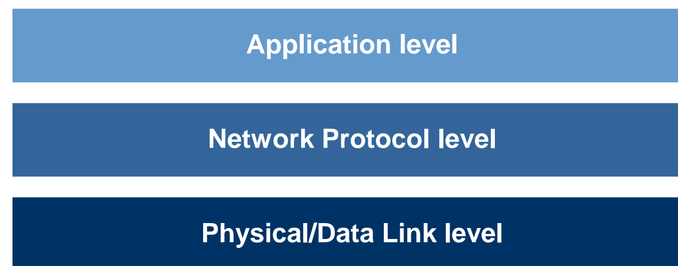


Figure 5: Basic Software Architecture

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.
- **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).

The above software architecture is described in more detail in Section 2.1.

1.7 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 1.7.1.

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 6 below. 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.

© Jennic 2010

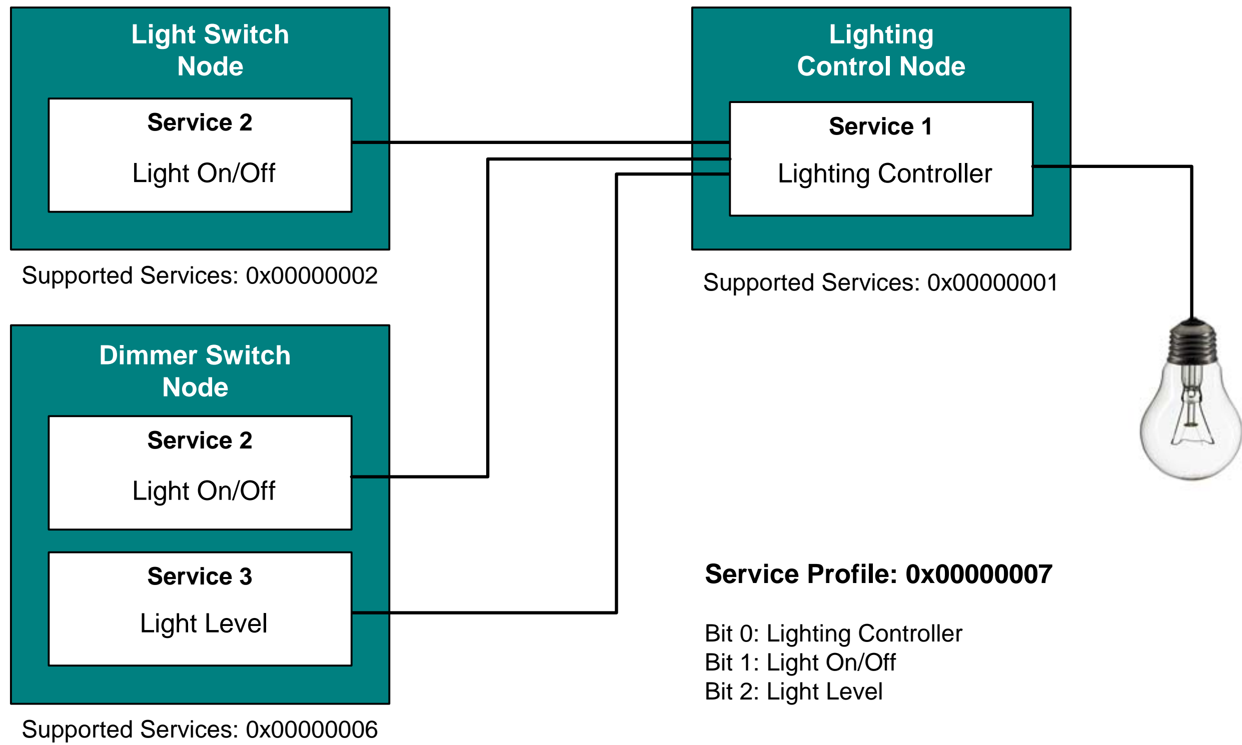


Figure 6: Example Lighting Control System

1.7.1 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 6 above, where the Service Profile is expressed as the hexadecimal value 0x00000007.

1.7.2 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 Jenie which services it supports. An application can also request all nodes that support a particular service. Jenie 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 3.5.

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.

© Jennic 2010

1.8 Bindings

As described in Section 1.7, 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 1.10).

The binding of services is illustrated in the lighting control system of Figure 6 above.

Jenie maintains a set 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

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 7 below, which also shows the bindings between services on different nodes.

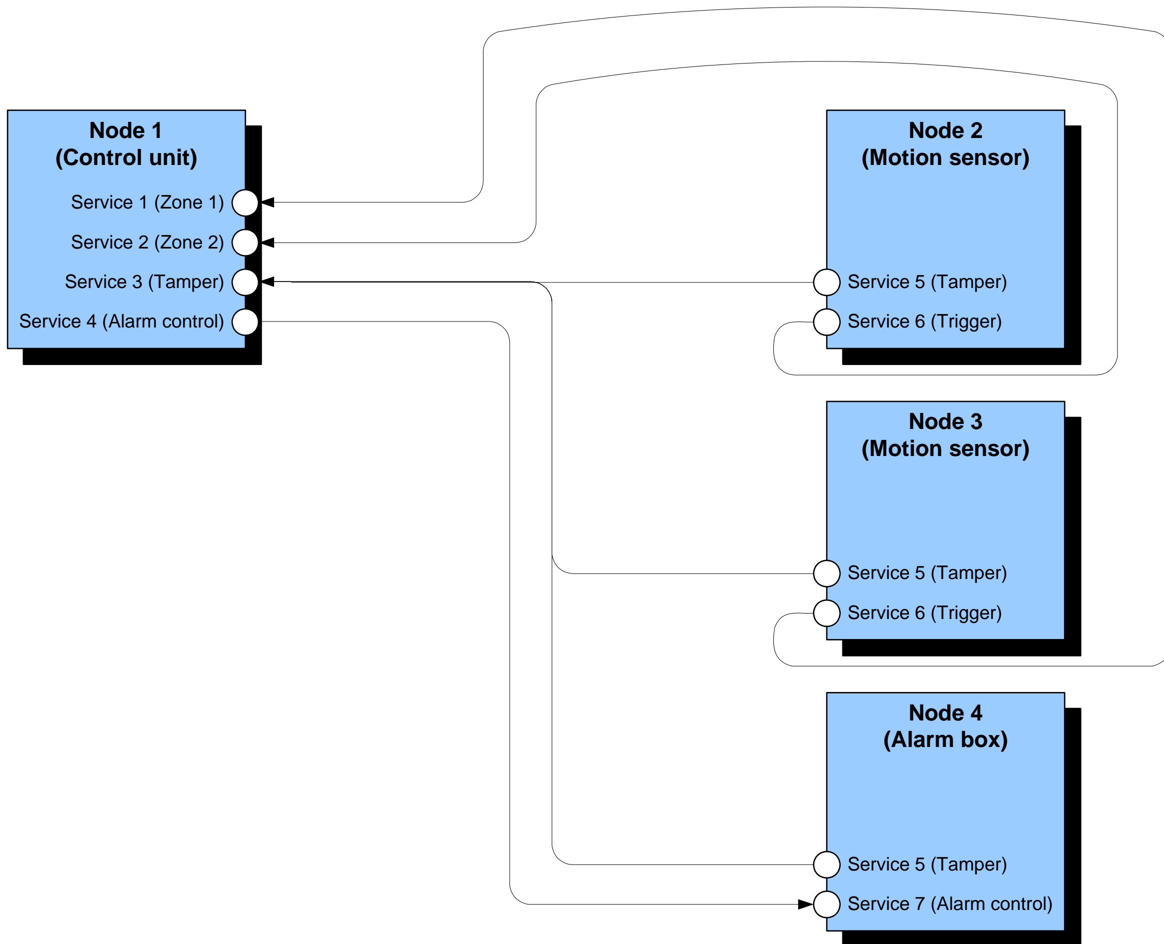


Figure 7: Bindings in Example System

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

1.9 Network Formation

The creation of a Jenie wireless network starts with the Co-ordinator.

Note: A Jenie network uses the Network Application ID (see Section 1.4) 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.

The procedure for starting and forming the network is then as follows:

On Co-ordinator

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.

On Other Devices

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).

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.

1.10 Data Transfer

Data passed between nodes can contain any kind of information, as is it not interpreted by Jenie. There are two ways of referring to nodes, depending on whether service binding is being used:

- **Using Addresses:** Data is sent to a particular node using the address of that node (the address obtained from the discovery stage - see Section 1.7.2). It is also possible to perform a broadcast to all Router nodes in the network.
- **Using Binding:** Data is sent from a service on the local node to one or more other services on remote nodes. The destination or destinations are determined by the binding relationships defined by the application - 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 8 below. For information on binding, refer to Section 1.8.

When data is received by a node, the address of the data source and the service (if applicable) are passed up to the application, together with the data itself.

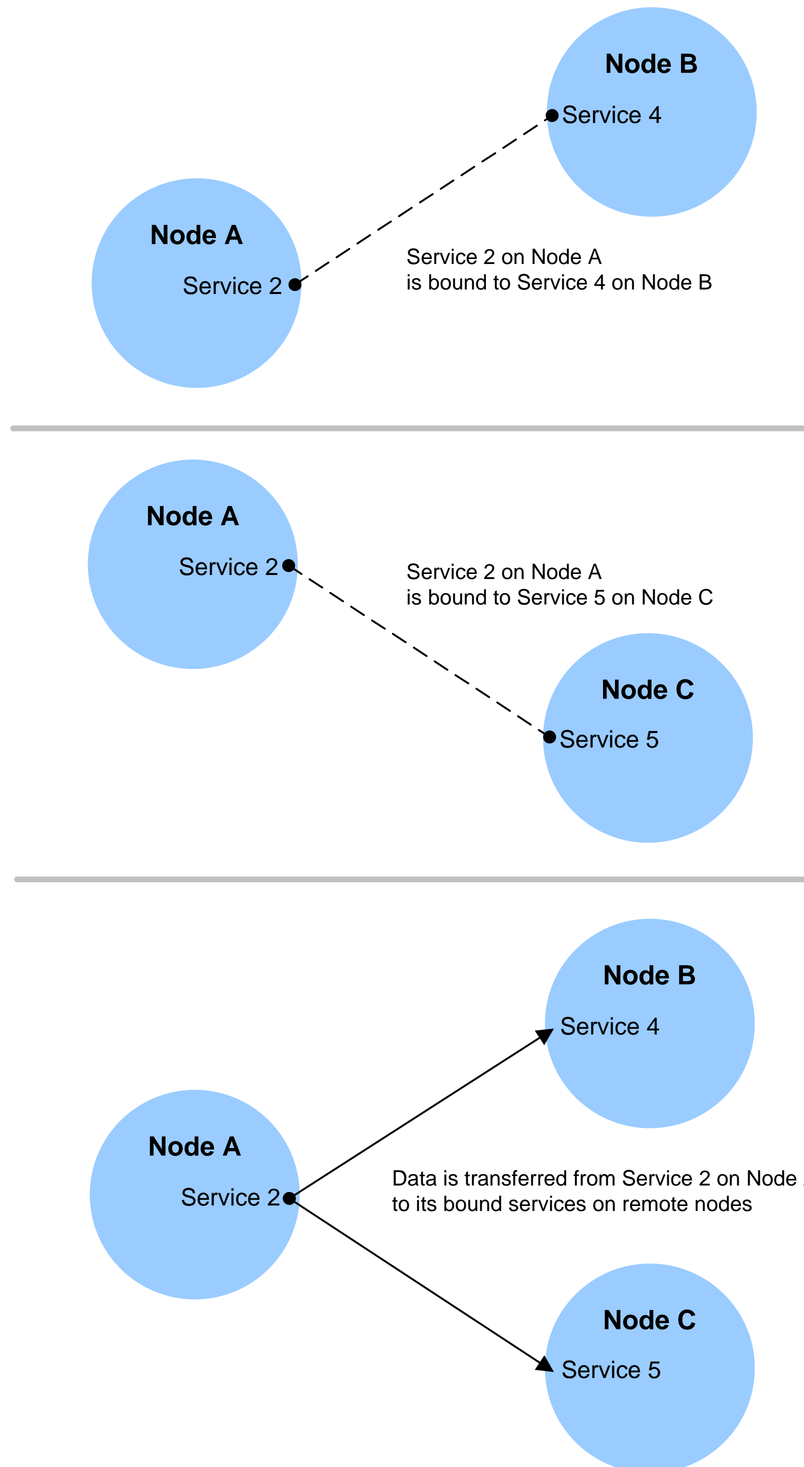


Figure 8: Data Transfer using Binding

1.11 Routing

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

- In a star network, all messages are routed by the Co-ordinator.
- In a tree network, unless a message is passed directly between parent and child (in either direction), the message must be routed by one or more routing nodes - that is, Routers and possibly the Co-ordinator (if the message reaches the top of the tree).

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

1.11.1 Neighbour and Routing Tables

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

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

Together, these tables give a Router knowledge of all descendant nodes in the tree 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.

1.11.2 Routing Process on a Node

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

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

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

1.12 Configurable Protocol Operations

This section describes a number of network protocol tasks:

- Message Acknowledgements (Section 1.12.1)
- Data Polling (Section 1.12.2)
- Auto-ping (Section 1.12.3)

Although these tasks are automatic during network operation, you may need to configure them when initialising the network in your application code.

1.12.1 Message Acknowledgements

When a message is sent from one node to another node, on receipt of the message 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: 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.

Acknowledgements can be enabled or disabled for an individual message transmission.

1.12.2 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 destination node may be in sleep mode. Consequently, the parent of the destination node buffers the data until the End Device is out of sleep mode and ready to receive data. It is the responsibility of the End Device to poll its parent to check whether there is pending data waiting to be delivered.

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).*

1.12.3 Auto-ping

A node may lose its parent and be unaware of this loss, particularly if data exchanges with its parent are infrequent. In Jenie, 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 the child must attempt 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 must 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 1.12.2). 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 1.12.2). 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).

© Jennic 2010

2. What is Jenie?

Jenie provides an easy-to-use interface which allows a user application to interact with the Network level (JenNet) and Physical/Data Link level (IEEE 802.15.4) of the wireless network software stack. Jenie functions/commands are used in the application code to:

- pass instructions and/or data down to the underlying software stack
- handle events received from the underlying software stack

The location of Jenie in the software stack is illustrated in Figure 9 below.

2.1 Jenie Architecture

As described in Section 1.6, the software that runs on each node of a wireless network is organised in levels ranging from high-level functions/commands, used directly by the application, down to the lowest level software that interacts with the node's radio transceiver. The specific implementation of the software stack levels in a Jenie network is illustrated and described below.

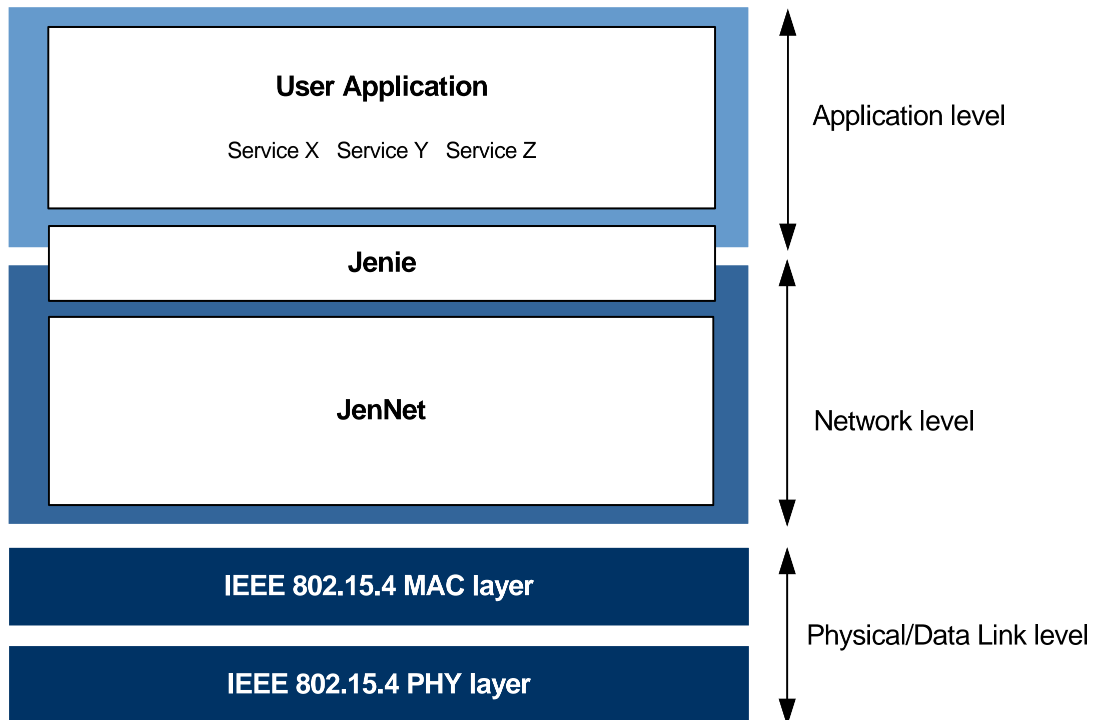


Figure 9: Detailed Software Architecture in Jenie

© Jennic 2010

The above diagram shows (from top to bottom):

Application Level

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

The user application interacts with the network through Jenie, which provides an easy-to-use interface used by the application code.

Network Level

This is the network layer that is implemented using the JenNet protocol. It handles network addressing and routing by invoking actions in the MAC layer (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

The network level interacts with the services in the Application layer through Jenie.

Tip: In order to develop wireless network applications using Jenie, no knowledge of the underlying JenNet network protocol is normally required. However, if you do require more information on JenNet, refer to the *JenNet Stack User Guide (JN-UG-3041)*.

Physical/Data Link Level

This is provided by the IEEE 802.15.4 standard. This level 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 packets or 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 wireless network applications using Jenie, no knowledge of IEEE 802.15.4 is required. However, if you do require more information on IEEE 802.15.4, refer to the *Jennic IEEE 802.15.4 Wireless Network User Guide (JN-UG-3024)*.

© Jennic 2010

2.2 Jenie Functionality

Jenie provides core functionality for managing network and system tasks, as well as functionality for interfacing with the hardware peripherals on the JN5139/JN5148 wireless microcontroller. These two functional areas are outlined below.

2.2.1 Core Functionality

Jenie provides functionality for implementing network management, data transfer and system tasks, as follows.

Network Management Tasks

The network management functionality provided by Jenie 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 provided by Jenie 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 provided by Jenie 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
- obtain the version number of a component on the node (this task verifies that the node is operating)
- handle hardware events

Note that ‘doze mode’ of the JN5139/JN5148 device is not supported by Jenie.

2.2.2 Hardware Functionality

Jenie also includes functionality for interacting with the integrated peripherals of the JN5139/JN5148 wireless microcontroller. These peripherals include:

- Analogue resources: ADC, DACs, comparators
- Digital I/O (DIOs)
- UARTs
- Timers
- Wake timers
- Serial Peripheral Interface (SPI)
- 2-Wire Serial Interface (SI)
- Intelligent Peripheral (IP) interface

For further information on controlling the on-chip peripherals, refer to Chapter 5.

Note: This Jenie hardware functionality is provided for Jennic customers who are maintaining Jenie applications for the JN5139 device or migrating Jenie applications from the JN5139 to the JN5148 device. Any new Jenie application development for the JN5139 or JN5148 device should instead use the Integrated Peripherals API, which is described in the *Integrated Peripherals API Reference Manual (JN-RM-2001)*.

2.3 Forms of Jenie

Jenie is available in two forms:

- **Jenie API:** This Application Programming Interface (API) comprises high-level functions that can be incorporated in an application running on a Jennic JN5139/JN5148 wireless microcontroller.
- **AT-Jenie:** This interface comprises serial commands that can be sent to a Jennic JN5139 wireless microcontroller from an application possibly running on a separate processor.

This User Guide describes the Jenie API, which is introduced further in the next section.

Tip: AT-Jenie provides an easy alternative to the Jenie API for application developers who do not wish to incorporate the Jenie function calls directly in their code. Note that AT-Jenie is currently available only for the JN5139 device.

© Jennic 2010

2.4 Jenie API

The Jenie API is an optimised Application Programming Interface providing a simple, easy-to-use yet powerful set of C functions designed to streamline application development for wireless networks. The API functions are used directly in application code to be run on the Jennic JN5139/JN5148 wireless microcontroller. This is illustrated in Figure 10 below.

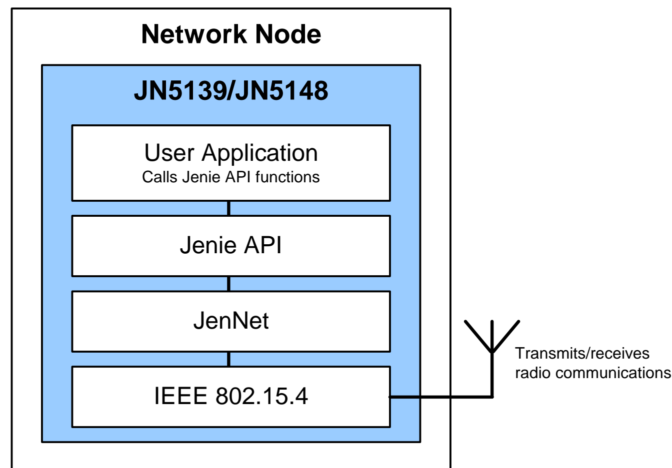


Figure 10: Application using Jenie API

Jenie functionality is outlined in Section 2.2. The Jenie API functions are described in the *Jenie API Reference Manual (JN-RM-2035)*.

2.5 Installing Jenie

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

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

- AT-Jenie Command Parser (JN-SW-4030 only)
- Jenie API
- JenNet protocol software
- IEEE 802.15.4 protocol software
- Integrated Peripherals API
- Board API

Note that you will only need the command parser if you wish to use the AT-Jenie serial command set (on the JN5139 device).

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

- Cygwin CLI
- Code::Blocks IDE (JN-SW-4031 only) or Eclipse IDE (JN-SW-4041 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 (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 Jennic SDK Installation Guide (JN-UG-3035 for the JN5139 SDK, JN-UG-3064 for the JN5148 SDK).*

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

3. 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 the *Jenie API Reference Manual (JN-RM-2035)*.

The Jenie API functions are divided into "application to stack" functions and "stack to application" (or "callback") functions. For further information, refer to Section 4.2.

Tip: Further guidance to application development using Jenie is provided through application templates, described in Chapter 4. You are strongly advised to use these templates.

© Jennic 2010

3.1 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.

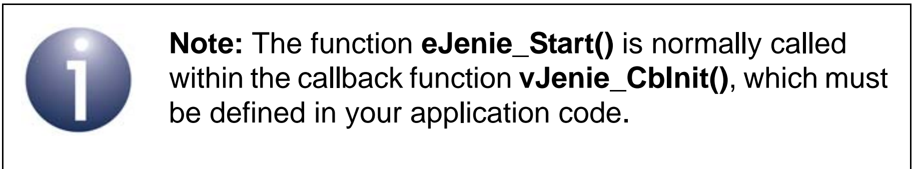
The network is first configured using the **vJenie_CbConfigureNetwork()** callback function, which acts as the entry point for the application code. This function allows network parameters to be set, including those listed in the table below (for full network parameter definitions, refer to the *Jenie API Reference Manual (JN-RM-2035)*).

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.

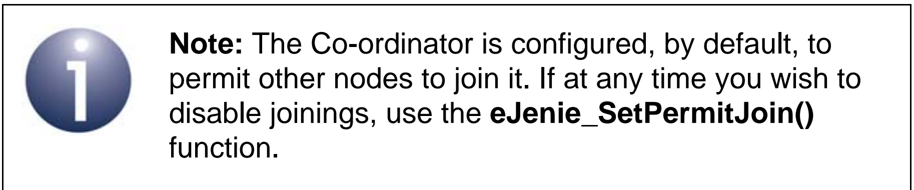
Further guidance on using some of the global parameters is provided in Appendix A.

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 3.2) 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.

3.2 Starting Other Nodes (Routers and End Devices)

Once the network has been started through the Co-ordinator, as described in Section 3.1, 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 the *Jenie API Reference Manual (JN-RM-2035)*).

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).

 © Jennic 2010

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 Appendix A. 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 1.9. 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.

3.3 Configuring the Radio Transmitter

The radio transmission power of the JN5139/JN5148 wireless microcontroller can be set using the Jenie API function **eJenie_RadioPower()**. The power levels for JN5139/ JN5148-based modules can be set as multiples of 6 dBm in the following ranges:

- Standard modules: -30 to 0 dBm (default: 0 dBm)
- High-power modules: -12 to +18 dBm (default: 18 dBm)

The power level can be set in these ranges but should normally be left at the default value. Note that 'boost mode' of the JN5139 device is not supported by Jenie.

The above function can also be used to switch the radio transmitter off and on.

3.4 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 the *JenNet Stack User Guide (JN-UG-3041)*.

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.

Caution: *In the current release of Jenie, 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()**.

3.5 Discovering Services

A node of a Jenie 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 Jenie 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 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.

3.5.1 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 1.7). 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 Jenie API 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()**.

3.5.2 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.

© Jennic 2010

3.6 Binding Services

In Jenie 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 Jenie API 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 3.5.2). Once a service binding has been created, messages can be sent from the local service to the remote service as described in Section 3.7.2.

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()**.

3.7 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 3.5) 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 3.7.3. 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 3.9.2.

3.7.1 Sending and Receiving Data using Addresses

Data can be sent to a remote node using the Jenie API 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()**.

3.7.2 Sending and Receiving Data using Bound Services

Data can be sent from a service to one or more bound services using the Jenie API function **eJenie_SendDataToBoundService()**. This method assumes the source and destination services have been bound as described in Section 3.6. 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()**.

3.7.3 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.

© Jennic 2010

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()** 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 variable *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.

3.8 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 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 function **u8Api_GetLastPktLqi()** to obtain the LQI value of the last packet received by the local node. A description of this function can be found in the JenNet appendix of the *Jenie API Reference Manual (JN-RM-2035)*.

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 (the power obtained from an LQI value of 0 can only be considered as the maximum possible power detected, while the power obtained from an LQI value of 255 can only be considered as the minimum possible power detected).*

© Jennic 2010

3.9 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.

Jenie 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 once 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 deriving from the on-chip comparators or DIOs, but 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.

© Jennic 2010

3.9.1 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.

3.9.2 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 variable *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 3.10 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.*

© Jennic 2010

3.10 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, Jenie 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.
- **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.

For full details of all functions, macros and parameters, refer to the *Jenie API Reference Manual (JN-RM-2035)*.

3.10.1 Network Context

In order to save network context data to external non-volatile memory, it is first necessary to set the global variable *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()**.

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 3.10.2.

In addition to *gJenie_RecoverFromJpdm*, the following global variables 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 variable, *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 rejoined 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 rejoins the network. Therefore, the children will all need to rejoin the network and it does not matter if some of them have already rejoined via new parents during the power loss.

3.10.2 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 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 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 3.10.1). 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.

3.11 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 variables 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 variables 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.

© Jennic 2010

4. Working with the Jenie API

This chapter describes application development using the Jenie API, based on the Jenie application templates available from Jennic. It also describes how to build your applications and download them to the target nodes.

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.*

4.1 Jenie Application Templates

The Jenie application templates provide a basis for your own application development for a wireless network. These templates are supplied in the Application Note *Jenie Application Templates (JN-AN-1061)*, available under **Application Notes** in the Support area of the Jennic web site (www.jennic.com/support).

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.

4.1.1 Pre-requisites

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.

4.1.2 Supplied Files

Three C source files are provided, one for each node type:

- **AN1061_JN_Coord.c** for the Co-ordinator
- **AN1061_JN_Router.c** for a Router
- **AN1061_JN_EndD.c** for an End Device

For each of the above applications, files are provided for building the binaries:

- Makefiles
- Code::Blocks project files (**.cbp**) for the JN5139 device
- Eclipse project files (**.project** and **.cproject**) for the JN5148 device

© Jennic 2010

4.2 Code Descriptions

This section describes the supplied application source code at function level - this is for a cold start. The code includes two types of function:

- "Application to stack" functions that are called in the application to interact with the software stack through Jenie. These functions are defined in the Jenie API.
- "Stack to application" functions that are called by the software stack through Jenie to interact with the application - these are referred to as "callback" functions. Their prototypes are included in the Jenie API but you must define their content in your application code.

The general structure of the application code is illustrated in Figure 11. The sub-sections which follow describe the code for the Co-ordinator, Router and End Device.

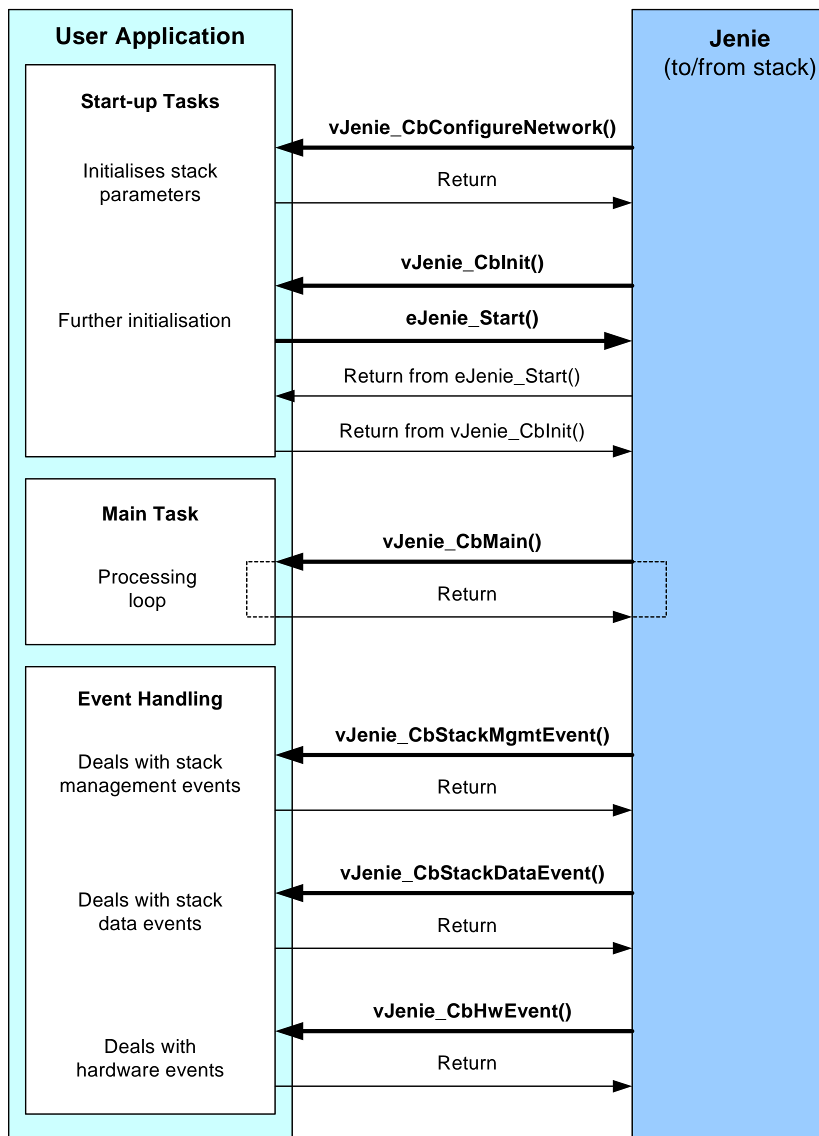


Figure 11: Application Code Overview

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 required.

4.2.1 Co-ordinator Code

The Co-ordinator application, **AN1061_JN_Coord.c**, is structured as illustrated in Figure 11 and described below:

1. The entry point from Jenie 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. Jenie 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, Jenie 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 Jenie, but between calls Jenie may generate events which are sent to the application. The application must define callback functions that can be invoked by Jenie 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 JN5139/JN5148 wireless microcontroller or carrier board.

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

© Jennic 2010

4.2.2 Router Code

The Router application, **AN1061_JN_Router.c**, is structured as illustrated in Figure 11 and described below (the overall structure is very similar to that of the Co-ordinator code).

1. The entry point from Jenie 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. Jenie 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, Jenie 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 Jenie, but between calls Jenie may generate events which are sent to the application. The application must define callback functions that can be invoked by Jenie 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 JN5139/JN5148 wireless microcontroller or carrier board.

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

© Jennic 2010

4.2.3 End Device Code

The End Device application, **AN1061_JN_EndD.c**, is structured as illustrated in Figure 11 and described below (the overall structure is very similar to that of the Co-ordinator and Router code):

1. The entry point from Jenie 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. Jenie 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, Jenie 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 Jenie, but between calls Jenie may generate events which are sent to the application. The application must define callback functions that can be invoked by Jenie 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 JN5139/JN5148 wireless microcontroller or carrier board.

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

© Jennic 2010

4.3 Building Your Application

Once you have created your source files (for example, **Coordinator.c**, **Router.c** and **EndDevice.c**), you must build the executables on a PC or workstation before downloading them to the relevant devices. There are two possible methods of building the applications, depending on your development environment:

- Makefiles (for CLI users)
- IDE (Eclipse for JN5148 users, Code::Blocks for JN5139 users)

These are described in the subsections below.

For all build methods, your project directory must be located in:

- **<JENNIC_SDK_ROOT>\cygwin\jennic\SDK\Application** for JN5139
- **<JENNIC_SDK_ROOT>\Application** for JN5148

where **<JENNIC_SDK_ROOT>** is the path into which the Jennic SDK was installed.

Note that the Jenie library file with which an application is linked depends on the node type, as follows:

- The Co-ordinator and Router applications are both linked to the library file **Jenie_TreeCRLib.a**.
- The End Device application is linked to the library file **Jenie_TreeEDLib.a**.

The relevant library file must be included in the makefile or project file, as appropriate.

4.3.1 Building Code using Makefiles

This section describes how to build your application code using a makefile.

You should base your makefiles on the examples supplied by Jennic in the Application Note *Jenie Application Templates (JN-AN-1061)*. There is a makefile for each node type, located in the **Build** sub-directory for the corresponding application. For example, the Co-ordinator makefiles are located in:

...\JN-AN-1061-Jenie-Application-Template\AN1061_JN_Coord\Build

Different makefiles are provided for JN5139 and JN5148 - a JN5148 makefile is simply called **Makefile** and a JN5139 makefile is called **Makefile_JN5139.mk**.

Build your code as follows:

1. Navigate to the **Build** directory for the application to be built and follow the instructions below for your chip type:

   For JN5139:

   At the command prompt, enter:

   ```
   make -f Makefile_JN5139.mk clean all
   ```

   For JN5148:

   At the command prompt, enter:

   ```
   make clean all
   ```

Tip: For the JN5148 device, you can alternatively enter the above command from the top level of the project directory, which will build the binaries for all applications.

In all the above cases, the binary file will be created in the relevant **Build** directory, the resulting filename indicating both the chip type (**JN5139** or **JN5148**) and networking stack (**JN** for Jenie) for which the application was built.

2. Load the resulting binary file into the board. To do this, use the Jennic JN51xx Flash Programmer, described in the *JN51xx Flash Programmer User Guide (JN-UG-3007)*.

4.3.2 Building Code using Eclipse (JN5148 only)

This section provides information on building application code for the JN5148 device using the Eclipse IDE.

Caution: *You must use the version of Eclipse provided by Jennic. This is described in the Eclipse IDE User Guide (JN-UG-3063).*

The build process in Eclipse uses the following files:

- A makefile for each application
- Eclipse project files (**.project** and **.cproject**), where each file covers all the applications in the project

Examples of these files are provided in the Application Note *Jenie Application Templates (JN-AN-1061)*. The project files are located in the top level of the project directory (**JN-AN-1061-Jenie-Application-Template**). The makefile for an application is simply called **Makefile** and is located in the application's **Build** sub-directory - for example, for the Co-ordinator:

...\JN-AN-1061-Jenie-Application-Template\AN1061_JN_Coord\Build

For further instructions on creating and building a project in Eclipse, refer to the *Eclipse IDE User Guide (JN-UG-3063)*.

4.3.3 Building Code using Code::Blocks (JN5139 only)

This section describes how to build application code for the JN5139 device using the Code::Blocks IDE.

***Caution:** You must use the version of Code::Blocks provided by Jennic. This is described in the Jennic Code::Blocks IDE User Guide (JN-UG-3028).*

You will need a Code::Blocks project file for each source file - for example, **5139_JN_Coord.cbp**, **5139_JN_Router.cbp** and **5139_JN_EndD.cbp**. Here, the prefix indicates the target chip.

You can base your project files on the examples supplied by Jennic in the Application Note *Jenie Application Templates (JN-AN-1061)*. The Code::Blocks project files are located in the directory:

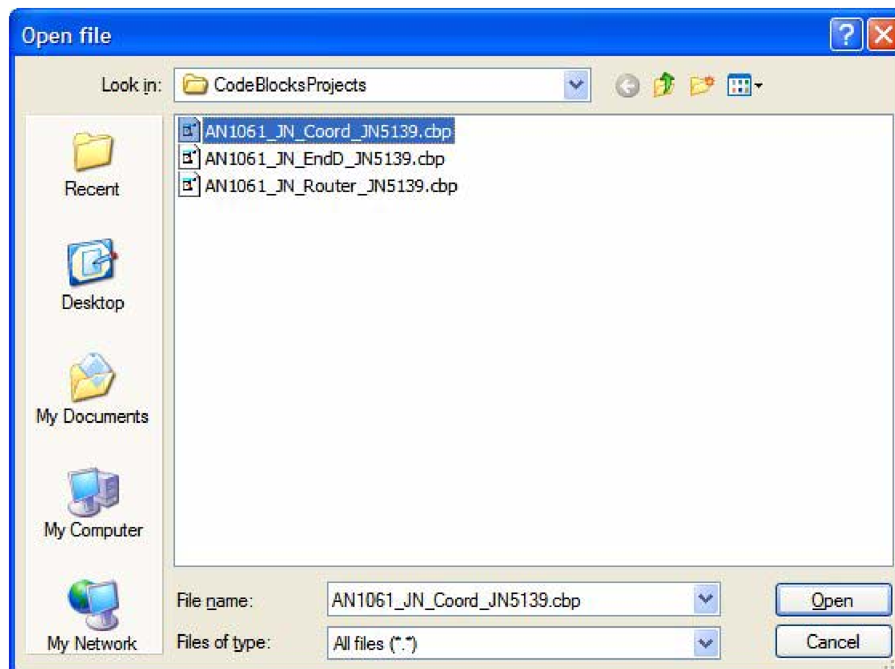
...\JN-AN-1061-Jenie-Application-Template\CodeBlocksProject

A project file is provided for each node type (Co-ordinator, Router, End Device).

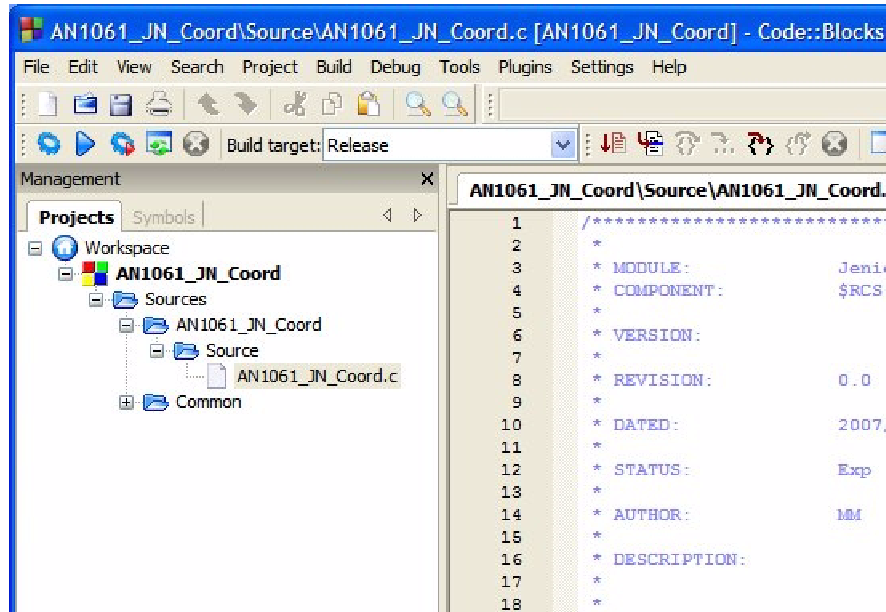
Follow the procedure below (you will need to do this for each application).

Step 1 Start Code::Blocks and open the project to be built by following the menu path **File > Open**.

Step 2 In the **Open file** screen, choose the project file for the application to be built (e.g. **5139_JN_Coord.cbp**).



Step 3 To display the source code, navigate to the required source file under the **Projects** tab of the left pane and double-click on the filename.



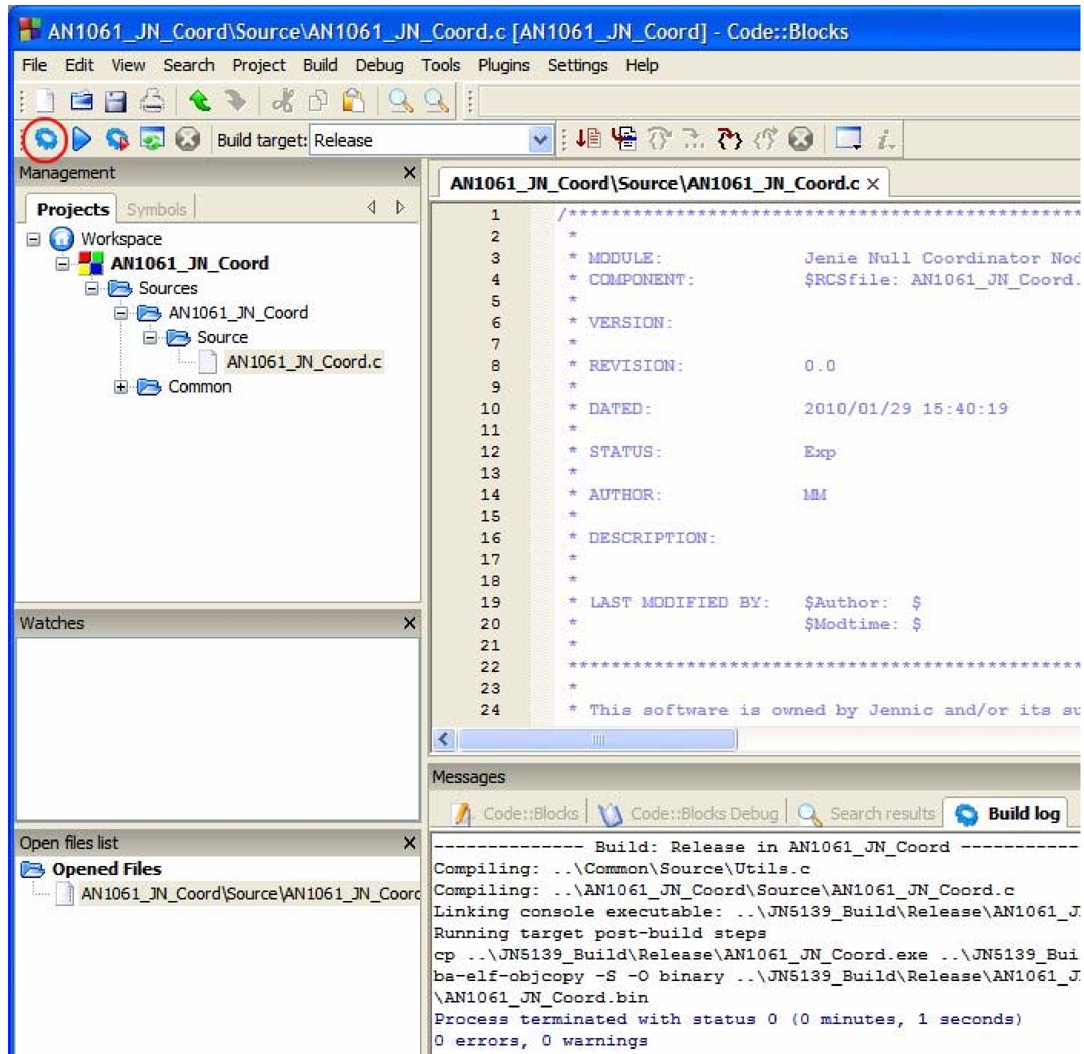
Step 4 Select the required build type, Debug or Release, by following the menu path **Build>Select target**.

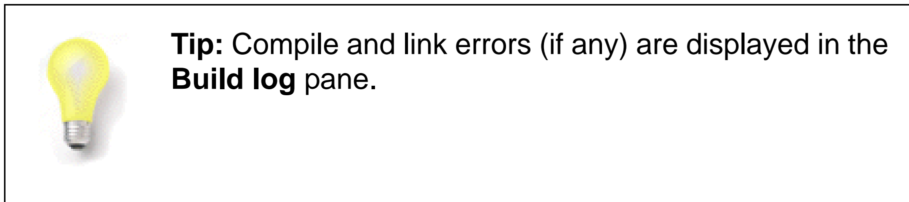
- If you are building for debug purposes, select Debug.
- If you are building for a final release, select Release.

Alternatively, you can make this choice using the **Build target** drop-down list in the Code::Blocks toolbar.

Note: Code that is built with the Debug setting will only run in debug mode on the local PC (and will not run on the JN5139 wireless microcontroller).

Step 5 To compile the application, click on the **Build** button (circled in red below) in the toolbar of Code::Blocks. To see the build log, click on the **Build log** tab in the bottom pane of the Code::Blocks window.





Tip: Compile and link errors (if any) are displayed in the **Build log** pane.

© Jennic 2010

4.4 Downloading Code to Nodes

Once you have built your application for the JN5139/JN5148 microcontroller, there are two possible ways of downloading your binary file to the target device:

- If you are using the Code::Blocks IDE or Eclipse IDE provided by Jennic, you can download your **.bin** file directly from the IDE - refer to the *Jennic Code::Blocks IDE User Guide (JN-UG-3028)* or the *Eclipse IDE User Guide (JN-UG-3063)*, as appropriate.
- Otherwise, you must run the Jennic JN51xx Flash Programmer to download your **.bin** file - refer to the *Jennic JN51xx Flash Programmer Application User Guide (JN-UG-3007)*.

5. Controlling Hardware Peripherals

The Jenie API includes a set of functions that can be used to interface with on-chip peripherals of the JN5139/JN5148 wireless microcontroller - these functions are collectively referred to as the Jenie Peripherals Interface (JPI). The JPI can be used to interact with the following peripherals:

- Analogue resources: ADC, DACs, comparators
- Digital I/O (DIOs)
- UARTs
- Timers
- Wake timers
- Serial Peripheral Interface (SPI)
- 2-wire Serial Interface (SI)
- Intelligent Peripheral (IP) interface

This chapter outlines the use of the JPI to control the ADC, DACs, comparators, DIOs, timers, wake timers, SPI, SI and IP interface, and also provides the essential hardware knowledge needed to use these peripherals. The JPI functions referenced are fully detailed in the *Jenie API Reference Manual (JN-RM-2035)*.

Important: The JPI library is provided for Jennic customers who are maintaining Jenie applications for the JN5139 device or migrating Jenie applications from the JN5139 to the JN5148 device. Any new Jenie application development for the JN5139 or JN5148 device should instead use the Integrated Peripherals API, which includes extra functionality. The functions of this API are provided in the file **AppHardwareApi.h** and are described in the *Integrated Peripherals API Reference Manual (JN-RM-2001)*.

For further information on the JN5139 or JN5148 integrated peripherals, refer to the Jennic data sheet (JN-DS-JN5139 or JN-DS-JN5148) for the relevant device.

5.1 ADC

The JN5139/JN5148 device includes a 12-bit Analogue-to-Digital Converter (ADC) - that is, an ADC which samples an analogue input (such as a temperature measurement) and converts it into a 12-bit digital output.

The ADC has the following features, which are configured/controlled using the Jenie functions indicated:

Note: The functions **vJPI_AnalogueConfigure()** and **vJPI_AnalogueEnable()**, referred to below, are used to configure all the analogue peripherals (ADC and DACs). The function **vJPI_AnalogueConfigure()** is used to configure properties that apply to all the analogue peripherals. The function **vJPI_AnalogueEnable()** can be used to configure any one of the analogue peripherals, the target device (e.g. ADC) being specified as a parameter.

- **Input source:** The ADC can take its input from one of six multiplexed sources, comprising four external pins, an on-chip temperature sensor and an internal voltage monitor. The input source is selected using the function **vJPI_AnalogueEnable()**.
- **Input voltage range:** The permissible range for the analogue input voltage can be defined relative to a reference voltage V_{ref}, which can be sourced internally or externally. The input voltage range can be selected as either 0-V_{ref} or 0-2V_{ref} (an input voltage outside this range results in a saturated digital output). The source of V_{ref} is selected using the function **vJPI_AnalogueConfigure()**. The analogue voltage range is selected using the **vJPI_AnalogueEnable()** function.
- **Clock:** The clock input for the ADC is provided by the 16-MHz on-chip clock, which is divided down. The target frequency is selected using the function **vJPI_AnalogueConfigure()** (this clock output is shared with the DACs). Currently, the only target frequency suitable for the ADC is 500 kHz.
- **Sampling interval:** The sampling interval determines the time over which the ADC will sample the analogue input before performing the conversion - in fact, the sampling occurs over three times this interval (see Figure 12). This interval can be set as a multiple of the ADC clock cycle (2, 4, 6 or 8), where this multiple is selected using the function **vJPI_AnalogueConfigure()**. Normally, it should be set to 2 - for details, refer to the Jennic data sheet for the relevant device.
- **Conversion mode:** The ADC can be configured to perform a single analogue-to-digital conversion (single-shot mode) or repeated conversions (continuous mode). The conversion mode is selected using the function **vJPI_AnalogueEnable()**. In either case, the total time to complete an individual conversion is given by *(3 x sampling interval) + (14 x clock period)*. This is illustrated in Figure 12. In the case of continuous conversion, after this time the next conversion will start, and this will continue indefinitely until stopped using the **vJPI_AnalogueDisable()** function.

- **Interrupt:** Interrupts can be enabled such that an interrupt is generated after each individual conversion. This is particularly useful for ADC continuous conversion. Interrupts can be enabled/disabled using the function **vJPI_AnalogueConfigure()**.
- **Regulator:** In order to minimise the amount of digital noise in the ADC, the device is powered (along with the DACs) from a separate regulator, sourced from the analogue supply VDD1. The regulator, and therefore power, can be enabled/disabled using the function **vJPI_AnalogueConfigure()**.

Once the ADC has been configured using the functions **vJPI_AnalogueConfigure()** and **vJPI_AnalogueEnable()**, you can start conversion using the function **vJPI_AnalogueStartSample()** - the ADC will perform either a single conversion or repeated conversions, depending on its configured mode (single shot or continuous). The 12-bit output of the latest sample can be read from the ADC using the function **u16JPI_AnalogueAdcRead()** (if interrupts are enabled, this read operation may be performed after receiving an interrupt from the ADC). The ADC can be disabled using the function **vJPI_AnalogueDisable()**.

ADC captures analogue input during this time

ADC uses this time to perform the conversion

3 x sampling interval

14 x clock period

Figure 12: ADC Sampling

5.2 DACs

The JN5139/JN5148 device includes two Digital-to-Analogue Converters (DACs), denoted DAC1 and DAC2. These are 11-bit DACs on the JN5139 device and 12-bit DACs on the JN5148 device. Each DAC can take a digital input of up to 11/12 bits and, from it, produce an analogue output as a proportional voltage on a dedicated pin (DAC1 or DAC2, as appropriate).

Note: On the JN5139 device, only one DAC can be used at any one time, since the two DACs share resources. If both DACs are to be used concurrently, they can be multiplexed.

Each DAC has the following features, which are configured/controlled using the Jenie functions indicated.

Note: The functions **vJPI_AnalogueConfigure()** and **vJPI_AnalogueEnable()**, referred to below, are used to configure all the analogue peripherals (ADC and DACs). The function **vJPI_AnalogueConfigure()** is used to configure properties that apply to all the analogue peripherals. The function **vJPI_AnalogueEnable()** can be used to configure any one of the analogue peripherals, the target device (e.g. DAC1) being specified as a parameter.

- **Output voltage range:** The maximum range for the analogue output voltage can be defined relative to a reference voltage V_{ref}, which can be sourced internally or externally. The output voltage range can be selected as either 0-V_{ref} or 0-2V_{ref}. The source of V_{ref} is selected using the function **vJPI_AnalogueConfigure()**. The analogue voltage range is selected using the function **vJPI_AnalogueEnable()**.
- **Clock:** The clock input for the DAC is provided by the 16-MHz on-chip clock, which is divided down. The target frequency is selected using the function **vJPI_AnalogueConfigure()** (this clock output is shared with the other DAC and ADC). Note that to make use of the full 11-bit or 12-bit input resolution of the DAC, the target frequency should be set to 250 kHz.
- **Sampling period:** The sampling period can be set as a multiple of the DAC clock cycle (2, 4, 6 or 8) and determines the conversion period of the DAC - that is, the time delay between submitting a digital value to the DAC and obtaining the converted analogue value on the output pin. The conversion period is given by *(3 x sampling interval) + (14 x clock period)*, as for the ADC. The sampling period multiple is selected using the function **vJPI_AnalogueConfigure()** (this value is shared with the other DAC and ADC).

- **Regulator:** In order to minimise the amount of digital noise in the DAC, the device is powered (along with the other DAC and ADC) from a separate regulator, sourced from the analogue supply VDD1. The separate regulator (and therefore power) can be enabled/disabled using the function **vJPI_AnalogueConfigure()**.

In addition to the above, the function **vJPI_AnalogueEnable()** takes as a parameter the first digital value to be converted - this is a 16-bit value, but only the 11/12 least significant bits are used (all other bits are ignored). Conversion begins as soon as **vJPI_AnalogueEnable()** is called for the DAC. Subsequent digital values for conversion can be submitted to the DAC using the **vJPI_AnalogueDacOutput()** function. The DAC can be disabled using the function **vJPI_AnalogueDisable()**.

5.3 Comparators

The JN5139/JN5148 device includes two comparators, denoted COMP1 and COMP2.

A comparator can be used to compare two analogue inputs - it changes its two-state digital output (high to low or low to high) when the difference between the inputs changes sense (positive to negative or negative to positive). As such, it can be used as a basis for measuring the frequency of a time-varying analogue input when compared with a constant reference input.

Thus, each comparator has two analogue inputs. One analogue input (on the 'positive' pin COMP1P or COMP2P) carries the externally sourced signal that is to be compared with a reference signal. The reference signal can be sourced internally or externally, as follows:

- externally from the 'negative' pin COMP1M or COMP2M
- internally from the analogue output of the corresponding DAC (DAC1 or DAC2)
- internally from the reference voltage V_{ref} (the source of V_{ref} is selected using the function **vJPI_AnalogueConfigure()**)

The reference signal is selected from the above options using the function **vJPI_ComparatorEnable().**

The comparator has two possible states - high or low. The comparator state is determined by the relative values of the two analogue input voltages - that is, by the instantaneous voltages of the signal under analysis, V_{sig}, and the reference signal, V_{refsig}. The relationships are as follows:

$$V_{sig} > V_{refsig} \Rightarrow \text{high}$$

$$V_{sig} < V_{refsig} \Rightarrow \text{low}$$

or in terms of differences:

$$V_{sig} - V_{refsig} > 0 \Rightarrow \text{high}$$

$$V_{sig} - V_{refsig} < 0 \Rightarrow \text{low}$$

Thus, as the signal levels vary with time, when V_{sig} rises above V_{refsig} or falls below V_{refsig}, the state of the comparator result changes. In this way, V_{refsig} is used as the threshold against which V_{sig} is assessed.

© Jennic 2010

In reality, this method of functioning is sensitive to noise in the analogue input signals causing spurious changes in the comparator state. This situation can be improved by using two different thresholds:

- An upper threshold, V_{upper}, for low-to-high transitions
- A lower threshold, V_{lower}, for high-to-low transitions

The thresholds V_{upper} and V_{lower} are defined such that they are above and below the reference signal voltage V_{refsig} by the same amount, where this amount is called the hysteresis voltage, V_{hyst}. That is:

$$V_{upper} = V_{refsig} + V_{hyst}$$

$$V_{lower} = V_{refsig} - V_{hyst}$$

The hysteresis voltage is selected using the **vJPI_ComparatorEnable()** function. It can be set to 0, 5, 10 or 20 mV. The selected hysteresis level should be larger than the noise level in the input signal.

The comparator two-threshold mechanism is illustrated in Figure 13 below for the case when the reference signal voltage V_{refsig} is constant.

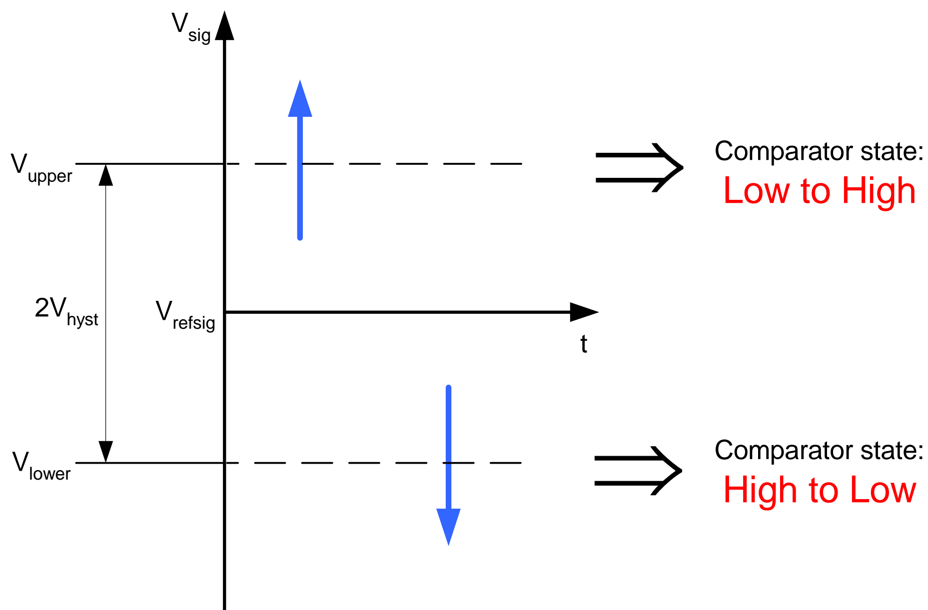


Figure 13: Upper and Lower Thresholds of Comparator

As well as configuring a specified comparator, the **vJPI_ComparatorEnable()** also starts the comparator. The current state of the comparator (high or low) can be obtained at any time using the function **bJPI_ComparatorStatus()**. This returns a boolean value - FALSE corresponds to low and TRUE corresponds to high. The comparator can be stopped using the **vJPI_ComparatorDisable()** function.

The comparators allow an interrupt to be generated on either a low-to-high or high-to-low transition. Interrupts can only be produced on transitions in one direction (and not both directions). The function **vJPI_ComparatorIntEnable()** is used to both enable/disable comparator interrupts and select the direction of the transitions that will trigger the interrupts.

A comparator interrupt can be used as a signal to wake a node from sleep - this is then referred to as a 'wake-up interrupt'. To use this feature, interrupts must be configured and enabled using **vJPI_ComparatorIntEnable()**, as described above. Note that during sleep, the reference signal V_{refsig} is taken from an external source via the 'negative' pin COMP1M or COMP2M. The wake-up interrupt status can be checked using the function **bJPI_ComparatorWakeStatus()**.

5.4 Digital I/O

The JN5139/JN5148 device includes 21 general-purpose digital input/output (DIO) pins, denoted DIO0 to DIO20. Each pin can be individually configured as an input or output. However, the DIO pins are shared with on-chip peripherals and are not available when the corresponding peripherals are enabled. From reset, all peripherals are disabled and the DIOs are configured as inputs.

The DIOs can be individually configured as inputs and outputs using the function **vJPI_DioSetDirection()**. The DIOs configured as outputs can then be individually set to on (high) and off (low) status using the function **vJPI_DioSetOutput()**. The status of all the DIOs configured as inputs can be obtained using the function **u32JPI_DioReadInput()**.

Each DIO has an associated pull-up resistor. The purpose of the 'pull-up' is to prevent the state of the pin from 'floating' when there is no external load connected to the DIO - that is, when enabled, the pull-up ties the pin to the high (on) state in the absence of an external load (or in the presence a weak external load). The pull-ups for all the DIOs can be enabled/disabled using the function **vJPI_DioSetPullup()** - by default, all pull-ups are enabled.

The DIOs can be used to wake the node from sleep. Any DIO pin configured as an input can be used for wake-up - a change of state of the DIO will trigger the wake signal. However, first the wake signal must be enabled on the relevant DIO and the transition (low-to-high or high-to-low) that will trigger the wake signal must also be selected. DIO wake signals are configured and enabled/disabled using the function **vJPI_DioWake()**. The wake status of the DIO pins can subsequently be obtained using the function **u32JPI_DioWakeStatus()**.

 © Jennic 2010

5.5 Timers

The JN5139 device includes 2 general-purpose timers/counters, denoted Timer 0 and Timer 1, and the JN5148 device includes a further timer, denoted Timer 2.

Caution: *The tick timer, also provided by the JN5139/ JN5148 device, is reserved for Jenie use and must not be directly used by your application.*

Each timer requires a source clock, which is selected when the timer is enabled using the function **vJPI_TimerEnable()** - the source clock options are described below.

The timers can be operated in the following modes: Timer, PWM, Delta-Sigma and Capture. These modes are summarised in the table below, along with the functions needed for each mode.

Mode	Description	Functions
Timer	The source clock is used to produce a pulse cycle defined by the number of clock cycles until a positive pulse edge and until a negative pulse edge. Interrupts can be generated on either or both edges. The pulse cycle can be produced just once in 'single-shot' mode or continuously in 'repeat' mode. Timer mode is described further in Section 5.5.1.	**vJPI_TimerEnable()** **vJPI_TimerStart()**
PWM	As for Timer mode, except the Pulse Width Modulated signal is output on a DIO (which depends on the specific timer used - see Table 5 below).	**vJPI_TimerEnable()** **vJPI_TimerStart()**
Delta-Sigma	The timer is used as a low-rate DAC. The converted signal is output on a DIO (which depends on the specific timer used - see Table 5 below) and requires simple filtering to give the analogue signal. Delta-Sigma mode is available in two options, NRZ and RTZ, and is described further in Section 5.5.2.	**vJPI_TimerEnable()** **vJPI_TimerStart()**
Capture	An external input signal is sampled on every tick of the source clock. The results of the capture allow the period and pulse width of the sampled signal to be calculated. Capture mode is described further in Section 5.5.3.	**vJPI_TimerEnable()** **vJPI_TimerStartCapture()** **u32JPI_TimerReadCapture()**

Table 4: Modes of Timer Operation

© Jennic 2010

Before using a timer, the following parameters must be configured for the specified timer using the function **vJPI_TimerEnable()**:

- **Source Clock:** Timer 0 and Timer 1 can be sourced from either an internal or external clock, while Timer 2 (JN5148 only) is always sourced internally. If the internal clock is used, this is the 16-MHz on-chip system clock. If an external clock is used, this must be connected to the DIO8 pin for Timer 0 or the DIO11 pin for Timer 1 (and the DIO pins must be explicitly enabled for use by the timer - see below). In addition to source clock selection, the function **vJPI_TimerEnable()** allows you to specify whether the clock is to be inverted.
- **Clock Divisor:** To obtain the timer frequency, the source clock frequency is divided by a factor $2^{prescale}$, where *prescale* is a user-configurable integer value in the range 0 to 16 (note that the value 0 leaves the clock frequency unchanged). For example, for a *prescale* value of 3, the 16-MHz system clock frequency is divided by 8 to give a timer frequency of 2 MHz.
- **Interrupts:** Each timer can be configured to generate interrupts on either or both of the following conditions:
  - On the rising edge of the timer output (at end of low period)
  - On the falling edge of the timer output (at the end of full timer period)
- **External Output:** The timer signal can be output externally, but this output must be explicitly enabled. For Timer 0, the DIO10 pin is used for this purpose. For Timer 1, the DIO13 pin is used. For Timer 2 (JN5148 only), the DIO11 pin is used. If a DIO pin is to be used by a timer, this use must be enabled (see below).
- **DIO Pins:** The timers can use certain DIO pins. The relevant pins for the two timers are summarised in the table below. Their use must be explicitly enabled.

DIO Pins for Timer 0	DIO Pins for Timer 1	DIO Pins for Timer 2	Function
8	11	-	External clock input
9	12	-	Capture input
10	13	11	PWM and Delta-Sigma output

Table 5: DIO Pins for Timers

Once a timer has been configured using the function **vJPI_TimerEnable()**, it is normally started in the required mode using one of two functions:

- **vJPI_TimerStart()** is used to start the timer in ‘Timer/PWM’ mode or ‘Delta-Sigma’ mode - these modes are described in Section 5.5.1 and Section 5.5.2, respectively.
- **vJPI_TimerStartCapture()** is used to start the timer in ‘Capture’ mode - this mode is described in .Section 5.5.3.

© Jennic 2010

5.5.1 Timer/PWM Mode

Timer mode allows a timer to produce a rectangular waveform of a specified period, where this waveform starts low and then goes high after a specified time. These times are specified when the timer is started using **vJPI_TimerStart()**, in terms of the following parameters:

- **Time to rise (*u16HighPeriod*):** This is the number of clock cycles from timer start before the (first) low-to-high transition. An interrupt can be generated at this transition.
- **Time to fall (*u16LowPeriod*):** This is the number of clock cycles from timer start before the (first) high-to-low transition (effectively the period of one pulse cycle). An interrupt can be generated at this transition.

These times and the timer signal are illustrated below in Figure 14.

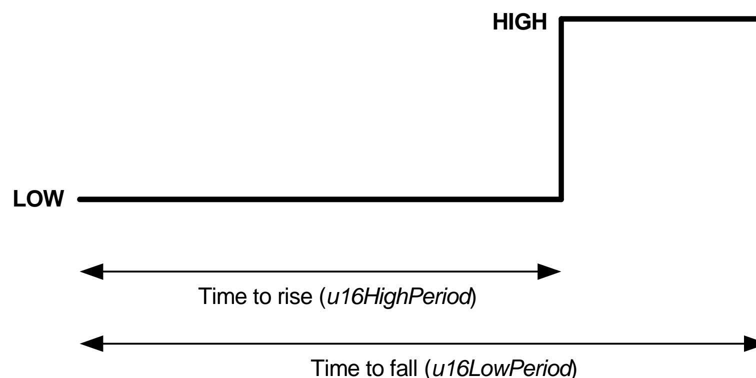


Figure 14: Timer Mode Signal

Within Timer mode, one of two sub-modes must be selected in the function **vJPI_TimerStart()**:

- **Single-shot mode:** The timer produces a single pulse cycle (as depicted in Figure 14) and then stops.
- **Repeat mode:** The timer produces a train of pulses (where the repetition rate is determined by the configured “time to fall” period - see above).

Once started, the timer can be stopped using the function **vJPI_TimerStop()**.

PWM (Pulse Width Modulation) mode is identical to Timer mode except the produced waveform is output on a DIO pin - DIO10 for Timer 0, DIO13 for Timer 1 and DIO11 for Timer 2 (JN5148 only). This output can be enable in the function **vJPI_TimerEnable()**.

5.5.2 Delta-Sigma Mode (NRZ and RTZ)

Delta-Sigma mode allows a timer to be used as a simple low-rate DAC. This requires the timer output to be enabled on the relevant DIO pin (DIO10 pin for Timer 0, DIO13 for Timer 1, DIO11 for Timer 2), and an RC (Resistor-Capacitor) circuit to be inserted between this pin and Ground (see Figure 15 on page 80).

A timer is started in Delta-Sigma mode using the function **vJPI_TimerStart()**. The value to be converted is digitally encoded by the timer as a pseudo-random waveform in which:

- the total number of clock cycles that make up one period of the waveform is fixed (at 2^{16} for NRZ and 2^{17} for RTZ - see below)
- the number of high clock cycles during one period is set to a number which is proportional to the value to be converted
- the high clock cycles are distributed randomly throughout a complete period

Thus, the capacitor will charge in proportion to the specified value such that, at the end of the period, the voltage produced is an analogue representation of the digital value. The voltage obtained on the capacitor depends on the value of the RC constant for the external circuit. This requires calibration - for example, you could determine the maximum possible voltage by measuring the voltage across the capacitor after a conversion with the high period set to the whole pulse period.

Two Delta-Sigma mode options are available, NRZ and RTZ:

- **NRZ (Non Return-to-Zero):** Delta-Sigma NRZ mode uses the 16-MHz system clock and the period of the waveform is fixed at 2^{16} clock cycles. The NRZ option means that clock cycles are implemented without gaps between them (see RTZ option below). You must define the number of clock cycles spent in the high state during the pulse cycle such that this high period is proportional to the value to be converted. This is set in the function **vJPI_TimerStart()**. For example, if you wish to convert values in the range 0-100 then 2^{16} clock cycles would correspond to 100, and to convert the value 25 you must set the number of high clock cycles to 2^{14} (a quarter of the pulse cycle). For an illustration, refer to Figure 15.
- **RTZ (Return-to-Zero):** Delta-Sigma RTZ mode is similar to the NRZ option, described above, except that after every clock cycle it inserts a blank (low) clock cycle. Thus, each pulse cycle takes twice as many clock cycles - that is, 2^{17}. Note that this does not affect the required number of high clock cycles to represent the digital value being converted. This mode doubles the conversion period, but improves linearity if the rise and fall times of the outputs are different from one another.

Note: For more information on 'Delta-Sigma' mode, refer to the data sheet for your Jennic wireless microcontroller. Also, refer to the Application Note *Using JN51xx Timers (JN-AN-1032)*, which includes the selection of the R and C values for the RC circuit.

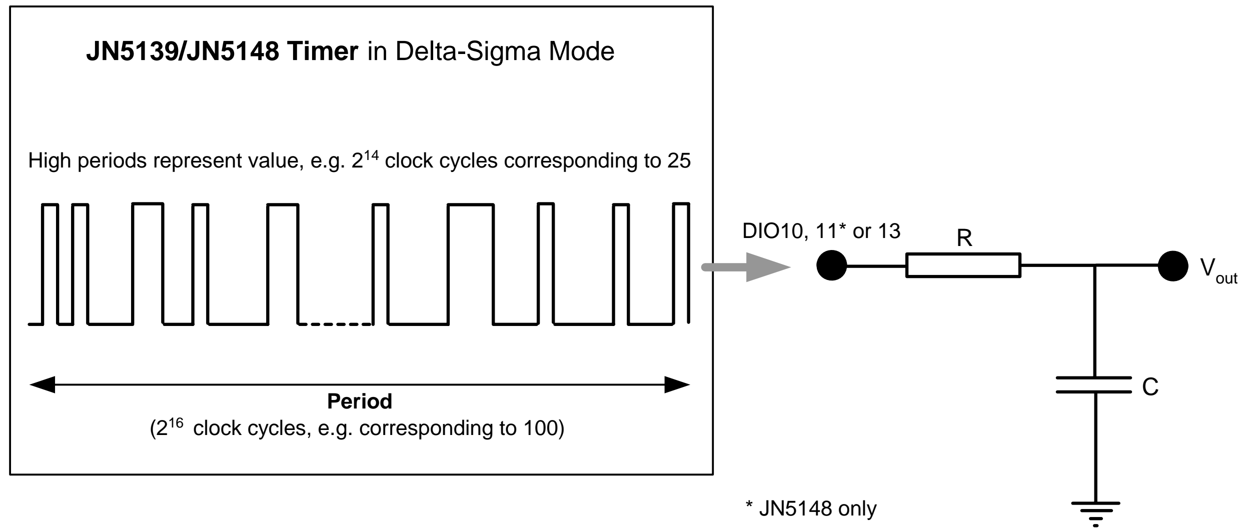


Figure 15: Delta-Sigma NRZ Mode Operation

5.5.3 Capture Mode

In Capture mode, Timer 0 or Timer 1 can be used to measure the pulse width of an external input (Capture mode is not available on Timer 2 of the JN5148 device). The external signal must be provided on the DIO9 pin (Timer 0) or DIO12 pin (Timer 1). The timer measures the number of clock cycles in the input signal from the start of capture to the next low-to-high transition and also to next the high-to-low transition. The number of clock cycles in the last pulse is then the difference between these measured values (see Figure 16). The pulse width in units of time is then given by:

Pulse width (in units of time) = Number of clock cycles in pulse X Clock cycle period

A timer is started in Capture mode using the function **vJPI_TimerStartCapture()**. The timer can be stopped and the most recent measurements obtained using the function **u32JPI_TimerReadCapture()**.

Note: Only the measurements for the last low-to-high and high-to-low transitions are stored and then returned when **u32JPI_TimerReadCapture()** is called. Therefore, it is important not to call this function during a pulse, as in this case the measurements will not give sensible results. To ensure that you obtain the capture results after a pulse has completed, you should enable interrupts on the falling edge when the timer is configured using **vJPI_TimerEnable()**.

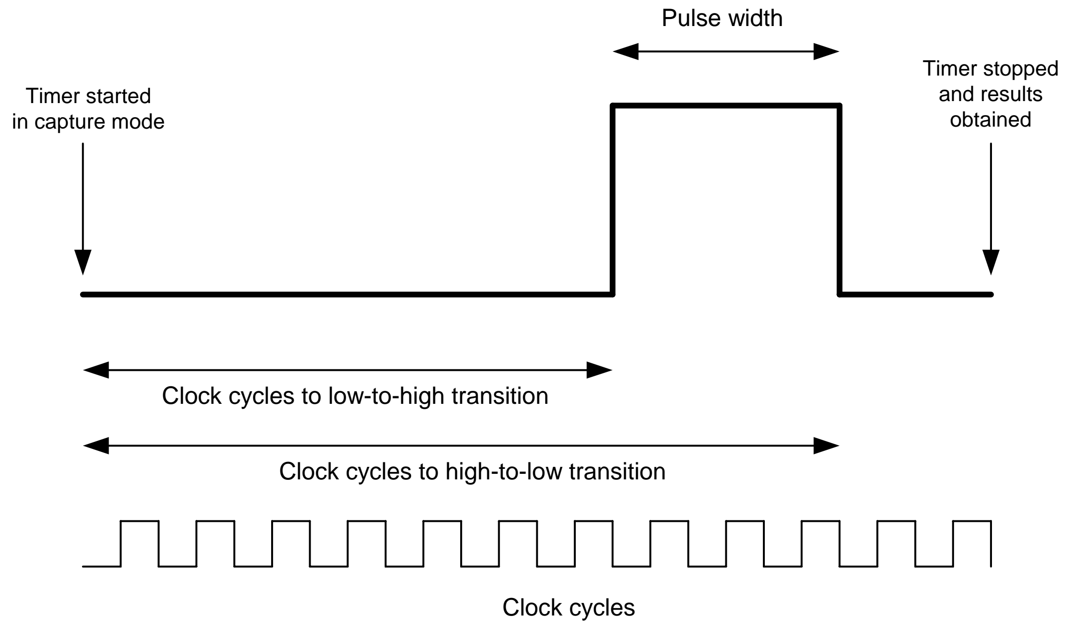


Figure 16: Capture Mode Operation

© Jennic 2010

5.6 Wake Timers

The JN5139/JN5148 device includes two wake timers, denoted Wake Timer 0 and Wake Timer 1. These are 32-bit timers on the JN5139 device and 35-bit timers on the JN5148 device, but the 35-bit timers operate only as 32-bit timers with JPI library. The wake timers can run while the device is in sleep mode (or while the CPU is running), and are generally used to time the sleep duration and wake the device at the end of the sleep period. A wake timer counts down from a programmed value and wakes the device when the count reaches zero by generating an interrupt or wake-up event.

Caution: *The wireless microcontroller can be put to sleep using the function* ***eJenie_Sleep()****, as described in Section 3.9. If this function is called after the function* ***eJenie_SetSleepPeriod()*** *to initiate a timed sleep, Wake Timer 0 is used and the above functions automatically call the Wake Timer functions mentioned below. Therefore, you must finish using Wake Timer 0 for any other purposes in your application code before calling* ***eJenie_SetSleepPeriod()*** *and* ***eJenie_Sleep()****.*

A wake timer is enabled using the functon **vJPI_WakeTimerEnable()**. This function allows you to enable/disable the interrupt that is generated when the counter reaches zero. The timer can then be started using the function **vJPI_WakeTimerStart()**. This function takes as a parameter the starting value for the countdown - this value must be specified in 32-kHz clock periods (thus, 32 corresponds to 1 millisecond). If enabled, the wake timer interrupt is generated on reaching zero.

A wake timer can be stopped at any time using the function **vJPI_WakeTimerStop()** - the counter will then remain at the value at which it was stopped and will not generate an interrupt.

The function **uint32JPI_WakeTimerRead()** can be used to obtain the current value of a wake timer. The function **u8JPI_WakeTimerStatus()** requests which timers are active (note that a timer remains active after it has fired). The function **u8JPI_WakeTimerFiredStatus()** requests which timers have fired (any timers that have fired are cleared as a result of this function call).

The wake timers are driven by the JN5139/JN5148 internal 32-kHz clock. However, this clock may run up to 30% fast or slow, depending on temperature, supply voltage and manufacturing tolerance. For cases in which accurate timing is required, a self-calibration facility is provided to time the 32-kHz clock against the chip's more accurate 16-MHz clock. This test is performed using Wake Timer 0. The result of this calibration allows you to calculate the required number of 32-kHz clock cycles to achieve the desired timer duration when starting a wake timer with the function **vJPI_WakeTimerStart()**. The calibration is performed using the function **u32JPI_WakeTimerCalibrate()**, as described below.

1. Wake Timer 0 must be disabled (using **vJPI_WakeTimerStop()**, if required).
2. Both wake timers (0 and 1) must be cleared by calling the function **u8JPI_WakeTimerFiredStatus()**.
3. The calibration is started using **u32JPI_WakeTimerCalibrate()**.

   This causes Wake Timer 0 to start counting down 20 clock periods of the internal 32-kHz clock. At the same time, a reference counter starts counting up from zero using the 16-MHz clock.
4. When the wake timer reaches zero, **u32JPI_WakeTimerCalibrate()** returns the number of 16-MHz clock cycles registered by the reference counter. Let this value be *n*.
   - If the clock is running at 32 kHz, $n = 10000$
   - If the clock is running slower than 32 kHz, $n > 10000$
   - If the clock is running faster than 32 kHz, $n < 10000$
5. You can then calculate the required number of 32-kHz clock periods (for **vJPI_WakeTimerStart()**) to achieve the desired timer duration. If *T* is the required duration in seconds, the appropriate number of 32-kHz clock periods, *N*, is given by:

$$N = \left(\frac{10000}{n}\right) \times 32000 \times T$$

   For example, if a value of 9000 is obtained for *n*, this means that the 32-kHz clock is running fast. Therefore, to achieve a 2-second timer duration, instead of requiring 64000 clock periods, you will need (10000/9000) x 32000 x 2 clock periods; that is, 71111 (rounded down).

Tip: To ensure that the device wakes in time for a scheduled event, it is better to under-estimate the required number of 32-kHz clock periods than to over-estimate them.

© Jennic 2010

5.7 Serial Peripheral Interface (SPI)

The Serial Peripheral Interface (SPI) on the JN5139/JN5148 wireless microcontroller allows high-speed synchronous data transfers between the JN5139/JN5148 and peripheral devices, without software intervention.

The JN5139/JN5148 device operates as the master on the SPI bus and all other devices connected to the bus are then expected to be slave devices under the control of the master's CPU. The SPI supports up to five slave devices, one of which is Flash memory, by default. If enabled, the additional slave-select lines use DIO0-DIO3.

Data transfer is full-duplex, so data is transmitted by both communicating devices at the same time. Data to be transmitted is stored in a FIFO buffer in the device. The data transaction size can be 8, 16 or 32 bits, and the data transfer order can be configured as LSB (least significant bit) or MSB (most significant bit) first.

Since the data transfer is synchronous, both transmitting and receiving devices use the same clock, provided by the SPI master. The SPI uses the 16-MHz clock, which may be divided down to allow bit rates from 250 kbps to 16 Mbps.

The clock edge on which data is latched is determined by the SPI mode of operation used (0, 1, 2 or 3), which is determined by two boolean parameters, clock polarity and phase, as indicated in the table below.

SPI Mode	Polarity	Phase	Description
0	0	0	Data latched on rising edge of clock
1	0	1	Data latched on falling edge of clock
2	1	0	Clock inverted and data latched on falling edge of clock
3	1	1	Clock inverted and data latched on rising edge of clock

Table 6: SPI Modes of Operation

An interrupt can be enabled, which is generated when the data transfer completes.

Before transferring data, the SPI master must select the slave(s) with which it wishes to communicate. Thus, the relevant slave-select line(s) must be asserted. It is usual for the SPI master to communicate with a single slave at a time, so not to receive data from multiple slaves simultaneously (unless the slave devices can be inhibited from transmitting data). An "Automatic Slave Selection" feature is provided, which only asserts the chosen slave-select line(s) during a data transfer.

© Jennic 2010

An SPI data transfer is performed using the JPI functions as follows:

1. The SPI master must first be configured using the function **vJPI_SpiConfigure()**. This function allows the configuration of:
   - Number of extra SPI slaves (in addition to Flash memory)
   - Clock divisor (for 16-MHz clock)
   - Data transfer order (LSB first or MSB first)
   - Clock polarity (unchanged or inverted)
   - Phase (latch data on leading edge or trailing edge of clock)
   - Automatic Slave Selection
   - SPI interrupts

   If SPI interrupts are enabled, a corresponding callback must be registered using the function **vJPI_SpiRegisterCallback()**.
2. The SPI slaves must be selected using the function **vJPI_SpiSelect()**. If "Automatic Slave Selection" is off, the relevant slave-select line(s) will be asserted immediately, otherwise the line(s) will only be asserted during a subsequent data transfer.
3. A data transfer is implemented using one of the following functions, depending on the transaction size:
   - **vJPI_SpiStartTransfer8()** for 8-bit data
   - **vJPI_SpiStartTransfer16()** for 16-bit data
   - **vJPI_SpiStartTransfer32()** for 32-bit data
4. The transfer is allowed to complete by waiting for a SPI interrupt (if enabled) to indicate completion, or by calling **vJPI_SpiWaitBusy()** which returns when the transfer has completed, or by periodically calling **bJPI_SpiPollBusy()** to check whether the SPI master is still busy.
5. Data received from a slave is read using one of the following functions, depending on the transaction size:
   - **u8JPI_SpiReadTransfer8()** for 8-bit data
   - **u16JPI_SpiReadTransfer16()** for 16-bit data
   - **u32JPI_SpiReadTransfer32()** for 32-bit data
6. If another transfer is required then Steps 3 to 5 must be repeated for the next data. Otherwise, if "Automatic Slave Selection" is off, the SPI slaves must be de-selected by calling **vJPI_SpiSelect(0)** or **vJPI_SpiStop()**.

A number of other SPI functions exist in the Jenie Peripherals Interface. The current SPI configuration can be obtained and saved using **vJPI_SpiReadConfiguration()**. If necessary, this saved configuration can later be restored in the SPI using the function **vJPI_SpiRestoreConfiguration()**.

5.8 Serial Interface (2-wire)

The JN5139/JN5148 device includes an industry-standard 2-wire synchronous Serial Interface (SI) that provides a simple and efficient method of data exchange between devices.

The Serial Interface is similar to an I^2C interface. It comprises two lines:

- Serial data line
- Serial clock line

The Serial Interface peripheral on the JN5139/JN5148 device acts as a master of the Serial Interface bus and can implement bi-directional communication with a slave device on the bus. Note that the Serial Interface bus on the JN5148 device can have more than one master, but multiple masters cannot use the bus at the same time (to avoid this, an arbitration scheme is provided - see later).

As a bus master, the JN5139/JN5148 provides the clock (on the clock line) for synchronous data transfers (on the data line) - this clock is scaled from the 16-MHz system clock.

The master can write data to or read data from a slave device. The protocol used is outlined below:

1. The master takes control of the Serial Interface bus by issuing a start bit (provided that no other master has control of the bus).
2. The master specifies the 7-bit address of the slave to communicate with.
3. The master indicates whether it intends to perform a read or write transaction with the slave.
4. If the slave with the specified address exists on the bus, the slave responds to the master with an ACK (aknowledgment).
5. Provided that the slave exists, the master continues to perform the required data transfer operation:
   - For a write operation, the master sends a series of data bytes to the slave, which responds to each byte with an ACK.
   - For a read operation, the master receives a series of data bytes from the slave, with the master responding to each byte (except the final one) with an ACK.
6. Once the transfer is complete, the master relinquishes control of the bus with a stop bit. Alternatively, it can issue another start condition and begin another data transfer from Step 1.

On the JN5148 device, an arbitration scheme exists to resolve conflicts caused by competing masters attempting to take control of the Serial Interface bus. If a master loses arbitration, it must wait and try again later.

Note: In order to implement data transfers on the SI bus, you are advised to study the protocol detailed in the I^2C Specification (available from www.nxp.com).

5.9 Intelligent Peripheral (IP) Interface

The Intelligent Peripheral (IP) interface is used for high-speed data exchanges between the JN5139/JN5148 device and a 'remote' processor, which may be a separate processor contained in the wireless network node. The data exchange requires minimal use of the CPU of this processor. This interface is based on the Serial Peripheral Interface (SPI) - see Section 5.7. The IP interface on the JN5139/JN5148 is an SPI slave - the remote processor must contain the SPI master (which initiates data transfers). The interface shares pins with DIO14-18.

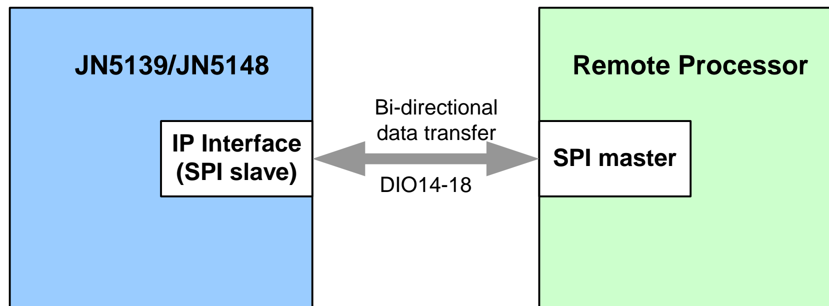


Figure 17: IP Interface as SPI Slave

The data transfer is bi-directional. The JN5139/JN5148 device uses a Transmit buffer and Receive buffer in a dedicated block of local memory for the data exchanges - each buffer in this IP memory block contains sixty-three 32-bit words. The IP_INT line is used by the JN5139/JN5148 device to indicate that it is ready to transmit data (held in a local Transmit buffer) to the remote processor. As the master device, the remote processor must initiate the transfer. Data can be transmitted and received simultaneously, but the clock edges on which receive data is sampled and transmit data is changed are separately configurable on the JN5139/JN5148 device.

Before using the IP interface, it must first be enabled using the function **vJPI_IpEnable()**. This function allows the transmit and receive clock edges to be selected, and the byte order for the data transfer to be set as Big Endian or Little Endian. The function **bJPI_IpSendData()** is used to indicate to the remote processor (via the IP_INT line) that the JN5139/JN5148 device is ready to transmit and receive data. It is then the responsibility of the remote processor, as the master, to initiate the data transfer. Two functions are provided to check from the JN5139/JN5148 side whether a data transfer has completed:

- **bJPI_IpTxDone()** can be used to check whether all data has been transmitted
- **bJPI_IpRxDataAvailable()** can be used to check whether data has been received

Received data can be read from the local Receive buffer using the function **bJPI_IpReadData()**.

 © Jennic 2010

6. Advanced Issues in Network Operation

This chapter deals with a range of Jenie network features and issues that are not covered in the description of basic network operation in Chapter 3. 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 Jenie global variables. These global variables can be set in the function **vJenie_CbConfigureNetwork()**, and are fully listed and described in the *Jenie API Reference Manual (JN-RM-2035)*.

6.1 Identifying the Network

As described in Section 1.4.1, Jenie uses two identifiers to distinguish a network from other Jenie networks operating in the same space - the Network Application ID and PAN ID. Two global variables 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 1.4.1 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” in on neighbouring networks - if the specified PAN ID is found elsewhere, the value of this global variable will be automatically adjusted until a unique value is obtained. In this respect, it does not matter which value you assign to this global variable (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 variable 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 Jenie 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 *TxOptions* flag `TXOPTION_SILENT` cleared, Jenie 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 4 tries in total.

Depending on the outcome of the send, Jenie 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: For **eJenie_SendData()**, if the *TxOptions* flag `TXOPTION_SILENT` or `TXOPTION_BDCAST` (broadcast) is set, the above stack events will not be generated.

© Jennic 2010

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 *TxOptions* flag `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.

© Jennic 2010

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 variable *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 1.11.1). The Neighbour table is small, since a node can have an absolute maximum of only 16 children. The Routing table, however, can potentially accommodate entries for a very large number of descendant nodes and therefore take up significant memory space. For this reason, the application is allowed some control over the Routing table, in order to limit the amount of memory space occupied by the table.

The Routing table is represented in memory as an array of structures, where each structure 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 variables:

- *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.

© Jennic 2010

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.

Jennic suggest that for very large networks, which may be long and thin with regular traffic, purging should be 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 route purging functions are provided:

- **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.

© Jennic 2010

6.3.3 Automatic Route Importation

Jenie/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 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.

 © Jennic 2010

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). In Jenie, this number is determined by the global variable *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, Jenie 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 1.12.3). If there is no other traffic on the link:

- A Router will periodically ping its parent at an interval determined by the global variable *gJenie_RouterPingPeriod* (in units of 100 ms).
- An End Device will periodically ping its parent at an interval determined by the global variable *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 variable *gJenie_MaxFailedPkts* (which is set to 5, by default). In this case, Jenie 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 variable *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), Jenie 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. Jenie 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, Jenie sends an E_JENIE_NETWORK_UP event to the application. Otherwise, the device goes to sleep for a period determined by the global variable *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.*

 © Jennic 2010

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 variable *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 variable *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 variable *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

© Jennic 2010

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 variable *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 variable *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 variable *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 variable 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 variable *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 variable.

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 variable *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 3.7.3.

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 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 Jenie, 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 variable *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 for 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 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 variable *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 and Caching

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.

In addition, in order to minimise the traffic from End Devices joining the network, the join results can be cached in their parent Routers. The Co-ordinator should then request this End Device data from the Routers by sending messages using the function **eJenie_SendData()**. This 'polling' method reduces the amount of unsolicited traffic in the network, with the data from all the immediate children of a Router being sent to the Co-ordinator in one message. This approach is particularly useful to avoid buffer timeouts for End Devices with long sleep durations.

'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, frantic 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 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. Jenie/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 Jenie/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. 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 variables to 0:

- *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 3.10). 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.

 © Jennic 2010

6.10 Key Performance Parameters

This section describes how certain key network parameters affect the performance of the network. The full set of Jenie/JenNet network parameters are listed and described in the *Jenie API Reference Manual (JN-RM-2035)*.

6.10.1 Broadcast TTL (Time To Live)

gJenie_MaxBcastTTL

The broadcast TTL (Time To Live) is represented by the global variable *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 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 variable *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 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 variable *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 variable *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 variable *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.

© Jennic 2010

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 variable *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 variable *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 variable setting but does not replace it.

Appendices

A. Hardware and Memory Usage

This appendix details the JN5139/JN5148 hardware and memory required by Jenie.

A.1 Hardware Resources

The JN5139/JN5148 hardware required by the Jenie/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 stack event.

A.2 Memory Resources

This section details the memory resources required by the Jenie/JenNet stack on the Jennic wireless microcontrollers. JN5139 and JN5148 memory resources are covered separately below.

JN5139 Memory Resources

From the 96 KB of RAM on the JN5139 wireless microcontroller, the exact memory resources required by the Jenie/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	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 1: JN5139 Memory Required by Jenie/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.

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

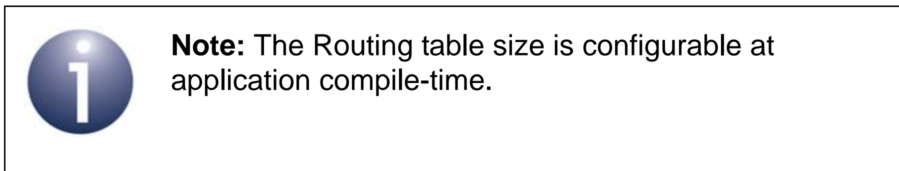
JN5148 Memory Resources

From the 128 KB of RAM on the JN5148 wireless microcontroller, the exact memory resources required by the Jenie/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	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 2: JN5148 Memory Required by Jenie/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.



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

B. Glossary

Term	Description
Address	A numeric value that is used to identify a network node. In Jenie, 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.
AT-Jenie	An ASCII- or binary-based serial command set which provides a high-level control interface 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 Jennic software stack. Among other functionality, it provides the physical interface to the network's transmission medium (radio).
Jenie	Jennic's proprietary easy-to-use interface between the application and the JenNet network level of the Jennic software stack. Available in the form of an API or a serial command set (AT-Jenie).
JenNet	Jennic's proprietary network protocol which sits on IEEE 802.15.4 in the Jennic software stack. An application interacts with JenNet through the Jenie interface.
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.

© Jennic 2010

Term	Description
Network Application ID	A 32-bit value that identifies the network application (e.g. a product). It is used in Jenie 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.
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 Jenie 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).

© Jennic 2010

Revision History

Version	Date	Comments
1.0	04-Dec-2007	First release
1.1	21-Feb-2008	Added chapter on controlling hardware peripherals and appendix on global variables
1.2	07-Mar-2008	Updated for Jenie v1.2
1.3	01-Apr-2008	Minor updates
1.4	09-July-2008	Updated for Jenie v1.3
1.5	24-Sep-2008	Updated with minor corrections
1.6	01-Dec-2008	Updated for Jenie v1.4. More information added in Appendix A.5.1 on buffering messages for End Device children. Added Appendix B on hardware and memory usage
1.7	27-Aug-2009	Minor updates/corrections made
1.8	17-Mar-2010	Modified for JN5148, added chapter on advanced issues and various other updates/corrections 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.

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