Jennic

TECHNOLOGY FOR A CHANGING WORLD

Jenie API Reference Manual

JN-RM-2035

Revision 1.8

17 March 2010

Contents

About this Manual **9**
Organisation 9
Conventions 10
Acronyms and Abbreviations 10
Related Documents 11
Feedback Address 11

1. Jenie Overview **13**
1.1 Core Functionality 13
1.2 Hardware Functionality 14

2. Jenie Functions **15**
2.1 "Application to Stack" Functions 16
2.1.1 Network Management Functions 16
eJenie_Start 17
eJenie_Leave 18
eJenie_RegisterServices 19
eJenie_RequestServices 20
eJenie_BindService 21
eJenie_UnBindService 22
eJenie_SetPermitJoin 23
bJenie_GetPermitJoin 24
eJenie_SetSecurityKey 25
2.1.2 Data Transfer Functions 26
eJenie_SendData 27
eJenie_SendDataToBoundService 29
eJenie_PollParent 30
2.1.3 System Functions 31
vJPDM_SaveContext 32
eJPDM_RestoreContext 33
vJPDM_EraseAllContext 34
eJenie_SetSleepPeriod 35
eJenie_Sleep 36
eJenie_RadioPower 38
u32Jenie_GetVersion 39
2.1.4 Statistics Functions 40
u16Jenie_GetRoutingTableSize 41
eJenie_GetRoutingTableEntry 42
u8Jenie_GetNeighbourTableSize 43
eJenie_GetNeighbourTableEntry 44

eJenie_ResetNeighbourStats 45

2.2 “Stack to Application” Functions 46

vJenie_CbConfigureNetwork 47

vJenie_CbInit 48

vJenie_CbMain 49

vJenie_CbStackMgmtEvent 50

vJenie_CbStackDataEvent 51

vJenie_CbHwEvent 52

3. Jenie Peripherals Interface (JPI) 53

3.1 General 54

u32JPI_Init 55

u8JPI_PowerStatus 56

vJPI_SwReset 57

vJPI_DriveResetOut 58

vJPI_HighPowerModuleEnable 59

vJPI_SysCtrlRegisterCallback 60

3.2 Analogue Peripherals 61

vJPI_AnalogueConfigure 62

vJPI_AnalogueEnable 63

vJPI_AnalogueDisable 64

vJPI_APRegisterCallback 65

bJPI_APRegulatorEnabled 66

vJPI_AnalogueStartSample 67

u16JPI_AnalogueAdcRead 68

bJPI_AdcPoll 69

vJPI_AnalogueDacOutput 70

bJPI_DacPoll 71

vJPI_ComparatorEnable 72

vJPI_ComparatorDisable 73

bJPI_ComparatorStatus 74

vJPI_ComparatorIntEnable 75

bJPI_ComparatorWakeStatus 76

3.3 Digital I/O 77

vJPI_DioSetDirection 78

vJPI_DioSetOutput 79

vJPI_DioSetPullup 80

u32JPI_DioReadInput 81

vJPI_DioWake 82

u32JPI_DioWakeStatus 84

3.4 UARTs 85

vJPI_UartEnable 87

vJPI_UartDisable 88

vJPI_UartSetClockDivisor 89

vJPI_UartSetBaudDivisor 90

vJPI_UartSetControl 91
vJPI_UartSetInterrupt 92
vJPI_UartSetRTSCTS 93
vJPI_UartReset 94
u8JPI_UartReadLineStatus 95
u8JPI_UartReadModemStatus 96
u8JPI_UartReadInterruptStatus 97
vJPI_UartWriteData 98
u8JPI_UartReadData 99
vJPI_Uart0RegisterCallback 100
vJPI_Uart1RegisterCallback 101
3.5 Timers 102
vJPI_TimerEnable 104
vJPI_TimerDisable 106
vJPI_TimerStart 107
vJPI_TimerStop 109
vJPI_TimerStartCapture 110
u32JPI_TimerReadCapture 111
u8JPI_TimerFired 112
vJPI_Timer0RegisterCallback 113
vJPI_Timer1RegisterCallback 114
vJPI_Timer2RegisterCallback (JN5148 Only) 115
3.6 Wake Timers 116
vJPI_WakeTimerEnable 118
vJPI_WakeTimerStart 119
vJPI_WakeTimerStop 120
u32JPI_WakeTimerRead 121
u8JPI_WakeTimerStatus 122
u8JPI_WakeTimerFiredStatus 123
u32JPI_WakeTimerCalibrate 124
3.7 Serial Peripheral Interface (SPI) 125
vJPI_SpiConfigure 126
vJPI_SpiReadConfiguration 128
vJPI_SpiRestoreConfiguration 129
vJPI_SpiSelect 130
vJPI_SpiStop 131
vJPI_SpiStartTransfer32 132
u32JPI_SpiReadTransfer32 133
vJPI_SpiStartTransfer16 134
u16JPI_SpiReadTransfer16 135
vJPI_SpiStartTransfer8 136
u8JPI_SpiReadTransfer8 137
bJPI_SpiPollBusy 138
vJPI_SpiWaitBusy 139
vJPI_SpiRegisterCallback 140

3.8 Serial Interface (2-Wire) 141

vJPI_SiConfigure 142

vJPI_SiSetCmdReg 143

vJPI_SiWriteData8 145

vJPI_SiWriteSlaveAddr 146

u8JPI_SiReadData8 147

bJPI_SiPollBusy 148

bJPI_SiPollTransferInProgress 149

bJPI_SiPollRxNack 150

bJPI_SiPollArbitrationLost 151

vJPI_SiRegisterCallback 152

3.9 Intelligent Peripheral Interface 153

vJPI_IpEnable 154

bJPI_IpSendData 155

bJPI_IpReadData 156

bJPI_IpTxDone 157

bJPI_IpRxDataAvailable 158

vJPI_IpRegisterCallback 159

Appendices 161

A. Global Network Parameters 161

B. Enumerated Types and Defines 164

B.1 For Core Jenie Functions 164

B.2 For Jenie Peripheral Interface Functions 166

C. Data Types 178

D. Stack Events 180

D.1 Stack Management Events 180

D.2 Data Events 182

E. Integrated Peripheral Interrupt Handling 184

E.1 Callback Function Prototype and Parameters 184

E.2 Handling Wake Interrupts 188

F. JenNet API 191

F.1 JenNet Functions 192

eApi_SendDataToExtNwk 193

vNwk_DeleteChild 194

vApi_SetScanSleep 195

vApi_SetBcastTTL 196

vApi_SetPurgeRoute 197

vApi_SetPurgeInterval 198

vNwk_SetBeaconCalming 199

vApi_SetUserBeaconBits 200

u16Api_GetUserBeaconBits 201

u8Api_GetLastPktLqi 202
u16Api_GetDepth 203
u8Api_GetStackState 204
u32Api_GetVersion 205
vApi_RegBeaconNotifyCallback 206
vApi_RegLocalAuthoriseCallback 207
vApi_RegNwkAuthoriseCallback 208
vApi_RegScanSortCallback 209

F.2 JenNet Network Parameters 210

G. Link Quality Indication (LQI) 214

About this Manual

This manual provides key reference information for developers using the Jenie Application Programming Interface (API) to produce wireless network applications for the Jennic JN5139 and JN5148 wireless microcontrollers. The manual provides detailed descriptions of functions of the Jenie API intended for programmers with a knowledge of C.

Tip: You should use this Reference Manual in conjunction with the *Jenie API User Guide (JN-UG-3042)*, which provides both relevant concept information and practical guidance on using Jenie to develop wireless network applications.

Tip: Jenie is also available in the form of the AT-Jenie serial command set, which provides a simpler alternative to the Jenie API. AT-Jenie is currently available only for the JN5139 device, and is described in separate user documentation - the *AT-Jenie User Guide (JN-UG-3043)* and the *AT-Jenie Reference Manual (JN-RM-2038)*.

Organisation

This manual consists of 3 chapters and 7 appendices, as follows:

- Chapter 1 introduces the Jenie API.
- Chapter 2 details the main functions of the Jenie API.
- Chapter 3 details the Jenie Peripherals Interface (JPI) functions of the API.
- The Appendices provide ancillary information needed to use the Jenie API: global network parameters, enumerated types, data types, stack events, hardware interrupts and LQI values. In addition, an appendix is provided which describes functions and network parameters of the JenNet layer, which sits below Jenie in the stack.

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]	Jenie Application Templates Application Note (JN-AN-1061)
[2]	Jenie Tutorial Application Note (JN-AN-1085)
[3]	Jenie API User Guide (JN-UG-3042)
[4]	JenNet Stack User Guide (JN-UG-3041)
[5]	IEEE 802.15.4 Wireless Networks User Guide (JN-UG-3024)
[6]	Board API Reference Manual (JN-RM-2003)
[7]	Application Queue API Reference Manual (JN-RM-2025)

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. Jenie Overview

Jennic's proprietary Jenie software provides an easy-to-use interface for developing wireless network applications for the Jennic JN5139 and JN5148 wireless microcontrollers. The Jenie Application Programming Interface (API) comprises C functions for controlling the wireless network and the on-chip hardware peripherals of the JN5139/JN5148 device. This functionality is outlined below.

Note: For a more complete introduction to Jenie, refer to the *Jenie API User Guide (JN-UG-3042)*.

1.1 Core Functionality

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

- **Management tasks:**
  - Configure and initialise network
  - Start a device as a Co-ordinator, Router or End Device
  - Determine whether a Router or Co-ordinator is accepting join requests
  - Advertise local node services and seek remote node services
  - Establish bindings between local and remote node services
  - Handle stack management events
- **Data transfer tasks:**
  - 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:**
  - Configure and start sleep mode
  - Configure, start and stop the radio transmitter
  - Obtain the version number of a component on the node
  - Handle hardware events

In addition to the functions of the Jenie API, functions of the JenNet layer (which sits below Jenie in the stack) are described in this manual in Appendix F. The JenNet functions are intended for advanced users who require more control over the network than is available through Jenie.

1.2 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

The part of the Jenie API concerned with the above peripherals is referred to as the Jenie Peripherals Interface (JPI).

Note 1: 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 is described in the *Integrated Peripherals API Reference Manual (JN-RM-2001)*.

Note 2: Jennic also provide separate software for controlling hardware resources on the JN5139/JN5148 carrier boards. This software is described in the *Board API Reference Manual (JN-RM-2003)*.

2. Jenie Functions

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

- “Application to Stack” functions, described in Section 2.1.
- “Stack to Application” functions, described in Section 2.2.

In addition, each of the above categories is sub-divided into functions that deal with management tasks, data transfer tasks and system tasks (see Chapter 1 for an overview of these three areas).

Note: In addition to the functions of the Jenie API, functions of the JenNet layer (which sits below Jenie in the stack) are described in this manual in Appendix F. The JenNet functions are intended for advanced users who require more control over the network than is available through Jenie.

2.1 “Application to Stack” Functions

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

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

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

2.1.1 Network Management Functions

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

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

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

Function	Page
eJenie_Start	17
eJenie_Leave	18
eJenie_RegisterServices	19
eJenie_RequestServices	20
eJenie_BindService	21
eJenie_UnBindService	22
eJenie_SetPermitJoin	23
bJenie_GetPermitJoin	24
eJenie_SetSecurityKey	25

eJenie_Start

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

Description

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

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

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

Parameters

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

Returns

One of:

E_JENIE_SUCCESS

E_JENIE_ERR_INVLD_PARAM

For explanations, refer to Appendix B.

eJenie_Leave

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

Description

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

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

Parameters

None

Returns

One of:

E_JENIE_SUCCESS

E_JENIE_ERR_UNKNOWN

E_JENIE_ERR_STACK_BUSY

For explanations, refer to Appendix B.

eJenie_RegisterServices

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

Description

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

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

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

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

Parameters

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

Returns

One of:

E_JENIE_SUCCESS

E_JENIE_DEFERRED

E_JENIE_ERR_UNKNOWN

E_JENIE_ERR_STACK_BUSY

For explanations, refer to Appendix B.

eJenie_RequestServices

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

Description

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

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

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

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

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

Parameters

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

Returns

One of:

E_JENIE_ERR_INVLD_PARAM
E_JENIE_DEFERRED
E_JENIE_ERR_UNKNOWN
E_JENIE_ERR_STACK_BUSY

For explanations, refer to Appendix B.

eJenie_BindService

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

Description

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

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

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

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

Parameters

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

Returns

One of:

E_JENIE_SUCCESS
E_JENIE_ERR_INVLD_PARAM
E_JENIE_ERR_STACK_RSRC

For explanations, refer to Appendix B.

eJenie_UnBindService

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

Description

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

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

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

Parameters

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

Returns

One of:

E_JENIE_SUCCESS

E_JENIE_ERR_INVLD_PARAM

E_JENIE_ERR_STACK_RSRC

For explanations, refer to Appendix B.

eJenie_SetPermitJoin

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

Description

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

Parameters

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

Returns

E_JENIE_SUCCESS

bJenie_GetPermitJoin

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

Description

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

Parameters

None

Returns

One of:

TRUE - joinings allowed

FALSE - joinings forbidden

eJenie_SetSecurityKey

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

Description

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

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

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

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

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

Parameters

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

Returns

One of:

E_JENIE_SUCCESS

E_JENIE_ERR_INVLD_PARAM

For explanations, refer to Appendix B.

2.1.2 Data Transfer Functions

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

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

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

Function	**Page**
eJenie_SendData	27
eJenie_SendDataToBoundService	29
eJenie_PollParent	30

eJenie_SendData

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

Description

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

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

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

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

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

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

Parameters

u64DestAddr	Address of the destination node. For a broadcast or to send data to the Co-ordinator, this parameter must be set to zero (for a broadcast, *u8TxFlags* must also be set appropriately)
***pu8Payload*	Pointer to the data to be sent
u16Length	Length of data to be sent, in bytes (for limits, see above)
u8TxFlags	Sets the transmission options. These options are detailed in Table 2 on page 165. The values can be logical ORed to simultaneously specify more than one option

Returns

One of:

E_JENIE_ERR_INVLD_PARAM

E_JENIE_DEFERRED

E_JENIE_ERR_UNKNOWN

E_JENIE_ERR_STACK_BUSY

For explanations, refer to Appendix B.

eJenie_SendDataToBoundService

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

Description

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

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

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

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

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

Parameters

u8Service	Service ID of local service from which data is to be sent
***pu8Payload*	Pointer to the data to be sent
u16Length	Length of data to be sent, in bytes (for limits, see above)
u8TxFlags	Sets the transmission options. These options are detailed in Table 2 on page 165. The values can be logical ORed to simultaneously specify more than one option

Returns

One of:

E_JENIE_ERR_INVLD_PARAM
E_JENIE_DEFERRED
E_JENIE_ERR_UNKNOWN
E_JENIE_ERR_STACK_BUSY

For explanations, refer to Appendix B.

eJenie_PollParent

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

Description

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

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

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

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

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

Parameters

None

Returns

One of:

- E_JENIE_DEFERRED
- E_JENIE_ERR_UNKNOWN
- E_JENIE_ERR_STACK_BUSY

For explanations, refer to Appendix B.

2.1.3 System Functions

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

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

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

Function	Page
vJPDM_SaveContext	32
eJPDM_RestoreContext	33
vJPDM_EraseAllContext	34
eJenie_SetSleepPeriod	35
eJenie_Sleep	36
eJenie_RadioPower	38
u32Jenie_GetVersion	39

vJPDM_SaveContext

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

Description

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

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

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

Note that for a Router, child tables and routing tables will not be saved.

Parameters

None

Returns

None

eJPDM_RestoreContext

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

Description

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

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

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

The above macro is defined as follows:

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

where:

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

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

size is the number of bytes in the buffer

Parameters

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

Returns

One of:

E_JENIE_SUCCESS

E_JENIE_ERR_INVLD_PARAM

For explanations, refer to Appendix B.

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

vJPDM_EraseAllContext

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

Description

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

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

Parameters

None

Returns

None

eJenie_SetSleepPeriod

teJenieStatusCode eJenie_SetSleepPeriod(
uint32 *u32SleepPeriodMs***);**

Description

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

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

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

Parameters

u32SleepPeriodMs	Sleep duration, in milliseconds

Returns

E_JENIE_SUCCESS

eJenie_Sleep

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

Description

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

The following sleep options can be specified:

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

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

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

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

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

Note that if a wake timer (see Section 3.6) is used to wake the device from "sleep with memory held", no event is generated via the **vJenie_CbHwEvent()** function or the registered system controller callback function (although a wake-up initiated by a DIO or comparator will generate a hardware event).

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

Parameters

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

Returns

E_JENIE_SUCCESS

E_JENIE_ERR_UNKNOWN

eJenie_RadioPower

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

Description

This function can be used to set the transmit power level of the radio transceiver, or to switch the radio transceiver on or off. The transmit power level can be set to a value in the range -30 to +18 dBm, in steps of 6 dBm, with the following restrictions depending on the module type:

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

In addition to the above values, 20 and 21 are used to switch the radio transmitter on and off, respectively (enumerations are available to do this). With the exception of these two special values, if the specified power level is not a multiple of 6, the power level set by this function is the nearest multiple of 6 dBm within the valid range.

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

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

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

An 'invalid parameter' error will be returned if an out-of-range power level is specified.

Parameters

iPowerLevel	Power level as multiple of 6 in the range -30 to +18, or one of: E_JENIE_RADIO_OFF - switch radio transceiver off E_JENIE_RADIO_ON - switch radio transceiver on
bHighPower	Enables high-power mode for a Jennic high-power module: TRUE - high-power mode enabled FALSE - high-power mode disabled

Returns

One of:

E_JENIE_SUCCESS

E_JENIE_ERR_INVLD_PARAM

For explanations, refer to Appendix B.

u32Jenie_GetVersion

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

Description

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

Parameters

eComponent	Component for which version number is needed, one of: E_JENIE_COMPONENT_JENIE (Jenie) E_JENIE_COMPONENT_NETWORK (JenNet) E_JENIE_COMPONENT_MAC (IEEE 802.15.4) E_JENIE_COMPONENT_CHIP (JN513x chip)

Returns

Version number of component, as described below:

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

2.1.4 Statistics Functions

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

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

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

Function	Page
u16Jenie_GetRoutingTableSize	41
eJenie_GetRoutingTableEntry	42
u8Jenie_GetNeighbourTableSize	43
eJenie_GetNeighbourTableEntry	44
eJenie_ResetNeighbourStats	45

u16Jenie_GetRoutingTableSize

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

Description

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

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

Parameters

None

Returns

Number of valid entries in the local Routing table

eJenie_GetRoutingTableEntry

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

Description

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

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

Parameters

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

Returns

One of:

E_JENIE_SUCCESS

E_JENIE_ERR_INVLD_PARAM

For explanations, refer to Appendix B.

u8Jenie_GetNeighbourTableSize

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

Description

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

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

Parameters

None

Returns

Number of entries in the Neighbour table

eJenie_GetNeighbourTableEntry

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

Description

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

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

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

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

Parameters

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

Returns

One of:

E_JENIE_SUCCESS

E_JENIE_ERR_INVLD_PARAM

For explanations, refer to Appendix B.

eJenie_ResetNeighbourStats

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

Description

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

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

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

Parameters

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

Returns

One of:

E_JENIE_SUCCESS
E_JENIE_ERR_INVLD_PARAM

For explanations, refer to Appendix B.

2.2 “Stack to Application” Functions

This section details the “Stack to Application” functions of the Jenie API. These are callback functions triggered by events from the underlying stack. They provide the opportunity for the application software to receive information and respond at defined points during program execution, such as at stack initialisation, or at regular intervals. You must define these functions in your application code, even those functions that are not used in your code (and are therefore empty).

The callback functions handle:

- stack management events
- data events
- hardware events

Stack management and data events are described in Appendix D. Hardware events are described in Appendix E.

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

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

Function	**Page**
vJenie_CbConfigureNetwork	47
vJenie_CbInit	48
vJenie_CbMain	49
vJenie_CbStackMgmtEvent	50
vJenie_CbStackDataEvent	51
vJenie_CbHwEvent	52

vJenie_CbConfigureNetwork

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

Description

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

Parameters

None

Returns

None

vJenie_CbInit

void vJenie_CbInit(bool_t *bWarmStart***);**

Description

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

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

Parameters

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

Returns

None

vJenie_CbMain

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

Description

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

Parameters

None

Returns

None

vJenie_CbStackMgmtEvent

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

Description

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

For further details of the stack management events, refer to Appendix D.1.

Parameters

eEventType	The type of stack management event received, one of: E_JENIE_REG_SVC_RSP E_JENIE_SVC_REQ_RSP E_JENIE_POLL_CMPLT E_JENIE_PACKET_SENT E_JENIE_PACKET_FAILED E_JENIE_NETWORK_UP E_JENIE_STACK_RESET E_JENIE_CHILD_JOINED E_JENIE_CHILD_LEAVE E_JENIE_CHILD_REJECTED
**pvEventPrim*	Pointer to event primitive (if relevant, or NULL if not)

Returns

None

vJenie_CbStackDataEvent

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

Description

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

For further details of the data events, refer to Appendix D.2.

Parameters

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

Returns

None

vJenie_CbHwEvent

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

Description

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

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

Parameters

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

Returns

None

3. Jenie Peripherals Interface (JPI)

This chapter details the Jenie Peripherals Interface (JPI), part of the Jenie API. These functions are used to interface with the on-chip peripherals of the Jennic JN5139/JN5148 wireless microcontroller. The functions are defined in the header file **JPI.h**.

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)*.

Peripheral control using these functions covers:

- General - see Section 3.1
- Analogue resources (ADC, DACs, comparators) - see Section 3.2
- Digital I/O (DIOs) - see Section 3.3
- UARTs - see Section 3.4
- Timers - see Section 3.5
- Wake timers - see Section 3.6
- Serial Peripheral Interface (SPI) - see Section 3.7
- 2-Wire Serial Interface (SI) master - see Section 3.8
- Intelligent Peripheral (IP) interface - see Section 3.9

Before using the JPI functions, you are advised to consult the following documentation for information on the JN5139/JN5148 integrated peripherals:

- Jenie API User Guide (JN-UG-3042)
- JN513x or JN5148 Data Sheet (JN-DS-JN513x or JN-DS-JN5148)

***Caution:** The Jenie 'Application to Stack' functions included in Chapter 2 must not be called from interrupt context. Therefore, these functions must not be called from within a user-defined callback function registered through the JPI. Instead, the application should set a flag to indicate that the call should be made later, outside of interrupt context.*

3.1 General

This section details the general functions that are not specific to an individual JN5139/JN5148 integrated peripheral.

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

Function	Page
u32JPI_Init	55
u8JPI_PowerStatus	56
vJPI_SwReset	57
vJPI_DriveResetOut	58
vJPI_HighPowerModuleEnable	59
vJPI_SysCtrlRegisterCallback	60

u32JPI_Init

```
uint32 u32JPI_Init(void);
```

Description

This function initialises the Jenie Peripherals Interface (JPI). The function should be called after every reset or wake-up, and before any other JPI functions are called.

Parameters

None

Returns

0 if initialisation failed, otherwise a 32-bit version number (most significant 16 bits are main revision, least significant 16 bits are minor revision)

u8JPI_PowerStatus

```
uint8 u8JPI_PowerStatus(void);
```

Description

This function obtains various settings for the JN5139/JN5148 device (see below).

Parameters

None

Returns

Returns the status information in bits 0-3 of the 8-bit return value:

Bit	Reads a '1' if...
0	Device has completed a sleep-wake cycle
1	RAM contents were retained during sleep
2	Analogue power domain is switched on
3	Protocol logic is operational - clock is enabled
4-7	Unused

vJPI_SwReset

```
void vJPI_SwReset (void);
```

Description

This function generates an internal reset, which completely re-starts the system.

***Caution:** Calling this function has the same effect as momentarily pulling the external RESETN line low. When RESETN is low, on-chip RAM is not powered. Therefore, as a result of this function call, data stored in RAM may be corrupted or lost.*

Parameters

None

Returns

None

vJPI_DriveResetOut

void vJPI_DriveResetOut(uint8 *u8Period***);**

Description

This function drives the ResetN line low for the specified time.

Note that one or more external devices may be connected to the ResetN line. Therefore, using this function to drive this line low may affect these external devices. For more information on the ResetN line and external devices, consult the datasheet for your wireless microcontroller.

Parameters

u8Period	Duration for which line will be driven low, in milliseconds

Returns

None

vJPI_HighPowerModuleEnable

void vJPI_HighPowerModuleEnable(bool_t *bRFTXEn*,
bool_t *bRFRXEn***);**

Description

This function allows the transmitter and receiver sections of a Jennic high-power module to be enabled or disabled. The transmitter and receiver sections must both be enabled or disabled at the same time (enabling only one of them is not supported). The function must be called before using the radio transceiver on a high-power module.

The function sets the CCA (Clear Channel Assessment) threshold to suit the gain of the attached Jennic high-power module.

Note that this function cannot be used with a high-power module from a manufacturer other than Jennic.

Parameters

bRFTXEn	Enable/disable setting for high-power module transmitter (must be same setting as for *bRFRXEn*): TRUE - enable transmitter FALSE - disable transmitter
bRFRXEn	Enable/disable setting for high-power module receiver (must be same setting as for *bRFTXEn*): TRUE - enable receiver FALSE - disable receiver

Returns

None

vJPI_SysCtrlRegisterCallback

```
void vJPI_SysCtrlRegisterCallback(
            PR_HWINT_APPCALLBACK prSysCtrlCallback);
```

Description

This function registers an application callback that will be called when the System Controller interrupt is triggered.

A System Controller interrupt can be caused by wake timer, comparator and DIO events. Note that the System Controller interrupt handler will clear the interrupt before invoking the callback function to deal with the interrupt. Also note that when a DIO or comparator event wakes the device from sleep with memory held, the registered callback function will be called before any other Jenie callback function.

The registered callback function is only preserved during sleep with memory held. If RAM is powered off during sleep and interrupts are required, the callback function must be re-registered before calling **u32JPI_Init()** on waking.

Interrupt handling is described in Appendix E.

Parameters

prSysCtrlCallback	Pointer to function to be called when a System Controller interrupt occurs

Returns

None

3.2 Analogue Peripherals

This section details the functions for controlling the analogue peripherals - ADC, DACs and comparators. The JN5139/JN5148 device has:

- One 12-bit ADC (Analogue-to-Digital Converter)
- Two DACs (Digital to Analogue Converters) - DAC1 and DAC2 - these are 11-bit DACs on the JN5139 device and 12-bit DACs on the JN5148 device
- Two comparators - COMP1 and COMP2

A comparator can be programmed to provide an interrupt when the difference between its inputs changes sense, and can also be used to wake the chip from sleep. The inputs to the comparator use dedicated pins on the chip.

Note 1: The analogue peripheral regulator must be enabled when configuring a comparator, but can be disabled once configuration is complete.

Note 2: If a comparator is to be used to wake the device from sleep mode then the DAC output option cannot be used, since the analogue power domain is turned off when the device enters sleep mode.

Note 3: 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.

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

Function	Page
vJPI_AnalogueConfigure	62
vJPI_AnalogueEnable	63
vJPI_AnalogueDisable	64
vJPI_APRegisterCallback	65
bJPI_APRegulatorEnabled	66
vJPI_AnalogueStartSample	67
u16JPI_AnalogueAdcRead	68
bJPI_AdcPoll	69
vJPI_AnalogueDacOutput	70
bJPI_DacPoll	71
vJPI_ComparatorEnable	72
vJPI_ComparatorDisable	73
bJPI_ComparatorStatus	74
vJPI_ComparatorIntEnable	75
bJPI_ComparatorWakeStatus	76

vJPI_AnalogueConfigure

```
void vJPI_AnalogueConfigure(bool_t bAPRegulator,
                            bool_t bIntEnable,
                            uint8 u8SampleSelect,
                            uint8 u8ClockDivRatio,
                            bool_t bRefSelect);
```

Description

This function configures common parameters for all on-chip analogue resources.

- The analogue peripheral regulator can be enabled - this dedicated power source minimises digital noise and is sourced from the analogue supply pin VDD1.
- Interrupts can be enabled that are generated after each ADC conversion (Analogue Peripherals interrupts are handled by a callback function registered using **vJPII_APRegisterCallback()**)
- The clock frequency (derived from the chip's 16-MHz clock) is specified.
- The 'sampling interval' is specified as a number of clock periods.
- The source of the reference voltage, V_{ref}, is specified.

For the ADC, the input signal is integrated over *3 x Sampling Interval*. For the ADC and DACs, the total conversion period (for a single value) is given by

(3 x sampling interval) + *(14 x clock period)*

Parameters

bAPRegulator	Enable/disable analogue peripheral regulator: E_JPI_AP_REGULATOR_ENABLE E_JPI_AP_REGULATOR_DISABLE
bIntEnable	Enable/disable interrupt when conversion/capture completes: E_JPI_AP_INT_ENABLE E_JPI_AP_INT_DISABLE
u8SampleSelect	Select sampling interval in terms of divided clock periods: E_JPI_AP_SAMPLE_2 (2 clock periods) E_JPI_AP_SAMPLE_4 (4 clock periods) E_JPI_AP_SAMPLE_6 (6 clock periods) E_JPI_AP_SAMPLE_8 (8 clock periods)
u8ClockDivRatio	Clock divide ratio: E_JPI_AP_CLOCKDIV_2MHZ (achieves 2 MHz) E_JPI_AP_CLOCKDIV_1MHZ (achieves 1 MHz) E_JPI_AP_CLOCKDIV_500KHZ (achieves 500 kHz) E_JPI_AP_CLOCKDIV_250KHZ (achieves 250 kHz) (500 kHz is recommended for ADC and 250 kHz for DACs)
bRefSelect	Select source of reference voltage, V_{ref}: E_JPI_AP_EXTREF (external from VREF pin) E_JPI_AP_INTREF (internal)

Returns

None

vJPI_AnalogueEnable

```
void vJPI_AnalogueEnable(
        teJPI_AnalogueChannel eChan,
        bool_t bInputRange,
        bool_t bContinuous,
        uint8 u8Source,
        bool_t bOutputHold,
        uint16 u16InitValue);
```

Description

This function configures and enables the specified analogue device (ADC or DAC). A parameter is relevant to both ADC and DACs unless otherwise stated. Note that:

- The source of V_{ref} is defined using **vJPI_AnalogueConfigure()**.
- For the ADC, the internal voltage monitor measures the voltage on the pin VDD1.
- The ADC can be configured to operate in single-shot mode or continuous mode.
- Only one of the DACs can be enabled at any one time - if both DACs are to be used concurrently, they can be multiplexed.
- For a DAC, the first value to be converted is specified through this function - the conversion will be started immediately after the function call. Subsequent values must be specified through **vJPI_AnalogueDacOutput()**.

Parameters

eChan	Analogue device to configure and enable: E_JPI_ANALOGUE_DAC_0 (DAC1) E_JPI_ANALOGUE_DAC_1 (DAC2) E_JPI_ANALOGUE_ADC (ADC)
bInputRange	Set input voltage range to either 0-V_{ref} or 0-2V_{ref}: E_JPI_AP_INPUT_RANGE_2 (0-2V_{ref}) E_JPI_AP_INPUT_RANGE_1 (0-V_{ref})
bContinuous	Enable/disable continuous conversion for ADC: E_JPI_ADC_CONTINUOUS (continuous mode) E_JPI_ADC_SINGLE_SHOT (single shot mode)
u8Source	Source for ADC conversions: E_JPI_ADC_SRC_ADC_1 (ADC1 input pin) E_JPI_ADC_SRC_ADC_2 (ADC2 input pin) E_JPI_ADC_SRC_ADC_3 (ADC3 input pin) E_JPI_ADC_SRC_ADC_4 (ADC4 input pin) E_JPI_ADC_SRC_TEMP (on-chip temperature sensor) E_JPI_ADC_SRC_VOLT (internal voltage monitor)
bOutputHold	Unused - set to 0 (FALSE)
u16InitValue	Initial digital value for DAC to convert (only lower 11/12 bits used)

Returns

None

vJPI_AnalogueDisable

void vJPI_AnalogueDisable(teJPI_AnalogueChannel *eChan***);**

Description

This function disables the specified analogue device (ADC or DAC).

Note that only one of the two DACs can be used at any one time. If both DACs are to be used, you must alternate between them - use **vJPI_AnalogueEnable()** to enable a DAC and then **vJPI_AnalogueDisable()** to disable the DAC before enabling the next one using **vJPI_AnalogueEnable()**, and so on.

Parameters

eChan	Analogue device to disable: E_JPI_ANALOGUE_DAC_0 (DAC1) E_JPI_ANALOGUE_DAC_1 (DAC2) E_JPI_ANALOGUE_ADC (ADC)

Returns

None

vJPI_APRegisterCallback

**void vJPI_APRegisterCallback(
PR_HWINT_APPCALLBACK** *prApCallback***);**

Description

This function registers an application callback that will be called when the Analogue Peripherals interrupt is triggered.

Note: Among the analogue peripherals, only the ADC generates Analogue Peripheral interrupts. The DACs do not generate interrupts and the comparators generate System Controller interrupts (see Section 3.1, page 60).

The registered callback function is only preserved during sleep with memory held. If RAM is powered off during sleep and interrupts are required, the callback function must be re-registered before calling **u32JPI_Init()** on waking.

Interrupt handling is described in Appendix E.

Parameters

prApCallback	Pointer to function to be called when the Analogue Peripherals interrupt occurs

Returns

None

bJPI_APRegulatorEnabled

```
bool_t bJPI_APRegulatorEnabled(void);
```

Description

This function enquires whether the analogue peripheral regulator has powered up. The function should be called after enabling the regulator through **vJPI_AnalogueConfigure()**. When the regulator is enabled, it will take a little time to start up - this period is 31.25 µs for the JN5139/JN5148 device.

Parameters

None

Returns

TRUE if powered up, FALSE otherwise

vJPI_AnalogueStartSample

```
void vJPI_AnalogueStartSample(void);
```

Description

This function starts the ADC sampling in single-shot or continuous mode, depending on which mode has been configured using **vJPI_AnalogueEnable()**:

- **Single-shot mode:** ADC will perform a single conversion and then stop (only valid if DACs are not enabled).
- **Continuous mode:** ADC will perform conversions repeatedly until stopped using the function **vJPI_AnalogueDisable()**.

If analogue peripheral interrupts have been enabled in **vJPI_AnalogueConfigure()**, an interrupt will be triggered when a result becomes available. Alternatively, if interrupts are disabled, you can use **bJPI_AdcPoll()** to check for a result. Once a conversion result becomes available, it should be read with **u16JPI_AnalogueAdcRead()**.

Parameters

None

Returns

None

u16JPI_AnalogueAdcRead

```
uint16 u16JPI_AnalogueAdcRead(void);
```

Description

This function reads the most recent ADC conversion result. The value is 12 bits wide.

If analogue peripheral interrupts have been enabled in **vJPI_AnalogueConfigure()**, you must call this read function from a callback function invoked when an interrupt has been generated to indicate that an ADC result is ready (this user-defined callback function is registered using the function **vJPI_APRegisterCallback()**). Alternatively, if interrupts have not been enabled, before calling the read function, you must first check whether a result is ready using the function **bJPI_AdcPoll()**.

Parameters

None

Returns

Most recent ADC conversion result (the result is contained in the least significant 12 bits of the 16-bit returned value)

bJPI_AdcPoll

```
bool_t bJPI_AdcPoll(void);
```

Description

This function can be used when the ADC is operating in single-shot mode or continuous mode, to check whether the ADC is still busy performing a conversion:

- In single-shot mode, the poll result indicates whether the sample has been taken and is ready to be read.
- In continuous mode, the poll result indicates whether a new sample is ready to be read.

You may wish to call this function before attempting to read the conversion result using **u16JPI_AnalogueAdcRead()**, particularly if you are not using the analogue peripheral interrupts.

Parameters

None

Returns

TRUE if ADC is busy, FALSE if conversion complete

vJPI_AnalogueDacOutput

void vJPI_AnalogueDacOutput(uint8 *u8Dac***,**
uint16 *u16Value***);**

Description

This function allows the next value for conversion by the specified DAC to be set. This value will be used for all subsequent conversions until the function is called again with a new value.Although a 16-bit value must be specified in this function:

- For the JN5148 device, only the 12 least significant bits will be used, since the chip features 12-bit DACs
- For the JN5139 device, only the 11 least significant bits will be used, since the chip features 11-bit DACs

Note: The first value to convert using the DAC is specified when the DAC is enabled through **vJPI_AnalogueEnable()**.

Parameters

u8Dac	Identity of DAC: E_JPI_ANALOGUE_DAC_0 (DAC1) E_JPI_ANALOGUE_DAC_1 (DAC2)
u16Value	Value to convert to analogue - only the 11 or 12 least significant bits will be used (see above)

Returns

None

bJPI_DacPoll

```
bool_t bJPI_DacPoll (void);
```

Description

This function checks whether the enabled DAC is busy performing a conversion.

A short delay (approximately 2 µs) after polling and checking the DAC is included to prevent lock-ups when further calls are made to the DAC.

Parameters

None

Returns

TRUE if DAC is busy, FALSE if conversion complete

vJPI_ComparatorEnable

```
void vJPI_ComparatorEnable(
                teJPI_Comparator eComparator,
                uint8 u8Hysteresis,
                uint8 u8SignalSelect);
```

Description

This function configures and enables the specified comparator. The reference signal and hysteresis setting must be specified.

The hysteresis voltage selected should be greater than:

- the noise level in the input signal on the comparator '+' pin (COMP1P or COMP2P), if comparing the signal on this pin with the internal reference voltage or DAC output
- the differential noise between the signals on the comparator '+' and '-' pins, if comparing the signals on these two pins

Note that the same hysteresis setting is used for both comparators, so if this function is called several times for different comparators, only the hysteresis value from the final call will be used.

Once enabled using this function, the comparator can be disabled using the function **vJPI_ComparatorDisable()**.

Parameters

eComparator	Identity of comparator: E_JPI_COMPARATOR_1 E_JPI_COMPARATOR_2
u8Hysteresis	Hysteresis setting (controllable from 0 to ±20mV): **JN5139:** E_JPI_COMP_HYSTERESIS_0MV (0 mV) E_JPI_COMP_HYSTERESIS_5MV (±5 mV) E_JPI_COMP_HYSTERESIS_10MV (±10 mV) E_JPI_COMP_HYSTERESIS_20MV (±20 mV) **JN5148:** E_JPI_COMP_HYSTERESIS_0MV (0 mV) E_JPI_COMP_HYSTERESIS_10MV (±5 mV) E_JPI_COMP_HYSTERESIS_20MV (±10 mV) E_JPI_COMP_HYSTERESIS_40MV (±20 mV)
u8SignalSelect	Reference signal to compare with input signal on comparator '+' pin: E_JPI_COMP_SEL_EXT (Comparator '-' pin) E_JPI_COMP_SEL_DAC (Related DAC output) E_JPI_COMP_SEL_BANDGAP (Fixed at V_{ref})

Returns

None

vJPI_ComparatorDisable

```
void vJPI_ComparatorDisable(
                    teJPI_Comparator eComparator);
```

Description

This function disables the specified comparator.

Parameters

eComparator	Identity of comparator: E_JPI_COMPARATOR_1 E_JPI_COMPARATOR_2

Returns

None

bJPI_ComparatorStatus

```
bool_t bJPI_ComparatorStatus(
                teJPI_Comparator eComparator);
```

Description

This function returns the status of the specified comparator.

Parameters

eComparator	Identity of comparator: E_JPI_COMPARATOR_1 E_JPI_COMPARATOR_2

Returns

- FALSE if the input signal is lower than reference signal
- TRUE if the input signal is higher than reference signal

vJPI_ComparatorIntEnable

```
void vJPI_ComparatorIntEnable(
                    teJPI_Comparator eComparator,
                    bool_t bIntEnable,
                    bool_t bRisingNotFalling);
```

Description

This function enables the comparator interrupt for the specified comparator. The interrupt can be used to wake the device from sleep, and as a normal interrupt.

If enabled, an interrupt is generated on one of the following conditions (which must be configured):

- The input signal rises above the reference signal (plus hysteresis level, if non-zero)
- The input signal falls below the reference signal (minus hysteresis level, if non-zero)

Comparator interrupts are handled by the System Controller callback function, registered using the function **vJPI_SysCtrlRegisterCallback()**.

Parameters

eComparator	Identity of comparator: E_JPI_COMPARATOR_1 E_JPI_COMPARATOR_2
bIntEnable	TRUE to enable interrupt FALSE to disable interrupt
bRisingNotFalling	TRUE - interrupt when input signal rises above reference FALSE - interrupt when input signal falls below reference

Returns

None

bJPI_ComparatorWakeStatus

```
uint8 u8JPI_ComparatorWakeStatus(
                    teJPI_Comparator eComparator);
```

Description

This function returns the wake-up interrupt status of the specified comparator. The value is cleared after reading.

Note: You cannot use this function to check whether a comparator was responsible for a recent wake-up event. You should determine the wake source as part of your System Controller callback function, as described in Appendix E.2.

Parameters

eComparator	Identity of comparator: E_JPI_COMPARATOR_1 E_JPI_COMPARATOR_2

Returns

- 0 if wake-up interrupt has not occurred
- Non-zero value if wake-up interrupt has occurred

3.3 Digital I/O

This section details the functions for controlling the digital I/O (DIO) pins. The JN5139/ JN5148 device includes 21 DIO pins, denoted DIO0 to DIO20. Each pin can be individually configured as an input or output. When configured as an input, a DIO can be used to generate interrupts and to wake the device from sleep. However, the pins for the DIO lines are shared with other peripherals (see list below) and are not available when those peripherals are enabled:

- UARTs
- Timers
- Serial Interface (2-wire)
- Serial Peripheral Interface (SPI)
- Intelligent Peripheral (IP) interface

For details of the shared pins, refer to the datasheet for your wireless microcontroller.

In addition to normal operation, when configured as inputs, the DIOs can be used to generate interrupts and wake the device from sleep.

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

Function	**Page**
vJPI_DioSetDirection	78
vJPI_DioSetOutput	79
vJPI_DioSetPullup	80
u32JPI_DioReadInput	81
vJPI_DioWake	82
u32JPI_DioWakeStatus	84

vJPI_DioSetDirection

void vJPI_DioSetDirection(uint32 *u32Inputs***,**
uint32 *u32Outputs***);**

Description

This function sets the direction for the DIO pins individually as either input or output. This is done through two bitmaps for inputs and outputs, *u32Inputs* and *u32Outputs* respectively. In these values, each of bits 0 to 20 represents a DIO pin, where bit 0 represents DIO0 and bit 20 represents DIO20 (bits 21-31 are ignored).

Note that:

- Not all DIO pins must be defined (in other words, *u32Inputs* logical ANDed with *u32Outputs* does not need to produce all zeros for bits 0-20).
- Any DIO pins that are not defined by a call to this function (the relevant bits being cleared in both bitmaps) will be left in their previous states.
- If a bit is set in both *u32Inputs* and *u32Outputs*, it will default to becoming an input.
- If a DIO is assigned to another peripheral which is enabled, this function call will not immediately affect the relevant pin. However, the DIO setting specified by this function will take effect if/when the relevant peripheral is subsequently disabled.
- This function does not change the DIO pull-up status - this must be done separately using **vJPI_DioSetPullup()**.

Parameters

u32Inputs	Bitmap of inputs - a bit set means that the corresponding DIO pin will become an input
u32Outputs	Bitmap of outputs - a bit set means that the corresponding DIO pin will become an output

Returns

None

vJPI_DioSetOutput

void vJPI_DioSetOutput(uint32 *u32On*, **uint32** *u32Off***);**

Description

This function sets individual DIO outputs on or off. This is done through two bitmaps for on-pins and off-pins, *u32On* and *u32Off* respectively. In these values, each of bits 0 to 20 represents a DIO pin, where bit 0 represents DIO0 and bit 20 represents DIO20 (bits 21-31 are ignored). Setting a bit in one of these bitmaps configures the corresponding DIO output as on or off, depending on the bitmap.

Note that:

- Not all DIO pins must be defined (in other words, *u32On* logical ANDed with *u32Off* does not need to produce all zeros for bits 0-20).
- Any DIO pins that are not defined by a call to this function (the relevant bits being cleared in both bitmaps) will be left in their previous states.
- If a bit is set in both *u32On* and *u32Off*, the DIO pin will default to off.
- This call has no effect on DIO pins that are not defined as outputs (see **vJPI_DioSetDirection()**) until a time when they are re-configured as outputs.
- If a DIO is assigned to another peripheral which is enabled, this function call will not affect the relevant DIO, until a time when the relevant peripheral is disabled.

Parameters

u32On	Bitmap of on-pins - a bit set means that the corresponding DIO pin will be set to on
u32Off	Bitmap of off-pins - a bit set means that the corresponding DIO pin will be set to off

Returns

None

vJPI_DioSetPullup

void vJPI_DioSetPullup(uint32 *u32On*, **uint32** *u32Off***);**

Description

This function sets the pull-ups on individual DIO pins as on or off. A pull-up can be set irrespective of whether the pin is an input or output. This is done through two bitmaps for ‘pull-ups on’ and ‘pull-ups off’, *u32On* and *u32Off* respectively. In these values, each of bits 0 to 20 represents a DIO pin, where bit 0 represents DIO0 and bit 20 represents DIO20 (bits 21-31 are ignored).

Note that:

- By default, the pull-ups are enabled (on) at power-up.
- Not all DIO pull-ups must be set (in other words, *u32On* logical ORed with *u32Off* does not need to produce all zeros for bits 0-20).
- Any DIO pull-ups that are not set by a call to this function (the relevant bits being cleared in both bitmaps) will be left in their previous states.
- If a bit is set in both *u32On* and *u32Off*, the corresponding DIO pull-up will default to off.
- If a DIO is assigned to another peripheral which is enabled, this function call will not immediately affect the relevant pin. However, the DIO pull-up setting specified by this function will take effect if/when the relevant peripheral is subsequently disabled.

Parameters

u32On	Bitmap of ‘pull-ups on’ - a bit set means that the corresponding pull-up will be turned on
u32Off	Bitmap of ‘pull-ups off’ - a bit set means that the corresponding pull-up will be turned off

Returns

None

u32JPI_DioReadInput

```
uint32 u32JPI_DioReadInput(void);
```

Description

This function returns the value of each of the DIO input pins (irrespective of whether the pins are used as inputs, as outputs or by other enabled peripherals).

Parameters

None

Returns

Bitmap:

Each of bits 0-20 corresponds to a DIO pin, where bit 0 represents DIO0 and bit 20 represents DIO20. The bit is set to 1 if the input is high or to 0 if the input is low. Bits 21-31 are always 0.

vJPI_DioWake

```
void vJPI_DioWake(uint32 u32Enable,
                  uint32 u32Disable,
                  uint32 u32Rising,
                  uint32 u32Falling);
```

Description

This function configures and enables/disables wake signals (interrupts) on the DIO input pins.

The function enables/disables wake interrupts on the DIO pins - that is, whether activity on a DIO input will be able to wake the device from sleep or doze mode. This is done through two bitmaps for 'wake enabled' and 'wake disabled', *u32Enable* and *u32Disable* respectively. In these values, each of bits 0 to 20 represents a DIO pin, where bit 0 represents DIO0 and bit 20 represents DIO20 (bits 21-31 are ignored). Setting a bit in one of these bitmaps enables/disables wake interrupts on the corresponding DIO, depending on the bitmap. Note that:

- Not all DIO wake interrupts must be defined (in other words, *u32Enable* logical ORed with *u32Disable* does not need to produce all zeros for bits 0-20).
- Any DIO wake interrupts that are not defined by a call to this function (the relevant bits being cleared in both bitmaps) will be left in their previous states.
- If a bit is set in both *u32Enable* and *u32Disable*, the corresponding DIO wake interrupt will default to disabled.
- This call has no effect on DIO pins that are not defined as inputs (see **vJPI_DioSetDirection()**).
- DIOs assigned to enabled JN5139/JN5148 peripherals are affected by this function.
- The DIO wake interrupt settings made with this function are retained during sleep.

The function also controls whether individual DIOs will generate wake interrupts on a rising or falling edge of the DIO input. This is done through two bitmaps for 'rising edge' and 'falling edge', *u32Rising* and *u32Falling* respectively. In these values, each of bits 0 to 20 represents a DIO pin, where bit 0 represents DIO0 and bit 20 represents DIO20 (bits 21-31 are ignored). Setting a bit in one of these bitmaps configures wake interrupts on the corresponding DIO to occur on a rising or falling edge, depending on the bitmap (by default, all DIO wake interrupts are 'rising edge'). Note that:

- Not all DIO wake interrupts must be configured (in other words, *u32Rising* logical ORed with *u32Falling* does not need to produce all zeros for bits 0-20).
- Any DIO wake interrupts that are not configured by a call to this function (the relevant bits being cleared in both bitmaps) will be left in their previous states.
- If a bit is set in both *u32Rising* and *u32Falling*, the corresponding DIO wake interrupt will default to 'rising edge'.

Note that DIO wake interrupts are handled by the System Controller callback function, registered using the function **vJPI_SysCtrlRegisterCallback()**.

Parameters

u32Enable	Bitmap for wake enable - a bit set means that the wake signal on the corresponding DIO pin will be enabled
u32Disable	Bitmap for wake disable - a bit set means that the wake signal on the corresponding DIO pin will be disabled
u32Rising	Bitmap for rising edge - a bit set means that the wake signal on the corresponding DIO pin will be triggered on a rising edge
u32Falling	Bitmap for falling edge - a bit set means that the wake signal on the corresponding DIO pin will be triggered on a falling edge

Returns

None

u32JPI_DioWakeStatus

```
uint32 u32JPI_DioWakeStatus(void);
```

Description

This function returns the interrupt status of each of the DIO input pins.

The returned value is a bitmap in which a bit is set if a wake interrupt has occurred on the corresponding DIO input pin (see below). Bit values are not valid for DIO pins that are assigned as outputs or assigned to another enabled peripheral. After reading the interrupt status, the value is cleared.

Note: You cannot use this function to check whether a DIO was responsible for a recent wake-up event. You should determine the wake source as part of your System Controller callback function, as described in Appendix E.2.

Parameters

None

Returns

Bitmask:

Each of bits 0-20 corresponds to a DIO pin, where bit 0 represents DIO0 and bit 20 represents DIO20. The bit is set if an interrupt associated with the pin has triggered. Bits 21-31 are always 0.

3.4 UARTs

This section details the functions for controlling the UARTs (Universal Asynchronous Receiver Transmitters). The JN5139/JN5148 device has two 16550-compatible UARTs, denoted UART0 and UART1, which can be independently enabled. UART0 is also used for the debugger.

Note the following:

UART Pins

Each UART uses four pins, shared with the DIOs, for the following signals: clear-to-send (CTS) input, request-to-send (RTS) output, transmit data output, receive data input.

On the JN5148 device, the pins normally used by a UART can alternatively be used to connect a JTAG emulator for debugging.

Receive FIFO Interrupt Operation

- Receiver interrupts for the UARTs are enabled using **vJPI_UartSetInterrupt()**.
- The "receive data available interrupt" is issued when the FIFO reaches its programmed trigger level. It is cleared as soon as the FIFO drops below its programmed trigger level. The FIFO trigger level can be set to 1, 4, 8 or 14 bytes using **vJPI_UartSetInterrupt()**.
- The function **u8JPI_UartReadInterruptStatus()** can be used to get the "receive data available status". This is set when the FIFO trigger level is reached and, like the interrupt, it is cleared when the FIFO drops below the trigger level.
- When Receiver FIFO interrupts are enabled, timeout interrupts also occur. A FIFO timeout interrupt will occur if the following conditions exist:
    - At least one character is in the FIFO
    - No character has entered the FIFO during a time interval in which at least four characters could potentially have been received
    - Nothing has been read from the FIFO during a time interval in which at least four characters could potentially have been read
- When a timeout interrupt occurs, it is cleared and the timer is reset by reading a character from the receive FIFO.

Enabling/Disabling the UARTs

Avoid using the UARTs while attempting to join the network. Only enable the UARTs on a device once it has joined a network. If UART output is important while trying to join the network, the relevant UART could be enabled at the start of any functions requiring UART output and disabled once the functions have completed.

The UARTs can be disabled before entering sleep mode in order to reduce current consumption while sleeping. In this case, when entering sleep mode, wait for the UART transmit buffers to empty before disabling the UART.

On waking or reset, the following recommendations should be applied:

- Following a cold start, do not enable the UARTs until the network is running, as indicated by an E_JENIE_NETWORK_UP event. This is the case following a device reset or sleep without memory held.
- Following a warm start, enable the UARTs as a part of the **vJenie_CbInit()** callback function. This is the case following sleep with memory held.

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

Function	**Page**
vJPI_UartEnable	87
vJPI_UartDisable	88
vJPI_UartSetClockDivisor	89
vJPI_UartSetBaudDivisor	90
vJPI_UartSetControl	91
vJPI_UartSetInterrupt	92
vJPI_UartSetRTSCTS	93
vJPI_UartReset	94
u8JPI_UartReadLineStatus	95
u8JPI_UartReadModemStatus	96
u8JPI_UartReadInterruptStatus	97
vJPI_UartWriteData	98
u8JPI_UartReadData	99
vJPI_Uart0RegisterCallback	100
vJPI_Uart1RegisterCallback	101

vJPI_UartEnable

```
void vJPI_UartEnable(uint8 u8Uart);
```

Description

This function enables the specified UART. It must be the first UART function called.

Be sure to enable the UART using this function before writing to the UART using the function **vJPI_UartWriteData()**, otherwise an exception will result.

If you wish to implement RTS/CTS flow control, you will need to call **vJPI_UartSetRTSCTS()** before calling **vJPI_UartEnable()** in order to take control of the DIOs used for RTS and CTS. The UARTs use certain DIO lines, as follows:

Signal	DIOs for UART0	DIOs for UART1
CTS	DIO4	DIO17
RTS	DIO5	DIO18
TxD	DIO6	DIO19
RxD	DIO7	DIO20

If a UART uses only the RxD and TxD lines, it is said to operate in 2-wire mode. If, in addition, it uses the RTS and CTS lines, it is said to operate in 4-wire mode.

Parameters

u8Uart	Identity of UART: E_JPI_UART_0 E_JPI_UART_1

Returns

None

vJPI_UartDisable

void vJPI_UartDisable(uint8 *u8Uart***);**

Description

This function disables the specified UART.

Be sure to re-enable the UART using **vJPI_UartEnable()** before attempting to write to the UART using the function **vJPI_UartWriteData()**, otherwise an exception will result.

Parameters

u8Uart	Identity of UART: E_JPI_UART_0 E_JPI_UART_1

Returns

None

vJPI_UartSetClockDivisor

void vJPI_UartSetClockDivisor(uint8 *u8Uart*,
uint8 *u8BaudRate***);**

Description

This function sets the baud rate for the specified UART to one of a number of standard rates.

The possible baud rates are:

- 4800 bps
- 9600 bps
- 19200 bps
- 38400 bps
- 76800 bps
- 115200 bps

To set the baud rate to other values, use the function **vJPI_UartSetBaudDivisor()**.

Parameters

u8Uart	Identity of UART: E_JPI_UART_0 E_JPI_UART_1
u8BaudRate	Desired baud rate: E_JPI_UART_RATE_4800 (4800 bps) E_JPI_UART_RATE_9600 (9600 bps) E_JPI_UART_RATE_19200 (19200 bps) E_JPI_UART_RATE_38400 (38400 bps) E_JPI_UART_RATE_76800 (76800 bps) E_JPI_UART_RATE_115200 (115200 bps)

Returns

None

vJPI_UartSetBaudDivisor

```
void vJPI_UartSetBaudDivisor(uint8 u8Uart,
                             uint16 u16Divisor);
```

Description

This function sets an integer divisor to derive the baud rate from a 1-MHz frequency for the specified UART. The function allows baud rates to be set that are not available through the function **vJPI_UartSetClockDivisor()**.

The baud rate produced is defined by:

baud rate = 1000000/*u16Divisor*

For example:

u16Divisor	**Baud rate (bit/s)**
1	1000000
2	500000
9	115200 (approx)
26	38400 (approx)

Parameters

u8Uart	Identity of UART: E_JPI_UART_0 E_JPI_UART_1
u16Divisor	Integer divisor (for 1-MHz clock)

Returns

None

vJPI_UartSetControl

```
void vJPI_UartSetControl(uint8 u8Uart,
                         bool_t bEvenParity,
                         bool_t bEnableParity,
                         uint8 u8WordLength,
                         bool_t bOneStopBit,
                         bool_t bRtsValue);
```

Description

This function sets various control bits for the specified UART.

Note that RTS cannot be controlled automatically, but only enabled or disabled under software control.

Parameters

u8Uart	Identity of UART: E_JPI_UART_0 E_JPI_UART_1
bEvenParity	Parity (even or odd): E_JPI_UART_EVEN_PARITY E_JPI_UART_ODD_PARITY
bEnableParity	Enable/disable parity check: E_JPI_UART_PARITY_ENABLE E_JPI_UART_PARITY_DISABLE
u8WordLength	Word length: E_JPI_UART_WORD_LEN_5 (Word is 5 bits) E_JPI_UART_WORD_LEN_6 (Word is 6 bits) E_JPI_UART_WORD_LEN_7 (Word is 7 bits) E_JPI_UART_WORD_LEN_8 (Word is 8 bits)
bOneStopBit	Number of stop bits - TRUE for 1 stop bit, FALSE for 1.5 or 2 stop bits, depending on word length - enumerated as: E_JPI_UART_1_STOP_BIT (TRUE) E_JPI_UART_2_STOP_BITS (FALSE)
bRtsValue	Enable or disable RTS: E_JPI_UART_RTS_HIGH (TRUE) - Disable RTS E_JPI_UART_RTS_LOW (FALSE) - Enable RTS

Returns

None

vJPI_UartSetInterrupt

```
void vJPI_UartSetInterrupt(uint8 u8Uart,
                           bool_t bEnableModemStatus,
                           bool_t bEnableRxLineStatus,
                           bool_t bEnableTxFifoEmpty,
                           bool_t bEnableRxData,
                           uint8 u8FifoLevel);
```

Description

This function enables or disables the interrupts generated by the specified UART and sets the receive FIFO level - that is, the number of words received by the FIFO in order to trigger a receive data interrupt.

UART interrupts are handled by callback functions registered using the following functions:

- **vJPI_Uart0RegisterCallback()** for UART0
- **vJPI_Uart1RegisterCallback()** for UART1

Note: If the receive FIFO level is set to a value greater than one, UART timeout interrupts (described on page 85) will automatically be enabled.

Parameters

u8Uart	Identity of UART: E_JPI_UART_0 E_JPI_UART_1
bEnableModemStatus	Enable/disable modem status interrupt (e.g. CTS change detected) - TRUE for enable, FALSE for disable
bEnableRxLineStatus	Enable/disable receive line status interrupt (e.g. framing error, parity error) - TRUE for enable, FALSE for disable
bEnableTxFifoEmpty	Enable/disable interrupt when transmit FIFO empty - TRUE for enable, FALSE for disable
bEnableRxData	Enable/disable interrupt when receive data seen - TRUE for enable, FALSE for disable
u8FifoLevel	Number of received words required to trigger a receive data interrupt: E_JPI_UART_FIFO_LEVEL_1 (1 word) E_JPI_UART_FIFO_LEVEL_4 (4 words) E_JPI_UART_FIFO_LEVEL_8 (8 words) E_JPI_UART_FIFO_LEVEL_14 (14 words)

Returns

None

vJPI_UartSetRTSCTS

void vJPI_UartSetRTSCTS(uint8 *u8Uart*, **bool_t** *bRTSCTSEn***);**

Description

This function instructs the specified UART to take or release control of the DIO lines used for RTS/CTS flow control.

UART0: DIO4 for CTS
DIO5 for RTS

UART1: DIO17 for CTS
DIO18 for RTS

The function must be called before **vJPI_UartEnable()** is called.

If a UART uses these two additional DIO lines, it is said to operate in 4-wire mode, otherwise it is said to operate in 2-wire mode.

Parameters

u8Uart	Identity of UART: E_JPI_UART_0 E_JPI_UART_1
bRTSCTSEn	Take or release control of DIOs for RTS/CTS: TRUE to take control FALSE to release control

Returns

None

vJPI_UartReset

void vJPI_UartReset(uint8 *u8Uart*,
bool_t *bTxReset*,
bool_t *bRxReset***);**

Description

This function resets the Transmit and Receive FIFOs. The character currently being transferred is not affected. The Transmit and Receive FIFOs can be reset individually or together.

The function also sets the FIFO trigger level to single-byte trigger. The FIFO interrupt trigger level can be set via **vJPI_UartSetInterrupt()**.

Parameters

u8Uart	Identity of UART: E_JPI_UART_0 E_JPI_UART_1
bTxReset	Transmit FIFO reset: TRUE to reset FALSE not to reset
bRxReset	Receive FIFO reset TRUE to reset FALSE not to reset

Returns

None

u8JPI_UartReadLineStatus

uint8 u8JPI_UartReadLineStatus(uint8 *u8Uart*);

Description

This function returns line status information for the specified UART as a bitmap.

Note that the following bits are cleared after reading:
E_JPI_UART_LS_ERROR
E_JPI_UART_LS_BI
E_JPI_UART_LS_FE
E_JPI_UART_LS_PE
E_JPI_UART_LS_OE

Parameters

u8Uart	Identity of UART: E_JPI_UART_0 E_JPI_UART_1

Returns

Bitmap:

Bit	Description
E_JPI_UART_LS_ERROR	This bit will be set if a parity error, framing error or break indication has been received
E_JPI_UART_LS_TEMT	This bit will be set if the transmit shift register is empty
E_JPI_UART_LS_THRE	This bit will be set if the transmit FIFO is empty
E_JPI_UART_LS_BI	This bit will be set if a break indication has been received (line held low for a whole character)
E_JPI_UART_LS_FE	This bit will be set if a framing error has been received
E_JPI_UART_LS_PE	This bit will be set if a parity error has been received
E_JPI_UART_LS_OE	This bit will be set if a receive overrun has occurred, i.e. the receive buffer is full but another character arrives
E_JPI_UART_LS_DR	This bit will be set if there is data in the receive FIFO

u8JPI_UartReadModemStatus

uint8 u8JPI_UartReadModemStatus(uint8 *u8Uart***);**

Description

This function obtains modem status information from the specified UART as a bitmap which includes the CTS and ‘CTS has changed’ status (which can be extracted as described below).

Parameters

u8Uart	Identity of UART: E_JPI_UART_0 E_JPI_UART_1

Returns

Bitmap in which:

- CTS input status is bit 4 (‘1’ indicates CTS is high, ‘0’ indicates CTS is low).
- ‘CTS has changed’ status is bit 0 (‘1’ indicates that CTS input has changed). If the return value logically ANDed with E_JPI_UART_MS_DCTS is non-zero, the CTS input has changed.

u8JPI_UartReadInterruptStatus

uint8 u8JPI_UartReadInterruptStatus(uint8 *u8Uart*);

Description

This function returns a pending interrupt for the specified UART as a bitmap.

Interrupts are returned one at a time, so there may need to be multiple calls to this function. If interrupts are enabled, the interrupt handler processes this activity and posts each interrupt to the queue or to a callback function.

Parameters

u8Uart	Identity of UART: E_JPI_UART_0 E_JPI_UART_1

Returns

Bitmap:

Bit range	Value/Enumeration	Description
Bit 0	0	More interrupts pending
	1	No more interrupts pending
Bits 1-3	E_JPI_UART_INT_MODEM	Modem status interrupt
	E_JPI_UART_INT_TX	Transmit FIFO empty interrupt
	E_JPI_UART_INT_RXDATA	Receive data available interrupt
	E_JPI_UART_INT_RXLINE	Receive line status interrupt
	E_JPI_UART_INT_TIMEOUT	Timeout interrupt

vJPI_UartWriteData

void vJPI_UartWriteData(uint8 *u8Uart*, **uint8** *u8Data***);**

Description

This function writes a data byte to the Transmit FIFO of the specified UART for transmission.

Before this function is called, the UART must be enabled using the function **vJPI_UartEnable()**, otherwise an exception will result.

Parameters

u8Uart	Identity of UART: E_JPI_UART_0 E_JPI_UART_1
u8Data	Byte to transmit

Returns

None

u8JPI_UartReadData

uint8 u8JPI_UartReadData(uint8 *u8Uart***);**

Description

This function returns a single byte read from the Receive FIFO of the specified UART. If the FIFO is empty, the value is not valid.

Refer to the description of **u8JPI_UartReadLineStatus()** for details of how to determine whether the FIFO is empty.

Parameters

u8Uart	Identity of UART: E_JPI_UART_0 E_JPI_UART_1

Returns

Received byte

vJPI_Uart0RegisterCallback

void vJPI_Uart0RegisterCallback(
            PR_HWINT_APPCALLBACK *prUart0Callback***);**

Description

This function registers an application callback that will be called when the UART0 interrupt is triggered.

The registered callback function is only preserved during sleep with memory held. If RAM is powered off during sleep and interrupts are required, the callback function must be re-registered before calling **u32JPI_Init()** on waking.

Interrupt handling is described in Appendix E.

Parameters

prUart0Callback	Pointer to function that is to be called when UART0 interrupt occurs.

Returns

None

vJPI_Uart1RegisterCallback

```
void vJPI_Uart1RegisterCallback(
            PR_HWINT_APPCALLBACK prUart1Callback);
```

Description

This function registers an application callback that will be called when the UART1 interrupt is triggered.

The registered callback function is only preserved during sleep with memory held. If RAM is powered off during sleep and interrupts are required, the callback function must be re-registered before calling **u32JPI_Init()** on waking.

Interrupt handling is described in Appendix E.

Parameters

prUart1Callback	Pointer to function that is to be called when UART1 interrupt occurs.

Returns

None

3.5 Timers

This section details the functions for controlling the general-purpose timers. The number of timers available depends on the device type:

- JN5139 has two timers: Timer 0 and Timer 1
- JN5148 has three timers: Timer 0, Timer 1 and Timer 2

These timers are distinct from the wake timers described in Section 3.6.

***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.*

The timers normally use the internal 16-MHz clock source of the JN5139/JN5148 device.

Modes of Operation

The timers can be operated in the following modes: Timer, PWM, Delta-Sigma, 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.	**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 DIO Usage 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 DIO Usage below) and requires simple filtering to give the analogue signal.	**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.	**vJPI_TimerEnable()**
		vJPI_TimerStartCapture()
		u32JPI_TimerReadCapture()

Table 1: Modes of Timer Operation

DIO Usage

The timers use the JN5139/JN5148 device's DIO pins as follows:

Timer 0 DIO pin	Timer 1 DIO pin	Timer 2 DIO pin	Function
8	11 *	-	Clock/gate input
9	12	-	Capture input
10	13	11 *	PWM and Delta-Sigma output

* DIO11 is shared by Timer 1 and TImer 2 on the JN5148 device, and their use must not confict.

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

Function	Page
vJPI_TimerEnable	104
vJPI_TimerDisable	106
vJPI_TimerStart	107
vJPI_TimerStop	109
vJPI_TimerStartCapture	110
u32JPI_TimerReadCapture	111
u8JPI_TimerFired	112
vJPI_Timer0RegisterCallback	113
vJPI_Timer1RegisterCallback	114
vJPI_Timer2RegisterCallback (JN5148 Only)	115

vJPI_TimerEnable

```
void vJPI_TimerEnable(teJPI_Timer eTimer,
                      uint8 u8Prescale,
                      uint8 mIntMask,
                      bool_t bOutputEn,
                      bool_t bTimerIOEn,
                      teJPI_TimerClockType eClockType);
```

Description

This function configures and enables the specified timer, and must be the first timer function called. The timer can be used in various modes, described in Table 1 on page 102.

The parameters of this enable function cover the following features:

- **Source clock** (*eClockType*): The clock for the timer is normally the internal 16-MHz system clock, which can be optionally inverted.
- **Prescaling** (*u8Prescale*): The timer's source clock is divided down to produce a slower clock for the timer, the divisor being $2^{Prescale}$. Therefore:

  Timer clock frequency = Source clock frequency / $2^{Prescale}$
- **Interrupts** (*mIntMask*): Interrupts can be generated in Timer mode on a low-to-high transition (rising output) and/or on a high-to-low transition (end of the timer period). Alternatively, timer interrupts can be disabled. If enabled, timer interrupts are handled by callback functions registered using:
  - **vJPI_Timer0RegisterCallback()** for Timer 0
  - **vJPI_Timer1RegisterCallback()** for Timer 1
  - **vJPI_Timer2RegisterCallback()** for Timer 2 (JN5148 only)
- **DIOs** (*bTimerIOEn*): Timer 0 uses DIO8-10, Timer 1 uses DIO11-13 and Timer2 uses DIO11. Use of these pins must be explicitly enabled.
- **Output** (*bOutputEn*): When operating in PWM mode or Delta-Sigma mode, the timer's signal is output on a DIO pin (DIO10 for Timer 0, DIO13 for Timer 1, DIO11 for Timer 2), which must be enabled. If this option is enabled, the other DIOs associated with the timer cannot be used for general-purpose input/output.

Parameters

eTimer	Identity of timer: E_JPI_TIMER_0 E_JPI_TIMER_1 E_JPI_TIMER_2 (JN5148 Only)
u8Prescale	Prescale index, in range 0 to 16, used to divide down clock (divisor is $2^{Prescale}$)
mIntMask	Mask for interrupt generation on timer events: 0x00 to disable timer interrupts E_JPI_TIMER_INT_PERIOD (interrupt at end of timed period) E_JPI_TIMER_INT_RISE (interrupt on low-to-high transition)

bOutputEn	Enable/disable output of timer signal on DIO pin (see above): TRUE - Enable FALSE - Disable
bTimerIOEn	Enable/disable use of DIO pins for timer (see above): TRUE - Enable FALSE - Disable
eClockType	Clock (internal or external) and its polarity (normal or inverted): E_JPI_TIMER_CLOCK_INTERNAL_NORMAL E_JPI_TIMER_CLOCK_INTERNAL_INVERTED E_JPI_TIMER_CLOCK_EXTERNAL_NORMAL E_JPI_TIMER_CLOCK_EXTERNAL_INVERTED

Returns

None

vJPI_TimerDisable

void vJPI_TimerDisable(teJPI_Timer *eTimer***);**

Description

This function disables the specified timer. As well as stopping the timer from running, the clock to the timer block is turned off in order to reduce power consumption. This means that any subsequent attempt to access the timer will be unsuccessful until **vJPI_TimerEnable()** is called to re-enable the block.

Caution: *An attempt to access the timer while it is disabled will result in an exception.*

Parameters

eTimer	Identity of timer: E_JPI_TIMER_0 E_JPI_TIMER_1 E_JPI_TIMER_2 (JN5148 Only)

Returns

None

vJPI_TimerStart

```
void vJPI_TimerStart(teJPI_Timer eTimer,
                     teJPI_TimerMode eTimerMode,
                     uint16 u16HighPeriod,
                     uint16 u16LowPeriod);
```

Description

This function starts the specified timer in Timer/PWM mode or Delta-Sigma mode.

- **Timer mode:** During one pulse cycle of this mode, the timer signal starts off low and then goes high. The output goes low until *u16HighPeriod* clock periods have passed, after which it goes high until *u16LowPeriod* clock periods since the timer was started. If timer interrupts have been enabled using **vJPI_TimerEnable()**, an interrupt will be triggered at the low-high transition and/or at the high-low transition.

  This mode is available in 'single-shot' and 'repeat' versions:

  - Single-shot mode produces one pulse cycle and stops
  - Repeat mode produces a train of pulses until the timer is stopped

  PWM (Pulse Width Modulation) mode is simply one of the above Timer modes with the timer output enabled on the appropriate DIO (see below).

- **Delta-Sigma mode:** This mode allows the timer to be used as a low-rate DAC with the output on the appropriate DIO (see below). An output waveform is produced consisting of a series of high and low clock cycles that are pseudo-randomly spaced in time. The number of high clock cycles (*u16HighPeriod*) within the total period of the waveform (comprising 65536 clock cycles) should be set to a value which is proportional to the value to be converted. Placing a Resistor-Capacitor (RC) circuit on the output pin will then produce an averaged analogue voltage.

  This mode is available in 'normal' and 'RTZ' versions:

  - Normal Delta-Sigma mode operates as described above.
  - RTZ (Return-to-Zero) Delta-Sigma mode operates as normal Delta-Sigma mode but a low clock cycle is inserted after every clock cycle (thus, there are 131072 clock cycles in the total period). Note that this does not affect the setting of *u16HighPeriod*.

If the timer signal is to be output (as in the case of PWM and Delta-Sigma modes), use DIO10 for Timer 0, DIO13 for Timer 1, DIO11 for Timer 2. These DIOs must be enabled for output in **vJPI_TimerEnable()**. For Delta-Sigma modes, an RC circuit must be inserted between the output pin and Ground

Note that the timer is started in Capture mode using a separate function, **vJPI_TimerStartCapture()**.

For more information on timer operation, refer to the *Jenie API User Guide (JN-UG-3042)*.

Parameters

eTimer	Identity of timer: E_JPI_TIMER_0 E_JPI_TIMER_1 E_JPI_TIMER_2 (JN5148 only)

eTimerMode	Mode of operation: E_JPI_TIMER_MODE_SINGLESHOT (Single-shot) E_JPI_TIMER_MODE_REPEATING (Repeat) E_JPI_TIMER_MODE_DELTASIGMA (Delta-Sigma) E_JPI_TIMER_MODE_DELTASIGMARTZ (Return-to-Zero)
u16HighPeriod	Number of clock periods after starting a timer before the output goes high (Timer mode) or number of high clock cycles representing value to be converted (Delta-Sigma mode)
u16LowPeriod	Number of clock periods after starting a timer before the output goes low in Timer mode (but parameter ignored in Delta-Sigma mode)

Returns

None

vJPI_TimerStop

```
void vJPI_TimerStop(teJPI_Timer eTimer);
```

Description

This function stops the specified timer if it has been started in Timer/PWM mode or Delta-Sigma mode using **vJPI_TimerStart()**.

Parameters

eTimer	Identity of timer: E_JPI_TIMER_0 E_JPI_TIMER_1 E_JPI_TIMER_2 (JN5148 only)

Returns

None

vJPI_TimerStartCapture

void vJPI_TimerStartCapture(teJPI_Timer *eTimer***);**

Description

This function starts the specified timer in Capture mode.

An input signal must be provided on pin DIO9 for Timer 0 or DIO12 for Timer 1 (Capture mode is not available on Timer 2 of the JN5148 device). The incoming signal is timed and the captured measurements are:

- number of clock cycles to the last low-to-high transition of the input signal
- number of clock cycles to the last high-to-low transition of the input signal

These values are placed in registers to be read later using the function **vJPI_TimerReadCapture()**. They allow the input pulse width to be determined.

Parameters

eTimer	Identity of timer: E_JPI_TIMER_0 E_JPI_TIMER_1

Returns

None

u32JPI_TimerReadCapture

uint32 u32JPI_TimerReadCapture(teJPI_Timer *eTimer***);**

Description

This function stops the timer and provides the results from a 'capture' started using **vJPI_TimerStartCapture()**.

The values returned are offsets from when the capture was originally started, as follows:

- number of clock cycles to the last low-to-high transition of the input signal
- number of clock cycles to the last high-to-low transition of the input signal

The width of the last pulse can be calculated from the difference of these results, provided that the results were requested during a low period. However, since it is not possible to be sure of this, the results obtained from this function may not always be valid for calculating the pulse width.

Parameters

eTimer	Identity of timer: E_JPI_TIMER_0 E_JPI_TIMER_1

Returns

32-bit value in which:

- The upper 16 bits (bits 31-16) represent the number of clock cycles up to the last low-to-high transition.
- The lower 16 bits (bits 15-0) represent the number of clock cycles up to the last high-to-low transition.

u8JPI_TimerFired

```
uint8 u8JPI_TimerFired(teJPI_Timer eTimer);
```

Description

This function obtains the interrupt status of the specified timer. The function also clears interrupt status after reading it.

Parameters

eTimer	Identity of timer: E_JPI_TIMER_0 E_JPI_TIMER_1 E_JPI_TIMER_2 (JN5148 only)

Returns

Bitmap:

Returned value logical ANDed with E_JPI_TIMER_INT_PERIOD - will be non-zero if interrupt for high-to-low transition (end of period) has been set

Returned value logical ANDed with E_JPI_TIMER_INT_RISE - will be non-zero if interrupt for low-to-high transition (output rising) has been set

vJPI_Timer0RegisterCallback

**void vJPI_Timer0RegisterCallback(
PR_HWINT_APPCALLBACK** *PrTimer0Callback***);**

Description

This function registers an application callback that will be called when the Timer 0 interrupt is triggered.

Interrupt handling is described in Appendix E.

Parameters

prTimer0Callback	Pointer to function that is to be called when Timer 0 interrupt occurs

Returns

None

vJPI_Timer1RegisterCallback

void vJPI_Timer1RegisterCallback(
        PR_HWINT_APPCALLBACK *PrTimer1Callback***);**

Description

This function registers an application callback that will be called when the Timer 1 interrupt is triggered.

Interrupt handling is described in Appendix E.

Parameters

prTimer1Callback	Pointer to function that is to be called when Timer 1 interrupt occurs

Returns

None

vJPI_Timer2RegisterCallback (JN5148 Only)

```
void vJPI_Timer2RegisterCallback(
        PR_HWINT_APPCALLBACK PrTimer2Callback);
```

Description

This function registers an application callback that will be called when the Timer 2 interrupt is triggered on a JN5148 device.

Interrupt handling is described in Appendix E.

Parameters

prTimer2Callback	Pointer to function that is to be called when Timer 2 interrupt occurs

Returns

None

3.6 Wake Timers

This section details the functions for controlling the 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 the JPI library.

Note: If you wish to use the full 35-bit wake timers on the JN5148 device, you should use the functions of the Integrated Peripherals API described in the *Integrated Peripherals API Reference Manual (JN-RM-2001)*.

The wake timers are normally used to time sleep periods and can be programmed to generate interrupts when the timeout period is reached. They can also be used while the CPU is running (but there is another set of timers with more functionality that can operate only while the CPU is running - see Section 3.5).

Wake Timer Calibration

The wake timers run at a nominal 32 kHz but to minimise complexity and hence power consumption, they may run at up to 30% fast or slow depending on temperature, supply voltage and manufacturing tolerance. A self-calibration facility is provided to time the 32-kHz clock against the 16-MHz clock if accurate timing is required.

Wake Timer Events

When a sleeping End Device is woken by a wake timer, this event is not presented to the user application either by the **vJenie_CbHwEvent()** callback function or by the callback function that is registered through **vJPI_SysCtrlRegisterCallback()**. However, since all other wake sources (DIO and comparator) do generate an event, it is possible to determine whether a wake timer caused the wake-up by a process of elimination.

The ‘wake timer fired’ status is cleared by the stack upon waking, so it is not possible to use the **u8JPI_WakeTimerFiredStatus()** function to determine whether the wake timer caused the wake-up. However, the wake timer value is not cleared by the stack and can be read with the **u32JPI_WakeTimerRead()** function. Thus, if the wake timer has fired, this function will return a high value, as the timer will have rolled over from 0 (if this value is greater than 0x80000000 then the wake is likely to be due to the timer firing).

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

Function	**Page**
vJPI_WakeTimerEnable	118
vJPI_WakeTimerStart	119
vJPI_WakeTimerStop	120
u32JPI_WakeTimerRead	121
u8JPI_WakeTimerStatus	122
u8JPI_WakeTimerFiredStatus	123
u32JPI_WakeTimerCalibrate	124

vJPI_WakeTimerEnable

void vJPI_WakeTimerEnable(uint8 *u8Timer***,**
bool_t *bIntEnable***);**

Description

This function allows the wake timer interrupt (which is generated when the timer fires) to be enabled/disabled. If this function is called for a wake timer that is already running, it will stop the wake timer.

The wake timer can be subsequently started using **vJPI_WakeTimerStart()**.

Wake timer interrupts are handled by the System Controller callback function, registered using the function **vJPI_SysCtrlRegisterCallback()**.

Parameters

u8Timer	Identity of timer: E_JPI_WAKE_TIMER_0 E_JPI_WAKE_TIMER_1
bIntEnable	Interrupt enable/disable: TRUE to enable interrupt when wake timer fires FALSE to disable interrupt

Returns

None

vJPI_WakeTimerStart

void vJPI_WakeTimerStart(uint8 *u8Timer*,
uint32 *u32Count***);**

Description

This function starts the specified wake timer with the specified count value. The wake timer will count down from this value, which is set according to the desired timer duration. On reaching zero, the timer 'fires', rolls over and continues to count down.

The count value, *u32Count*, is set as the required number of 32-kHz periods. Thus:

Timer duration (in seconds) = *u32Count* / 32000

Note that the 32-kHz internal clock, which drives the wake timer, may be running up to 30% fast or slow. For accurate timings, you are advised to first calibrate the clock using the function **u32JPI_WakeTimerCalibrate()** and adjust the specified count value accordingly.

If you wish to enable interrupts for the wake timer, you must call **vJPI_WakeTimerEnable()** before calling **vJPI_WakeTimerStart()**. The wake timer can be subsequently stopped using **vJPI_WakeTimerStop()** and can be read using **u32JPI_WakeTimerRead()**. Stopping the timer does not affect interrupts that have been set using **vJPI_WakeTimerEnable()**.

Parameters

u8Timer	Identity of timer: E_JPI_WAKE_TIMER_0 E_JPI_WAKE_TIMER_1
u32Count	Count value in 32-kHz periods, i.e. 32 is 1 millisecond

Returns

None

vJPI_WakeTimerStop

```
void vJPI_WakeTimerStop(uint8 u8Timer);
```

Description

This function stops the specified wake timer.

Note that no interrupt will be generated.

Parameters

u8Timer	Identity of timer: E_JPI_WAKE_TIMER_0 E_JPI_WAKE_TIMER_1

Returns

None

u32JPI_WakeTimerRead

uint32 u32JPI_WakeTimerRead(uint8 *u8Timer***);**

Description

This function obtains the current value of the specified wake timer counter (which counts down), without stopping the counter.

Note that on reaching zero, the timer ‘fires’, rolls over to 0xFFFFFFFF and continues to count down. The count value obtained using this function then allows the application to calculate the time that has elapsed since the wake timer fired.

Parameters

u8Timer	Identity of timer: E_JPI_WAKE_TIMER_0 E_JPI_WAKE_TIMER_1

Returns

Current value of wake timer counter.

u8JPI_WakeTimerStatus

```
uint8 u8JPI_WakeTimerStatus(void);
```

Description

This function determines which wake timers are active. It is possible to have more than one wake timer active at the same time. The function returns a bitmap where the relevant bits are set to show which wake timers are active.

Note that a timer remains active after its countdown has reached zero (when the timer rolls over and continues to count down).

Parameters

None

Returns

Bitmap:

Returned value logical ANDed with E_JPI_WAKE_TIMER_MASK_0 will be non-zero if Wake Timer 0 is active

Returned value logical ANDed with E_JPI_WAKE_TIMER_MASK_1 will be non-zero if Wake Timer 1 is active

u8JPI_WakeTimerFiredStatus

```
uint8 u8JPI_WakeTimerFiredStatus(void);
```

Description

This function determines which wake timers have fired (by having passed zero). The function returns a bitmap where the relevant bits are set to show which timers have fired. Any fired timer status is cleared as a result of this call.

Note: If you wish to use this function to check whether a wake timer caused a wake-up event, you must call it before **u32JPI_Init()**. Alternatively, you can determine the wake source as part of your System Controller callback function. For more information, refer to Appendix E.2.

Parameters

None

Returns

Bitmap:

Returned value logical ANDed with E_JPI_WAKE_TIMER_MASK_0 will be non-zero if Wake Timer 0 has fired
Returned value logical ANDed with E_JPI_WAKE_TIMER_MASK_1 will be non-zero if Wake Timer 1 has fired

u32JPI_WakeTimerCalibrate

uint32 u32JPI_WakeTimerCalibrate(void);

Description

This function requests a calibration of the 32-kHz internal clock (on which the wake timers run) against the more accurate 16-MHz system clock. Note that the 32-kHz clock has a tolerance of ±30%.

Wake Timer 0 is used in this calibration and must first be disabled, if necessary.

The returned result, n, is interpreted as follows:

- n = 10000 ⇒ clock running at 32 kHz
- n > 10000 ⇒ clock running slower than 32 kHz
- n < 10000 ⇒ clock running faster than 32 kHz

The returned value can be used to adjust the time interval value used to program a wake timer. If the required timer duration is T seconds, the count value N that must be specified in **vJPI_WakeTimerStart()** is given by N = (10000/n) x 32000 x T.

Note that before calling the calibration function, both wake timers (0 and 1) must be cleared using the function **u8JPI_WakeTimerFiredStatus()**, otherwise an incorrect result will be returned.

Parameters

None

Returns

Calibration measurement, n (see above)

3.7 Serial Peripheral Interface (SPI)

This section details the functions for controlling the Serial Peripheral Interface (SPI) on the JN5139/JN5148 device. The SPI allows high-speed synchronous data transfer between the JN5139/JN5148 and peripheral devices. The JN5139/JN5148 operates as a master on the SPI bus and all other devices connected to the SPI are expected to be slave devices under the control of the master's CPU.

The SPI master can be used to communicate with up to five attached peripherals, including the Flash memory. It can transfer 8, 16 or 32 bits without software intervention and can keep the slave select lines asserted between transfers, when required, to allow longer transfers to be performed.

As well as dedicated pins for Data In, Data Out, Clock and Slave Select 0, the SPI master can be configured to enable up to 4 more slave select lines which appear on DIO0 to DIO3. Slave-select 0 is assumed to be connected to Flash memory and is read during the boot sequence.

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

Function	Page
vJPI_SpiConfigure	126
vJPI_SpiReadConfiguration	128
vJPI_SpiRestoreConfiguration	129
vJPI_SpiSelect	130
vJPI_SpiStop	131
vJPI_SpiStartTransfer32	132
u32JPI_SpiReadTransfer32	133
vJPI_SpiStartTransfer16	134
u16JPI_SpiReadTransfer16	135
vJPI_SpiStartTransfer8	136
u8JPI_SpiReadTransfer8	137
bJPI_SpiPollBusy	138
vJPI_SpiWaitBusy	139
vJPI_SpiRegisterCallback	140

vJPI_SpiConfigure

```
void vJPI_SpiConfigure(uint8 u8SlaveEnable,
                       bool_t bLsbFirst,
                       bool_t bPolarity,
                       bool_t bPhase,
                       uint8 u8ClockDivider,
                       bool_t bInterruptEnable,
                       bool_t bAutoSlaveSelect);
```

Description

This function configures and enables the SPI master.

The function allows the number of extra SPI slaves (of the master) to be set. By default, there is one SPI slave (the Flash memory) with a dedicated IO pin for its select line. Depending on how many additional slaves are enabled, up to four more select lines can be set, which use DIO pins 0 to 3. For example, if two additional slaves are enabled, DIO 0 and 1 will be assigned. Note that once reserved for SPI use, DIO lines cannot be subsequently released by calling this function again (and specifying a smaller number of SPI slaves).

The following features are also configurable using this function:

- Data transfer order - whether the least significant bit is transferred first or last
- Clock polarity and phase, which together determine the SPI mode (0, 1, 2 or 3) and therefore the clock edge on which data is latched (for more information on SPI modes, refer to the *Jenie API User Guide (JN-UG-3042)*):
  - SPI Mode 0: polarity=0, phase=0
  - SPI Mode 1: polarity=0, phase=1
  - SPI Mode 2: polarity=1, phase=0
  - SPI Mode 3: polarity=1, phase=1
- Clock divisor - the value (in the range 1 to 63) used to derive the SPI clock from the 16-MHz base clock (the 16-MHz clock is divided by twice the specified value)
- SPI interrupt - generated when an API transfer has completed (SPI interrupts are handled by a callback function registered using **vJPI_SpiRegisterCallback()**). Note that interrupts are only worth using if the SPI clock frequency is much less than 16 MHz
- Automatic slave selection - enable the programmed slave-select line or lines (see **vJPI_SpiSelect()**) to be automatically asserted at the start of a transfer and de-asserted when the transfer completes. If not enabled, the slave-select lines will reflect the value set by **vJPI_SpiSelect()** directly.

Parameters

u8SlaveEnable	Number of extra SPI slaves to control. Valid values are 0 to 4 - higher values are truncated to 4
bLsbFirst	Enable/disable data transfer with the least significant bit (LSB) transferred first: TRUE - enable FALSE - disable

bPolarity	Clock polarity: TRUE - inverted FALSE - unchanged
bPhase	Phase: TRUE - latch on trailing edge of clock FALSE - latch on leading edge of clock
u8ClockDivider	Clock divisor in the range 0 to 63 - 16-MHz clock is divided by 2 x *u8ClockDivider,* but 0 is a special value used when no clock division is required (to obtain a 16-MHz SPI bus clock)
bInterruptEnable	Enable/disable interrupt when an SPI transfer has completed: TRUE - enable FALSE - disable
bAutoSlaveSelect	Enable/disable automatic slave selection: TRUE - enable FALSE - disable

Note that *bPolarity* and *bPhase* are named differently in the library header file **JPI.h**.

Returns

None

vJPI_SpiReadConfiguration

void vJPI_SpiReadConfiguration(
tSpiConfiguration ****ptConfiguration*);**

Description

This function obtains the current configuration of the SPI bus.

This function is intended to be used in a system where the SPI bus is used in multiple configurations to allow the state to be restored later using the function **vJPI_SpiRestoreConfiguration()**. Therefore, no knowledge is needed of the configuration details.

Parameters

**ptConfiguration*	Pointer to location to receive obtained SPI configuration

Returns

None

vJPI_SpiRestoreConfiguration

```
void vJPI_SpiRestoreConfiguration(
                        tSpiConfiguration *ptConfiguration);
```

Description

This function restores the SPI bus configuration using the configuration previously obtained using **vJPI_SpiReadConfiguration()**.

Parameters

**ptConfiguration*	Pointer to SPI configuration to be restored

Returns

None

vJPI_SpiSelect

```
void vJPI_SpiSelect(uint8 u8SlaveMask);
```

Description

This function sets the active slave-select line(s) to use.

The slave-select lines are asserted immediately if “Automatic Slave Selection” is disabled, or otherwise only during data transfers. The number of valid bits in *u8SlaveMask* depends on the setting of *u8SlaveEnable* in a previous call to **vJPI_SpiConfigure()**, as follows:

u8SlaveEnable	Valid bits in *u8SlaveMask*
0	Bit 0
1	Bits 0, 1
2	Bits 0, 1, 2
3	Bits 0, 1, 2, 3
4	Bits 0, 1, 2, 3, 4

Parameters

u8SlaveMask	Bitmap - one bit per slave-select line

Returns

None

vJPI_SpiStop

```
void vJPI_SpiStop(void);
```

Description

This function clears any active slave-select lines. It has the same effect as **vJPI_SpiSelect(0)**.

Parameters

None

Returns

None

vJPI_SpiStartTransfer32

```
void vJPI_SpiStartTransfer32(uint32 u32Out);
```

Description

This function starts a 32-bit data transfer to/from the selected slave(s).

It is assumed that **vJPI_SpiSelect()** has been called to set the slave(s) to communicate with. If interrupts are enabled for the SPI master, an interrupt will be generated when the transfer has completed.

Parameters

u32Out	32 bits of data to transmit

Returns

None

u32JPI_SpiReadTransfer32

```
uint32 u32JPI_SpiReadTransfer32(void);
```

Description

This function obtains the received data after a 32-bit SPI transfer has completed.

Parameters

None

Returns

Received data (32 bits)

vJPI_SpiStartTransfer16

void vJPI_SpiStartTransfer16(uint16 *u16Out***);**

Description

This function starts a 16-bit data transfer to/from the selected slave(s).

It is assumed that **vJPI_SpiSelect()** has been called to set the slave(s) to communicate with. If interrupts are enabled for the SPI master, an interrupt will be generated when the transfer has completed.

Parameters

u16Out	16 bits of data to transmit

Returns

None

u16JPI_SpiReadTransfer16

```
uint16 u16JPI_SpiReadTransfer16(void);
```

Description

This function obtains the received data after a 16-bit SPI transfer has completed.

Parameters

None

Returns

Received data (16 bits)

vJPI_SpiStartTransfer8

void vJPI_SpiStartTransfer8(uint8 *u8Out***);**

Description

This function starts an 8-bit transfer to/from the selected slaves(s).

It is assumed that **vJPI_SpiSelect()** has been called to set the slave(s) to communicate with. If interrupts are enabled for the SPI master, an interrupt will be generated when the transfer has completed.

Parameters

u8Out	8 bits of data to transmit

Returns

None

u8JPI_SpiReadTransfer8

```
uint8 u8JPI_SpiReadTransfer8(void);
```

Description

This function obtains the received data after a 8-bit SPI transfer has completed.

Parameters

None

Returns

Received data (8 bits)

bJPI_SpiPollBusy

```
bool_t bJPI_SpiPollBusy(void);
```

Description

This function polls the SPI master to determine whether it is currently busy performing a data transfer.

Parameters

None

Returns

TRUE if the SPI master is performing a transfer, FALSE otherwise

vJPI_SpiWaitBusy

```
void vJPI_SpiWaitBusy(void);
```

Description

This function waits for the SPI master to complete a transfer and then returns.

Parameters

None

Returns

None

vJPI_SpiRegisterCallback

void vJPI_SpiRegisterCallback(
PR_HWINT_APPCALLBACK *prSpiCallback***);**

Description

This function registers an application callback that will be called when the SPI interrupt is triggered.

The registered callback function is only preserved during sleep with memory held. If RAM is powered off during sleep and interrupts are required, the callback function must be re-registered before calling **u32JPI_Init()** on waking.

Interrupt handling is described in Appendix E.

Parameters

prSpiCallback	Pointer to function to be called when SPI interrupt occurs

Returns

None

3.8 Serial Interface (2-Wire)

This section details the functions for controlling the 2-Wire Serial Interface (SI) master on the JN5139/JN5148 device. The Serial Interface is logic-compatible with similar interfaces such as I^{2}C and SMbus.

The SI master on the JN5139/JN5148 can implement bi-directional communication with a slave device on the SI bus. Note that the SI 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.

When enabled, this interface uses DIO14 as a clock and DIO15 as a bi-directional data line. The clock is scaled from the 16-MHz system clock.

Note: Before implementing data transfers on the SI bus, you are strongly advised to study the protocol detailed in the I^{2}C Specification (available from www.nxp.com).

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

Function	Page
vJPI_SiConfigure	142
vJPI_SiSetCmdReg	143
vJPI_SiWriteData8	145
vJPI_SiWriteSlaveAddr	146
u8JPI_SiReadData8	147
bJPI_SiPollBusy	148
bJPI_SiPollTransferInProgress	149
bJPI_SiPollRxNack	150
bJPI_SiPollArbitrationLost	151
vJPI_SiRegisterCallback	152

vJPI_SiConfigure

```
void vJPI_SiConfigure(bool_t bSiEnable,
                      bool_t bInterruptEnable,
                      uint16 u16PreScaler);
```

Description

This function is used to enable/disable and configure the 2-wire Serial Interface (SI) master on the JN5139/JN5148 device. This function must be called to enable the SI block before any other SI master function is called.

The operating frequency, derived from the 16-MHz system clock using the specified prescaler *u16PreScaler*, is given by:

Operating frequency = 16/[(*PreScaler* + 1) x 5] MHz

SI interrupts are handled by a callback function registered using the function **vJPI_SiRegisterCallback()**.

Parameters

bSiEnable	Enable Serial Interface (master): TRUE - enable FALSE - disable
bInterruptEnable	Enable Serial interface interrupt: TRUE - enable FALSE - disable
u16PreScaler	16-bit clock prescaler (see above)

Returns

None

vJPI_SiSetCmdReg

```
void vJPI_SiSetCmdReg(bool_t bSetSTA,
                      bool_t bSetSTO,
                      bool_t bSetRD,
                      bool_t bSetWR,
                      bool_t bSetAckCtrl,
                      bool_t bSetIACK);
```

Description

This function configures the combination of I^{2}C-protocol commands for a transfer on the SI bus and starts the transfer of the data held in the SI master's transmit buffer.

Up to four commands can be used to perform an I^{2}C-protocol transfer - Start, Stop, Write, Read. This function allows these commands to be combined to form a complete or partial transfer sequence. The valid command combinations that can be specified are summarised below.

Start	Stop	Read	Write	Resulting Instruction to SI Bus
0	0	0	0	No active command (idle)
1	0	0	1	Start followed by Write
1	1	0	1	Start followed by Write followed by Stop
0	1	1	0	Read followed by Stop
0	1	0	1	Write followed by Stop
0	0	0	1	Write only
0	0	1	0	Read only
0	1	0	0	Stop only

The above command combinations will result in the function returning TRUE, while command combinations that are not in the above list are invalid and will result in a FALSE return code.

The function must be called immediately after **vJPI_SiWriteSlaveAddr()**, which puts the destination slave address (for the subsequent data transfer) into the transmit buffer. It must then be called immediately after **vJPI_SiWriteData8()** to start the transfer of data (from the transmit buffer).

To implement a data transfer on the SI bus, you must follow the process described in the I^{2}C Specification.

Parameters

bSetSTA	Generate START bit to gain control of the SI bus (must not be enabled with STOP bit): E_JPI_SI_START_BIT E_JPI_SI_NO_START_BIT
bSetSTO	Generate STOP bit to release control of the SI bus (must not be enabled with START bit): E_JPI_SI_STOP_BIT E_JPI_SI_NO_STOP_BIT
bSetRD	Read from slave (cannot be enabled with slave write): E_JPI_SI_SLAVE_READ E_JPI_SI_NO_SLAVE_READ
bSetWR	Write to slave (cannot be enabled with slave read): E_JPI_SI_SLAVE_WRITE E_JPI_SI_NO_SLAVE_WRITE
bSetAckCtrl	Send ACK or NACK to slave after each byte read: E_JPI_SI_SEND_ACK (to indicate ready for next byte) E_JPI_SI_SEND_NACK (to indicate no more data required)
bSetIACK	Generate interrupt acknowledge (set to clear pending interrupt): E_JPI_SI_IRQ_ACK E_JPI_SI_NO_IRQ_ACK

Returns

None

vJPI_SiWriteData8

void vJPI_SiWriteData8(uint8 *u8Out***);**

Description

This function writes a single data byte to the Transmit register of the Serial Interface.

The contents of the transmit buffer will not be transmitted on the SI bus until the function **vJPI_SiSetCmdReg()** is called.

Parameters

u8Out	8 bits of data to transmit

Returns

None

vJPI_SiWriteSlaveAddr

```
void vJPI_SiWriteSlaveAddr(uint8 u8SlaveAddress,
                           bool_t bReadStatus);
```

Description

This function is used in setting up communication with a slave device. In this function, you must specify the address of the slave (see below) and the operation (read or write) to be performed on the slave. The function puts this information in the SI master's transmit buffer, but the information will be not transmitted on the SI bus until the function **vJPI_SiSetCmdReg()** is called.

You must specify a 7-bit slave address and the operation (read or write) to be performed on the slave. To implement a data transfer on the SI bus, you must follow the process described in the I^{2}C Specification.

Parameters

u8SlaveAddress	7-bit slave address
bReadStatus	Operation to perform on slave (read or write): TRUE to configure a read FALSE to configure a write

Returns

None

u8JPI_SiReadData8

```
uint8 u8JPI_SiReadData8(void);
```

Description

This function obtains 8-bit data received over the Serial Interface bus.

Parameters

None

Returns

Data read from receive buffer of SI master

bJPI_SiPollBusy

```
bool_t bJPI_SiPollBusy(void);
```

Description

This function checks whether the SI bus is busy (could be in use by another master).

Parameters

None

Returns

TRUE if busy, FALSE otherwise

bJPI_SiPollTransferInProgress

```
bool_t bJPI_SiPollTransferInProgress(void);
```

Description

This function checks whether a transfer is in progress on the SI bus.

Parameters

None

Returns

TRUE if a transfer is in progress, FALSE otherwise

bJPI_SiPollRxNack

```
bool_t bJPI_SiPollRxNack(void);
```

Description

This function checks whether a NACK or an ACK has been received from the slave device. If a NACK has been received, this indicates that the SI master should stop sending data to the slave.

Parameters

None

Returns

TRUE if NACK has occurred

FALSE if ACK has occurred

bJPI_SiPollArbitrationLost

```
bool_t bJPI_SiPollArbitrationLost(void);
```

Description

This function checks whether arbitration has been lost (by the local master) on the SI bus.

Parameters

None

Returns

TRUE if arbitration loss has occurred, FALSE otherwise

vJPI_SiRegisterCallback

```
void vJPI_SiRegisterCallback(
            PR_HWINT_APPCALLBACK prSiCallback);
```

Description

This function registers a user-defined callback function that will be called when a Serial Interface interrupt is triggered on the SI master.

Note that this function can be used to register the callback function for a SI slave as well as for the SI master. The SI interrupt handler will determine whether a SI interrupt has been generated on a master or slave, and then invoke the relevant callback function.

The registered callback function is only preserved during sleep modes in which RAM remains powered. If RAM is powered off during sleep and interrupts are required, the callback function must be re-registered before calling **u32JPI_Init()** on waking.

Interrupt handling is described in Appendix E.

Parameters

prSiCallback	Pointer to function to be called when Serial Interface interrupt occurs

Returns

None

3.9 Intelligent Peripheral Interface

This section details the functions for controlling the Intelligent Peripheral (IP) interface of the JN5139/JN5148 device.

The Intelligent Peripheral interface is a SPI slave interface and uses pins shared with Digital IO signals DIO14-18. The interface is designed to allow message passing and data transfer. Data received and transmitted on the IP interface is copied directly to and from a dedicated area of memory without intervention from the CPU. This memory area, the Intelligent Peripheral memory block, contains receive and transmit buffers, each comprising sixty-three 32-bit words.

For more details of the data message format, refer to the Jennic data sheet for the relevant wireless microcontroller.

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

Function	Page
vJPI_IpEnable	154
bJPI_IpSendData	155
bJPI_IpReadData	156
bJPI_IpTxDone	157
bJPI_IpRxDataAvailable	158
vJPI_IpRegisterCallback	159

vJPI_IpEnable

void vJPI_IpEnable(bool_t *bTxEdge***,**
bool_t *bRxEdge***,**
bool_t *bEndian***);**

Description

This function initialises and enables the Intelligent Peripheral (IP) interface on the JN5139/JN5148 device.

The function allows the clock edges to be selected on which receive data will be sampled and transmit data will be changed (but see Caution below). It also allows Intelligent Peripheral interrupts to be enabled/disabled.

The function also requires the byte order (Big or Little Endian) of the data for the IP interface to be specified.

Caution: *Only one mode of the IP interface (SPI mode 0) is supported, in which data is transmitted on a negative clock edge and received on a positive clock edge. The parameters bTxEdge and bRxEdge must be set accordingly (both to 0).*

Parameters

bTxEdge	Clock edge that transmit data is changed on (see Caution): E_JPI_IP_TXPOS_EDGE (data changed on +ve edge, to be sampled on -ve edge) E_JPI_IP_TXNEG_EDGE (data changed on -ve edge, to be sampled on +ve edge)
bRxEdge	Clock edge that receive data is sampled on (see Caution): E_JPI_IP_RXPOS_EDGE (data sampled on +ve edge) E_JPI_IP_RXNEG_EDGE (data sampled on -ve edge)
bEndian	Byte order (Big or Little Endian) of data over the IP interface: E_JPI_IP_BIG_ENDIAN E_JPI_IP_LITTLE_ENDIAN

Returns

None

bJPI_IpSendData

```
bool_t bJPI_IpSendData(uint8 u8Length,
                       uint8 *pau8Data);
```

Description

This function is used to indicate that data is ready to be transmitted across the IP interface to the remote processor (the SPI master).

The function requires the data length to be specified, as well as a pointer to the buffer containing the data. The data should be stored in the buffer according to the byte order (Big or Little Endian) specified in the function **vJPI_IpEnable()**.

The function copies the specified data to a local Transmit buffer, ready to be sent when the master device initiates the transfer. The IP_INT pin is also asserted to indicate to the master that data is ready to be sent.

The data length is transmitted in the first 32-bit word of the data payload. It is the responsibility of the SPI master receiving the data to retrieve the data length from the payload.

Parameters

u8Length	Length of data to be sent (in 32-bit words)
**pau8Data*	Pointer to a buffer containing the data to be sent

Returns

TRUE if successful, FALSE if unable to send

bJPI_IpReadData

bool_t bJPI_IpReadData(uint8 ****pu8Length*,**
uint8 ****pau8Data*);**

Description

This function is used to read data received from the remote processor (the SPI master).

The function must provide a pointer to a buffer to receive the data and a pointer to a buffer to receive the data length.

Data is stored in the specified buffer according to the specified byte order (Big or Little Endian) specified in the function **vJPI_IpEnable()**.

After the data has been read, the IP interface will indicate to the SPI master that the interface is ready to receive more data.

Parameters

**pu8Length*	Pointer to length of buffer to receive data (in 32-bit words)
**pau8Data*	Pointer to buffer to receive data

Returns

TRUE if data read successfully, FALSE if unable to read

bJPI_IpTxDone

```
bool_t bJPI_IpTxDone(void);
```

Description

This function checks whether data copied to the local Transmit buffer has been sent to the remote processor (the SPI master).

Parameters

None

Returns

TRUE if data sent, FALSE if incomplete

bJPI_IpRxDataAvailable

```
bool_t bJPI_IpRxDataAvailable(void);
```

Description

This function checks whether data from the remote processor (the SPI master) has been received in the local Receive buffer.

Parameters

None

Returns

TRUE if Receive buffer contains data, FALSE otherwise

vJPI_IpRegisterCallback

```
void vJPI_IpRegisterCallback(
        PR_HWINT_APPCALLBACK prIpCallback);
```

Description

This function registers an application callback that will be called when the Intelligent Peripheral interrupt is triggered. The Interrupt is triggered when either a transmit or receive transaction has completed.

The registered callback function is only preserved during sleep modes in which RAM remains powered. If RAM is powered off during sleep and interrupts are required, the callback function must be re-registered before calling **u32JPI_Init()** on waking.

Interrupt handling is described in Appendix E.

Parameters

prIpCallback	Pointer to function to be called when Intelligent Peripheral interrupt occurs

Returns

None

Appendices

The appendices contains all the ancillary information that you need in order to use the functions of the Jenie API. This information includes global parameters, enumerations, data types, structures, events and interrupts. In addition, functions and network parameters of the JenNet layer (which sits below Jenie in the stack) are described.

A. Global Network Parameters

This appendix details the global network parameters that can be set in the **vJenie_CbConfigureNetwork()** callback function (otherwise their default values will be used). For further information on using some of these variables, refer to the relevant appendix of the *Jenie API User Guide (JN-UG-3042)*.

Parameter Name	**Description**	**Default Value**	**Range**
gJenie_PanID	16-bit PAN ID to identify network (if no existing network with same PAN ID). **Co-ordinator only**	0xAAAA	0-0xFFFE
gJenie_NetworkApplicationID	32-bit Network Application ID used to identify and form network.	0xAAAA AAAA	0-0xFFFFFFFF
gJenie_Channel	The 2.4-GHz channel to be used by the network, or auto-scan (see *gJenie_ScanChannels* below). **Co-ordinator only**	0	0: Auto-scan 11-26: Channel
gJenie_ScanChannels	Bitmap (32 bits) of the set of channels to consider when performing an auto-scan of the 2.4-GHz band for a suitable channel to use. The Co-ordinator will select the quietest channel from those available (auto-scan must have been enabled via *gJenie_Channel.*). Other node types will scan the possible channels to search for network.	0x07FFF800 (all channels)	0x00000800 - 0x07FFF800 (Bit 11 set ⇒ Ch 11, Bit 12 set ⇒ Ch 12,...)
gJenie_MaxChildren	Maximum number of children the node can have. **Co-ordinator and Routers only**	10	0-16
gJenie_MaxSleepingChildren	Maximum number of children that can be End Devices (nodes capable of sleeping). This value must be less than or equal to *gJenie_MaxChildren*. The remaining child nodes are reserved exclusively for Routers, although any number of children can be Routers. **Co-ordinator and Routers only**	8	0- *gJenie_MaxChildren*

Table 1: Global Network Parameters

gJenie_MaxBcastTTL	Determines the maximum number of hops that a broadcast message sent from the local node can make. Set this value to one less than the desired maximum (so the value 0 corre-sponds to one hop).	5	0-255
gJenie_MaxFailedPkts	Number of missed communications (MAC acknowledgments) before parent considered to be lost (and node must try to find a new parent).	5	0-255 Zero value disables the feature
gJenie_RoutingEnabled	Enables/disables routing capability of the node (must be disabled for End Devices).	0	0: Disable routing 1: Enable routing For End Devices, always set to 0
gJenie_RoutingTableSize	Number of elements in array used to store the Routing table. Should be set to a value slightly larger than the maximum number of network nodes, to allow for nodes leaving and joining. **Co-ordinator and Routers only**	-	0-1000 Note that the upper limit may be restricted by the amount of available RAM. Each Routing table entry uses 12 bytes.
gJenie_RoutingTableSpace	Pointer to the Routing table array in memory. The Routing table is an array of structures of type **tsJenieRoutingTable,** where this array is declared in the application. **Co-ordinator and Routers only**	NULL	-
gJenie_RouterPingPeriod	Time between auto-pings generated by a Router (to its parent). Set in units of 100 ms. The same value should be set in all routing nodes in the network. **Co-ordinator and Routers only**	50 (5 seconds)	0-6553 Zero value disables pings. Non-zero val-ues below 50 are not recommended
gJenie_EndDevicePingInterval	Number of sleep cycles between auto-pings generated by an End Device (to its parent). **End Devices only**	1	0-255 Zero value disables pings
gJenie_EndDeviceScanSleep	Amount of time following a failed scan that an End Device waits (sleeps) before starting another scan. Set in milliseconds. **End Devices only**	10000 or 0x2710 (10 seconds)	0xC8-0xFFFFFFEB Values below 0x3E8 (1 second) are not recommended for large networks
gJenie_EndDevicePollPeriod	Time between auto-poll data requests sent from an End Device (while awake) to its parent. Set in units of 100 ms. **End Devices only**	50 or 0x32 (5 seconds)	0-0xFFFFFFFF Zero value disables auto-polling

Table 1: Global Network Parameters

gJenie_EndDeviceChildActivityTimeout	Timeout period for communication (excluding data polling) from an End Device child. If no message is received from the End Device within this period, the child is assumed lost and is removed from the Neighbour table (and Routing tables higher in the network). **Co-ordinator and Routers only**	0 (Timeout disabled)	0-0xFFFFFFFF Timeout is value set multiplied by 100 ms 0 disables the timeout but this is not advised, as child slots may fill with inactive End Devices, preventing other devices from joining.
gJenie_RecoverFromJpdm	Indicates whether network context data is to be recovered from external non-volatile memory during a cold start following power loss to the on-chip memory. Data must have been previously saved to external memory using **vJPDM_SaveContext()**.	0	0: Disable recovery 1: Enable recovery
gJenie_RecoverChildrenFromJpdm	Enables the recovery of child/neighbour table when restoring context data from non-volatile memory. Context recovery must also be enabled using *gJenie_RecoverFromJpdm*. **Co-ordinator and Routers only**	1	0: Disable recovery 1: Enable recovery
gJpdmSector	Number of sector where context will be saved in external non-volatile memory.	3	Positive integer
gJpdmSectorSize	Size of sector where context data will be saved in external non-volatile memory.	32768 (32K)	Size in bytes
gJpdmFlashType	Type of Flash memory device used as external non-volatile memory.	Auto-detect	E_FL_CHIP_ST_M25P10_A E_FL_CHIP_SST_25VF010 E_FL_CHIP_ATMEL_AT25F512 E_FL_CHIP_CUSTOM E_FL_CHIP_AUTO
gJpdmFlashFuncTable	Pointer to function table for custom Flash memory device.	NULL	-

Table 1: Global Network Parameters

B. Enumerated Types and Defines

The following enumerated types and defines are used by the Jenie API functions. They are presented as those relevant to the core Jenie functions, included in the header file **Jenie.h** and detailed in Chapter 2, and those relevant to the Jenie Integrated Peripheral functions, included in the header file **JPI.h** and detailed in Chapter 3.

B.1 For Core Jenie Functions

Return Status (teJenieStatusCode)

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

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

Node Type (teJenieDeviceType)

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

Component (teJenieComponent)

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

Radio Transceiver (teJenieRadioPower)

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

Poll Status (teJeniePollStatus)

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

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

TXOPTION #defines

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

Table 2: TXOPTION #defines

B.2 For Jenie Peripheral Interface Functions

Separate sets of JPI data types and enumerations are provided for the JN5139 and JN5148 devices. These are detailed in the sub-sections below.

B.2.1 JN5139 Versions

Peripheral (teJPI_Device)

Device types, used to identify interrupt source:

```
typedef enum
{
    E_JPI_DEVICE_TICK_TIMER = 0, /* Tick timer */
    E_JPI_DEVICE_SYSCTRL = 2,    /* System controller */
    E_JPI_DEVICE_BBC,            /* Baseband controller */
    E_JPI_DEVICE_AES,            /* Encryption engine */
    E_JPI_DEVICE_PHYCTRL,        /* Phy controller */
    E_JPI_DEVICE_UART0,          /* UART 0 */
    E_JPI_DEVICE_UART1,          /* UART 1 */
    E_JPI_DEVICE_TIMER0,         /* Timer 0 */
    E_JPI_DEVICE_TIMER1,         /* Timer 1 */
    E_JPI_DEVICE_SI,             /* Serial Interface (2 wire) */
    E_JPI_DEVICE_SPIM,           /* SPI master */
    E_JPI_DEVICE_INTPER,         /* Intelligent peripheral */
    E_JPI_DEVICE_ANALOGUE        /* Analogue peripherals */
} teJPI_Device;
```

System Control Item (teJPI_Item)

Individual System Controller interrupts:

```
typedef enum
{
    E_JPI_SYSCTRL_WK0   = 26,    /* Wake timer 0 */
    E_JPI_SYSCTRL_WK1   = 27,    /* Wake timer 1 */
    E_JPI_SYSCTRL_COMP0 = 28,    /* Comparator 0 */
    E_JPI_SYSCTRL_COMP1 = 29,    /* Comparator 1 */
} teJPI_Item;
```

Analogue Peripheral (teJPI_AnalogueChannel)

```
typedef enum {
    E_JPI_ANALOGUE_DAC_0,
    E_JPI_ANALOGUE_DAC_1,
    E_JPI_ANALOGUE_ADC
} teJPI_AnalogueChannel;
```

Comparator (teJPI_Comparator)

```
typedef enum {
    E_JPI_COMPARATOR_0,
    E_JPI_COMPARATOR_1
} teJPI_Comparator;
```

Timer (teJPI_Timer)

```
typedef enum {
    E_JPI_TIMER_0,
    E_JPI_TIMER_1
} teJPI_Timer;
```

Timer Mode (teJPI_TimerMode)

```
typedef enum {
    E_JPI_TIMER_MODE_SINGLESHOT,
    E_JPI_TIMER_MODE_REPEATING,
    E_JPI_TIMER_MODE_DELTASIGMA,
    E_JPI_TIMER_MODE_DELTASIGMARTZ
} teJPI_TimerMode;
```

Timer Clock Type (teJPI_TimerClockType)

```
typedef enum {
    E_JPI_TIMER_CLOCK_INTERNAL_NORMAL,
    E_JPI_TIMER_CLOCK_INTERNAL_INVERTED,
    E_JPI_TIMER_CLOCK_EXTERNAL_NORMAL,
    E_JPI_TIMER_CLOCK_EXTERNAL_INVERTED
} teJPI_TimerClockType;
```

Analogue Peripheral (ADC and DACs) #defines

```
#define E_JPI_ADC_SRC_ADC_1        0
#define E_JPI_ADC_SRC_ADC_2        1
#define E_JPI_ADC_SRC_ADC_3        2
#define E_JPI_ADC_SRC_ADC_4        3
#define E_JPI_ADC_SRC_TEMP         4
#define E_JPI_ADC_SRC_VOLT         5
#define E_JPI_AP_REGULATOR_ENABLE  TRUE
#define E_JPI_AP_REGULATOR_DISABLE FALSE
#define E_JPI_AP_SAMPLE_2          0
#define E_JPI_AP_SAMPLE_4          1
#define E_JPI_AP_SAMPLE_6          2
#define E_JPI_AP_SAMPLE_8          3
#define E_JPI_AP_CLOCKDIV_2MHZ     0
#define E_JPI_AP_CLOCKDIV_1MHZ     1
#define E_JPI_AP_CLOCKDIV_500KHZ   2
#define E_JPI_AP_CLOCKDIV_250KHZ   3
#define E_JPI_AP_INPUT_RANGE_2     TRUE
#define E_JPI_AP_INPUT_RANGE_1     FALSE
#define E_JPI_AP_GAIN_2            TRUE
#define E_JPI_AP_GAIN_1            FALSE
#define E_JPI_AP_EXTREF            TRUE
#define E_JPI_AP_INTREF            FALSE
#define E_JPI_ADC_CONVERT_ENABLE   TRUE
#define E_JPI_ADC_CONVERT_DISABLE  FALSE
#define E_JPI_ADC_CONTINUOUS       TRUE
#define E_JPI_ADC_SINGLE_SHOT      FALSE
#define E_JPI_AP_INT_ENABLE        TRUE
#define E_JPI_AP_INT_DISABLE       FALSE
#define E_JPI_DAC_RETAIN_ENABLE    TRUE
#define E_JPI_DAC_RETAIN_DISABLE   FALSE
```

Comparator #defines

```
#define E_JPI_COMP_HYSTERESIS_0MV  0
#define E_JPI_COMP_HYSTERESIS_5MV  1
#define E_JPI_COMP_HYSTERESIS_10MV 2
#define E_JPI_COMP_HYSTERESIS_20MV 3
#define E_JPI_AP_COMPARATOR_MASK_1 1
#define E_JPI_AP_COMPARATOR_MASK_2 2
#define E_JPI_COMP_SEL_EXT         0x00
#define E_JPI_COMP_SEL_DAC         0x01
#define E_JPI_COMP_SEL_BANDGAP     0x03
```

UART #defines

```
#define E_JPI_UART_0                0
#define E_JPI_UART_1                1
#define E_JPI_UART_RATE_4800        0
#define E_JPI_UART_RATE_9600        1
#define E_JPI_UART_RATE_19200       2
#define E_JPI_UART_RATE_38400       3
#define E_JPI_UART_RATE_76800       4
#define E_JPI_UART_RATE_115200      5
#define E_JPI_UART_WORD_LEN_5       0
#define E_JPI_UART_WORD_LEN_6       1
#define E_JPI_UART_WORD_LEN_7       2
#define E_JPI_UART_WORD_LEN_8       3
#define E_JPI_UART_FIFO_LEVEL_1     0
#define E_JPI_UART_FIFO_LEVEL_4     1
#define E_JPI_UART_FIFO_LEVEL_8     2
#define E_JPI_UART_FIFO_LEVEL_14    3
#define E_JPI_UART_LS_ERROR         0x80
#define E_JPI_UART_LS_TEMT          0x40
#define E_JPI_UART_LS_THRE          0x20
#define E_JPI_UART_LS_BI            0x10
#define E_JPI_UART_LS_FE            0x08
#define E_JPI_UART_LS_PE            0x04
#define E_JPI_UART_LS_OE            0x02
#define E_JPI_UART_LS_DR            0x01
#define E_JPI_UART_MS_DCTS          0x01
#define E_JPI_UART_INT_MODEM        0
#define E_JPI_UART_INT_TX           1
#define E_JPI_UART_INT_RXDATA       2
#define E_JPI_UART_INT_RXLINE       3
#define E_JPI_UART_INT_TIMEOUT      6
#define E_JPI_UART_TX_RESET         TRUE
#define E_JPI_UART_RX_RESET         TRUE
#define E_JPI_UART_TX_ENABLE        FALSE
#define E_JPI_UART_RX_ENABLE        FALSE
#define E_JPI_UART_EVEN_PARITY      TRUE
#define E_JPI_UART_ODD_PARITY       FALSE
#define E_JPI_UART_PARITY_ENABLE    TRUE
#define E_JPI_UART_PARITY_DISABLE   FALSE
#define E_JPI_UART_1_STOP_BIT       TRUE
#define E_JPI_UART_2_STOP_BITS      FALSE
#define E_JPI_UART_RTS_HIGH         TRUE
#define E_JPI_UART_RTS_LOW          FALSE
```

Timer #defines

```
#define E_JPI_TIMER_INT_PERIOD     1
#define E_JPI_TIMER_INT_RISE       2
#define E_JPI_TIMER_INTERRUPT_RISING   (0U)
#define E_JPI_TIMER_INTERRUPT_COMPLETE (1U)
```

Wake Timer #defines

```
#define E_JPI_WAKE_TIMER_0         0
#define E_JPI_WAKE_TIMER_1         1
#define E_JPI_WAKE_TIMER_MASK_0    1
#define E_JPI_WAKE_TIMER_MASK_1    2
```

SPI #defines

```
#define E_JPI_SPIM_MSB_FIRST       FALSE
#define E_JPI_SPIM_LSB_FIRST       TRUE
#define E_JPI_SPIM_TXPOS_EDGE      FALSE
#define E_JPI_SPIM_TXNEG_EDGE      TRUE
#define E_JPI_SPIM_RXPOS_EDGE      FALSE
#define E_JPI_SPIM_RXNEG_EDGE      TRUE
#define E_JPI_SPIM_INT_ENABLE      TRUE
#define E_JPI_SPIM_INT_DISABLE     FALSE
#define E_JPI_SPIM_AUTOSLAVE_ENBL  TRUE
#define E_JPI_SPIM_AUTOSLAVE_DSABL FALSE
#define E_JPI_SPIM_SLAVE_ENBLE_0   0x1
#define E_JPI_SPIM_SLAVE_ENBLE_1   0x2
#define E_JPI_SPIM_SLAVE_ENBLE_2   0x4
#define E_JPI_SPIM_SLAVE_ENBLE_3   0x8
```

Serial Interface (SI) #defines

```
#define E_JPI_SI_INT_AL            0x20
#define E_JPI_SI_SLAVE_RW_SET      FALSE
#define E_JPI_SI_START_BIT         TRUE
#define E_JPI_SI_NO_START_BIT      FALSE
#define E_JPI_SI_STOP_BIT          TRUE
#define E_JPI_SI_NO_STOP_BIT       FALSE
#define E_JPI_SI_SLAVE_READ        TRUE
#define E_JPI_SI_NO_SLAVE_READ     FALSE
#define E_JPI_SI_SLAVE_WRITE       TRUE
#define E_JPI_SI_NO_SLAVE_WRITE    FALSE
#define E_JPI_SI_SEND_ACK          FALSE
#define E_JPI_SI_SEND_NACK         TRUE
#define E_JPI_SI_IRQ_ACK           TRUE
#define E_JPI_SI_NO_IRQ_ACK        FALSE
```

Intelligent Peripheral (IP) #defines

```
#define E_JPI_IP_MAX_MSG_SIZE      0x3F
#define E_JPI_IP_TXPOS_EDGE        FALSE
#define E_JPI_IP_TXNEG_EDGE        TRUE
#define E_JPI_IP_RXPOS_EDGE        FALSE
#define E_JPI_IP_RXNEG_EDGE        TRUE
#define E_JPI_IP_BIG_ENDIAN        TRUE
#define E_JPI_IP_LITTLE_ENDIAN     FALSE
```

DIO #defines

```
#define E_JPI_DIO0_INT             0x00000001
#define E_JPI_DIO1_INT             0x00000002
#define E_JPI_DIO2_INT             0x00000004
#define E_JPI_DIO3_INT             0x00000008
#define E_JPI_DIO4_INT             0x00000010
#define E_JPI_DIO5_INT             0x00000020
#define E_JPI_DIO6_INT             0x00000040
#define E_JPI_DIO7_INT             0x00000080
#define E_JPI_DIO8_INT             0x00000100
#define E_JPI_DIO9_INT             0x00000200
#define E_JPI_DIO10_INT            0x00000400
#define E_JPI_DIO11_INT            0x00000800
#define E_JPI_DIO12_INT            0x00001000
#define E_JPI_DIO13_INT            0x00002000
#define E_JPI_DIO14_INT            0x00004000
#define E_JPI_DIO15_INT            0x00008000
#define E_JPI_DIO16_INT            0x00010000
#define E_JPI_DIO17_INT            0x00020000
#define E_JPI_DIO18_INT            0x00040000
#define E_JPI_DIO19_INT            0x00080000
#define E_JPI_DIO20_INT            0x00100000
```

B.2.2 JN5148 Versions

Peripheral (teJPI_Device)

Device types, used to identify interrupt source:

```
typedef enum
{
    E_JPI_DEVICE_AUDIOFIFO  = 0,    /* Sample FIFO */
    E_JPI_DEVICE_I2S        = 1,    /* 4-wire DAI */
    E_JPI_DEVICE_SYSCTRL    = 2,    /* System controller */
    E_JPI_DEVICE_BBC        = 3,    /* Baseband controller */
    E_JPI_DEVICE_AES        = 4,    /* Encryption engine */
    E_JPI_DEVICE_PHYCTRL    = 5,    /* Phy controller */
    E_JPI_DEVICE_UART0      = 6,    /* UART 0 */
    E_JPI_DEVICE_UART1      = 7,    /* UART 1 */
    E_JPI_DEVICE_TIMER0     = 8,    /* Timer 0 */
    E_JPI_DEVICE_TIMER1     = 9,    /* Timer 1 */
    E_JPI_DEVICE_SI         = 10,   /* 2-wire SI */
    E_JPI_DEVICE_SPIM       = 11,   /* SPI master */
    E_JPI_DEVICE_INTPER     = 12,   /* Intelligent peripheral */
    E_JPI_DEVICE_ANALOGUE   = 13,   /* Analogue peripherals */
    E_JPI_DEVICE_TIMER2     = 14,   /* Timer 2 */
    E_JPI_DEVICE_TICK_TIMER = 15    /* Tick timer */
} teJPI_Device;
```

System Control Item (teJPI_Item)

Individual System Controller interrupts:

```
typedef enum
{
    E_JPI_SYSCTRL_PC0   = 22,   /* Pulse Counter 0 */
    E_JPI_SYSCTRL_PC1   = 23,   /* Pulse Counter 1 */
    E_JPI_SYSCTRL_VFES  = 24,   /* VBO Falling  */
    E_JPI_SYSCTRL_VRES  = 25,   /* VBO Rising */
    E_JPI_SYSCTRL_WK0   = 26,   /* Wake timer 0 */
    E_JPI_SYSCTRL_WK1   = 27,   /* Wake timer 1 */
    E_JPI_SYSCTRL_COMP0 = 28,   /* Comparator 0 */
    E_JPI_SYSCTRL_COMP1 = 29,   /* Comparator 1 */
    E_JPI_SYSCTRL_RNDES = 30,   /* Random number generator */
    E_JPI_SYSCTRL_CKES  = 31    /* Clock change  */
} teJPI_Item;
```

Analogue Peripheral (teJPI_AnalogueChannel)

```
typedef enum {
    E_JPI_ANALOGUE_DAC_0,
    E_JPI_ANALOGUE_DAC_1,
    E_JPI_ANALOGUE_ADC
} teJPI_AnalogueChannel;
```

Comparator (teJPI_Comparator)

```
typedef enum {
    E_JPI_COMPARATOR_0,
    E_JPI_COMPARATOR_1
} teJPI_Comparator;
```

Timer (teJPI_Timer)

```
typedef enum {
    E_JPI_TIMER_0,
    E_JPI_TIMER_1,
    E_JPI_TIMER_2
} teJPI_Timer;
```

Timer Mode (teJPI_TimerMode)

```
typedef enum {
    E_JPI_TIMER_MODE_SINGLESHOT,
    E_JPI_TIMER_MODE_REPEATING,
    E_JPI_TIMER_MODE_DELTASIGMA,
    E_JPI_TIMER_MODE_DELTASIGMARTZ
} teJPI_TimerMode;
```

Timer Clock Type (teJPI_TimerClockType)

```
typedef enum {
    E_JPI_TIMER_CLOCK_INTERNAL_NORMAL,
    E_JPI_TIMER_CLOCK_INTERNAL_INVERTED,
    E_JPI_TIMER_CLOCK_EXTERNAL_NORMAL,
    E_JPI_TIMER_CLOCK_EXTERNAL_INVERTED
} teJPI_TimerClockType;
```

Analogue Peripheral (ADC and DACs) #defines

```
#define E_JPI_ADC_SRC_ADC_1            0
#define E_JPI_ADC_SRC_ADC_2            1
#define E_JPI_ADC_SRC_ADC_3            2
#define E_JPI_ADC_SRC_ADC_4            3
#define E_JPI_ADC_SRC_TEMP             4
#define E_JPI_ADC_SRC_VOLT             5
#define E_JPI_AP_REGULATOR_ENABLE      TRUE
#define E_JPI_AP_REGULATOR_DISABLE     FALSE
#define E_JPI_AP_SAMPLE_2              0
#define E_JPI_AP_SAMPLE_4              1
#define E_JPI_AP_SAMPLE_6              2
#define E_JPI_AP_SAMPLE_8              3
#define E_JPI_AP_CLOCKDIV_2MHZ         0
#define E_JPI_AP_CLOCKDIV_1MHZ         1
#define E_JPI_AP_CLOCKDIV_500KHZ       2
#define E_JPI_AP_CLOCKDIV_250KHZ       3
#define E_JPI_AP_INPUT_RANGE_2         TRUE
#define E_JPI_AP_INPUT_RANGE_1         FALSE
#define E_JPI_AP_GAIN_2                TRUE
#define E_JPI_AP_GAIN_1                FALSE
#define E_JPI_AP_EXTREF                TRUE
#define E_JPI_AP_INTREF                FALSE
#define E_JPI_ADC_CONVERT_ENABLE       TRUE
#define E_JPI_ADC_CONVERT_DISABLE      FALSE
#define E_JPI_ADC_CONTINUOUS           TRUE
#define E_JPI_ADC_SINGLE_SHOT          FALSE
#define E_JPI_AP_CAPT_INT_ENABLE       0x1U
#define E_JPI_AP_INT_ENABLE            TRUE
#define E_JPI_AP_INT_DISABLE           FALSE
#define E_JPI_DAC_RETAIN_ENABLE        TRUE
#define E_JPI_DAC_RETAIN_DISABLE       FALSE
```

Comparator #defines

```
#define E_JPI_COMP_HYSTERESIS_0MV  0
#define E_JPI_COMP_HYSTERESIS_10MV 1
#define E_JPI_COMP_HYSTERESIS_20MV 2
#define E_JPI_COMP_HYSTERESIS_40MV 3
#define E_JPI_AP_COMPARATOR_MASK_1 1
#define E_JPI_AP_COMPARATOR_MASK_2 2
#define E_JPI_COMP_SEL_EXT         0x00
#define E_JPI_COMP_SEL_DAC         0x01
#define E_JPI_COMP_SEL_BANDGAP     0x03
```

UART #defines

```
#define E_JPI_UART_RATE_4800       0
#define E_JPI_UART_RATE_9600       1
#define E_JPI_UART_RATE_19200      2
#define E_JPI_UART_RATE_38400      3
#define E_JPI_UART_RATE_76800      4
#define E_JPI_UART_RATE_115200     5
#define E_JPI_UART_WORD_LEN_5      0
#define E_JPI_UART_WORD_LEN_6      1
#define E_JPI_UART_WORD_LEN_7      2
#define E_JPI_UART_WORD_LEN_8      3
#define E_JPI_UART_FIFO_LEVEL_1    0
#define E_JPI_UART_FIFO_LEVEL_4    1
#define E_JPI_UART_FIFO_LEVEL_8    2
#define E_JPI_UART_FIFO_LEVEL_14   3
#define E_JPI_UART_LS_ERROR        0x80
#define E_JPI_UART_LS_TEMT         0x40
#define E_JPI_UART_LS_THRE         0x20
#define E_JPI_UART_LS_BI           0x10
#define E_JPI_UART_LS_FE           0x08
#define E_JPI_UART_LS_PE           0x04
#define E_JPI_UART_LS_OE           0x02
#define E_JPI_UART_LS_DR           0x01
#define E_JPI_UART_MS_CTS          0x10
#define E_JPI_UART_MS_DCTS         0x01
#define E_JPI_UART_INT_MODEM       0
#define E_JPI_UART_INT_TX          1
#define E_JPI_UART_INT_RXDATA      2
#define E_JPI_UART_INT_RXLINE      3
#define E_JPI_UART_INT_TIMEOUT     6
#define E_JPI_UART_TX_RESET        TRUE
#define E_JPI_UART_RX_RESET        TRUE
#define E_JPI_UART_TX_ENABLE       FALSE
#define E_JPI_UART_RX_ENABLE       FALSE
#define E_JPI_UART_EVEN_PARITY     TRUE
#define E_JPI_UART_ODD_PARITY      FALSE
#define E_JPI_UART_PARITY_ENABLE   TRUE
#define E_JPI_UART_PARITY_DISABLE  FALSE
#define E_JPI_UART_1_STOP_BIT      TRUE
#define E_JPI_UART_2_STOP_BITS     FALSE
#define E_JPI_UART_RTS_HIGH        TRUE
#define E_JPI_UART_RTS_LOW         FALSE
```

Timer #defines

```
#define E_JPI_TIMER_INT_PERIOD      1
#define E_JPI_TIMER_INT_RISE        2
#define E_JPI_TIMER_INTERRUPT_RISING    (1U)
#define E_JPI_TIMER_INTERRUPT_COMPLETE  (2U)
```

Wake Timer #defines

```
#define E_JPI_WAKE_TIMER_0          0
#define E_JPI_WAKE_TIMER_1          1
```

SPI #defines

```
#define E_JPI_SPIM_MSB_FIRST        FALSE
#define E_JPI_SPIM_LSB_FIRST        TRUE
#define E_JPI_SPIM_TXPOS_EDGE       FALSE
#define E_JPI_SPIM_TXNEG_EDGE       TRUE
#define E_JPI_SPIM_RXPOS_EDGE       FALSE
#define E_JPI_SPIM_RXNEG_EDGE       TRUE
#define E_JPI_SPIM_INT_ENABLE       TRUE
#define E_JPI_SPIM_INT_DISABLE      FALSE
#define E_JPI_SPIM_AUTOSLAVE_ENBL   TRUE
#define E_JPI_SPIM_AUTOSLAVE_DSABL FALSE
#define E_JPI_SPIM_SLAVE_ENBLE_0    0x1
#define E_JPI_SPIM_SLAVE_ENBLE_1    0x2
#define E_JPI_SPIM_SLAVE_ENBLE_2    0x4
#define E_JPI_SPIM_SLAVE_ENBLE_3    0x8
```

Serial Interface (SI) #defines

```
#define E_JPI_SI_INT_AL             0x20
#define E_JPI_SI_SLAVE_RW_SET       FALSE
#define E_JPI_SI_START_BIT          TRUE
#define E_JPI_SI_NO_START_BIT       FALSE
#define E_JPI_SI_STOP_BIT           TRUE
#define E_JPI_SI_NO_STOP_BIT        FALSE
#define E_JPI_SI_SLAVE_READ         TRUE
#define E_JPI_SI_NO_SLAVE_READ      FALSE
#define E_JPI_SI_SLAVE_WRITE        TRUE
#define E_JPI_SI_NO_SLAVE_WRITE     FALSE
#define E_JPI_SI_SEND_ACK           FALSE
#define E_JPI_SI_SEND_NACK          TRUE
#define E_JPI_SI_IRQ_ACK            TRUE
#define E_JPI_SI_NO_IRQ_ACK         FALSE
```

Intelligent Peripheral (IP) #defines

```
#define E_JPI_IP_MAX_MSG_SIZE      0x3E
#define E_JPI_IP_TXPOS_EDGE        FALSE
#define E_JPI_IP_TXNEG_EDGE        TRUE
#define E_JPI_IP_RXPOS_EDGE        FALSE
#define E_JPI_IP_RXNEG_EDGE        TRUE
#define E_JPI_IP_BIG_ENDIAN        TRUE
#define E_JPI_IP_LITTLE_ENDIAN     FALSE
```

DIO #defines

```
#define E_JPI_DIO0_INT             0x00000001
#define E_JPI_DIO1_INT             0x00000002
#define E_JPI_DIO2_INT             0x00000004
#define E_JPI_DIO3_INT             0x00000008
#define E_JPI_DIO4_INT             0x00000010
#define E_JPI_DIO5_INT             0x00000020
#define E_JPI_DIO6_INT             0x00000040
#define E_JPI_DIO7_INT             0x00000080
#define E_JPI_DIO8_INT             0x00000100
#define E_JPI_DIO9_INT             0x00000200
#define E_JPI_DIO10_INT            0x00000400
#define E_JPI_DIO11_INT            0x00000800
#define E_JPI_DIO12_INT            0x00001000
#define E_JPI_DIO13_INT            0x00002000
#define E_JPI_DIO14_INT            0x00004000
#define E_JPI_DIO15_INT            0x00008000
#define E_JPI_DIO16_INT            0x00010000
#define E_JPI_DIO17_INT            0x00020000
#define E_JPI_DIO18_INT            0x00040000
#define E_JPI_DIO19_INT            0x00080000
#define E_JPI_DIO20_INT            0x00100000
```

C. Data Types

The following data types are used by the Jenie API.

tsJenieSecKey (Security Key)

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

tsJenie_RoutingEntry (Routing Table Entry)

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

tsJenie_NeighbourEntry (Neighbour Table Entry)

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

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

tsScanElement (Scan Results)

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

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

For more information on `u8LinkQuality`, refer to the Note on page 178.

D. Stack Events

The sub-sections below detail the stack events (management and data) that can be handled by the Jenie API callback functions described in Section 2.2.

Note: Hardware events for the JN5139/JN5148 integrated peripherals are described in Appendix E.

D.1 Stack Management Events

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

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

Table 3: Stack Management Events

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

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

tsSvcReqRsp

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

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

tsPollCmplt

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

For details of the enumerated type `teJeniePollStatus`, refer to Appendix B.1.

tsChildJoined

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

tsChildLeave

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

tsChildRejected

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

tsNwkStartUp

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

D.2 Data Events

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

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

Table 4: Data Events

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

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

tsData

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

tsDataToService

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

tsDataAck

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

tsDataToServiceAck

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

E. Integrated Peripheral Interrupt Handling

Interrupts from the JN5139/JN5148 integrated peripherals are handled by a set of peripheral-specific callback functions. These callbacks are user-defined and can be registered using the corresponding callback registration functions from the Jenie API. The callback registration functions for the different peripherals are specified in the table below (which also gives the page of the function description in this manual).

Peripheral	Registration Function and Page
Comparator, DIO, WakeTimer	"vJPI_SysCtrlRegisterCallback" on page 60
ADC, DAC	"vJPI_APRegisterCallback" on page 65
UART0	"vJPI_Uart0RegisterCallback" on page 100
UART1	"vJPI_Uart1RegisterCallback" on page 101
Timer0	"vJPI_Timer0RegisterCallback" on page 113
Timer1	"vJPI_Timer1RegisterCallback" on page 114
Timer2 (JN5148 only)	"vJPI_Timer2RegisterCallback (JN5148 Only)" on page 115
SPI Master	"vJPI_SpiRegisterCallback" on page 140
2-wire Serial Interface (SI)	"vJPI_SiRegisterCallback" on page 152
Intelligent Peripheral Interface	"vJPI_IpRegisterCallback" on page 159

Table 5: Peripherals and Callback Registration Functions

For example, you can define your own UART interrupt handler and register this callback function using the **vJPI_Uart0RegisterCallback()** function (for UART0).

E.1 Callback Function Prototype and Parameters

All peripheral-specific callback functions must have the following prototype:

```
PRIVATE void vHwDeviceIntCallback(uint32 u32DeviceId,
                                  uint32 u32ItemBitmap);
```

where

- *u32DeviceId* is an enumerated value indicating the peripheral that generated the interrupt - see Appendix E.1.1 below
- *u32ItemBitmap* is a 32-bit bitmask indicating the individual interrupt source within the peripheral (except for the UARTs, for which the parameter returns an enumerated value) - see Appendix E.1.2 below

Before calling the callback function, the library clears the source of the interrupt, so there is no possibility of the processor entering a state of permanently trying to handle the same interrupt due to a malformed callback function. This also means that it is possible to have a NULL callback function.

The UARTs are the exception to this rule - when generating a 'receive data available' or 'timeout indication' interrupt, the UARTs will only clear the interrupt when the data is read from the UART receive buffer. It is therefore vital that (if UART interrupts are to be enabled) the callback function handles the 'receive data available' and 'timeout indication' interrupts by reading the data from the UART before returning.

Note: If the Jennic Application Queue API is being used, the issue with UART interrupts is handled by this API, so the application does not have to cope with it. For more information about the Application Queue API, refer to the *Application Queue API Reference Manual (JN-RM-2025)*.

E.1.1 Peripheral Interrupt Enumerations (u32DeviceId)

The parameter *u32DeviceID* in the user-defined interrupt handler identifies the on-chip peripheral that has generated the interrupt. A set of enumerations are provided for this purpose, as detailed in the table below.

Enumeration	Value (JN5139)	Value (JN5148)	Interrupt Source
E_JPI_DEVICE_AUDIOFIFO	-	0	Sample FIFO *
E_JPI_DEVICE_I2S	-	1	Digital Audio Interface (DAI) *
E_JPI_DEVICE_SYSCTRL	2	2	System Controller (comparator, DIO or wake timer)
E_JPI_DEVICE_BBC	3	3	Baseband Controller
E_JPI_DEVICE_AES	4	4	Encryption Engine
E_JPI_DEVICE_PHYCTRL	5	5	PHY Controller
E_JPI_DEVICE_UART0	6	6	UART0
E_JPI_DEVICE_UART1	7	7	UART1
E_JPI_DEVICE_TIMER0	8	8	Timer 0
E_JPI_DEVICE_TIMER1	9	9	Timer 1
E_JPI_DEVICE_SI	10	10	2-Wire Serial Interface
E_JPI_DEVICE_SPIM	11	11	SPI Master
E_JPI_DEVICE_INTPER	12	12	Intelligent Peripheral Interface
E_JPI_DEVICE_ANALOGUE	13	13	Analogue Peripherals (ADC or DAC)
E_JPI_DEVICE_TIMER2	-	14	Timer 2 (JN5148 only)
E_JPI_DEVICE_TICK_TIMER	0	15	Tick Timer

Table 6: Enumerations for *u32DeviceID*

* JN5148 features not supported by JPI library

E.1.2 Peripheral Interrupt Sources (u32ItemBitmap)

The parameter *u32ItemBitmap* is a 32-bit bitmask indicating the individual interrupt source within the peripheral (except for the UARTs, for which the parameter returns an enumerated value). The bits and their meanings are detailed in the tables below.

System Controller

Mask	Bit	Description
E_JPI_SYSCTRL_COMP0_MASK E_JPI_SYSCTRL_COMP1_MASK	28 29	Comparator 0 event Comparator 1 event
E_JPI_SYSCTRL_WK0_MASK E_JPI_SYSCTRL_WK1_MASK	26 27	Wake Timer 0 event Wake Timer 1 event
E_JPI_DIO0_INT E_JPI_DIO1_INT E_JPI_DIO2_INT : E_JPI_DIO20_INT	0 1 2 : 20	DIO0 event DIO1 event DIO2 event : DIO20 event

Analogue Peripherals

Mask	Bit	Description
E_JPI_AP_INT_STATUS_MASK	0	Asserted to indicate capture complete or new sample ready

Timers

These values are identical for the two timers.

Mask	Bit	Description
E_JPI_TIMER_RISE_MASK	1	Interrupt status, generated on timer rising edge, end of low period - will be non-zero if interrupt for timer 'output going high' has been set
E_JPI_TIMER_PERIOD_MASK	0	Interrupt status, generated on timer falling edge, end of period - will be non-zero if interrupt for timer 'period complete' has been set

SPI Master

Mask	Bit	Description
E_JPI_SPIM_TX_MASK	1	Asserted to indicate transfer has completed

Serial Interface (SI)

Mask	Bit	Description
E_JPI_SI_RXACK_MASK	7	Asserted if no acknowledgement is received from the addressed slave
E_JPI_SI_BUSY_MASK	6	• Asserted if a START signal is detected • Cleared if a STOP signal is detected
E_JPI_SI_AL_MASK	5	Asserted to indicate loss of arbitration
E_JPI_SI_ACK_CTRL_MASK	2	Acknowledgement control: • 0 indicates sent ACK • 1 indicates sent NACK
E_JPI_SI_TIP_MASK	1	Asserted to indicate transfer in progress
E_JPI_SI_INT_STATUS_MASK	0	Interrupt status - interrupt indicates loss of arbitration or that byte trans-fer has completed

Intelligent Peripheral (IP) Interface

Mask	Bit	Description
E_JPI_IP_INT_STATUS_MASK	6	Asserted to indicate transaction has completed, i.e. Slave Select gone high, and TXGO or RXGO gone low
E_JPI_IP_TXGO_MASK	1	• Asserted when Transmit data is copied to the internal buffer • Cleared when it has been transmitted
E_JPI_IP_RXGO_MASK	0	• Asserted when device is in ‘ready to receive’ state • Cleared when data receive is complete

UARTs

For the UART interrupts, *u32ItemBitmap* returns the following enumerated values. These values are identical for the two UARTs:

Enumeration	Value	Description
E_JPI_UART_INT_TIMEOUT	6	Timeout indication *
E_JPI_UART_INT_RXLINE	3	Receive line status
E_JPI_UART_INT_RXDATA	2	Receive data available *
E_JPI_UART_INT_TX	1	Transmit FIFO empty
E_JPI_UART_INT_MODEM	0	Modem status

* When this interrupt occurs, the received data byte is passed to the application via bits 15-8 of the *u32Bitmap* parameter of the **vJenie_CbHwEvent()** callback function. In the case of an E_JPI_UART_INT_RXDATA interrupt, any other bytes remaining in the UART's FIFO can be detected by checking the E_JPI_UART_LS_DR bit using the **u8JPI_UartReadLineStatus()** function, and then read from the UART using the **u8JPI_UartReadData()** function.

E.2 Handling Wake Interrupts

The JN5139/JN5148 wireless microcontroller can be woken from sleep by any of the following sources:

- Wake timer
- DIO
- Comparator

The above wake sources are outlined in Appendix E.2.1, Appendix E.2.2 and Appendix E.2.3.

For the device to be woken by one of the above wake sources, interrupts must be enabled for that source at some point before the device goes to sleep. The handling of these interrupts is described below.

DIO and Comparator Interrupts

Interrupts from the DIOs and comparators are handled by the user-defined System Controller callback function which is registered using the function **vJPI_SysCtrlRegisterCallback()**. The callback function must be registered before the device goes to sleep. If the device wakes from sleep with memory held and there are any System Controller interrupts pending, the Jenie restart will result in the callback function being invoked and the interrupts being cleared. An interrupt bitmask *u32ItemBitmap* is passed into the callback function and the particular source of the interrupt (DIO or comparator) can be obtained from this bitmask by logical ANDing it with the masks for the System Controller detailed in Appendix E.1.2.

Caution: *During sleep without memory held, the registered callback function is lost. It must therefore be re-registered on waking, but will not be available in time to process the interrupt that woke the device. Therefore, the identity of this wake source will be lost.*

Wake Timer Interrupts

When a sleeping End Device is woken by a wake timer, this event is not presented to the user application either by the **vJenie_CbHwEvent()** callback function or by the callback function that is registered through **vJPI_SysCtrlRegisterCallback()**. However, since the other wake sources (DIO and comparator) do generate interrupts that are handled by the System Controller callback function (see above), it is possible to determine whether a wake timer caused the wake-up by a process of elimination.

The 'wake timer fired' status is cleared by the stack upon waking, so it is not possible to use the **u8JPI_WakeTimerFiredStatus()** function to determine whether the wake timer caused the wake-up. However, the wake timer value is not cleared by the stack and can be read with the **u32JPI_WakeTimerRead()** function. Thus, if the wake timer has fired, this function will return a high value, as the timer will have rolled over from 0 (if this value is greater than 0x80000000 then the wake is likely to be due to the timer firing).

E.2.1 Wake Timer

There are two wake timers (0 and 1) on the JN5139/JN5148 wireless microcontroller. These timers run at a nominal 32 kHz and are able to operate during sleep periods. When a running wake timer expires during sleep, an interrupt can be generated which wakes the device. The functions for controlling the wake timers are detailed in Section 3.6.

Interrupts for a wake timer can be enabled using the function **vJPI_WakeTimerEnable()**. The timed period for a wake timer is set when the wake timer is started.

E.2.2 DIO

There are 21 DIO lines (0-20) on the JN5139/JN5148 wireless microcontroller. The device can be woken from sleep on the change of state of any DIOs that have been configured as inputs and as wake sources. The functions for controlling the DIOs are detailed in Section 3.3.

The directions of the DIOs (input or output) are configured using the function **vJPI_DioSetDirection()**. Wake interrupts can then be enabled on DIO inputs using the function **vJPI_DioSetOutput()**.

E.2.3 Comparator

There are two comparators (1 and 2) on the JN5139/JN5148 wireless microcontroller. The device can be woken from sleep by a comparator interrupt when either of the following comparator events occurs:

- The comparator's input voltage rises above the reference voltage.
- The comparator's input voltage falls below the reference voltage.

The functions for controlling the comparators are detailed in Section 3.2.

Interrupts for a comparator are configured and enabled using the function **vJPI_ComparatorEnable()**.

F. JenNet API

This appendix details the functions and network parameters of the JenNet API, which may be used in conjunction with the Jenie API to access features provided by the underlying JenNet stack layer. The JenNet functions provide additional control over how nodes join a network, inter-network communication and the operation of the Jenie/JenNet stack.

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

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

MAC_Addr_s

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

MAC_ExtAddr_s

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

All other structures used are declared in **JenNetApi.h**.

F.1 JenNet Functions

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

Function	**Page**
eApi_SendDataToExtNwk	193
vNwk_DeleteChild	194
vApi_SetScanSleep	195
vApi_SetBcastTTL	196
vApi_SetPurgeRoute	197
vApi_SetPurgeInterval	198
vNwk_SetBeaconCalming	199
vApi_SetUserBeaconBits	200
u16Api_GetUserBeaconBits	201
u8Api_GetLastPktLqi	202
u16Api_GetDepth	203
u8Api_GetStackState	204
u32Api_GetVersion	205
vApi_RegBeaconNotifyCallback	206
vApi_RegLocalAuthoriseCallback	207
vApi_RegNwkAuthoriseCallback	208
vApi_RegScanSortCallback	209

eApi_SendDataToExtNwk

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

Description

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

The destination address is specified using the `MAC_Addr_s` structure (shown on page 191), which allows the application to specify the destination PAN ID and either a 64-bit extended address (IEEE/MAC address) or a 16-bit short address (as used in IEEE 802.15.4).

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

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

Parameters

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

Returns

E_JENNET_DEFERRED

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

E_JENNET_ERROR

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

vNwk_DeleteChild

```
void vNwk_DeleteChild(MAC_ExtAddr_s *psNodeAddr);
```

Description

This function is used on a parent node to force an immediate child to leave the network by deleting its entry in the local Neighbour table. The node to be removed is specified using its 64-bit IEEE/MAC address in the `MAC_ExtAddr_s` structure (shown on page 191).

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

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

Parameters

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

Returns

None

vApi_SetScanSleep

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

Description

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

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

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

Parameters

u32ScanSleepDuration	Time, in milliseconds, to sleep after scan timeout

Returns

None

vApi_SetBcastTTL

void vApi_SetBcastTTL(uint8 *u8MaxTTL***);**

Description

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

u8MaxTTL = Desired maximum number of broadcast hops - 1

Parameters

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

Returns

None

vApi_SetPurgeRoute

```
void vApi_SetPurgeRoute(bool_t bPurge);
```

Description

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

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

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

Parameters

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

Returns

None

vApi_SetPurgeInterval

```
void vApi_SetPurgeInterval(uint32 u32Interval);
```

Description

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

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

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

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

Parameters

u32Interval	Route maintenance period in units of 100 ms

Returns

None

vNwk_SetBeaconCalming

void vNwk_SetBeaconCalming(bool *bState*);

Description

This function enables/disables 'beacon calming'.

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

Parameters

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

Returns

None

vApi_SetUserBeaconBits

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

Description

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

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

Parameters

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

Returns

None

u16Api_GetUserBeaconBits

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

Description

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

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

Parameters

None

Returns

16 bits of user data read from the beacon payload

u8Api_GetLastPktLqi

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

Description

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

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

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

Parameters

None

Returns

The LQI value of the last packet received

u16Api_GetDepth

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

Description

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

Parameters

None

Returns

Number of hops from Co-ordinator

u8Api_GetStackState

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

Description

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

Parameters

None

Returns

The state of the JenNet stack, one of:

E_JENNET_IDLE (0x00)

E_JENNET_ENERGY_SCAN (0x01)

E_JENNET_WAITING_FOR_ENERGY_SCAN (0x02)

E_JENNET_ACTIVE_SCAN (0x03)

E_JENNET_WAITING_FOR_ACTIVE_SCAN (0x04)

E_JENNET_ASSOCIATE (0x05)

E_JENNET_ASSOCIATE_SKIP_ESTABLISH_ROUTE (0x06)

E_JENNET_WAITING_FOR_ASSOCIATE (0x07)

E_JENNET_WAITING_FOR_ASSOCIATE_SKIP_ESTABLISH_ROUTE (0x08)

E_JENNET_START_COORD (0x09)

E_JENNET_START_COORD_SKIP_ESTABLISH_ROUTE (0x0A)

E_JENNET_ESTABLISH_ROUTE (0x0B)

E_JENNET_WAITING_FOR_ESTABLISH_ROUTE (0x0C)

E_JENNET_RUNNING (0x0D)

E_JENNET_WAITING_FOR_BACKOFF (0x0E)

E_JENNET_SLEEP (0x0F)

u32Api_GetVersion

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

Description

This allows a stack version text string to be obtained.

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

Parameters

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

Returns

None

vApi_RegBeaconNotifyCallback

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

Description

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

The prototype for the callback function is detailed below.

Parameters

prCallback	Pointer to callback function

Returns

None

Callback Function

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

Description

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

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

Parameters

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

Returns

TRUE	Accept the beacon for sorting
FALSE	Delete the beacon

vApi_RegLocalAuthoriseCallback

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

Description

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

Parameters

prCallback	Pointer to callback function

Returns

None

Callback Function

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

Description

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

Parameters

**psAddr*	Pointer to IEEE/MAC address

Returns

TRUE Joining process continues

FALSEJoining is denied

vApi_RegNwkAuthoriseCallback

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

Description

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

Parameters

prCallback	Pointer to callback function

Returns

None

Callback Function

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

Description

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

Parameters

**psAddr*	Pointer to the IEEE/MAC address

Returns

TRUE	Joining process continues
FALSE	Joining is denied

vApi_RegScanSortCallback

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

Description

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

Parameters

prCallback	Pointer to callback function

Returns

None

Callback Function

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

Description

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

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

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

Parameters

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

Returns

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

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

F.2 JenNet Network Parameters

This section describes certain JenNet network parameters. Some of these parameters are duplicated in the Jenie network parameters, detailed in Appendix A. The Jenie values are loaded into the JenNet parameters by **vJenie_CbConfigureNetwork()**, which occurs once at the program start.

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

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

Co-ordinator Parameters

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

Table 7: Co-ordinator Parameters

General Parameters

Parameter Name	Description	Default Value	Range
gInternalTimer	The timer to be used as an internal timer: Timer 0, Timer 1 or the Tick Timer. The valid values are shown to the right and defined in the header file **AppHardwareApi.h**.	E_AHI_DEVICE _TICK_TIMER	E_AHI_DEVICE_TICK_TIMER E_AHI_DEVICE_TIMER0 E_AHI_DEVICE_TIMER1
gMaxBcastTTL	Determines the maximum number of hops that a broadcast message sent from the local node can make. Set this value to one less than the desired maximum (so the value 0 corresponds to one hop).	5	0-255
gMaxFailedPkts	Number of missed communications (MAC acknowledgments) before parent considered to be lost (and node must try to find a new parent).	5	1-255
gMinBeaconLQI	Minimum valid radio signal strength (as an LQI value) of a beacon - the stack rejects beacons with signal strength less than this value.	0	0-255 For information on LQI values, refer to Appendix G.
gNetworkID	32-bit Network Application ID used to identify an individual application/ network.	0xAAAAAAAA	0-0xFFFFFFFF
gScanChannels	Bitmap (32 bits) of the set of channels to consider when performing an auto-scan of the 2.4-GHz band for a suitable channel to use. The Co-ordinator will select the quietest channel from those available (auto-scan must have been enabled via *gChannel.*). Other node types will scan the possible channels to search for network.	0x07FFF800 (All Channels)	0x00000800 -0x07FFF800 (Bit 11 set ⇒ Ch 11, Bit 12 set ⇒ Ch 12,...)

Table 8: General Parameters

Co-ordinator/Router Parameters

Parameter Name	Description	Default Value	Range
gEDChildActivityTimeout *	Timeout period for communication (excluding data polling) from an End Device child. If no message is received from the End Device within this period, the child is assumed lost and is removed from the Neighbour table (and Routing tables higher in the network), provided End Device purging has been enabled through *gRouterPurgeInactiveED*.	0	0-0xFFFFFFFF Timeout is value set multiplied by 100 ms
gMaxChildren	Maximum number of children that the node can have.	10	0-16
gMaxSleepingChildren	Maximum number of children that can be End Devices (nodes capable of sleeping). This value must be less than or equal to *gMaxChildren*. The remaining child nodes are reserved exclusively for Routers, although any number of children can be Routers.	8	1-*gMaxChildren*
bPermitExtNwkPkts	Enables/disables reception of packets from external networks. Do not configure in **vJenie_CbConfigureNetwork()**.	FALSE (disabled)	TRUE - enable FALSE - disable
gRouteImport	Enables/disables the ability of routing nodes to import routes from child nodes that have children.	TRUE (enabled)	TRUE - enable FALSE - disable
gRouterEnableAutoPurge	Enables/disables the auto-purge facility which removes inactive nodes from the network.	TRUE (enabled)	TRUE - enable FALSE - disable
gRoutingTableSize	Number of elements in array used to store the Routing table. Should be set to a value slightly larger than the maximum number of network nodes, to allow for nodes leaving and joining. Set 0 for End Devices.	0	0-1000 Note that the upper limit may be restricted by the amount of available RAM. Each Routing table entry uses 12 bytes.
gRouterPingPeriod **	Time between auto-pings generated by a Router (to its parent). Set in units of 10 ms. The same value should be set in all routing nodes in the network.	500 (5 seconds)	500-65535
gRouterPurgeInactiveED *	Enable/disable the timeout on End Device activity - see the parameter *gEDChildActivityTimeout*.	FALSE (disabled)	TRUE - enable FALSE - disable

Table 9: Co-ordinator/Router Parameters

Parameter Name	Description	Default Value	Range
gpvRoutingTableSpace	Pointer to space allocated for the Routing table. This space must be equal to [sizeof(tsJenieRoutingTable) x *gRoutingTableSize*], or NULL in the case of an End Device.	NULL	Pointer

Table 9: Co-ordinator/Router Parameters

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

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

End Device Parameters

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

Table 10: End Device Parameters

G. Link Quality Indication (LQI)

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.

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).*

Revision History

Version	Date	Comments
1.0	29-Nov-2007	First release
1.1	21-Feb-2008	JPI function lists inserted and minor modifications made
1.2	07-Mar-2008	Updated for Jenie v1.2 - Statistics functions added
1.3	09-July-2008	Updated for Jenie v1.3
1.4	25-Sep-2008	Updated with minor corrections
1.5	04-Dec-2008	Updated for Jenie v1.4 - *gJenie_MaxChildren* variable modified
1.6	05-June-2009	Updated with JenNet API and other minor corrections
1.7	27-Aug-2009	Updated with minor corrections and new LQI appendix
1.8	17-Mar-2010	Modified for JN5148, JPI function descriptions modified/updated and various other updates/corrections made

Important Notice

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