RemoTI API

Document Number: SWRA268I

Table Of Contents

List of Figures

List of Tables

Acronyms and Definitions

AP	Application Processor
API	Application Programming Interface
AREQ	Asynchronous Request by AP of SPI interface
CERC	Consumer Electronics Remote Control – deprecated for ZRC
CTL	Controller
ID	Identifier
LQI	Link Quality Indication
MAC	Medium Access Control
NP	Network Processor
NPI	Network Processor Interface
NSDU	Network layer Service Data Unit
NV	Non Volatile Memory
OSAL	Operating System Abstraction Layer
PAN	Personal Area Network
POLL	Poll Request by target of SPI interface
RF4CE	Radio Frequency for Consumer Electronics
RCN	RF4CE Network Layer
RO	Read-Only
RPC	Remote Procedure Call
RTI	RemoTI Application Framework
RW	Read-Write
SPI	Serial Peripheral Interface
SREQ	Synchronous Request by AP of SPI interface
Target	Consumer Electronics device
TGT	Target
ZID	ZigBee Input Device – the 2nd Application Profile specified for RF4CE
ZRC	ZigBee Remote Control – the 1st Application Profile specified for RF4CE

References

[R1] IEEE Std. 802.15.4-2006, Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (WPANs)

[R2] ZigBee RF4CE Specification (ZigBee Alliance document 094945r00ZB)

[R3] ZigBee RF4CE CERC Profile Specification (ZigBee Alliance document 094946r00ZB)

[R4] ZigBee RF4CE Vendor ID List (ZigBee Alliance document 094949r00ZB)

[R5] ZigBee RF4CE Device Type List (ZigBee Alliance document 094950r00ZB)

[R6] ZigBee RF4CE Profile ID List (ZigBee Alliance document 094951r00ZB)

[R7] CC253x Users Guide, SWRU191, Texas Instruments Inc, **http://www.ti.com/lit/swru191**.

[R8] RemoTI Host Processor Sample Application and Porting Guide, SWRA259, Texas Instruments Inc, *can be found in the installation folder of RemoTI*.

[R9] RemoTI OS Abstraction Layer Application Programming Interface, SWRA417, Texas Instruments Inc, *can be found in the installation folder of RemoTI*.

[R10] ZigBee RF4CE ZID Profile Specification (ZigBee Alliance document 105557r18ZB), **http://www.zigbee.org/Standards/ZigBeeInputDevice/download.aspx**

1. Introduction

1.1 Interface Layers

RemoTI stack provides two layers of interfaces. One is RemoTI (RTI) API and the other is network layer (RCN) API. The RemoTI (RTI) API provides an interface to an application framework to simplify application development. RemoTI application framework (RTI) is implemented by use of network layer (RCN) API and removes some of the RCN layer complexity. RTI also includes API for sleep mode and test mode on top of network layer features. RTI application framework, implements the profile layer of RF4CE. The profile layer controls features such as discovery and pairing. RTI is based on the ZRC profile, but can support multiple profiles through extensions. RCN API is optionally provided so that customer can build their application on top of RCN API directly, to leverage full flexibility of network layer interface such as to perform non-ZRC profile discovery and pairing. On the other hand, RCN API does not have non-network layer API such as sleep mode and test mode.

If RemoTI application framework (RTI) is in use and application accesses RTI API, application is not advised to use RCN API directly as such use will conflict with RTI using RCN API. If application is built without RTI, application must access RCN API directly to control RemoTI network layer.

The ZID Profile extension to the RTI (also referred to as a co-layer to the RTI) is enabled by defining to TRUE one of the corresponding compiler directives (FEATURE_ZID_ADA or FEATURE_ZID_CLD). Enabling the ZID Profile extension empowers the RTI layer to implement the specified details of the ZID state machine for configuration and mandatory unpairing on failure. This simplifies the work of the Application layer to sending or receiving ZID data reports. If the ZID Profile extension is not enabled, then all ZID traffic will be received via the normal data indication mechanism defined later in this document. Defining FEATURE_ZID_ADA to TRUE will enable operation as ZID Adaptor, defining FEATURE_ZID_CLD will enable operation as ZID Class Device.

1.2 Software architecture

A RemoTI application can interface with the RemoTI stack in two configurations – either executing directly on the SoC (CC253x) or executing on a host processor that communicates with the SoC over a serial (UART, SPI, I2C or USB as CDC or HID) interface.

1.2.1 Remote

The first configuration is called the remote configuration and is typically used to develop remote applications. The end product might consist of a Keypad, LED’s, other user interface options and even IR generation. The software architecture for this configuration is illustrated in Figure 1.

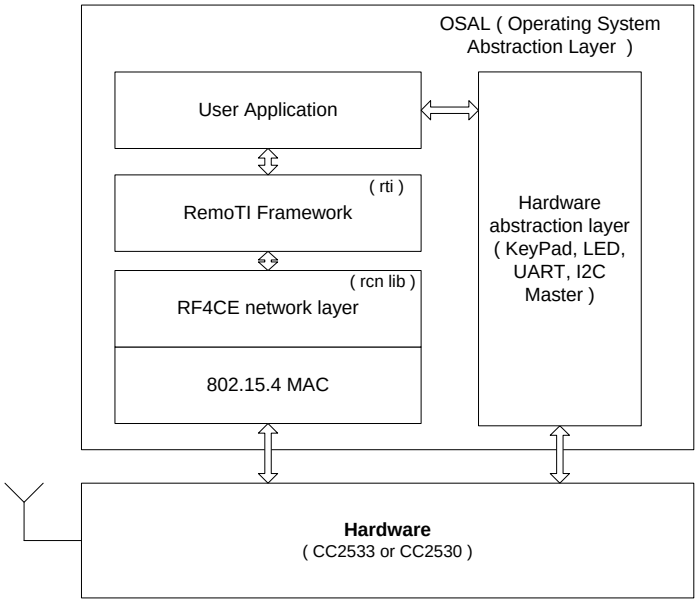

Figure 1: Remote Configuration

The remote configuration includes RemoTI Framework which provides RTI interface. If user application is implemented directly on top of RCN, the RemoTI Framework has to be excluded as RemoTI Framework will conflict with user application in its control of RCN.

1.2.2 Network processor

The second configuration is called the network processor configuration and it can be used to develop Target devices as well as more advanced remote control devices that have more sophisticated user interface technologies. The architecture for this configuration is illustrated in the figure below.

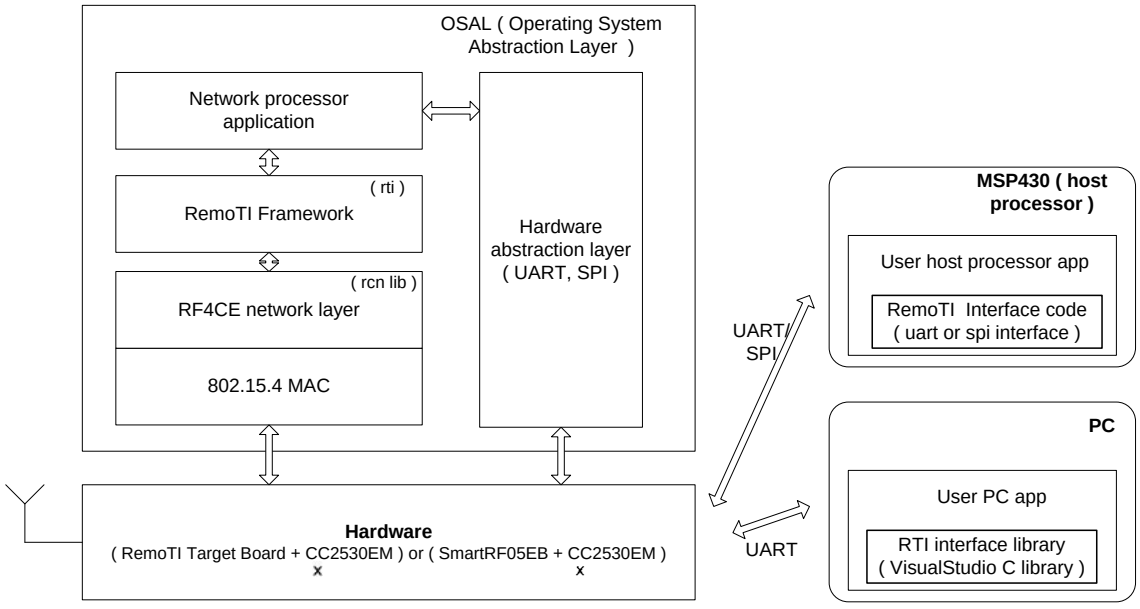

Figure 2: Network processor configuration

User host processor application can use either RTI API or RCN API as far as RemoTI interface code is ported on the host processor. No change of configuration on the network processor side is required to access RCN API. Use of both RTI API and RCN API by the user host processor application is not possible except for use of a certain complementary RTI API functions together with RCN API functions.

Such use will be described later. Porting RemoTI interface code on the host processor is described in **Error! Reference source not found.**.

PC RTI interface library provided in RemoTI software package includes RCN surrogate module as well and hence, user PC application can access RCN API without changing PC RTI interface library.

1.3 Software components

The RemoTI stack consists of the following software components.

1.3.1 OSAL

This is a simple operation system environment for the SoC. It includes features for task management, message passing, queuing, memory management, timers etc. This component is included as source code.

1.3.2 Hardware abstraction layer

This component provides an abstract interface to the hardware available on chip and on the board. It includes firmware for all the serial communication interfaces, Keypad on the remote control and LED's. This code is included in source to allow the user to modify to suit the hardware available on their product.

1.3.3 RCN library

This is the core RF4CE stack and includes the RF4CE network layer and security functions, and IEEE 802.15.4 High-Level MAC layer. This component in included as an object code library. There are several versions of this library and builds for two different use cases – a controller-only and a combined version. The controller-only version may be used for devices that only need the remote control functions to save code space.

The library files are, then, available in the form of near code model and banked code model. Near code model cannot exceed 64-KB of address space for the code space and hence is only for CC253xF64 or for any other product that only utilizes 64-KB of space for executable code.

There is also a build that takes the controller-only, non-banked build a step further in order to try to absolutely minimize memory and code usage by defining the `RCN_FEATURE_EXTRA_PAIR_INFO` build flag to FALSE.

Features	Near Code Model	Banked Code Model	*Minimal Memory/Code Model*
Controller feature only	rcnctrl-CC253x.lib	rcnctrl-CC253x-banked.lib	rcnctrl-min-CC253x.lib
Combined features	rcnsuper-CC253x.lib	rcnsuper-CC253x-banked.lib	N/A

Table 1: RCN library files

1.3.4 RTI

This component is the application framework for the RF4CE device. This component is included as source code.

1.3.5 RTI interface library for PC

This is a Visual Studio C library that implements an RPC mechanism for the RemoTI API and RCN API. It communicates with the hardware (operating in a network processor configuration) over a serial link and makes the RemoTI API and RCN API available for PC applications. Using this library, the developer can develop PC applications that utilize RemoTI API and RCN API.

1.3.6 RTI interface code for host processor

This is software that implements an RPC mechanism for RemoTI API and RCN API over any of the available serial interfaces. It makes the RemoTI API available for applications running on a host

processor. An open source software for linux host is available at **https://github.com/TI-LPRF-Software/RemoTI-Linux**.

2. RTI API Overview

The RTI API provides a framework for application development using TI's RF4CE software and development kits. RTI provides a simplified API that allows the developer to rapidly generate application prototypes using the Texas Instruments RF4CE software stack with the ZRC and/or ZID application profiles (hereafter referred to as RemoTI stack).

The following chapters describe the application programming interface for the RemoTI stack. They are sub-divided into the following categories

- Configuration Interface: allows the application to configure the RemoTI stack
- Application Framework Interface: provides interface to the stack networking features.
- Test Interface: provides an interface to test points in the RemoTI stack

2.1 RTI API and Node Type

The following table indicates which of the RTI APIs can be used by a Controller and/or Target application. A box containing the symbol ♦ means the API can be used by that node type.

RTI API	Controller	*Target*
RTI_InitReq	♦	♦
RTI_InitCnf	♦	♦
RTI_PairReq	♦	♦
RTI_PairCnf	♦	♦
RTI_PairAbortReq	♦	♦
RTI_PairAbortCnf	♦	♦
RTI_AllowPairReq		♦
RTI_AllowPairCnf		♦
RTI_AllowPairAbortReq		♦
RTI_UnpairReq	♦	♦
RTI_UnpairCnf	♦	♦
RTI_UnpairInd	♦	♦
RTI_SendDataReq	♦	♦
RTI_SendDataCnf	♦	♦
RTI_ReceiveDataInd	♦	♦
RTI_StandbyReq		♦
RTI_StandbyCnf		♦
RTI_RxEnableReq	♦	♦
RTI_RxEnableCnf	♦	♦

RTI API	Controller	*Target*
RTI_EnableSleepReq	♦	♦
RTI_EnableSleepCnf	♦	♦
RTI_DisableSleepReq	♦	♦
RTI_DisableSleepCnf	♦	♦

Table 2: API Usage by Node Type

3. RTI Configuration Interface

3.1 Overview

The Configuration Interface allows the application to configure the RTI stack. There are three items of configuration that can be accessed by the application layer:

- Configuration Parameters: used to control the behavior of the RTI
- State Attributes: the current network state information maintained by the RTI stack
- Constants: read-only information defined by the RF4CE specification

3.2 RTI_ReadItemEx and RTI_WriteItemEx

These functions are used to read and write the Configuration Interface tables. For the Configuration Parameters table, the writes should take place prior to calling **`RTI_InitReq()`** in order to be used by the RTI stack (please see section 4.2 for additional detail). The State Attributes table has a limited number of items that can be written (some of which are only permitted in test mode), and can be read to monitor various attributes of the RTI stack. The Constants table is of course read-only.

3.2.1 Prototype

```
rStatus_t RTI_ReadItemEx( uint8 profileId, uint8 itemId, uint8 len, uint8 *pValue )
rStatus_t RTI_WriteItemEx( uint8 profileId, uint8 itemId, uint8 len, uint8 *pValue )
```

3.2.2 Input Parameters

profileId:	The ID of the Profile corresponding to the 'itemId' parameter (i.e. the profile that specifies the attribute identifier or the proprietary extension to a profile that does not conflict with specified Id's) The supported Profile Id's are listed in rti_constants.h.
itemId:	The index used to identify the Configuration Interface item to be read or written. Please see Table 3, Table 4, and Table 5 for available item identifiers and their descriptions.
len:	The number of bytes to read or write.
***pValue:**	A byte pointer to storage containing data to be written.

3.2.3 Output Parameters

***pValue:**	A byte pointer to storage used to place read data.

3.2.4 Return

rStatus_t:	The resulting status from a call to RTI_ReadItem or RTI_WriteItem. Possible values

include (please see Table 6 for status descriptions):

- RTI_SUCCESS
- RTI_ERROR_NOT_PERMITTED
- RTI_ERROR_NO_PAIRING_INDEX
- RTI_ERROR_INVALID_PARAMETER
- RTI_ERROR_UNKNOWN_PARAMETER
- RTI_ERROR_UNSUPPORTED_ATTRIBUTE
- RTI_ERROR_OSAL_NV_OPER_FAILED
- RTI_ERROR_OSAL_NV_ITEM_UNINIT
- RTI_ERROR_OSAL_NV_BAD_ITEM_LEN

3.2.5 Notes

None.

3.3 Configuration Parameters

3.3.1 Overview

The Configuration Parameters table contains parameters that are configured by the application layer prior to starting the RTI stack. The RTI stack reads these parameter values during the initialization API call. Any modification of these values while the RTI stack is operational will not take effect until the next RTI stack initialization.

These values are stored by the RTI stack in non-volatile (NV) memory and persist across a device reset. The application can restore the configuration parameters to their default settings by setting the StartupOption parameter appropriately.

3.3.2 Parameter List

The following table contains the parameters of the Configuration Parameters table:

Parameter	Item ID	Length	Default	*Description*
StartupOption	0xA0	1	1	*rti.h definition:* `RTI_CP_ITEM_STARTUP_CTRL` *Valid for both target and controller.* *This attribute controls the behavior of the RTI stack upon start-up. It takes the following values:* *0 - Restore NIB state attributes from the saved NIB, and start the RTI stack* *1 - Clear the NIB state attributes (i.e. reset to default values), and start the RTI stack* *2 - Clear the Configuration Parameters attributes (i.e. reset to default values), clear the NIB state attributes (i.e. reset to default values), and start the RTI stack*
Node Capabilities				
Node Capabilities	0xA1	1	0xF	*rti.h definition:* `RTI_CP_ITEM_NODE_CAPABILITIES` *Valid for both target and controller.* *This attribute indicates the node capabilities of the node. Each bit indicates the following:* *Bit 0: 0 – controller type, 1 – target type* *Bit 1: 0 – battery powered, 1 – AC mains powered*

Parameter	Item ID	Length	Default	*Description*
				Bit 2: 0 – Security incapable, 1 – Security capable *Bit 3: 0 – Channel normalization incapable, 1 – Channel normalization capable* *Bit 4 – 7: reserved.*
SupportedTargetTypes	0xA2	6	2	*rti.h definition:* `RTI_CP_ITEM_NODE_SUPPORTED_TGT_TYPES` *Valid for controller only.* *This attribute lists the target types supported by the controller node. Each byte in this attribute will indicate a target type that the node supports. The possible device types are defined in Table 1 of* **Error! Reference source not found.***. A byte may be set to reserved value (0x00) if all supported valid device types are already indicated by other bytes of the attribute.*
Application Capabilities				
ApplicationCapabilities	0xA3	1	0x12	*rti.h definition:* `RTI_CP_ITEM_APPL_CAPABILITIES` *Valid for both target and controller.* *This attribute indicates the device capabilities of application specific information. Each bit represents the following:* *Bit 0: 0 – UserString is not specified, 1 – UserString is specified* *Bit 1-2: Number of supported device types* *Bit 3: Reserved* *Bit 4-6: Number of supported profiles* *Bit 7: Reserved*
DeviceTypeList	0xA4	1..3	1	*rti.h definition:* `RTI_CP_ITEM_APPL_DEV_TYPE_LIST` *Valid for both target and controller.* *This attribute lists the device types supported by this node. The possible device types are defined in Table 1 of* **Error! Reference source not found.***.*
ProfileIdList	0xA5	1..7	1	*rti.h definition:* `RTI_CP_ITEM_APPL_PROFILE_ID_LIST` *Valid for both target and controller.* *This attribute lists the profile Ids supported by this node. The available profiles are defined in* ***Error! Reference source not found.****.*
Standby Information				
DefaultStandbyActivePeriod	0xA6	2	14	*rti.h definition:* `RTI_CP_ITEM_STDBY_DEFAULT_ACTIVE_PERIOD` *Valid for target only.* *This attribute indicates the default receiver on time for a power saving device, in units of ms. This attribute will be*

Parameter	Item ID	Length	Default	*Description*
				used as default value of the state attribute StandbyActivePeriod when clearing state attributes.
DefaultStandbyDutyCycle	0xA7	2	330	*rti.h definition:* `RTI_CP_ITEM_STDBY_DEFAULT_DUTY_CYCLE` *Valid for target only.* *This attribute indicates the duty cycle for a power saving device, in units of ms, and is used to set the state attribute StandbyDutyCycle when power saving mode is entered by* `RTI_StandbyReq`.
Vendor Information				
VendorId	0xA8	2	7	*rti.h definition:* `RTI_CP_ITEM_VENDOR_ID` *Valid for both target and controller.* *This attribute indicates the manufacturer specific vendor identifier for this node. The available vendor identifiers are defined in Table 1 of* **Error! Reference source not found.**.
VendorName	0xA9	1..7 characters	Empty	*rti.h definition:* `RTI_CP_ITEM_VENDOR_NAME` *Valid for both target and controller.* *This attribute contains the manufacturer specific vendor identification string for this node, if one if provided.*
Discovery Information				
DiscoveryDuration	0xAA	2	100	*rti.h definition:* `RTI_CP_ITEM_DISCOVERY_DURATION` *Valid for both target and controller.* *This attribute indicates the maximum time to wait for discovery responses to be sent from potential target nodes, in units of milliseconds.*
DefaultDiscoveryLqi-Threshold	*0xAB*	*1*	*0*	*rti.h definition:* `RTI_CP_ITEM_DEFAULT_DISCOVERY_LQI_THRESHOLD` *Valid for both target and controller.* *This attribute indicates the default LQI threshold below which discovery requests will be rejected. This attribute will be used as default value of the state attribute DiscoveryLqiThreshold when clearing state attributes.*

Table 3: Configuration Parameters Table

3.4 State Attributes

3.4.1 Overview

The State Attributes table contains attributes that reflect the current operational state of the RF4CE device. These attributes are maintained by the RTI stack (i.e. by the network layer). The application may read any attribute, and may also write certain attributes as well.

These attributes are maintained in RAM by the RTI stack but a backup copy is also saved in NV memory. The backup copy can either be restored or cleared upon start-up depending on the Configuration Parameters' *StartupOption* parameter.

3.4.2 Attribute list

The following table contains the attributes of the State Attributes table:

Attribute	Item ID	Length	Default	*Description*
StandbyActivePeriod	0x60	2	Range	*rti.h definition:* `RTI_SA_ITEM_STANDBY_ACTIVE_PERIOD` *Valid for target only.* *This attribute is RW, and is used to obtain the active period of a device, in units of milliseconds.* *Note that this state attribute is reset to the DefaultStandbyActivePeriod configuration parameter value when RTI initializes with a StartupOption that instructs clearing of state attributes.* *The range of values include:* *MinStandbyActivePeriod..0xFFFF*
CurrentChannel	0x61	1	15	*rti.h definition:* `RTI_SA_ITEM_CURRENT_CHANNEL` *Valid for target only.* *This attribute is RW, and contains the current logical channel that was chosen when the RC PAN was formed (Target device only). The range of values include:* *15, 20, 25*
DiscoveryLQIThreshold	0x62	1	0	*rti.h definition:* `RTI_SA_ITEM_DISCOVERY_LQI_THRESHOLD` *Valid for target only.* *This attribute is RW. The LQI threshold below which discovery requests will be rejected. The range is from 0 to 0xff.* *Note that this state attribute is reset to the DefaultDiscoveryLQIThreshold configuration parameter value when RTI initializes with a StartupOption that instructs clearing of state attributes.*

Attribute	Item ID	Length	Default	*Description*
DutyCycle	0x64	2	0	*rti.h definition:* `RTI_SA_ITEM_DUTY_CYCLE` *Valid for target only.* *This attribute is RW. The duty cycle of a device in milliseconds. A value of 0 indicates the device is not using power saving feature of RF4CE (duty cycling).* *Note that this state attribute is reset to the DefaultDutyCycle configuration parameter value when RTI initializes with a StartupOption that instructs clearing of state attributes.*
FrameCounter	0x65	4	Range	*rti.h definition:* `RTI_SA_ITEM_FRAME_COUNTER` *Valid for both target and controller.* *This attribute is RO (RW in test mode), and contains the current frame count added to the transmitted NPDU. The range of values include:* *0x0000_0000..0xFFFF_FFFF*
InPowerSave	0x67	1	0	*rti.h definition:* `RTI_SA_ITEM_IN_POWER_SAVE` *Valid for target only.* *This attribute is RO, and is used to obtain whether the device is operating in power save mode. The range of values include:* *0 – Device is not operating in power save mode* *1 – Device is operating in power save mode*
MaxFirstAttemptCSMABackoffs	0x6a	1	4	*rti.h definition:* `RTI_SA_ITEM_MAX_FIRST_ATTEMPT_CSMA_BACKOFFS` *Valid for both target and controller.* *This attribute is RW. The maximum number of backoffs the MAC CSMA-CA algorithm will attempt before declaring a channel access failure for the first transmission attempt. Valid range is between 0 and 5*
MaxFirstAttemptFrameRetries	0x6b	1	3	*rti.h definition:* `RTI_SA_ITEM_MAX_FIRST_ATTEMPT_FRAME_RETRIES` *Valid for both target and controller.* *This attribute is RW. The maximum number of MAC retries allowed after a transmission failure for the first transmission attempt. Valid range is between 0 and 7.*
ResponseWaitTime	0x6d	1	50	*rti.h definition:* `RTI_SA_ITEM_RESPONSE_WAIT_TIME` *Valid for both target and controller.*

Attribute	Item ID	Length	Default	*Description*
				This attribute is RW, and is used to obtain or specify the maximum time a device will wait for a response following a request, in units of ms.
ScanDuration	0x6e	1	3	*rti.h definition:* `RTI_SA_ITEM_SCAN_DURATION` *Valid for target only.* *This attribute is RW, and is used to obtain or specify the duration of a scanning operation, as specified by* ***Error! Reference source not found.****. The range of values include:* *0..14*
UserString	0x6f	15	Empty	*rti.h definition:* `RTI_SA_ITEM_USER_STRING` *Valid for both target and controller.* *This attribute is RW. The user defined character string used to identify this node.*
IEEEAddress	0x84	8	N/A	*rti.h definition:* `RCN_NIB_IEEE_ADDRESS` *Valid for both target and controller.* *This attribute is RW.* *Note that this attribute is overwritten at every power cycle with a commissioned IEEE address if such commissioned IEEE address is available.*
PanID	0x85	2	0xFFFF	*rti.h definition:* `RTI_SA_ITEM_PAN_ID` *Valid for target only.* *This attribute is RW, and is used to obtain or specify the PAN identifier of the node. The range of values include:* *0..0xFFFE – The PAN identifier of the node* *0xFFFF – Used when broadcasting, when the target is a Controller, or when transmission uses an IEEE address*
ShortAddress	0x86	2	0xFFFF	*rti.h definition:* `RTI_SA_ITEM_SHORT_ADDRESS` *Valid for target only.* *This attribute is RW, and is used to obtain or specify the short address of the node. It can have the following values:* *0..0xFFFE – The PAN identifier of the node* *0xFFFF – Used when broadcasting, when the target is a Controller, or when transmission uses an IEEE address*
AgilityEnable	0x87	1	1	*rti.h definition:* `RTI_SA_ITEM_AGILITY_ENABLE`

Attribute	Item ID	Length	Default	*Description*
				Valid for target only. *This attribute is RW, and is used to enable or disable frequency agility. It can have the following values:* *0 – Disable frequency agility* *1 – Enable frequency agility* *This attribute is not retained in non-volatile memory and hence the attribute value is reset to default value after every power cycle.*
TransmitPower	0x88	1	127	*rti.h definition:* `RTI_SA_ITEM_TRANSMIT_POWER` *Valid for both target and controller.* *This attribute is RW. Transmit power level in dBm of network layer packets except for key seed frames shall be determined by this attribute. The negative value must be represented as two's complement.* *The transmit power level is rounded up to the closest power level included in the recommended transmit power level register settings of the radio processor. If the attribute value is higher than the highest transmit power level among the recommended register settings, the transmit power level is set to such highest transmit power level among the recommended register settings. For example, the default value 127 is rounded down to 4dBm for CC2530.* *This attribute is not retained in non-volatile memory and hence the attribute value is reset to default value after every power cycle.*
Pairing Table (PT) Related Attributes				
NumberOfActivePTEntries	0xB0	1	Range	*rti.h definition:* `RTI_SA_ITEM_PT_NUMBER_OF_ACTIVE_ENTRIES` *Valid for both target and controller.* *This attribute is RO, and is used to obtain the number of active (i.e. valid) pairing table entries. The range of values include:* *0.. MaxPairingTableEntries*
CurrentPTEntryIndex	0xB1	1	Range	*rti.h definition:* `RTI_SA_ITEM_PT_CURRENT_ENTRY_INDEX` *Valid for both target and controller.* *This attribute is RW, and is used to obtain or set the pairing table reference index that will be or is being used by the RTI for subsequent pairing table attribute accesses. The range of values include:* *0.. (MaxPairingTableEntries-1) – Valid index* ***RTI_INVALID_PAIRING_REF*** *– Invalid Index*
CurrentPTEntry	*0xB2*	*Size of*	*Invalid*	*rti.h definition:*

Attribute	Item ID	Length	Default	*Description*

		entry[1]	*Entry*	RTI_SA_ITEM_PT_CURRENT_ENTRY
				
				Valid for both target and controller.
				
				This attribute is RW, and is used to obtain or set a pairing table entry indexed by CurrentPTEntryIndex attribute above.
				The attribute content corresponds to the following C structure type:
				typedef struct
				`{`
				`  uint8 pairingRef;`
				`  uint16 srcNwkAddress;`
				`  uint8 logicalChannel;`
				`  sAddrExt_t ieeeAddress;`
				`  uint16 panId;`
				`  uint16 nwkAddress;`
				`  uint8 recCapabilities;`
				`  uint8 securityKeyValid;`
				`  uint8 securityKey[RCN__SEC_KEY_LENGTH];`
				`  uint16 vendorIdentifier;`
				`  uint8 devTypeList[RCN_MAX_NUM_DEV_TYPES];`
				`  uint32 recFrameCounter;`
				`  uint8 profileDiscs[RCN_PROFILE_DISCS_SIZE];`
				`} rcnNwkPairingEntry_t;`
				Note that the C structure type could vary from a software version to another. The type definition will be included in interface header file.

Table 4: State Attributes Table

3.5 Constants

3.5.1 Overview

The Constants table contains constants that are defined and used by the RTI stack. The constants are read-only, and can not be modified by the application.

3.5.2 Constant list

The following table contains the constants of the Constants table:

Constant	Item ID	Length	Value	*Description*
SoftwareVersion	0xC0	1	x.y.z x – major revision y – minor revision z –	*rti.h definition:* RTI_CONST_ITEM_SW_VERSION *This constant specifies the software version. This value is specified as follows:* *0b'xxxyyyzz' where:*

[1] Size of *CurrentPTEntry* can be determined from C structure representing a pairing table entry.

Constant	Item ID	Length	Value	*Description*
			incremental revision	*xxx: Software Major Version* *yyy: Software Minor Version* *zz: Incremental Version*
MaxPairingTableEntries	0xC1	1	10	*rti.h definition:* `RTI_CONST_ITEM_MAX_PAIRING_TABLE_ENTRIES` *This constant specifies the maximum number of entries supported in the pairing table.*
ProtocolIdentifier	0xC2	1	0xCE	*rti.h definition:* `RTI_CONST_ITEM_NWK_PROTOCOL_IDENTIFIER` *This constant specifies the identifier being used by this device.*
ProtocolVersion	0xC3	1	0x01	*rti.h definition:* `RTI_CONST_ITEM_NWK_PROTOCOL_VERSION` *This constant specifies the protocol implemented on this device.*
OAD Image Id	0xD0	2	0x0000-0xFFFF	*Application specific utilization of this constant.*
RNP Image Id	*0xD1*	*1*	*0x00* *0x01* *0x02*	*RNP without any ZID functionality.* *RNP with ZID CLD-only functionality.* *RNP with ZID ADA-only functionality.*

Table 5: Constant List Table

3.6 RTI_ReadItem and RTI_WriteItem - Deprecated

These functions have been deprecated in favor of the newer, extended version of the API: RTI_ReadItemEx **and** RTI_WriteItemEx. Legacy code can continue to use this deprecated API by including the newer header file "rti_deprecated.h". Legacy code can be ported to the new API by setting the newly added 'profileId' parameter to `RTI_PROFILE_RTI` (defined in rti_constants.h).

3.7 ZID Attribute Read & Write

The ZID Profile Attributes enumerated in zid_profile.h can be read and written using the API described in RTI_ReadItemEx **and** RTI_WriteItemEx by setting the 'profileId' parameter to ZID: RTI_PROFILE_ZID (defined in rti.h). There may be proprietary extensions to the specified attributes in order to further facilitate use, control and interaction with the ZID Profile extension of the RemoTI stack. Such proprietary extensions would be found in zid_common.h and would of course not conflict with valid ranges specified by the ZID Profile.

3.8 ZID Non-Standard Descriptor Component Read/Write Interface

Since ZID Non-Standard Descriptor Components can be rather large, they are stored in NV memory using the same fragmenting scheme that is used to transfer them over the air. In order to hide this complexity from the application, a set of application helper functions has been created to allow applications to read/write the complete component without having to worry about how they are fragmented in NV. These application helper functions can be thought of as a "thin" abstraction layer that

lies on top of the RTI API; it is thus running where the application resides. This interface is captured in two files: zid_ada_app_helper.c and zid_cld_app_helper.c. The first file can be included by a ZID Adaptor side application, while the second file can be included by a ZID class device side application. Details about these functions can be found in the following sections.

3.8.1 zidAdaAppHlp_ReadNonStdDescComp

This function can be used by an adaptor side application to read non-standard descriptor components that have been pushed to it by a class device.

3.8.1.1 Prototype

```
rStatus_t
zidAdaAppHlp_ReadNonStdDescComp( uint8 pairIdx,
                                 uint8 descNum,
                                 zid_non_std_desc_comp_t *pBuf )
```

3.8.1.2 Input Parameters

pairIdx:	Pairing table index of class device whose non-standard descriptor component is to be read.
descNum:	Non-standard descriptor number. Range is 0 to *aplcMaxNonStdDescCompsPerHID* – 1.
pBuf:	Pointer to buffer to place non-standard descriptor component as defined in Table 9 of ZID Profile, r18.

3.8.1.3 Output Parameters

None.

3.8.1.4 Return

If successful, this function will return `RTI_SUCCESS`. If not successful, meaning the item was not able to be read from NV, this function will return `RTI_ERROR_INVALID_PARAMETER`.

3.8.2 zidCldAppHlp_WriteNonStdDescComp

This function can be used by a class device side application to store a non-standard descriptor component into NV so that it can be pushed to a paired ZID adaptor.

3.8.2.1 Prototype

```
rStatus_t
zidCldAppHlp_WriteNonStdDescComp( uint8 descNum,
                                  zid_non_std_desc_comp_t *pBuf )
```

3.8.2.2 Input Parameters

descNum:	Non-standard descriptor number. Range is 0 to *aplcMaxNonStdDescCompsPerHID* – 1.
pBuf:	Pointer to buffer to place non-standard descriptor component as defined in Table 9 of ZID Profile, **Error! Reference source not found.**.

3.8.2.3 Output Parameters

None.

3.8.2.4 Return

If successful, this function will return `RTI_SUCCESS`. If not successful, meaning the item was not able to be written to NV, this function will return `RTI_ERROR_INVALID_PARAMETER`.

3.8.3 zidCldAppHlp_WriteNullReport

This function can be used by a class device side application to store a NULL report corresponding to a non-standard descriptor component into NV so that it can be pushed to a paired ZID adaptor.

3.8.3.1 Prototype

```
rStatus_t
zidCldAppHlp_WriteNullReport( uint8 descNum,
```

```
                                    zid_null_report_t *pBuf )
```

3.8.3.2 Input Parameters

descNum:	Non-standard descriptor number that NULL report is for. Range is 0 to *aplcMaxNonStdDescCompsPerHID* – 1.
pBuf:	Pointer to NULL report as defined in Table 14 of ZID Profile, **Error! Reference source not found.**.

3.8.3.3 Output Parameters

None.

3.8.3.4 Return

If successful, this function will return RTI_SUCCESS. If not successful, meaning the item was not able to be written to NV, this function will return RTI_ERROR_INVALID_PARAMETER.

4. RTI Application Profile Interface

4.1 Overview

The RTI API allows a developer to access the application profile specified by the RF4CE standard. Some of the API's are Controller specific, some are Target specific, and some can be used by either the Controller or the Target. Please see a summary of which type of application can use which API in section 2.1.

4.2 RTI_InitReq

This function is used by the Controller or Target application to initialise the RemoTI stack and begin network operation. This function will load the Configuration Parameters table from NV memory, so as to capture any updates made by the application, and base start-up operation on the *StartupUp* parameter (please see Table 3 for details). The application is therefore expected to update the Configuration Parameters table (using the RTI_WriteItem API) prior to making this call since once the call to RTI_InitReq is made by the application, subsequent changes to the Configuration Parameters table, while they will take place, will not be used by the RTI stack until a reset occurs.

4.2.1 Prototype

```
void RTI_InitReq( void )
```

4.2.2 Input Parameters

None.

4.2.3 Output Parameters

None.

4.2.4 Return

None.

4.2.5 Notes

A call to RTI_InitCnf will occur as a consequence of this function call.

4.3 RTI_InitCnf

This function is used by the RTI stack to return the results of an application call to RTI_InitReq. This function is to be completed by the application, and as such, constitutes a callback.

4.3.1 Prototype

```
void RTI_InitCnf( rStatus_t status )
```

4.3.2 Input Parameters

status:	The resulting status from a call to RTI_InitReq. Possible values include (please see Table 6 for status descriptions): • RTI_SUCCESS • RTI_ERROR_RCN_INVALID_INDEX • RTI_ERROR_RCN_UNSUPPORTED_ATTRIBUTE • RTI_ERROR_UNKNOWN_PARAMETER

4.3.3 Output Parameters

None.

4.3.4 Return

None.

4.3.5 Notes

This call can be made to the application before the application's call to RTI_InitReq has returned.

4.4 RTI_PairReq

This function is used by either a Controller application or a Target application to initiate a pairing process with a Target. The pairing process actually consists of a discovery operation followed by a pairing operation.

4.4.1 Prototype

```
void RTI_PairReq( void )
```

4.4.2 Input Parameters

None.

4.4.3 Output Parameters

None.

4.4.4 Return

None.

4.4.5 Notes

The parameters to be used during pairing must be configured prior to calling this API. These parameters include ProfileIDs, Application Capabilities and Node Capabilities, see Configuration Parameters Table. A call to RTI_PairCnf will occur as a consequence of this function call.

4.5 RTI_PairCnf

This function is used by the RTI stack to return the results of an application call to RTI_PairReq. This function is to be completed by the application, and as such, constitutes a callback.

4.5.1 Prototype

```
void RTI_PairCnf( rStatus_t status,
                  uint8     dstIndex,
                  uint8     devType )
```

4.5.2 Input Parameters

status:	The resulting status from a call to RTI_PairReq. Possible values include (please see Table 6 for status descriptions): • RTI_SUCCESS • RTI_ERROR_NOT_PERMITTED • RTI_ERROR_OUT_OF_MEMORY • RTI_ERROR_MAC_TRANSACTION_EXPIRED • RTI_ERROR_MAC_TRANSACTION_OVERFLOW • RTI_ERROR_MAC_NO_RESOURCES • RTI_ERROR_MAC_UNSUPPORTED • RTI_ERROR_MAC_BAD_STATE • RTI_ERROR_MAC_TX_ABORTED • RTI_ERROR_MAC_NO_TIME • RTI_ERROR_MAC_CHANNEL_ACCESS_FAILURE

	• RTI_ERROR_MAC_ACK_PENDING • RTI_ERROR_MAC_NO_ACK
dstIndex:	The pairing table index of the paired device, or RTI_INVALID_PAIRING_REF if an error resulted.
devType:	The pairing table index device type, or RTI_INVALID_DEVICE_TYPE if an error resulted.

4.5.3 Output Parameters

None.

4.5.4 Return

None.

4.5.5 Notes

This call can be made to the application before the application's call to RTI_PairReq has returned.

4.6 RTI_PairAbortReq

This function is used by an application to abort an on-going pairing process with a Target.

4.6.1 Prototype

```
void RTI_PairAbortReq( void )
```

4.6.2 Input Parameters

None.

4.6.3 Output Parameters

None.

4.6.4 Return

None.

4.6.5 Notes

A call to `RTI_PairAbortCnf` will occur as a consequence of this function call.

4.7 RTI_PairAbortCnf

This function is used by the RTI stack to return the results of an application call to `RTI_PairAbortReq`. This function is to be completed by the application, and as such, constitutes a callback.

4.7.1 Prototype

```
void RTI_PairAbortCnf( rStatus_t status )
```

4.7.2 Input Parameters

status:	The resulting status from a call to RTI_PairAbortReq. Possible values include (please see Table 6 for status descriptions): • RTI_SUCCESS • RTI_ERROR_PAIR_COMPLETE

4.7.3 Output Parameters

None.

4.7.4 Return

None.

4.7.5 Notes

This call can be made to the application before the application's call to `RTI_PairAbortReq` has returned.

Pairing process may complete when pairing cannot be aborted in the middle.

4.8 RTI_AllowPairReq

This function is used by the Target application to ready the node for a pairing request, and thereby allow this node to respond.

4.8.1 Prototype

```
void RTI_AllowPairReq( void )
```

4.8.2 Input Parameters

None.

4.8.3 Output Parameters

None.

4.8.4 Return

None.

4.8.5 Notes

A call to `RTI_AllowPairCnf` will occur as a consequence of this function call.

4.9 RTI_AllowPairCnf

This function is used by the RTI stack to return the results of an application call to `RTI_AllowPairReq`. This function is to be completed by the application, and as such, constitutes a callback.

4.9.1 Prototype

```
void RTI_AllowPairCnf( rStatus_t status,
                       uint8     dstIndex,
                       uint8     devType )
```

4.9.2 Input Parameters

status:	The resulting status from a call to RTI_AllowPairReq. Possible values include (please see Table 6 for status descriptions): • RTI_SUCCESS • RTI_ERROR_OSAL_NO_TIMER_AVAIL • RTI_ERROR_ALLOW_PAIRING_TIMEOUT
dstIndex:	The pairing table index of the paired device, or `RTI_INVALID_PAIRING_REF` if an error resulted.
devType:	The pairing table index device type, or `RTI_INVALID_DEVICE_TYPE` if an error resulted.

4.9.3 Output Parameters

None.

4.9.4 Return

None.

4.9.5 Notes

This call can be made to the application before the application's call to `RTI_AllowPairReq` has returned.

4.10 RTI_AllowPairAbortReq

This function is used by a Target application to cancel previous `RTI_AllowPairReq()` call.

4.10.1 Prototype

```
void RTI_AllowPairAbortReq( void )
```

4.10.2 Input Parameters

None.

4.10.3 Output Parameters

None.

4.10.4 Return

None.

4.10.5 Notes

This function call does not trigger any callback function. RTI may still complete on-going allow-pairing request triggered by `RTI_AllowPairReq()` if it cannot be aborted and hence may generate `RTI_AllowPairCnf()` callback function in case the request was not cancelled.

4.11 RTI_UnpairReq

This function is used by either the Controller application or the Target application to trigger removing pairing entry.

4.11.1 Prototype

```
void RTI_UnpairReq( uint8 dstIndex )
```

4.11.2 Input Parameters

dstIndex:	Specifies the index to the pairing table entry which is desired to be removed.

4.11.3 Output Parameters

None.

4.11.4 Return

None.

4.11.5 Notes

A call to `RTI_UnpairCnf` will occur as a consequence of this function call.

4.12 RTI_UnpairCnf

This function is used by the RTI stack to return the results of an application call to `RTI_UnpairReq`. This function is to be completed by the application, and as such, constitutes a callback.

4.12.1 Prototype

```
void RTI_UnpairCnf( rStatus_t status,
                    uint8     dstIndex )
```

4.12.2 Input Parameters

status:	The resulting status from a call to RTI_UnpairReq. Possible values include (please see Table 6 for status descriptions): • RTI_SUCCESS • RTI_ERROR_NOT_PERMITTED • RTI_ERROR_OUT_OF_MEMORY • RTI_ERROR_MAC_TRANSACTION_EXPIRED • RTI_ERROR_MAC_TRANSACTION_OVERFLOW • RTI_ERROR_MAC_NO_RESOURCES • RTI_ERROR_MAC_UNSUPPORTED • RTI_ERROR_MAC_BAD_STATE • RTI_ERROR_MAC_TX_ABORTED • RTI_ERROR_MAC_NO_TIME • RTI_ERROR_MAC_CHANNEL_ACCESS_FAILURE • RTI_ERROR_MAC_ACK_PENDING • RTI_ERROR_MAC_NO_ACK
dstIndex:	The pairing table index of the removed pairing entry.

4.12.3 Output Parameters

None.

4.12.4 Return

None.

4.12.5 Notes

This call can be made to the application before the application's call to RTI_UnpairReq has returned.

Regardless of the resulting status value, the pairing entry is removed from the local pairing table when this callback function is called.

4.13 RTI_UnpairInd

This function is used by the RTI stack to indicate that a paired device has removed a pairing table entry of this device. This function is to be completed by the application, and as such, constitutes a callback.

4.13.1 Prototype

```
void RTI_UnpairInd( uint8     dstIndex )
```

4.13.2 Input Parameters

dstIndex:	The pairing table index of the paired device which requested removal of the pairing entry.

4.13.3 Output Parameters

None.

4.13.4 Return

None.

4.13.5 Notes

By the time this call is made, the pairing entry corresponding to the pairing table index is removed.

4.14 RTI_SendDataReq

This function is used by the Controller or Target application to send a data packet to another RF4CE device.

4.14.1 Prototype

```
void RTI_SendDataReq( uint8  dstIndex,
                      uint8  profileId,
                      uint16 vendorId,
                      uint8  txOptions,
                      uint8  len,
                      uint8  *pData )
```

4.14.2 Input Parameters

dstIndex:	Specifies the index to the pairing table entry which contains the information required to transmit the data. Note that this parameter is ignored if the txOptions parameter specifies a broadcast transmission. Before sending ZID-specific commands/reports, if more than just the ZID Profile is being supported, then the Application has the burden to check whether the currently selected pairing table entry supports ZID (this check is made on the CurrentPTEntry.profileDiscs[], bit 2 or masked as 0x04).
profileId:	Specifies the identifier of the profile which indicates the format of the transmitted data. Possible values include: • RTI_PROFILE_ZRC • RTI_PROFILE_ZID
vendorId:	If txOptions parameter specifies this data is vendor specific, then this field specifies the identifier of the vendor transmitting the data. Possible values include (please see **Error! Reference source not found.** for more detail): • RTI_VENDOR_PANASONIC • RTI_VENDOR_SONY • RTI_VENDOR_SAMSUNG • RTI_VENDOR_PHILIPS • RTI_VENDOR_FREESCALE • RTI_VENDOR_OKI • RTI_VENDOR_TEXAS_INSTRUMENTS If on the other side the data is not vendor specific but rather standard, this parameter is ignored
txOptions:	Specifies the transmission options to be used. One or more of the following transmission options can be specified (please see **Error! Reference source not found.** for additional details): • RTI_TX_OPTION_BROADCAST • RTI_TX_OPTION_IEEE_ADDRESS • RTI_TX_OPTION_ACKNOWLEDGED • RTI_TX_OPTION_SECURITY • RTI_TX_OPTION_SINGLE_CHANNEL

- RTI_TX_OPTION_CHANNEL_DESIGNATOR
- RTI_TX_OPTION_VENDOR_SPECIFIC

Note that ZID has pre-defined “transmission pipes” which can be easily set using the constants pre-defined by `ZID_TX_OPTION_`... macros in zid_common.h (e.g. `ZID_TX_OPTION_INTERRUPT_PIPE`).

len: Specifies the number of bytes of data to be sent.

***pData:** Specifies a pointer to the data to be sent.

4.14.3 Output Parameters

None.

4.14.4 Return

None.

4.14.5 Notes

A call to RTI_SendDataCnf will occur as a consequence of this function call. There is no queue in the stack so RTI_SendDataCnf may be called with an unsuccessful status before this API returns. Thus it is important to manage the state before calling this API.

4.15 RTI_SendDataCnf

This function is used by the RTI stack to return the results of an application call to `RTI_SendDataReq`. This function is to be completed by the application, and as such, constitutes a callback.

4.15.1 Prototype

```
void RTI_SendDataCnf( rStatus_t status )
```

4.15.2 Input Parameters

status: The resulting status from a call to RTI_SendDataReq. Possible values include (please see Table 6 for status descriptions):

- RTI_SUCCESS
- RTI_ERROR_NO_PAIRING_INDEX
- RTI_ERROR_OUT_OF_MEMORY
- RTI_ERROR_NOT_PERMITTED
- RTI_ERROR_MAC_BEACON_LOSS
- RTI_ERROR_MAC_PAN_ID_CONFLICT
- RTI_ERROR_MAC_SCAN_IN_PROGRESS

4.15.3 Output Parameters

None.

4.15.4 Return

None.

4.15.5 Notes

This call can be made to the application before the application’s call to `RTI_SendDataReq` has returned. Hence it is important to control the application state before calling `RTI_SendDataReq`.

4.16 RTI_ReceiveDataInd

This function is used by the RTI stack to indicate to the Controller or Target application that data has been received from another RF4CE device. This function is to be completed by the application, and as such, constitutes a callback. Note that the ZID Profile extension to the RemoTI stack intercepts incoming

ZID data frames that are specific to the implementation of the ZID state machine and configuration (thus the importance of having all ZID attributes properly configured before initiating pairing).

4.16.1 Prototype

```
void RTI_ReceiveDataInd( uint8  srcIndex,
                         uint8  profileId,
                         uint16 vendorId,
                         uint8  rxLQI,
                         uint8  rxFlags,
                         uint8  len,
                         uint8  *pData )
```

4.16.2 Input Parameters

srcIndex:	Specifies the index to the pairing table entry which contains the information about the source node the data was received from.
profileId:	Specifies the identifier of the profile which indicates the format of the transmitted data. Possible values include: • RTI_PROFILE_ZRC • RTI_PROFILE_ZID
vendorId:	If the RxFlags parameter specifies that the data is vendor specific, this field indicates the identifier of the vendor transmitting the data. Possible values include (please see **Error! Reference source not found.** for more detail): • RTI_VENDOR_PANASONIC • RTI_VENDOR_SONY • RTI_VENDOR_SAMSUNG • RTI_VENDOR_PHILIPS • RTI_VENDOR_FREESCALE • RTI_VENDOR_OKI • RTI_VENDOR_TEXAS_INSTRUMENTS If the RxFlags parameter specifies that the data is not vendor specific this parameter shall be ignored.
rxLQI:	Specifies the Link Quality Indication.
rxFlags:	Specifies the reception indication flags. One or more of the following reception indication flags can be specified (please see **Error! Reference source not found.** for additional details): • RTI_RX_FLAGS_BROADCAST • RTI_RX_FLAGS_SECURITY • RTI_RX_FLAGS_VENDOR_SPECIFIC
len:	Specifies the number of bytes received.
***pData:**	Specifies a pointer to the data received.

4.16.3 Output Parameters

None.

4.16.4 Return

None.

4.16.5 Notes

None.

4.17 RTI_StandbyReq

This function is used by the Target application to place this node into standby mode. The properties of the standby consist of the active period and the duty cycle. These values are set by the application by setting

the corresponding parameters in the Configuration Parameters table. Note that duty cycle has to be greater than active period. Otherwise, software behaviour is unpredictable. The only exception is that duty cycle value zero (0) is used to indicate that no duty cycling is requested for the standby mode operation, i.e., standby mode when duty cycle value is zero is effectively disabling receiver forever.

4.17.1 Prototype

```
void RTI_StandbyReq ( uint8 mode )
```

4.17.2 Input Parameters

mode:	Specifies whether standby mode is enabled or disabled. The following values are permitted: • RTI_STANDBY_OFF • RTI_STANDBY_ON

4.17.3 Output Parameters

None.

4.17.4 Return

None.

4.17.5 Notes

A call to `RTI_StandbyCnf` will occur as a consequence of this function call.

This node will automatically come out of standby mode when it receives a data packet, and return to standby mode when nothing else is to be done. Standby mode will also persist across warm resets. The application is responsible for explicitly enabling and disabling standby using the `RTI_StandbyReq` API.

4.18 RTI_StandbyCnf

This function is used by the RTI stack to return the results of an application call to `RTI_StandbyReq`. This function is to be completed by the application, and as such, constitutes a callback.

4.18.1 Prototype

```
void RTI_StandbyCnf( rStatus_t status )
```

4.18.2 Input Parameters

status:	The resulting status from a call to RTI_StandbyReq. Possible values include (please see Table 6 for status descriptions): • RTI_SUCCESS • RTI_ERROR_INVALID_PARAMETER • RTI_ERROR_UNSUPPORTED_ATTRIBUTE • RTI_ERROR_INVALID_INDEX • RTI_ERROR_UNKNOWN_STATUS_RETURNED

4.18.3 Output Parameters

None.

4.18.4 Return

None.

4.18.5 Notes

This call can be made to the application before the application's call to `RTI_StandbyReq` has returned.

4.19 RTI_RxEnableReq

This function is used by the Controller or the Target application to enable the radio receiver, enable the radio receiver for a specified amount of time, or disable the radio receiver. The target node operates in standby mode once its received is initially turned off by use of this function call, regardless of prior `RTI_StandbyReq` function call. Vice versa, `RTI_StandbyReq` function call overrules previous `RTI_RxEnableReq` function call.

4.19.1 Prototype

```
void RTI_RxEnableReq ( uint16 duration )
```

4.19.2 Input Parameters

duration:	A value that specifies if the receiver is off, on for a specified duration and then off, or on till another function call to change the state of receiver. Possible values include a value from 1 to 0xFFFE (in 34milliseconds) or: • RTI_RX_ENABLE_OFF • RTI_RX_ENABLE_ON

4.19.3 Output Parameters

None.

4.19.4 Return

None.

4.19.5 Notes

A call to `RTI_RxEnableCnf` will occur as a consequence of this function call.

4.20 RTI_RxEnableCnf

This function is used by the RTI stack to return the results of an application call to `RTI_RxEnableReq`. This function is to be completed by the application, and as such, constitutes a callback.

4.20.1 Prototype

```
void RTI_RxEnableCnf( rStatus_t status )
```

4.20.2 Input Parameters

status:	The resulting status from a call to RTI_RxEnableReq. Possible values include (please see Table 6 for status descriptions): • RTI_SUCCESS • RTI_ERROR_INVALID_PARAMETER

4.20.3 Output Parameters

None.

4.20.4 Return

None.

4.20.5 Notes

This call can be made to the application before the application's call to `RTI_RxEnableReq` has returned.

4.21 RTI_EnableSleepReq

This function is used by the Controller or the Target application to enable power savings mode for this node.

4.21.1 Prototype

```
void RTI_EnableSleepReq ( void )
```

4.21.2 Input Parameters

None.

4.21.3 Output Parameters

None.

4.21.4 Return

None.

4.21.5 Notes

To gain power saving for Target one must first call RTI_StandbyReq, otherwise the radio and MCU will remain ON constantly. A call to `RTI_EnableSleepCnf` will occur as a consequence of this function call.

4.22 RTI_EnableSleepCnf

This function is used by the RTI stack to return the results of an application call to `RTI_EnableSleepReq`. This function is to be completed by the application, and as such, constitutes a callback.

4.22.1 Prototype

```
void RTI_EnableSleepCnf( rStatus_t status )
```

4.22.2 Input Parameters

status:	The resulting status from a call to RTI_EnableSleepReq. Possible values include (please see Table 6 for status descriptions): • RTI_SUCCESS

4.22.3 Output Parameters

None.

4.22.4 Return

None.

4.22.5 Notes

This call can be made to the application before the application's call to `RTI_EnableSleepReq` has returned.

4.23 RTI_DisableSleepReq

This function is used by the Controller or the Target application to disable power savings mode for this node.

4.23.1 Prototype

```
void RTI_DisableSleepReq ( void )
```

4.23.2 Input Parameters

None.

4.23.3 Output Parameters

None.

4.23.4 Return

None.

4.23.5 Notes

A call to `RTI_DisableSleepCnf` will occur as a consequence of this function call.

4.24 RTI_DisableSleepCnf

This function is used by the RTI stack to return the results of an application call to `RTI_DisableSleepReq`. This function is to be completed by the application, and as such, constitutes a callback.

4.24.1 Prototype

```
void RTI_DisableSleepCnf( rStatus_t status )
```

4.24.2 Input Parameters

status:	The resulting status from a call to RTI_DisableSleepReq. Possible values include (please see Table 6 for status descriptions): • RTI_SUCCESS

4.24.3 Output Parameters

None.

4.24.4 Return

None.

4.24.5 Notes

This call can be made to the application before the application's call to `RTI_DisableSleepReq` has returned.

5. RTI Test Mode interface

This interface allows application to access test modes in the RemoTI stack. The functions described in this chapter are compiled only when FEATURE_TEST_MODE compile flag is defined.

5.1 RTI_TestModeReq

This command is used to place the radio in different test modes.

```
void RTI_TestModeReq( uint8 mode, int8 txPower, uint8 channel )
```

Test mode	*Description*
0x00	*The device will transmit unmodulated carrier with the specified frequency and transmit power*
0x01	*The device will transmit pseudo-random data with the specified frequency and transmit power*
0x02	*The device will have the radio placed in receive mode on the specified frequency.*

Note that executing this command will leave the radio in a different configuration than is needed for regular operation. It is expected that the device will have to be reset before it can be used again for regular RF operations. . Also, the RemoTI software should be idling when this function is called. For instance, no non-test-mode RTI calls must be made after calling this function and the device should not enter power savings mode (e.g., OSAL power manager must be in PWRMGR_ALWAYS_ON state prior to this call. See **Error! Reference source not found.**). To ensure RemoTI software is idling, it is recommended to reset the device before calling this function.

Input parameter txPower indicates transmit power level in dBm unit. The actual transmit power level is set to the closest higher or equal transmit power level included in recommended register settings of the radio processor. If the txPower is greater than the highest transmit power level in recommended register settings, the actual transmit power level is set to the highest transmit power level in recommended register settings.

Input parameter channel designates MAC channel number to set the device on.

5.2 RTI_TestRxCounterGetReq

This function retrieves 16 bit receiver counter value which is updated when RTI is set to test mode 0x02 and resets the counter value if resetFlag is set to TRUE.

```
uint16 RTI_TestRxCounterGetReq( uint8 resetFlag )
```

5.3 RTI_SwResetReq

This function resets processor.

```
void RTI_SwResetReq( void )
```

6. Summary of RTI Return Status Values

The status parameter that is returned from the RemoTI stack may take one of the following values.

Status	Value	*Description*
RTI_SUCCESS	0x00	*Operation completed normally.*
RTI_ERROR_INVALID_INDEX	0xF9	*An invalid pairing table index was passed as a parameter.*
RTI_ERROR_INVALID_PARAMETER	0xE8	*An invalid parameter was passed to an API call.*
RTI_ERROR_UNSUPPORTED_ATTRIBUTE	0xF4	*An item identifier to access a table attribute was invalid.*
RTI_ERROR_NOT_PERMITTED	0xB4	*An attempt was made to pair when the RTI stack was not ready to perform a discovery/pair operation.*
RTI_ERROR_NO_ORG_CAPACITY	0xB0	*A pairing operation required additional pairing table space, but the pairing table of the pairing originator is full.*
RTI_ERROR_NO_REC_CAPACITY	0xB1	*A pairing operation required additional pairing table space, but the pairing table of the pairing recipient is full.*
RTI_ERROR_NO_RESPONSE	0xB3	*A request to pair timed out.*
RTI_ERROR_FRAME_COUNTER_EXPIRED	0xB6	*Frame counter has reached its maximum value.*
RTI_ERROR_DISCOVERY_ERROR	0xB7	*Failed pairing because more than one target device have been discovered.*
RTI_ERROR_DISCOVERY_TIMEOUT	0xB8	*Failed pairing as no target device has been discovered.*
RTI_ERROR_SECURITY_TIMEOUT	0xB9	*Failed pairing as security key exchange timed out.*
RTI_ERROR_SECURITY_FAILURE	0xBA	*Failed pairing as security key verification failed.*
RTI_ERROR_NO_SECURITY_KEY	0xBD	*Requested encryption while no security key is set up.*
RTI_ERROR_OUT_OF_MEMORY	0xBE	*The operation requested required a memory resource that is not available.*
RTI_ERROR_ALLOW_PAIRING_TIMEOUT	0x23	*A timeout occurred before a pairing operation took place.*
RTI_ERROR_UNKNOWN_STATUS_RETURNED	0x20	*A status from one of the RTI's subsystems returned a status that currently is not mapped to a RTI status.*
RTI_ERROR_OSAL_NO_TIMER_AVAIL	0x08	*The operation requested required an OS timer resource that is not available.*
RTI_ERROR_OSAL_NV_OPER_FAILED	0x0A	*The operation requested required access to OS NV memory, which failed.*
RTI_ERROR_OSAL_NV_ITEM_UNINIT	0x09	*The operation requested required access to an OS NV memory item that has not been previously initialized.*
RTI_ERROR_OSAL_NV_BAD_ITEM_LEN	0x0C	*The operation requested required access to an OS NV memory item using an invalid length.*
RTI_ERROR_MAC_TRANSACTION_EXPIRED	0xF0	*The operation requested required a MAC*

Status	Value	*Description*
		transaction which timed out.
RTI_ERROR_MAC_TRANSACTION_OVERFLOW	0xF1	*The operation requested required a MAC transaction that failed because a data queue was already full.*
RTI_ERROR_MAC_NO_RESOURCES	0x1A	*The operation requested required a MAC resource that is unavailable.*
RTI_ERROR_MAC_BAD_STATE	0x19	*The operation requested required a MAC operation that failed because it was received in a state that was not allowed.*
RTI_ERROR_MAC_CHANNEL_ACCESS_FAILURE	0xE1	*The operation requested required a MAC operation that failed because of activity on the channel.*
RTI_ERROR_MAC_NO_ACK	0xE9	*The operation requested required a MAC operation that failed because no acknowledgement was received.*
RTI_ERROR_MAC_BEACON_LOST	0xE0	*The operation requested required a MAC operation that failed because the beacon was lost following a synchronization request.*
RTI_ERROR_MAC_PAN_ID_CONFLICT	0xEE	*The operation requested required a MAC operation that failed because a PAN identifier conflict was detected.*
RTI_ERROR_MAC_SCAN_IN_PROGRESS	0xFC	*The operation requested required a MAC scan operation that failed because a scan was already in progress.*
RTI_ERROR_FAILED_TO_DISCOVER	0x21	*The RTI stack failed to discovery, as part of API pair request.*
RTI_ERROR_FAILED_TO_PAIR	0x22	*The RTI stack discovered, but failed to pair a target.*
RTI_ERROR_FAILED_TO_CONFIGURE_ZRC	0x50	*The configuration phase following the pairing process failed at the ZRC profile.*
RTI_ERROR_FAILED_TO_CONFIGURE_ZID	0x51	*The configuration phase following the pairing process failed at the ZID profile.*
RTI_ERROR_NO_PAIRING_INDEX	0xB2	*The pairing table reference passed was not valid.*
RTI_ERROR_ALLOW_PAIRING_TIMEOUT	0x23	*The target RTI stack was set to allow pairing, but none occurred before it timed out.*
RTI_ERROR_PAIR_COMPLETE	0x24	*Pairing process already completed and hence cannot be aborted.*
RTI_ERROR_SYNCHRONOUS_NPI_TIMEOUT	*0xFF*	*The synchronous request via the inter-processor interface timed out.*

Table 6: RTI API Return Status Summary

7. RCN API

7.1 Overview

RCN API implements network layer primitives defined in **Error! Reference source not found.**.

RCN API comprises of C functions callable from application and callback functions callable from RCN. The C functions callable from application correspond to request and response primitives defined in the RF4CE network layer specification. The callback function corresponds to confirmation and indication primitives. The application/upper layer has to implement RCN callback functions.

Parameters of the primitives are implemented either as C structure fields or as function arguments and return values. They should be interpreted in the same way as in RF4CE network layer specification with the exception of parameters relating to time durations. The parameters representing time duration use unit of milliseconds for every relevant RCN API, instead of symbol unit as defined in the RF4CE network layer specification. Such time duration parameter shall be of uint16 (16 bit) data type and whenever symbol value 0xffffff indicate a special purpose in RF4CE network layer specification, the value of 0xffff will be used for the same purpose (for example, as in `NLME-RX-ENABLE.request`).

Request and responses are accepted by RCN only in a proper state. For example, RCN will send confirmation with failure (not permitted) when `NLME-DISCOVERY.request` is issued without `NLME-RESET.request (SetDefaultNIB=TRUE)` and `NLME-START.request` after cold boot up.

Function prototypes and structure definitions are included in "`rcn_nwk.h`" file and NIB attribute indices are defined in "`rcn_attribs.h`" along with pairing table structure.

Support of certain RCN API functions are removed from controller-specific library (See 1.3.3) to optimize code size. See Table 7 for API functions supported in each library. Note that such selection is tied only to which library to use and that a controller node can still support discovery recipient feature (`RCN_NlmeAutoDiscoveryReq` and `RCN_NlmeDiscoveryRsp`) if combined library is used.

RCN API	Controller only library	*Combined library*
`RCN_CbackEvent`	♦	♦
`RCN_CbackRxCount`	♦	♦
`RCN_NldeDataAlloc`	♦	♦
`RCN_NldeDataReq`	♦	♦
`RCN_NlmeAutoDiscoveryReq`		♦
`RCN_NlmeAutoDiscoveryAbortReq`		♦
`RCN_NlmeDiscoveryReq`	♦	♦
`RCN_NlmeDiscoveryAbortReq`	♦	♦
`RCN_NlmeDiscoveryRsp`		♦
`RCN_NlmeGetReq`	♦	♦
`RCN_NlmePairReq`	♦	♦
`RCN_NlmePairRsp`		♦
`RCN_NlmeResetReq`	♦	♦

RCN API	Controller only library	*Combined library*
RCN_NlmeRxEnableReq	♦	♦
RCN_NlmeSetReq	♦	♦
RCN_NlmeStartReq	♦	♦
RCN_NlmeUnpairReq	♦	♦
RCN_NlmeUnpairRsp	♦	♦
RCN_NlmeUpdateKeyReq	♦	♦

Table 7: RCN API support

7.2 RCN_CbackEvent

This is a callback function which application should implement to receive RCN events corresponding to confirmation and indication primitives.

7.2.1 Prototype

```
void RCN_CbackEvent( rcnCbackEvent_t *pEvent )
```

7.2.2 Input Parameters

pEvent: Pointer to an event structure.

7.2.3 Output Parameters

None.

7.2.4 Return

None.

7.2.5 Notes

Event structure corresponds to union of network layer indication and confirmation primitives. One RCN_CbackEvent() function call will be made per one such primitive with an exception which will be described in a separate section. "eventId" field of the event structure indicates which primitive the event is associated with. The value of callback event id is defined in "rcn_nwk.h" file as "RCN_NLDE_DATA_IND", "RCN_NLDE_DATA_CNF", "RCN_NLME_COMM_STATUS_IND", etc. using similar names as standard primitive names.

"_IND" postfix corresponds to indication primitive and "_CNF" postfix corresponds to confirmation primitive.

Event structure includes "prim" field which is a union of all primitive structures such as "dataInd" field for "RCN_NLDE_DATA_IND" event, "dataCnf" for "RCN_NLDE_DATA_CNF" event, "commStatusInd" field for "RCN_NLME_COMM_STATUS_IND" event and so on. The individual union field is a C structure which looks very similar to a standard primitive, with a few exceptions which will be described in separate sections.

An example RCN_CbackEvent() function implementation will look like the following:

```
void RCN_CbackEvent( rcnCbackEvent_t *pEvent )
{
switch (pEvent->eventId)
{
case RCN_NLDE_DATA_IND:
```

```
  /* handle NLDE-DATA.indication primitive */
  /* … */
  break;
case RCN_NLDE_DATA_CNF:
  /* NLDE-DATA.confirm primitive */
  if (pEvent->prim.dataCnf.status == RCN_SUCCESS)
  {
    /* Status parameter of NLDE-DATA.confirm is SUCCESS */
    /* handle successful data confirm */
    /* … */
  }
  else
  {
     /* handle other cases */
     /* … */
  }
  break;
/* other case statements */
/* … */
}
}
```

7.3 RCN_CbackRxCount

This is another callback function from RCN.

RCN shall call this function to indicate that a CRC passed MAC frame was received. The application should increment its receive counter for receive test mode implementation upon this function call.

7.3.1 Prototype

void RCN_CbackRxCount(void)

7.3.2 Input Parameters

None.

7.3.3 Output Parameters

None.

7.3.4 Return

None.

7.4 RCN_NldeDataAlloc

This function is a special function to allocate a memory block where NSDU should be copied into before making `RCN_NldeDataReq()` function call.

This function is introduced to optimize memory copy operation for NLDE-DATA.request service provided by RCN.

7.4.1 Prototype

*uint8 RCN_NldeDataAlloc(rcnNldeDataReq_t *pPrim)*

7.4.2 Input Parameters

pPrim:	Pointer to a C structure corresponding to NLDE-DATA.request primitive.

7.4.3 Output Parameters

pPrim:	Nsdu field of the C structure pointed by this pointer will be set to address of newly allocated memory block if this function successfully allocates memory.

7.4.4 Return

Either RCN_SUCCESS for successful memory allocation or RCN_ERROR_OUT_OF_MEMORY in case memory block for NSDU cannot be allocated.

7.4.5 Notes

Application should fill in all rcnNldeDataReq_t structure fields except for "internal" and "nsdu" fields, before calling this function. Actual network layer payload should be copied into memory block pointed by nsdu after this function returns.

7.5 RCN_NldeDataReq

This function corresponds to sending NLDE-DATA.request primitive to network layer.

7.5.1 Prototype

```
void RCN_NldeDataReq( rcnNldeDataReq_t *pPrim )
```

7.5.2 Input Parameters

pPrim:	Pointer to a C structure corresponding to NLDE-DATA.request primitive.

7.5.3 Output Parameters

None.

7.5.4 Return

None.

7.5.5 Notes

A call to RCN_CbackEvent will occur as a consequence of this function call, corresponding to NLDE-DATA.confirm primitive. Its argument pEvent->eventId shall be set to RCN_NLDE_DATA_CNF and pEvent->prim.dataCnf will be the valid union field in such callback.

This function does not queue data transmission request when the request cannot be handled immediately. Hence, application has to queue data transmissions by itself, if necessary. When this function cannot handle data transmission request immediately, RCN_CbackEvent() will be called immediately with pEvent->prim.dataCnf.status == RCN_ERROR_NOT_PERMITTED.

Application must not set pPrim->nsdu by itself, but rather call RCN_NldeDataAlloc() function to allocate memory for NSDU.

The following is an example code making RCN_NldeDataReq() call.

```
{
static uint8 myExampleCommand[] = { 0xFF, 0x12, 0x34 };
/* Generate NLDE-DATA.request for myExampleCommand */
rcnNldeDataReq_t myPrim;

myPrim.pairingRef = destinationPairingRef;
myPrim.profileId = 0x01; // ZRC profile
myPrim.vendorId = 0x07; // TI vendor specific command
myPrim.txOptions = RCN_TX_OPTION_ACKNOWLEDGED |
                   RCN_TX_OPTION_VENDOR_SPECIFIC;
myPrim.nsduLength = sizeof(myExampleCommand);

/* allocate memory for NSDU */
if (RCN_NldeDataAlloc(&myPrim) == RCN_SUCCESS)
{
  /* Copy NSDU into allocated memory buffer */
  myMemcpy( myPrim.nsdu, myExampleCommand, sizeof(myExampleCommand) );
```

```
  /* Issue the primitive */
  RCN_NldeDataReq( &myPrim );
}
else
{
   /* handle memory allocation failure */
}
}
```

7.6 RCN_NlmeDiscoveryReq

This function corresponds to sending NLME-DISCOVERY.request to network layer.

7.6.1 Prototype

*void RCN_NlmeDiscoveryReq(rcnNlmeDiscoveryReq_t *pPrim)*

7.6.2 Input Parameters

pPrim:	Pointer to a C structure corresponding to NLME-DISCOVERY.request primitive.

7.6.3 Output Parameters

None.

7.6.4 Return

None.

7.6.5 Notes

One or more calls to RCN_CbackEvent() will occur as a consequence of this function call, corresponding to NLME-DISCOVERY.confirm.

NLME-DISCOVERY.confirm is implemented in multiple event callbacks in RCN. RCN will call RCN_CbackEvent() with its argument pEvent->eventId set to RCN_NLME_DISCOVERED_EVENT and its pEvent->prim.discoveredEvent union field valid one per each node descriptors of NLME-DISCOVERY.confirm primitive. Upon making all such callbacks for every node descriptors, RCN will call RCN_CbackEvent() with its argument pEvent->eventId set to RCN_NLME_DISCOVERY_CNF and its pEvent->prim.discoveryCnf union field valid to complete the NLME-DISCOVERY.confirm primitive.

In other words, node descriptor list is split into individual pEvent->prim.discoveredEvent structures and pEvent->prim.discoveryCnf structure. The discoveredEvent structure contains a single node descriptor structure and discoveryCnf structure contain all the remaining parameters of NLME-DISCOVERY.confirm.

The reason the single primitive is split into multiple event callbacks is to optimize memory usage. Application can make decision on whether or not to store node descriptor as it gets callback for RCN_NLME_DISCOVERED_EVENT instead of network layer having to allocate memory to store all node descriptors and pass them all up to application which application may choose to use only one of them. ZRC profile application, for example, will not use more than one node descriptor.

The following is an example code snippet for RCN_CbackEvent() function handling NLME-DISCOVERY.confirm for ZRC profile.

```
void RCN_CbackEvent( rcnCbackEvent_t *pEvent )
{
static rcnNlmeDiscoveredEvent_t myNodeDescCache;
static enum
{
  NO_VALID_DESC, /* no valid descriptor */
```

```
  VALID_DESC     /* at least one valid descriptor */
} discoveryState = NO_VALID_DESC;

switch (pEvent->eventId)
{
case RCN_NLME_DISCOVERED_EVENT:
  /* handle Node Descriptor of NLME-DISCOVERY.confirm */
  /* ZRC profile will use only one node descriptor and hence
   * only one node descriptor storage is provided and it will be
   * overwritten. */
  if (pEvent->prim.discoveredEvent.status == RCN_SUCCESS)
  {
    /* Status field of the node descriptor is SUCCESS */
    myMemcpy(myNodeDescCache, pEvent->prim.discoveredEvent,
             sizeof(myNodeDescCache));
    discoveryState = VALID_DESC;
  }
  break;
case RCN_NLME_DISCOVERY_CNF:
  /* NLME-DISCOVERY.confirm primitive conclusion */
  if (pEvent->prim.discoveryCnf.status == RCN_SUCCESS &&
      discoveryState == VALID_DESC)
  {
     /* Successful discovery.
Use myNodeDescCache to build NLME-PAIR.request primitive */
    /* … */
  }
  else
  {
     /* handle discovery failure */
     /* … */
  }
  discoveryState = NO_VALID_DESC; // Reset the state
  break;
/* other case statements */
/* … */
}
}
```

7.7 RCN_NlmeDiscoveryAbortReq

This function aborts discovery procedure triggered by a prior `RCN_NlmeDiscoveryReq()` call.

7.7.1 Prototype

`void RCN_NlmeDiscoveryAbortReq( void )`

7.7.2 Input Parameters

None.

7.7.3 Output Parameters

None.

7.7.4 Return

None.

7.7.5 Notes

This function does not correspond to any functionality in **Error! Reference source not found.**. When this function is called, the discovery procedure is aborted immediately or when the pending MAC transmission of discovery request command frame is completed. However, due to race condition, the application which

issued this function call may still get RCN_CbackEvent() callback for RCN_NLME_DISCOVERY_CNF event.

A call to RCN_CbackEvent() will occur as a consequence of this function call with eventId field set to RCN_NLME_DISCOVERY_ABORT_CNF. The application may use this callback event to know when to initiate other actions such as transmitting a data.

7.8 RCN_NlmeDiscoveryRsp

This function corresponds to sending NLME-DISCOVERY.response to network layer.

7.8.1 Prototype

```
void RCN_NlmeDiscoveryRsp( rcnNlmeDiscoveryRsp_t *pPrim )
```

7.8.2 Input Parameters

pPrim:	Pointer to a C structure corresponding to NLME-DISCOVERY.response primitive.

7.8.3 Output Parameters

None.

7.8.4 Return

None.

7.8.5 Notes

A call to RCN_CbackEvent() will occur as a consequence of this function call corresponding to NLME-COMM-STATUS.indication primitive. Its argument pEvent->eventId shall be set to RCN_NLME_COMM_STATUS_IND and pEvent->prim.commStatusInd will be the valid union field in such callback.

7.9 RCN_NlmeGetReq

This function corresponds to sending NLME-GET.request to network layer and receiving NLME-GET.confirm from network layer.

7.9.1 Prototype

```
Uint8 RCN_NlmeGetReq( uint8 attribute, uint8 attributeIndex, uint8 *pValue )
```

7.9.2 Input Parameters

attribute:	NIBAttribute parameter of NLME-GET.request primitive
attributeIndex:	NIBAttributeIndex parameter of NLME-GET.request primitive
pValue:	Pointer to a buffer to where to store the resulting NIBAttributeValue of NLME-GET.confirm primitive

7.9.3 Output Parameters

pValue:	The buffer which this argument points to, will be overwritten with NIBAttributeValue of NLME-GET.confirm primitive resulting from this function call.

7.9.4 Return

Status parameter value of NLME-GET.confirm primitive resulting from NLME-GET.request service triggered by this function call.

7.9.5 Notes

This function call will be blocked until RCN completes NLME-GET.request service.

This function supports not only all the standard NIB attributes but also certain custom attributes described in Table 8.

Custom Attribute	Index Literal	*Note*
Node Capabilities	RCN_NIB_NWK_NODE_CAPABILITIES	*This corresponds to nwkcNodeCapabilities constant. RCN allows application to program this constant value. But it is not expected that application will change the value once it is written.*
IEEE address	RCN_NIB_IEEE_ADDRESS	*This should be used as a read-only item.*
PAN Id	RCN_NIB_PAN_ID	*PAN Id as a target node. This should be used as a read-only item.*
Network address	RCN_NIB_SHORT_ADDRESS	*Network address as a target node. This should be used as a read-only item.*
Frequency agility flag	RCN_NIB_AGILITY_ENABLE	*This attribute turns on and off the frequency agility feature. When this attribute is set to 1, frequency agility is turned on. When it is set to 0, frequency agility is turned off.*
Transmit Power Level	RCN_NIB_TRANSMIT_POWER	*Transmit power level in dBm of network layer packets except for key seed frames shall be determined by this attribute. The negative value must be represented as two's complement.* *See Table 4 for details.*

Table 8: Custom Attributes

7.10 RCN_NlmePairReq

This function corresponds to sending NLME-PAIR.request to network layer.

7.10.1 Prototype

```
void RCN_NlmePairReq( rcnNlmePairReq_t *pPrim )
```

7.10.2 Input Parameters

pPrim:	Pointer to a C structure corresponding to NLME-PAIR.request primitive.

7.10.3 Output Parameters

None.

7.10.4 Return

None.

7.10.5 Notes

A call to `RCN_CbackEvent()` will occur as a consequence of this function call, corresponding to `NLME-PAIR.confirm` primitive. Its argument `pEvent->eventId` shall be set to `RCN_NLME_PAIR_CNF` and `pEvent->prim.pairCnf` will be the valid union field in such callback.

7.11 RCN_NlmePairRsp

This function corresponds to sending `NLME-PAIR.response` to network layer.

7.11.1 Prototype

```
void RCN_NlmePairRsp( rcnNlmePairRsp_t *pPrim )
```

7.11.2 Input Parameters

pPrim:	Pointer to a C structure corresponding to NLME-PAIR.response primitive.

7.11.3 Output Parameters

None.

7.11.4 Return

None.

7.11.5 Notes

A call to `RCN_CbackEvent()` will occur as a consequence of this function call, corresponding to `NLME-COMM-STATUS.indication` primitive. Its argument `pEvent->eventId` shall be set to `RCN_NLME_COMM_STATUS_IND` and `pEvent->prim.commStatusInd` will be the valid union field in such callback.

7.12 RCN_NlmeResetReq

This function corresponds to sending `NLME-RESET.request` to network layer.

7.12.1 Prototype

```
void RCN_NlmeResetReq( uint8 setDefaultNib )
```

7.12.2 Input Parameters

setDefaultNib:	SetDefaultNIB parameter of NLME-RESET.request primitive.

7.12.3 Output Parameters

None.

7.12.4 Return

None.

7.12.5 Notes

No callback will be made as a consequence to this function call. `NLME-RESET.request` is assumed to always succeed in RCN implementation. When the function is called by a host application to reset RCN on a network processor over serial communication, communication failure could actually occur. Such case should be handled in NPI implementation on the host processor.

7.13 RCN_NlmeRxEnableReq

This function corresponds to sending NLME-RX-ENABLE.request to network layer.

7.13.1 Prototype

uint8 RCN_NlmeRxEnableReq(uint16 rxOnDurationInMs)

7.13.2 Input Parameters

rxOnDurationInMs:	RxOnDuration parameter of NLME-RX-ENABLE.request primitive. However, the unit is in milliseconds instead of symbols and it supports only up to 0xfffe milliseconds. 0xffff is used to turn on receiver forever till next RCN_NlmeRxEnableReq() call.

7.13.3 Output Parameters

None.

7.13.4 Return

RCN_SUCCESS, MAC_PAST_TIME, MAC_ON_TIME_TOO_LONG or MAC_INVALID_PARAMTER. Return value correspond to status field value of NLME-RX-Enable.confirm primitive. See **Error! Reference source not found.** for the primitive status field value details.

7.13.5 Notes

No callback will be made as a consequence to this function call. The return value shall correspond to Status parameter of NLME-RX-ENABLE.confirm primitive resulting from this request. Host application could get return value of RCN_ERROR_COMMUNICATION in case the serial communication with the network processor fails.

7.14 RCN_NlmeSetReq

This function corresponds to sending NLME-SET.request to network layer.

7.14.1 Prototype

*uint8 RCN_NlmeSetReq(uint8 attribute, uint8 attributeIndex, uint8 *pValue)*

7.14.2 Input Parameters

attribute:	NIBAttribute parameter of NLME-SET.request primitive.
attributeIndex:	NIBAttributeIndex parameter of NLME-SET.request primitive.
pValue:	Pointer to a buffer that contains NIBAttributeValue of NLME-SET.request primitive.

7.14.3 Output Parameters

None.

7.14.4 Return

RCN_SUCCESS, RCN_ERROR_UNSUPPORTED_ATTRIBUTE or RCN_ERROR_INVALID_INDEX. Return values correspond to Status field value of NLME-SET.confirm primitive described in **Error! Reference source not found.**.

7.14.5 Notes

No callback will be made as a consequence to this function call. The return value shall correspond to Status parameter of NLME-SET.confirm primitive resulting from this request. Host application could get return value of RCN_ERROR_COMMUNICATION in case the serial communication with the network processor fails.

This function can also be used for non-standard state attributes such as RCN_NIB_AGILITY_ENABLE and RCN_NIB_TRANSMIT_POWER. See Table 8 for details.

7.15 RCN_NlmeStartReq

This function corresponds to sending NLME-START.request to network layer.

7.15.1 Prototype

void RCN_NlmeStartReq(void)

7.15.2 Input Parameters

None.

7.15.3 Output Parameters

None.

7.15.4 Return

None.

7.15.5 Notes

A call to RCN_CbackEvent() will occur as a consequence of this function call, corresponding to NLME-START.confirm primitive. Its argument pEvent->eventId shall be set to RCN_NLME_START_CNF and pEvent->prim.startCnf will be the valid union field in such callback.

7.16 RCN_NlmeUnpairReq

This function corresponds to sending NLME-UNPAIR.request to network layer.

7.16.1 Prototype

void RCN_NlmeUnpairReq(uint8 pairingRef)

7.16.2 Input Parameters

pairingRef:	PairingRef parameter of NLME-UNPAIR.request primitive.

7.16.3 Output Parameters

None.

7.16.4 Return

None.

7.16.5 Notes

A call to RCN_CbackEvent() will occur as a consequence of this function call, corresponding to NLME-UNPAIR.confirm primitive. Its argument pEvent->eventId shall be set to RCN_NLME_UNPAIR_CNF and pEvent->prim.unpairCnf will be the valid union field in such callback.

7.17 RCN_NlmeUnpairRsp

This function corresponds to sending `NLME-UNPAIR.response` to network layer.

7.17.1 Prototype

```
void RCN_NlmeUnpairRsp( uint8 pairingRef )
```

7.17.2 Input Parameters

pairingRef:	PairingRef parameter of NLME-UNPAIR.response primitive.

7.17.3 Output Parameters

None.

7.17.4 Return

None.

7.18 RCN_NlmeAutoDiscoveryReq

This function corresponds to sending `NLME-AUTO-DISCOVERY.request` to network layer.

7.18.1 Prototype

```
void RCN_NlmeAutoDiscoveryReq( rcnNlmeAutoDiscoveryReq_t *pPrim )
```

7.18.2 Input Parameters

pPrim:	Pointer to a C structure corresponding to NLME-AUTO-DISCOVERY.request primitive.

7.18.3 Output Parameters

None.

7.18.4 Return

None.

7.18.5 Notes

A call to `RCN_CbackEvent()` will occur as a consequence of this function call, corresponding to `NLME-AUTO-DISCOVERY.confirm` primitive. Its argument `pEvent->eventId` shall be set to `RCN_NLME_AUTO_DISCOVERY_CNF` and `pEvent->prim.autoDiscoveryCnf` will be the valid union field in such callback.

7.19 RCN_NlmeAutoDiscoveryAbortReq

This function aborts auto discovery procedure triggered by a prior `RCN_NlmeAotoDiscoveryReq()` call.

7.19.1 Prototype

```
void RCN_NlmeAutoDiscoveryAbortReq( void )
```

7.19.2 Input Parameters

None.

7.19.3 Output Parameters

None.

7.19.4 Return

None.

7.19.5 Notes

This function does not correspond to any functionality in **Error! Reference source not found.**. When this function is called, the auto discovery procedure is aborted immediately. However, due to race condition, the application which issued this function call may still get `RCN_CbackEvent()` callback for `RCN_NLME_AUTO_DISCOVERY_CNF` event.

No `RCN_CbackEvent()` call is made specifically in response to this function call.

7.20 RCN_NlmeUpdateKeyReq

This function corresponds to sending `NLME-UPDATE-KEY.request` to network layer.

7.20.1 Prototype

```
uint8 RCN_NlmeUpdateKeyReq( uint8 pairingRef, uint8 *pNewLinkKey )
```

7.20.2 Input Parameters

pairingRef:	PairingRef parameter of NLME-UPDATE-KEY.request primitive.
pNewLinkKey:	Pointer to NewLinkKey parameter of NLME-UPDATA-KEY.request primitive.

7.20.3 Output Parameters

None.

7.20.4 Return

`RCN_SUCCESS`, `RCN_ERROR_NO_PAIRING` or `RCN_ERROR_NOT_PERMITTED`. Return values correspond to the Status field values of `NLME-UPDATE-KEY.confirm` primitive, described in **Error! Reference source not found.**.

7.20.5 Notes

No callback will be made as a consequence to this function call. The return value shall correspond to Status parameter of `NLME-UPDATE-KEY.confirm` primitive resulting from this request. Host application could get return value of `RCN_ERROR_COMMUNICATION` in case the serial communication with the network processor fails.

7.21 Asynchronous RCN_CbackEvent() calls

All `RCN_CbackEvent()` calls are asynchronous to request or response function calls. That is, the callback is made without blocking the function call. However, the callbacks described in the request or response function calls are triggered only by such function calls.

There are certain network layer indication primitives that could be issued irrespective to any requests or responses that an application sends to network layer, such as `NLDE-DATA.indication`, `NLME-DISCOVERY.indication`, `NLME-PAIR.indication` and `NLME-UNPAIR.indication`. They are truly asynchronous to any requests or responses.

`RCN_CbackEvent()` callback function is used to notify the occurrence of such primitive as well. Table 9 describes eventId and valid union field for such primitives.

Primitive	pEvent->eventId value	*Valid union field*

Primitive	pEvent->eventId value	*Valid union field*
NLDE-DATA.indication	RCN_NLDE_DATA_IND	pEvent->prim.dataInd.
NLME-DISCOVERY.indication	RCN_NLME_DISCOVERY_IND	pEvent->prim.discoveryInd.
NLME-PAIR.indication	RCN_NLME_PAIR_IND	pEvent->prim.pairInd.
NLME-UNPAIR.indication	RCN_NLME_UNPAIR_IND	pEvent->prim.unpairInd.

Table 9: Asynchronous Callbacks

Among the primitive structures dataInd (corresponding to RCN_NLDE_DATA_IND) includes a pointer. "nsdu" field is a pointer to a data buffer that corresponds to nsdu parameter of NLDE-DATA.indication primitive. Neither the data buffer nor the pointer is valid once the callback function returns and hence application must copy the content of the buffer to its own managed buffer if it needs to access the data content after the callback returns.

8. Using RCN API from Host Application

Host application can choose to use RCN API instead of RTI API with the same RemoTI network processor. Host application in such a case has to indicate which API to use at the beginning. By default, after boot up or reset of network processor, network processor is configured to work with RTI API calls from host application. In order to switch to RCN API mode, host application should call RCN_NlmeResetReq() before making other calls. Either TRUE or FALSE value of setDefaultNib parameter will work to switch mode to RCN API mode. As a general rule, RTI API functions should not be used when RCN API is used. However, there are exceptions in both not using RCN calls before RCN_ResetReq() call and not using RTI API functions mixed with RCN API functions.

RCN_NlmeGetReq() and RCN_NlmeSetReq() can be used regardless of network processor mode. That is, it can be used before calling RCN_NlmeResetReq(). This is to address a typical use case of RCN_NlmeSetReq() to custom attributes such as node capabilities before calling RCN_NlmeResetReq(TRUE) and RCN_NlmeStartReq() to start up a node for cold boot (first boot up for the node). There is no reason to mix use RCN_NlmeGetReq() and RCN_NlmeSetReq() together with RTI API otherwise because RTI API function RTI_ReadItem() and RTI_WriteItem() supports access to all NIB attributes and custom attributes supported by network layer.

On the other hand, RTI_EnableSleepReq(), RTI_DisableSleepReq() and callbacks thereof can be used together with other RCN API functions. As standard network layer does not provide interface to enable and disable entire system power savings which may affect serial communication itself, these functions are exceptionally allowed to be used together with RCN API functions.

9. General Information

9.1 Document History

Table 10: Document History

Revision	Date	Description/Changes
1.0	2009-04-15	Initial release.

Revision	Date	Description/Changes
1.1	2009-06-30	Added RTI_UnpairReq, RTI_UnpairCnf, RTI_UnpairInd, RTI_PairAbortReq, RTI_PairAbortCnf, RTI_AllowPairAbortReq, RCN_NlmeDiscoveryAbortReq and RCN_NlmeAutoDiscoveryAbortReq functions. Corrected RTI_SendDataCnf result code list. Corrected rcnNwkPairingEntry_t structure in Table 4. Corrected Table 6 and added hex values. Corrected hardware platform names in Figure 1 and Figure 2.
swra268a	2009-09-18	Revision numbering scheme was corrected.
swra268b	2009-10-27	Missing RTI return status values were added.
swra268c	2010-07-06	Added references to CC2533.
swra268d	2011-03-02	Changed CERC to ZRC; added extended RTI_Read/WriteEx() and deprecatd the old RTI_Read/Write(); added ZID.
swra268e	2011-10-06	Added description of Non-Standard Descriptor read/write helper interface.
swra268f	2012-09-20	Changed description of PanID and ShortAddress to RW.
swra268g	2012-10-02	Fixed default value of DiscoveryLQIThreshold SA item.
swra268h	2012-10-31	Updated representation of software version number
swra268i	2012-11-20	Updated for latest release. - Changed name of reference SWRU191 - Rephrased description of RTI - Added reference to embedded Linux host support - Changed default nwkScanDuration from 6 to 3 - Updated ZID revision reference from r15 to r18 - Corrected ZID mask - Formatting

Address Information

Texas Instruments Norway AS
Gaustadalléen 21
N-0349 Oslo
NORWAY
Phone: +47 22 95 85 44
Fax: +47 22 95 85 46
Web site: http://www.ti.com/lpw

Texas Instruments Incorporated
Low-Power RF Software Development
9276 Scranton Road, Suite 450
San Diego CA 92121
United States of America
Phone: +1 858 638 4294
Fax: +1 858 638 4202
Web site: http://www.ti.com/lpw

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		Applications	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
		Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265