NFC Commissioning User Guide

JN-UG-3112
Revision 1.0
19 Nov 2015

Contents

About this Manual 3
Organisation 3
Conventions 3
Acronyms and Abbreviations 3
Related Documents 3
Support Resources 3
Trademarks 4

1 NFC Commissioning 5
1.1 System Overview 5
1.1.1 NFC Commissioning Implementation Options 6
1.2 NTAG Format 10

2 NFC Software Architecture 11
2.1 NFC Component 11
2.2 NDEF Component 11
2.3 NSC Component 11
2.4 NTAG Component 11
2.5 I2C Driver Component 11

3 API Reference 12
3.1 NFC Component API 12
3.2 NDEF Component API 13
3.3 NSC Component API 15
3.4 API NTAG Component 16
3.5 API I2C Driver 18

4 API Workflow 20
4.1 NTAG Creation by the Device 20
4.2 Device Commissioning at Boot 20

Appendices 21
Appendix A - Source File Descriptions 21
Appendix B – Preprocessing Macro Descriptions 22

About this Manual

This manual describes the fundamentals of NFC commissioning and its implementation in a ZigBee network.

Organisation

This manual consists of 4 chapters, as follows:

- Chapter 1 introduces the concept of Near Field Communication (NFC) used for out of band commissioning of ZigBee devices
- Chapter 2 describes the software components used for NFC commissioning
- Chapter 3 details the API used for NFC Commissioning
- Chapter 4 gives a brief description of the API workflow

Conventions

Files, folders, functions and parameter types are represented in **bold** type.

Function parameters are represented in *italics* type.

Code fragments are represented in the Courier typeface.

Acronyms and Abbreviations

NFC	Near Field Communication
NDEF	NFC Data Exchange Format
NTAG	NFC tag
API	Application Programming Interface
MAC	Media Access Control
ECC	Elliptic Curve Cryptography
ECDH	Elliptic Curve Diffie-Hellman

Related Documents

JN-AN-1221	ZigBee HA Lighting with NFC Application Note

Support Resources

To access JN516x support resources such as SDKs, Application Notes and User Guides, visit the Wireless Connectivity TechZone:

www.nxp.com/techzones/wireless-connectivity

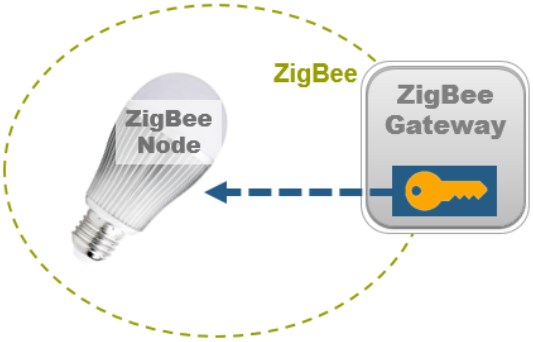
All NXP resources referred to in this manual can be found at the above address, unless otherwise stated.

Trademarks

All trademarks are the property of their respective owners.

1 NFC Commissioning

ZigBee devices need a commissioning phase to get access to a ZigBee network. They need to get a network key (NK) from the network coordinator.

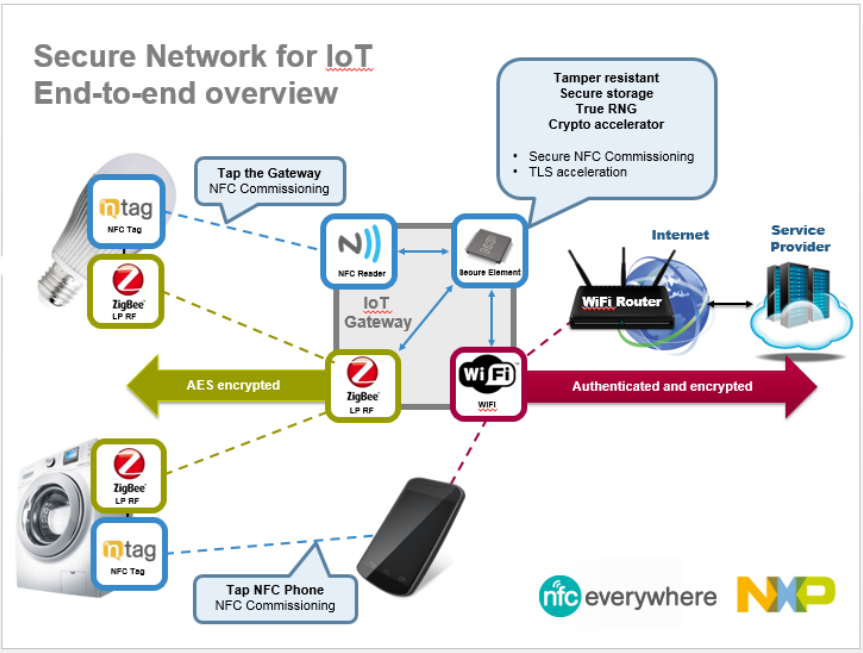


NFC can offer an easy method to transfer the network key data from the coordinator to an end device which is powered or unpowered, via two different ways:

- Tapping the coordinator
- Using a third-part device like a smartphone

1.1 System Overview

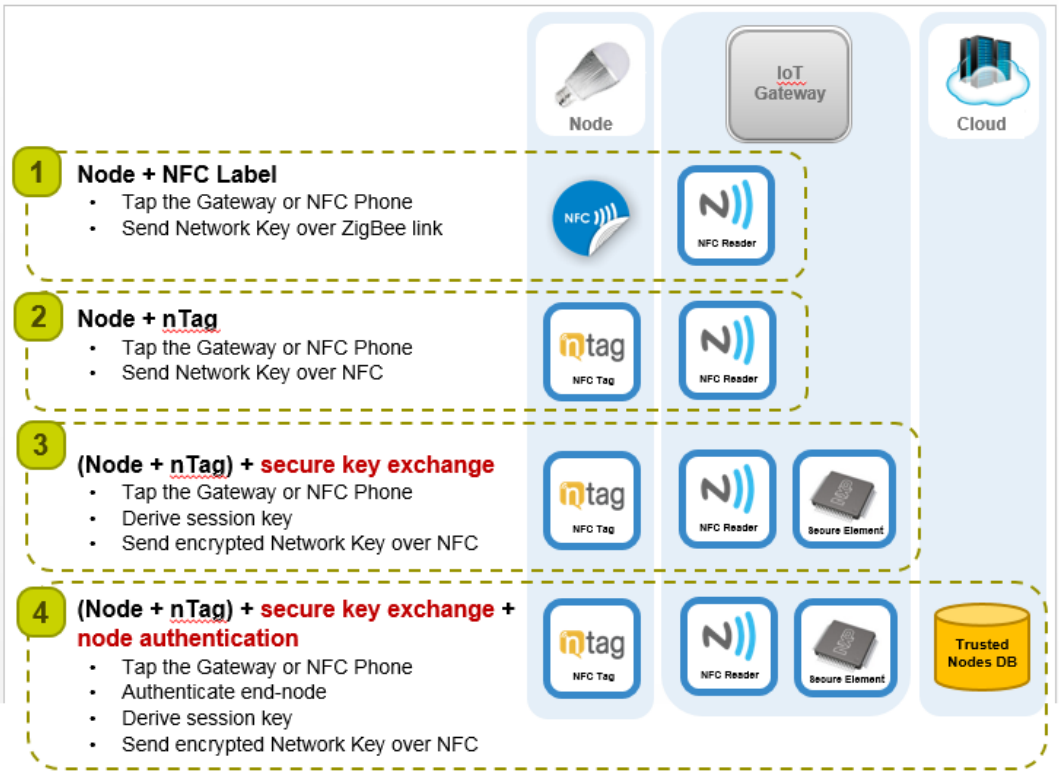
NFC feature can operate in various ways depicted in the following diagram:



NFC provides an easy and safe way of commissioning devices into low power radio networks independently of the network's communication protocol (ZigBee, WiFi, Bluetooth or Bluetooth Low Energy).

1.1.1 NFC Commissioning Implementation Options

There are different scenarios where NFC can be used to commission devices as shown below:



Scenario 1 depicts the simplest form of NFC integration. An NFC label is used to store specific data related to the node – such data includes the device type and the MAC address. This information is transferred to the gateway during an NFC "tap" between the end node and the gateway where the data is stored in a white list. When the device is powered "ON", the gateway which is in commissioning mode, will recognize the MAC address and commission the device.

Moreover, the NFC label and the ZigBee device must be treated as a pair at production level, i.e. each ZigBee microcontroller must be associated to a label which can be difficult to manage logistically.

Although a low-cost solution, it remains quite restrictive in its usage.

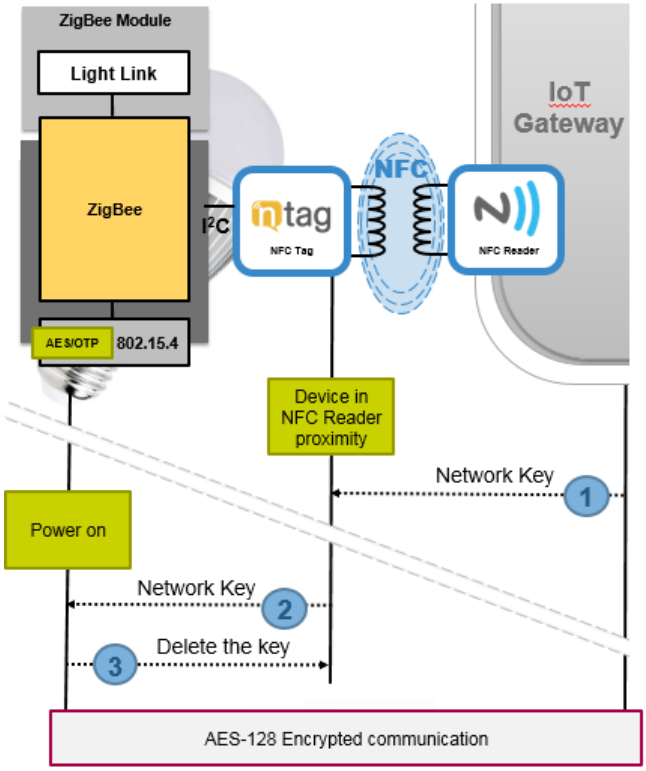
Scenario 2 uses an NTAG I²C device on the same PCB hardware as the JN516x device. More information on the NTAG I²C device can be found here: http://www.nxp.com/products/identification_and_security/smart_label_and_tag_ics/connected_tag_solutions/series/NT3H1101_NT3H1201.html

The NTAG device includes information such as the MAC address, the Link key of the device, the device type and a join command. This information is written automatically by the ZigBee microcontroller into the NTAG at the first power up of the device. It is recommended that this be done in production.

During an NFC action, this information is transferred to the gateway reader, and the reader writes in the NTAG the network information. The device information can then be stored in a whitelist or a database in the gateway host.

When the device is powered ON, the information contained in the NTAG are transferred to the ZigBee microcontroller. The “join” command is then used to commission the device onto the network, all radio data packets are encrypted with the network key. Note that the JN516x in the end node also erases all the network information contained in the NTAG to ensure that the NTAG cannot be read to get access to sensitive information.

Scenario 2 has been selected to demonstrate the ZigBee commissioning via NFC in the JN516x-EK004 Evluation Kit.

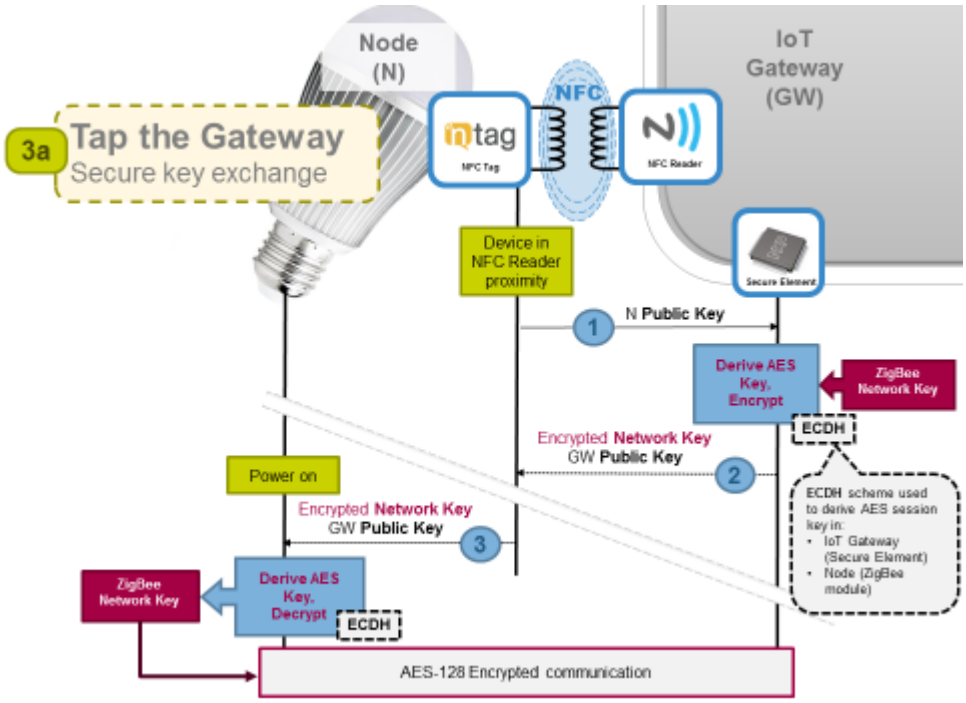


- No encryption (protected by short range of NFC)
- Out of band (NFC only) instant key and network parameters exchange
- Faster commissioning
- Possibility to issue commands: reset, decommission, etc.

To prevent the exposure of critical ZigBee information in the NTAG-I²C, secure commissioning, scenarios 3 and 4, have been implemented by NXP and can be used as guidelines to help developers with their applications. These implementations require Elliptic curve cryptography, and the said curves can be purchased through licensing either at Certicom or TLS MBED or any other curve providers. As such, due to licensing fees, these implementations are not provided by NXP in their evaluation kits or as a reference design.

Scenario 3 uses symmetric key cryptography. The end device’s private key is stored in the ZigBee microcontroller while its public key is stored in the NTAG device. The public key is transferred to the gateway during an NFC “tap”. The gateway uses the node’s public key and its own private key to encrypt the Network

key using Elliptic Curve Diffie-Hellman cryptography. Then in the same NFC action, the gateway transfers the Encrypted network key and the gateway public key to the NTAG device.



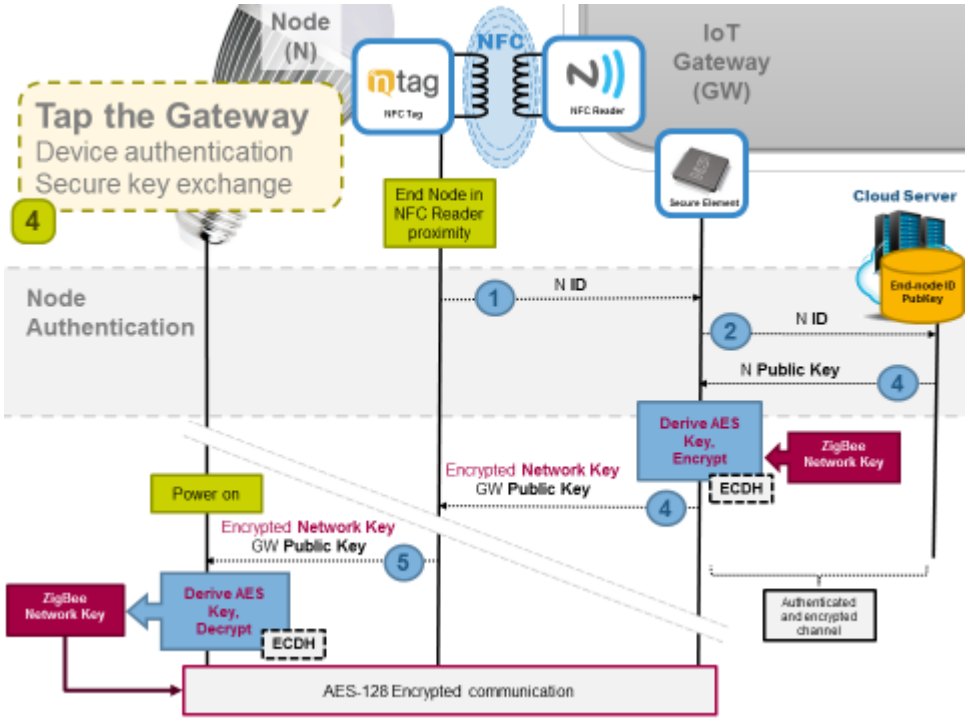
When the device is powered “ON”, the NTAG is read. The firmware in the micro controller recognizes the tag’s NDEF information in the network data section and re-initializes the software. The software decrypts the network key and then launches the ZigBee application. This is done sequentially as the data decryption is requires exclusive use of the CPU.

Note that in scenario 3, a secure element is used to run the cryptographic software. The secure element is a tamper proof IC and therefore adds extra security both physically and at the software level.

Scenario 4 reuses the basic concept of symmetric key cryptographic, and also includes authentication.

In this implementation the node has a specific identification which is linked to a private key securely stored in the ZigBee device. This ID is transferred to the gateway during an NFC tap. The gateway authenticates the device through the cloud and via a white list. Once it is authenticated the cloud server sends the corresponding public key to the gateway where it is used with the gateway’s private key to encrypt the network key.

The NFC reader then transfers the encrypted network key and the gateway’s public key to the end device.

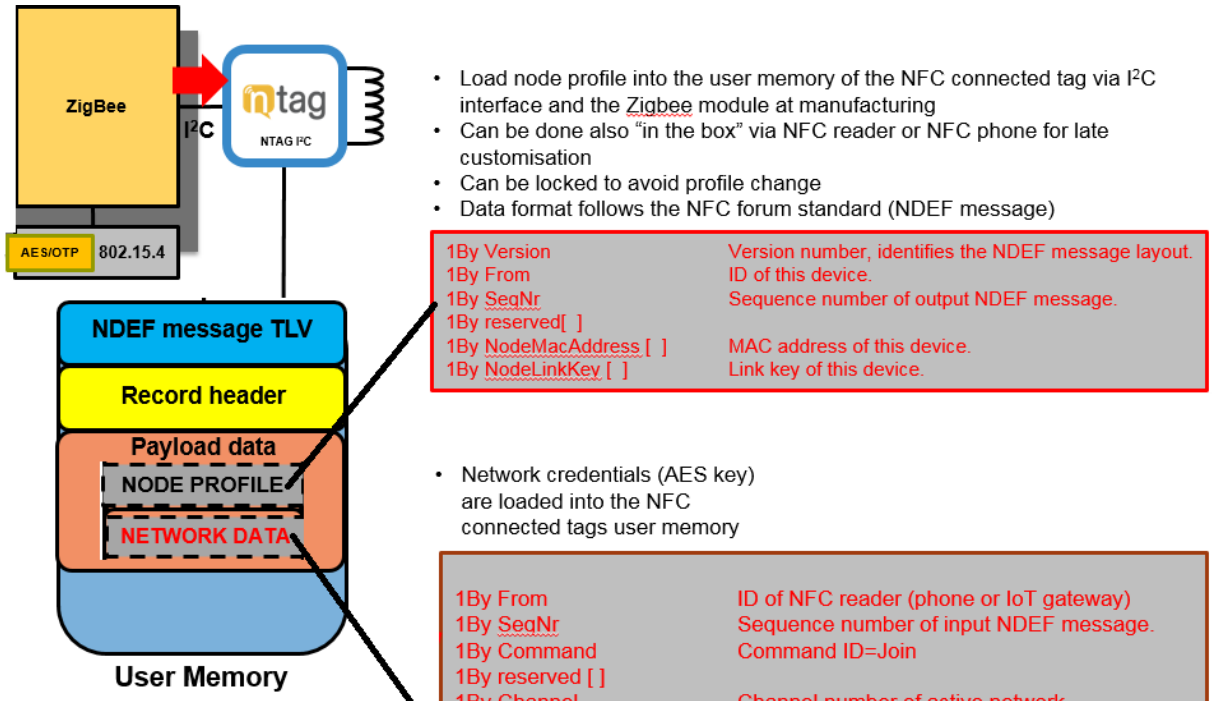


When the device is powered ““ON””, the NTAG is read. The firmware in the micro controller recognizes the tag’s NDEF information and re-initializes the software. The software decrypts the network key and then launches the ZigBee application. This is done sequentially as the data decryption is requires exclusive use of the CPU.

For further security, a secure element is implemented in the gateway to provide a secure connection to the cloud and also to run all the encryption software. Note that the solution can be made even more secure by implementing a secure element on the node side to securely store the end node’s private key.

1.2 NTAG Format

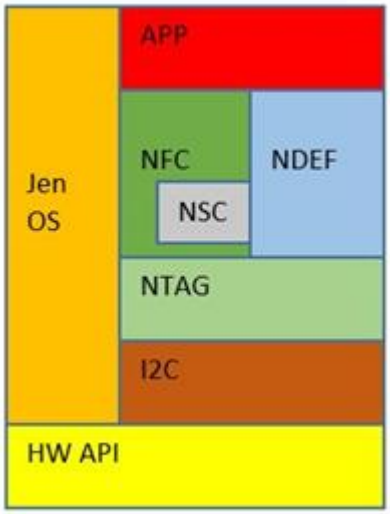
The exchange of data between the node device and the gateway is done via the standard NTAG format from the NFC forum. The structure is as followed:



Node profile data is written by the device as described in Chapter 4.1.

2 NFC Software Architecture

The software structure is organized in several layers as shown below:



2.1 NFC Component

This component implements the NFC loop task.

2.2 NDEF Component

This component implements the NDEF layer driver. Functions in this layer can be used by the application(s) to read/write in the NTAG. To be able to do so, each application must register its own unique NDEF type and use the lock and unlock API functions of this layer to get/release exclusive access to the NTAG EEPROM. This layer will use a callback function to notify the application of possible modifications in the NDEF message in the NTAG after a NFC reader (eg. smart phone) has accessed the tag data.

2.3 NSC Component

This component implements the NFC Secure Commissioning module and uses the LIBSLL library when the “NFC_COMMISSIONING” compilation switch is set.

2.4 NTAG Component

This component implements the NTAG I2C driver.

2.5 I2C Driver Component

This file implements the low-level I2C driver.

3 API Reference

3.1 NFC Component API

void	**vNfcInit** (void)
	This function initializes the NFC module (this source file), initializes the ntag layer (source file **ntag.c**), and takes care of initializing the NFC Secure Commissioning module (source file **nsc.c**) when NFC_COMMISSIONING is enabled. After the NTAG-I2C IC has been successfully initialized, the EEPROM of the NTAG is read out to check whether there is a Secure Join instruction in the NTAG.
void	**vNfcInitPostProcessing** (bool bUseNTAGirqs)
	This function does the NFC Secure Commissioning post processing (when NFC_COMMISSIONING is enabled) after reset of the CPU, as post processing of function **vNfcInit()**. Optionally it enables the IRQ detection for the field detect (FD) pin of the NTAG.
void	**vNfcTagFdPinIRQ** (void)
	This function is called when the Field Detect (FD) pin of the NTAG-I2C IC goes low, because of the presence of a NFC field.
void	**vNfcToggleDBG** (void)
	This function toggles the Nfc debug trace output when DEBUG_NFC is defined.

3.2 NDEF Component API

teNDEF_Status	**eNdefRegisterMsg** (const uint8 *pu8NdefType, uint8 u8NdefTypeLength, uint8 *pu8NdefId, uint8 u8NdefIdLength, uint32 u32NdefPayloadLength, void(*pCallbackFunc)(**ndef_rec_info_t** *))
	This function tries to register the NDEF message with the specified NDEF payload type, optional NDEF payload identifier, and callback function pointer. All applications that want to read/write a NDEF msg in the NTAG should register its payload type as soon as possible after reset of the CPU (after JenOS is started). When the last application registers its NDEF payload type, then the function **eNdefFormatNtag()** is called by this function.
void	**vNdefParser** (void)
	This function reads the NTAG EEPROM (from start of user memory till the TLV terminator) and parses the TLVs that it finds. During this parsing it has exclusive R/W access to the NTAG. Before calling the callback function belonging to the parsed NDEF message, the lock on the NTAG is released. In the applications callback function, use the functions eNdefLockNtag() / eNdefReadMsg() / eNdefWriteMsg / eNdefUnlockNtag() to access the payload of the related NDEF.
teNDEF_Status	**eNdefLockNtag** (const uint8 *pu8NdefType)
	This function tries to get exclusive R/W access to the specified NDEF in the NTAG EEPROM. After having gained exclusive R/W access the functions eNdefReadMsg() and eNdefWriteMsg() can be used to read and/or write to the specified NDEF payload area. The exclusive R/W access must be released by calling the function eNdefUnlockNtag().
teNDEF_Status	**eNdefUnlockNtag** (const uint8 *pu8NdefType)
	This function releases the exclusive R/W access to the specified NDEF in the NTAG EEPROM, but only in the case that this application actually holds the lock.
teNDEF_Status	**eNdefReadMsg** (const uint8 *pu8NdefType, uint32 u32NdefOffset, uint8 *pu8Data, uint32 u32DataLength)

	This function reads NDEF payload data from the NTAG EEPROM. This function checks whether the caller only reads from his own NDEF payload area.

teNDEF_Status	**eNdefWriteMsg** (const uint8 *pu8NdefType, uint32 u32NdefOffset, uint8 *pu8Data, uint32 u32DataLength)
	This function writes NDEF payload data into the NTAG EEPROM. This function checks whether the caller only writes to his own NDEF payload area.

teNDEF_Status	**eNdefFormatNtag** (void)
	This function writes the TLVs, NDEF headers and TLV terminator into the NTAG EEPROM. This function can be used to repair the NTAG format as designed by the application. RETURNS: This function returns E_NDEF_SUCCESS when the NTAG is formatted. One of the following errors is returned when it couldn't format the NTAG: E_NDEF_ERR_NTAG_ALREADY_LOCKED, E_NDEF_ERR_NTAG_NO_MEM_LOCK, E_NDEF_ERR_WRITE_HDR or E_NDEF_ERR_WRITE_TERMINATOR.

void	**vNdefToggleDBG** (void)
	This function toggles the Ndef debug trace output when DEBUG_NFC is defined.

3.3 NSC Component API

void	**vNscInit** (void)
	This function initializes the NSC module (this source file), and registers a NDEF payload type with the NDEF layer driver.
void	**vNscDoSslDecrypt** (void)
	This function initializes the Elliptic Curve library (libssl), and handles the ECC decryption of the Secure Join command that was read out of the NTAG at startup. The Elliptic Curve library can only be initialized successfully when there is enough RAM available (heap and stack).
void	**vNscInitPostProcessing** (void)
	This function handles the actual joining or leaving just after startup, and initiates the writing of ECC public key and link_info into the NTAG.
void	**vNscActOnCommand** (void)
	This function initiates the NFC Secure Commissioning when a join or leave command have just been written into the NTAG via the RF side of the NTAG IC.
void	**vNscJoinNetworkFinished** (void)
	This function initiates the writing of NFC secure join/leave info into the NTAG, and clears the join command from the NTAG.
void	**vNscLeaveNetworkFinished** (void)
	This function initiates the writing of NFC secure join/leave info into the NTAG, clears the leave command from the NTAG, and clears the ECC persistent data.
void	**vNscEraseCommissioningData** (void)
	This function clears the ECC persistent data.
void	**vNscToggleDBG** (void)
	This function toggles the Nsc debug trace output when DEBUG_NFC is defined.

3.4 API NTAG Component

bool	**bNtagPower** (**ntag_power_cmd_t** cmd)
	This function checks the availability of the NTAG-I2C, which is useful when it is an optional module in the device. In some devices the NTAG IC can be powered down to save energy. When the NTAG IC is powered on the manufacturer ID and validation registers are read and validated.
bool	**bNtagSetup** (uint16 u16Wdt, uint8 end_block)
	This function sets the NTAG watchdog, plus the LAST_NDEF_BLOCK which when read from RF side results in the flag NDEF_DATA_READ in NS register to be set.
bool	**bNtagGetNsReg** (uint8 *pu8regval)
	Using the I2C driver, read the NS register of the NTAG-I2C. Since the flag NS_REG_NDEF_DATA_READ is automatically reset on reading the NS_REG, the bool bNtagDataAvailable is used to remember that NS_REG_NDEF_DATA_READ was set. After reading out the NTAGs EEPROM the function vNtagClearNdefDataRead() is used to clear bool bNtagDataAvailable.
void	**vNtagClearNdefDataRead** (void)
	This function clears the bNtagDataAvailable boolean.
bool	**bNtagTry2SetI2cLock** (void)
	This function can be used to try to set the I2C_LOCKED bit, which means that the NTAG EEPROM is locked for read/write access from the I2C side. The function **bNtagReleaseI2cLock()** can be used to release the lock from the I2C side on the NTAG EEPROM.
bool	**bNtagStartRead** (uint16 u16Offset)
	Set the reader to u16Offset of the start of the stream.
uint32	**u32NtagRead** (uint8 *pu8Buffer, uint32 u32Nbyte)
	Read or skip the requested bytes from the NTAG-I2C IC. When pu8Buffer is equal to NULL, requested bytes will be skipped in the byte stream.

bool	**bNtagStartWrite** (uint16 u16Offset)
	Set the writer to u16Offset of the start of the stream.
uint32	**u32NtagWrite** (uint8 *pu8Buffer, uint32 u32Nbyte)
	Write the passed bytes to the NTAG-I2C IC.
bool	**bNtagFlush** (void)
	Writes the remaining bytes from wr_buf to the NTAG-I2C IC.
bool	**bNtagReleaseI2cLock** (void)
	Instruct the NTAG to release the EEPROM lock by I2C side.

3.5 API I2C Driver

void	**i2c_init** (bool bUseDio16Dio17, bool bI2Cclk400khz)
	This function initializes the I2C hardware block of the JN516x. When multiple I2C slaves are connected to the I2C bus, concurrent access to these slaves via this driver must be prevented by a semaphore. However, since these semaphores are only available when JenOS is started, the boolean bBusLockingEnabled is introduced to make access to these slaves possible, even before JenOS is started. After this init the I2C bus locking is enabled. Use the function **i2c_EnableBusLocking()** to disable/enable I2C bus locking when needed before JenOS is started.
void	**i2c_EnableBusLocking** (bool_t bEnable)
	This function sets the internal boolean bBusLockingEnabled.
void	**i2c_LockBus** (void)
	This function locks the I2C bus depending on the internal boolean bBusLockingEnabled.
void	**i2c_UnlockBus** (void)
	This function unlocks the I2C bus depending on the internal boolean bBusLockingEnabled.
bool	**i2c_BusWriteReg** (uint8 u8Address, uint8 u8Command, uint8 u8Length, uint8 *pu8Data)
	This function writes data to a I2C slave.
bool	**i2c_BusReadData** (uint8 u8Address, uint8 u8Command, uint8 u8Length, uint8 *pu8Data)
	This function reads data from a I2C slave.
bool	**i2c_ReadNtagReg** (uint8 u8Address, uint8 u8Command, uint8 u8Reg, uint8 *pu8Data)
	This function reads out a specified NTAG register.
bool	**i2c_CheckSlaveAvailable** (uint8 u8Address)
	This function checks if a I2C slave with the given slave address is communicating on the I2C bus.

void	**i2c_ToggleDBG** (void)
	This function toggles the debug output if DEBUG_I2C_DRIVER is defined.

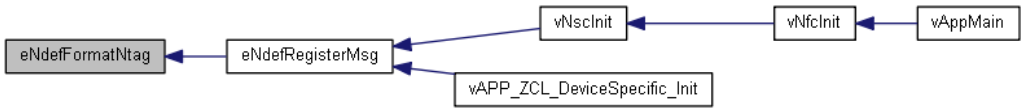
4 API Workflow

4.1 NTAG Creation by the Device

At device start-up, writes the TLVs, NDEF headers and TLV terminator into the NTAG EEPROM using the NTAG formats provided by the application.

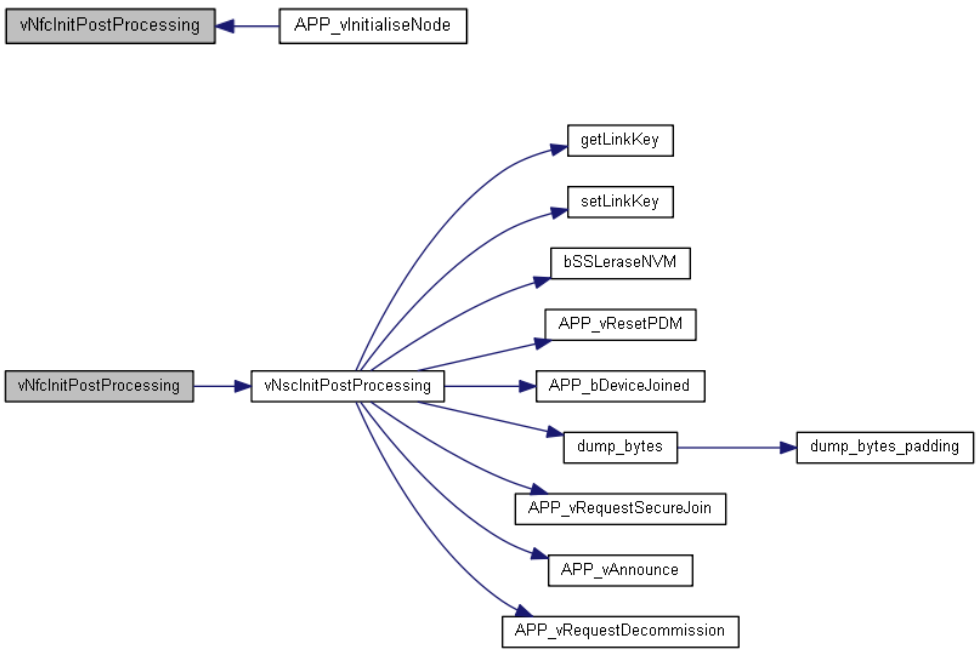
This process is used either to:

- Create the NTAG on the first power up (possibly during production test)
- Repair the NTAG (possibly during reboot by the user after a malfunction)



4.2 Device Commissioning at Boot

This process only takes place if NTAG contains payload information with network data provided by RF from the Gateway coordinator (see Chapter 1.2) for data format).



Appendices

Appendix A - Source File Descriptions

NFC Component Files

A number of common files are used across all device types and are located within the **\Common\Source** folder. The following table gives a brief description of each of the common files.

Filename	Description
nfc.c / h	Implements the NFC loop task using the NFC module
ntag.c / h	Implements the I2C NTAG driver
ndef.c/h	Implements the NDEF layer driver to read / write NDEF messages
nsc.c / h	Implements the secure NFC commissioning using the NFC module

Appendix B – Preprocessing Macro Descriptions

Compile-time macros to manipulate ZigBee NFC commissioning functionality are defined in the **app_config.h** file for the respective device.

The compile-time macros to configure NFC components are as follows:

Macro	Description
NFC_SUPPORT	Enable use of NFC
NFC_COMMISSIONING	Build flag to enable NFC commissioning (activate NSC component)
SUPPORT_LIBSSL	Enable ECC secure commissioning via LIBSSL library
NTAG_FD_PIN	NTAG field detector DIO pin number
CONFIG_NUMBER_OF_APP_NDEF_MSG	Number of NDEF messages registered by the application
CONFIG_NDEF_MAX_TYPE_LENGTH	Max of bytes of the NDEF message payload

Revision History

Version	Date	Description
1.0	19-Nov-2015	First release

Important Notice

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the *Terms and conditions of commercial sale* of NXP Semiconductors.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

NXP Semiconductors

For the contact details of your local NXP office or distributor, refer to:

www.nxp.com

