UM10802

NXP NFC Reader Library API

Rev. 1.0 — 07 April 20144 **User manual**

Document information

Info	Content
Keywords	NFC Reader Library, P2P, CLRC663, PN512, LPC1769, ISO18092, Discovery Loop, LLCP, SNEP, NFC Forum Tag Type Operation, NFC Forum, MIFARE, ISO14443.
Abstract	This document provides detailed description of the NXP NFC Reader Library API.

Revision history

Rev	Date	Description
1.0	20140407	First Release

Contact information

For more information, please visit: http://www.nxp.com

For sales office addresses, please send an email to: salesaddresses@nxp.com

1. Audience

This document is intended to be used by software designers, developers and integrators willing to develop NFC applications for NXP's contactless reader ICs. The developer should have prior knowledge and experience in C programming language and structured programming in general.

2. Abstract

This document provides detailed description of the NXP NFC Reader Library API and it is complement of a NXP NFC Reader Library User Manual [37]. This user manual is intended to help software developers, implementers and integrators to get familiar with the NFC Reader Library [3] and to learn how to work with it.

The document is divided in sections: after the introductory Sections 1 and 2, Section 3 provides an overview of the NFC Reader Library and its layered architecture. Sections 4, 5, 6, 7 and 8 provide a detailed description of the NFC Reader Library API.

Detailed description of the implementation and how to use it is explained in the user manual NXP NFC Reader Library User Manual [37].

3. Introduction

3.1 Overview of the NFC Reader Library

The NXP NFC Reader Library [3] is a modular software library written in C language, which provides an API that enables customers to create their own software stack and applications for the NXP contactless reader ICs. This API facilitates the most common operations required in NFC applications such as reading or writing data into contactless cards or tags, exchanging data with other NFC-enabled devices or allowing NFC reader ICs to emulate cards as well.

The NFC Reader Library is designed as a versatile and multi-layered architecture. From bottom to top, the NFC Reader Library is composed of the following layers:

- Bus Abstraction Layer (BAL): Implements the communication interface between the host device and the contactless reader IC.
- Hardware Abstraction Layer (HAL): Implements the hardware specific elements of the contactless reader IC and executes native commands of the chip.
- Protocol Abstraction Layer (PAL): Implements the functions for contactless card activation and contactless card protocols.
- Application Layer (AL): Implements the commands to work with several contactless smart card technologies.
- NFC Forum Tag Type Operations (TOP): Implements an API for developers to perform read and write operations on top of the four Tag Types defined in the NFC Forum specifications.
- NFC Activity: Implements a routine for sensing the RF field to detect the presence of contactless smart cards, NFC tags or other NFC-enabled devices in close proximity.

- NFC P2P Package: Implements P2P functionality based on the NFC Forum defined P2P protocol stack allowing two NFC devices to exchange data when they are brought into proximity.

The NFC Reader Library also includes an additional layer named:

- Common Layer: Implements utilities independent of any card or hardware being used during the project development.

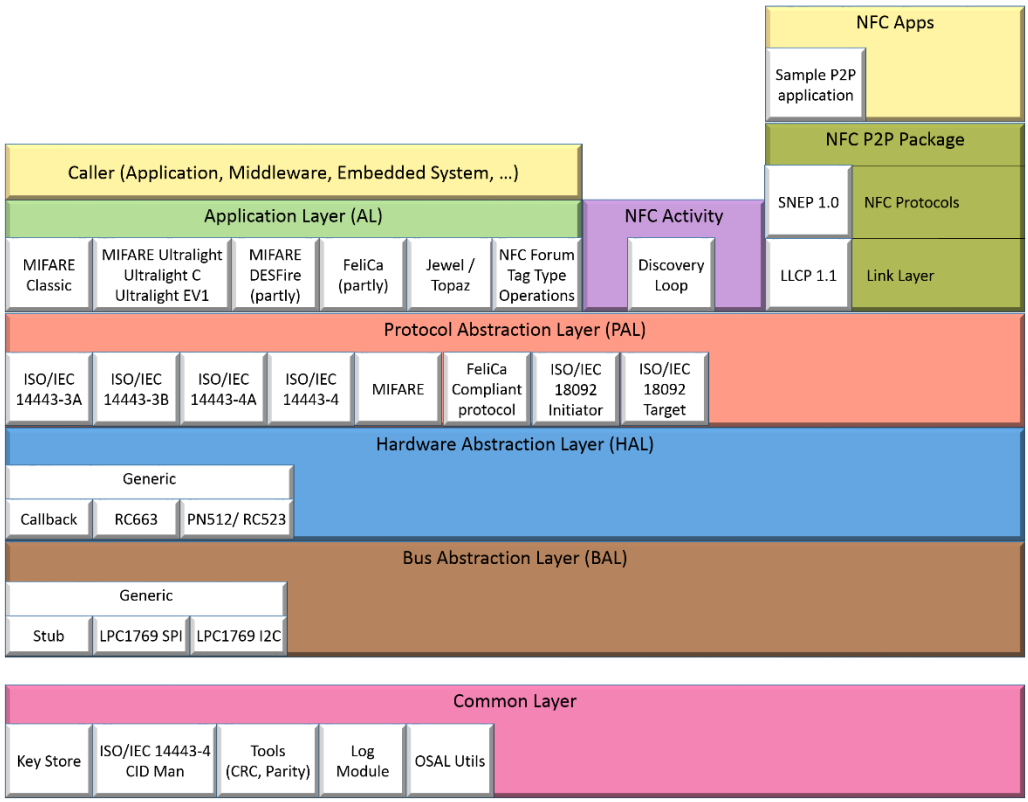


Fig 1. Layered Structure of the NFC Reader Library

3.2 NFC Reader Library Software Stack

The main advantage provided by this modular and multi-layered approach is flexibility. The Application Layer (AL), the NFC Activity component, the NFC P2P Package and the Protocol Abstraction Layer (PAL) are hardware-independent. This means that their functionality is not bound to or dependent on any specific hardware. Therefore, the developers can use them seamlessly on top of any of the supported contactless reader ICs implemented on the Hardware Abstraction Layer (HAL).

Similarly, the Application Layer (AL), the NFC Activity component, the NFC P2P Package, the Protocol Abstraction Layer (PAL) and the Hardware Abstraction Layer (HAL) are also platform-independent. This means that their functionality is not dependent to any specific underlying communication interface with the host. Therefore, the developers can use them seamlessly with any communication interface supported in the Bus Abstraction Layer (BAL).

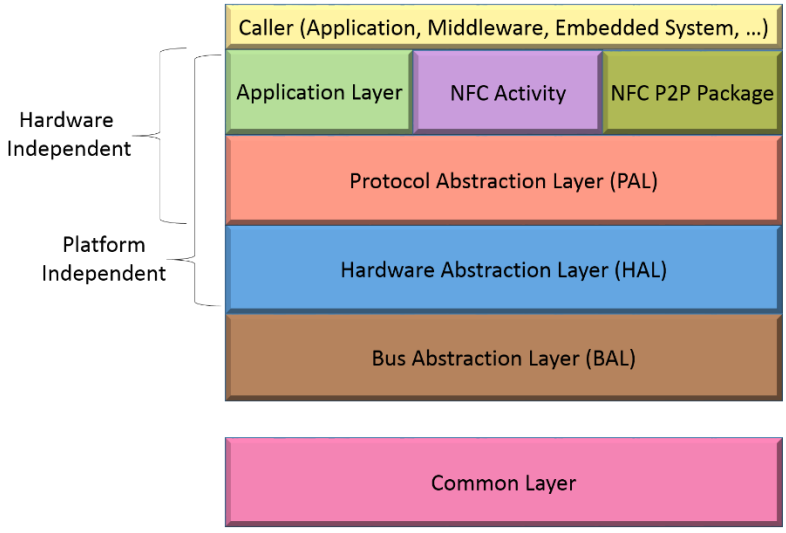


Fig 2. Hardware and Platform independent layers of the NFC Reader Library

In the following subsections, more details on the components and functionalities implemented in each layer are provided.

3.2.1 Bus Abstraction Layer

The Bus Abstraction Layer implements the communication interface between the host device and the contactless reader IC. The host device sends reader IC specific commands and generic commands containing addresses and data bytes. The reader IC responds to the host with data received from contactless cards or related information in requested registers. The NFC Reader Library supports following communication interfaces:

- **LPC1769 SPI**: Enables the communication with the LPC1769 board using the SPI communication interface.
- **LPC1769 I2C:** Enables the communication with the LPC1769 board using the I2C communication interface.
- **Stub**: General-purpose component for the implementation of customer specific communication buses.

3.2.2 Hardware Abstraction Layer

The Hardware Abstraction Layer (HAL) is responsible for the configuration and the execution of native commands of a particular contactless reader IC. These functions are mainly:

- Reading and writing from and into the reader's registers.
- RF field management, receiver and transmitter configuration.
- Timers' configuration.
- Resolving interrupt sources from the reader chip.
- FIFO management.

The NFC Reader Library currently supports the following contactless readers:

- **PN512** [14]: MFRC523 [11], MFRC522 [13]: Highly integrated reader ICs supporting ISO/IEC 14443 Type A, ISO/IEC 14443 Type B, FeliCa and ISO/IEC 18092.
- **CLRC663** [12]: Highly integrated reader IC with the highest RF output power fronted supporting ISO/IEC 14443 Type A and Type B, FeliCa and Passive Initiator mode according to ISO/IEC 18092; and its derivatives (MFRC631 [15], MFRC630 [16], SLRC610 [17]).

The NFC Reader Library is built in a way where upper layers are hardware independent. However, the developer must take into account the NFC capabilities of the selected NFC reader IC. For instance, the CLRC663 reader IC only supports passive communication mode whereas PN512 reader IC supports both active and passive communication modes.

3.2.3 Protocol Abstraction Layer

The protocol abstraction layer inherits hardware-independent implementation of the contactless protocol to be used for the communication.The NFC Reader Library supports the following ISO/IEC contactless standards protocols:

- **ISO14443-3A** [18]: Contactless Proximity card air interface communication at 13.56MHz for the Type A and Jewel contactless cards.
- **ISO14443-3B** [18]**:** Contactless Proximity card air interface communication at 13.56MHz for the Type B contactless cards.
- **ISO14443-4** [18]**:** Specifies a half-duplex block transmission protocol featuring the special needs of a contactless environment and defines the activation and deactivation sequence of the protocol.
- **ISO14443-4A** [18]: Transmission protocol for Type A contactless cards.
- **MIFARE (R):** Contains support for MIFARE authentication and data exchange.
- **FeliCa** (JIS: X6319) [9]: Contactless RFID smart card system from Sony.
- **ISO/IEC 18092 Initiator** [19]: NFC Interface and Protocol standard that enables NFC Data Exchange protocol. Component for devices acting as communication initiators, which implies RF field generation and transmission of communication establishment request. Both active and passive modes are supported.
- **ISO/IEC 18092 Target** [19]: NFC Interface and Protocol standard that enables NFC Data Exchange protocol. Component for devices acting as communication targets, which implies listening of the RF field and the response to the communication establishment requests. Both active and passive modes are supported.

3.2.4 Application Layer

The application layer implements the commands of contactless smart cards.The Application Layer enables the developer to access a particular card API by using its command set (e.g. reading, writing, modifying a sector etc.). The contactless card APIs provided is the following:

- **MIFARE Classic** [4]: MIFARE Classic is compliant with ISO/IEC 14443 Type A up to layer 3 and available with 1k and 4k memory and 7 Byte as well as 4 Byte UIDs.

- **MIFARE Ultralight** [5]**, MIFARE Ultralight EV1** [6] **and MIFARE Ultralight C** [7]: MIFARE Ultralight is compliant with ISO/IEC 14443 Type A up to layer 3.
- **MIFARE DESFire** [8]: MIFARE DESFire is fully compliant with ISO/IEC14443A (part 1 - 4) and uses a subset of ISO/IEC7816-4 commands. The selectable cryptographic methods include 2KTDES, 3KTDES and AES128. The highly secure microcontroller based IC is Common Criteria EAL4+ certified. The NFC Reader Library implements the non-export controlled command set.
- **FeliCa** [9]: FeliCa is a contactless smart card developed by Sony, commonly used in Japan. The command set is partly supported in the NFC Reader Library.
- **Jewel/Topaz** [10]: Jewel tags are compliant with ISO/IEC 14443 Type A up to layer 3, except for the anticollision procedure. They define a 7 byte UID and 120 bytes memory configured in 15 blocks of 8 bytes.
- **NFC Forum Tag Type Operations (TOP):** Provides an abstraction of the underlying hardware (tags) on which the data is stored. The TOP API facilitates the execution of read and write operations on NFC Forum tags as the NFC Reader Library translates these calls to the required specific read and write tag commands. The TOP API relies and leverages on the Application Layer components.

3.2.5 NFC Activity

This component provides an easy way to set the contactless reader IC in a Discovery Loop for detecting NFC contactless tags and P2P devices within the contactless reader IC RF field range.

- **Discovery Loop:** Executes a loop running in a single thread. The application is blocked until the Discovery Loop procedure is finished since the OSAL layer does not provide thread creation capabilities. The Discovery Loop uses MCU timers for measuring guard time intervals between technology detection.

Note: Depending on the manufacturer implementation, the Discovery Loop is also referred to as the polling loop.

3.2.6 NFC P2P Package

This layer implements the NFC Forum standardized protocol stack for a Peer to Peer communication with a NFC device. The NFC P2P package functionalities include the correct management of the logical link between peers – according to LLCP protocol - and the implementation of a client / server based architecture for the exchange of NDEF messages delivered by an upper protocol layer of the P2P application – according to SNEP protocol –.

- **Logical Link Control Protocol (LLCP) [20]:** LLCP is a link protocol layer that specifies the procedural means for transferring of upper layer information units between two NFC devices. It defines the logical link management and the synchronous exchange of data between peers in a connection-oriented or connectionless manner.
- **Simple NDEF Exchange Protocol (SNEP) [21]:** SNEP is an application-level protocol running on top of LLCP suitable for exchanging of application data units, in the form of NDEF messages between two NFC Devices. SNEP is a request/response protocol based on a client/server architecture.

3.2.7 Common Layer

The NFC Reader Library includes a set of utilities which are grouped and encapsulated together in an independent layer called Common Layer. These utilities are not bound to any specific card or hardware, and as such they are functional regardless of the reader IC used. The modules implemented in the Common Layer are the following:

- **Tools**: This module provides 5, 8, 16 and 32 bit length CRC software calculation in addition to the parity encoding and decoding.
- **Key Store**: Key handling software module for storing cryptographic keys used in the authentication and encryption operations. Only the NFC Reader Library Export Controlled version supports high secure key storage capabilities.
- **ISO14443-4 CID Manager**: This module is used when a CID needs to be assigned to an ISO/IEC 14443-4 PICC or a CID is released by the PICC.
- **Log**: Useful module during debugging phase which enables a software tracing mechanism that records information about components during project execution in order to show them on the screen or store them to a file.
- **OSAL utils**: This module provides an API for timer and memory management related applications in a software and hardware independent way for an easier and quicker development.

3.2.8 Building a Project from bottom to top

In order to use the NFC Reader Library, a stack of components has to be initialized from bottom to top. Every component in the software stack has to be initialized before it can be used. The referred initialization of each layer generates a data context which feeds the immediate upper layer. Some of the components may need a data context coming from the same layer to be used as an entry point.

For instance, if we aim to develop a MIFARE DESFire application, we must previously initialize the ISO/IEC14443 components of the underlying PAL layer. But in order to use ISO/IEC14443 components, we must have previously initialized the contactless reader component from the HAL layer, which similarly requires the previous initialization of the communication interface between the contactless reader and the MCU in the BAL layer.

The Fig 3 illustrates the mentioned implementation for the initialization procedure of a MIFARE DESFire application using a CLRC663 contactless reader and a MCU host.

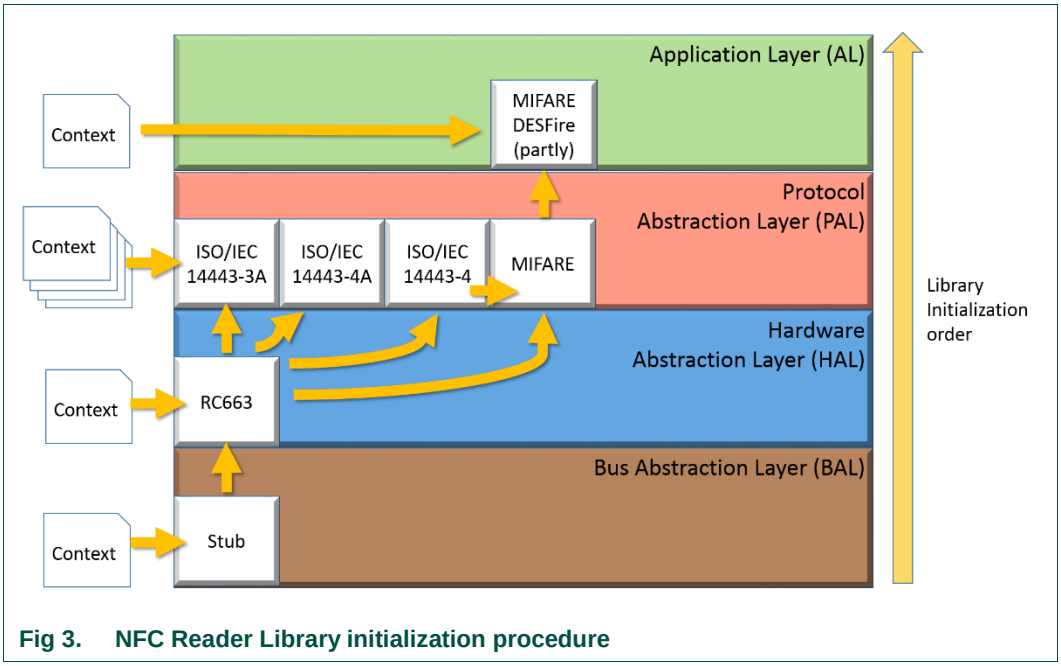


Fig 3. NFC Reader Library initialization procedure

3.3 NFC Reader Library and NFC Operating Modes

The NFC Reader Library provides developers with different APIs for building NFC applications with NXP reader ICs. The NFC Reader Library should be initialized according to the NFC application requirements and the NFC operating modes that will be used. It is recommended to initialize only the required components in order to reduce the code size. The NFC Reader Library implements the relevant NFC Forum specifications associated to each operating mode.

- Read/Write mode: Support of NFC Forum Tag Type Operation specification to allow hardware independent operations on top of the four NFC Forum Type Tags.
- Peer to Peer mode: Support of LLCP link layer protocol and SNEP application level protocol to ensure a reliable communication with NFC Forum devices.
- Card Emulation: Support for card emulation will be implemented and made available in future software releases.

The allowed transfer speeds and modulation schemes for each operation are out of the scope of this document. For further details, please refer to the corresponding standards' documentation.

3.3.1 Read/Write Mode

The Read/Write mode allows a NFC reader to perform read and write operations on any contactless tag or card. The content of the card might be protected or be public.

For those use cases where the customer aims to reach as much audience as possible, e.g. smart advertising, Read/Write mode leverages on the NFC Forum Data Exchange Format (NDEF) for the data encapsulation and NFC Forum Tag Type platforms to provide a hardware-independent solution.

In order to operate on Read/Write mode, the layers and components to be considered are shown in Fig 4.

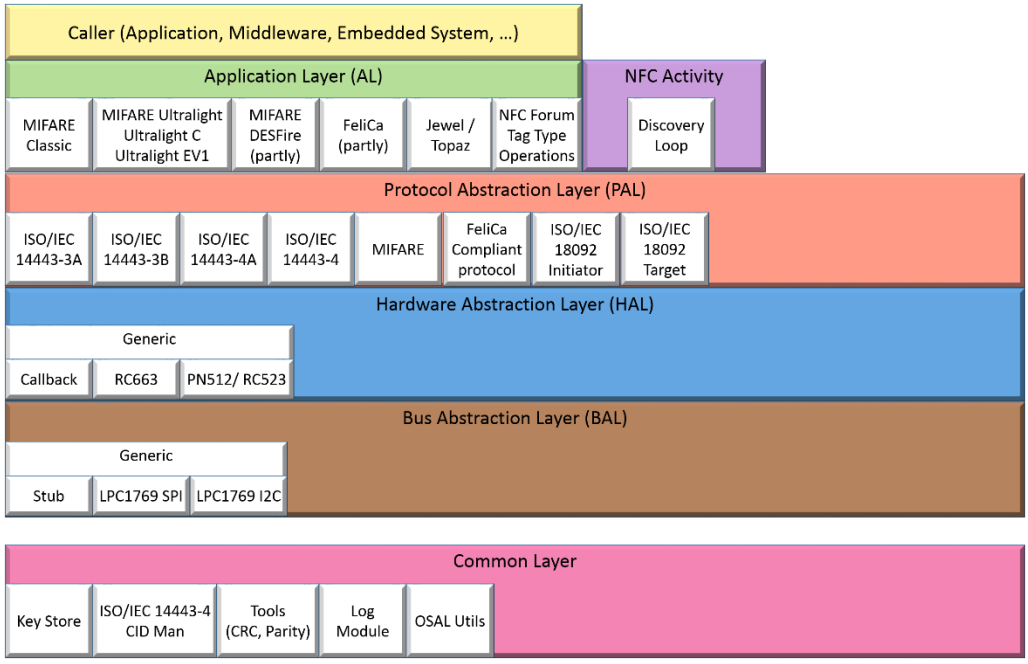


Fig 4. NFC Reader Library - Read and Write relevant modules

3.3.2 Peer-to-Peer Mode

The Peer-to-Peer (P2P) mode allows two NFC devices to exchange information with each other when they are brought into close proximity. The NFC P2P mode establishes a bidirectional channel between the two NFC devices to exchange data such as contacts, URLs, Bluetooth or Wi-Fi pairing information, and others.

The device starting the communication is called the Initiator device and the responding device is called the Target device. P2P is the only mode supporting both Active and Passive communication modes. In active communication mode both Initiator and Target generate their own RF field. In passive communication mode, the target modulates the RF field generated by the Initiator.

In order to enable the communication between existing NFC Forum devices, the NFC Forum has released the LLCP link layer protocol specification and the SNEP application layer specification.

If P2P is the selected operation mode, the layers and components to be considered are shown in Fig 5.

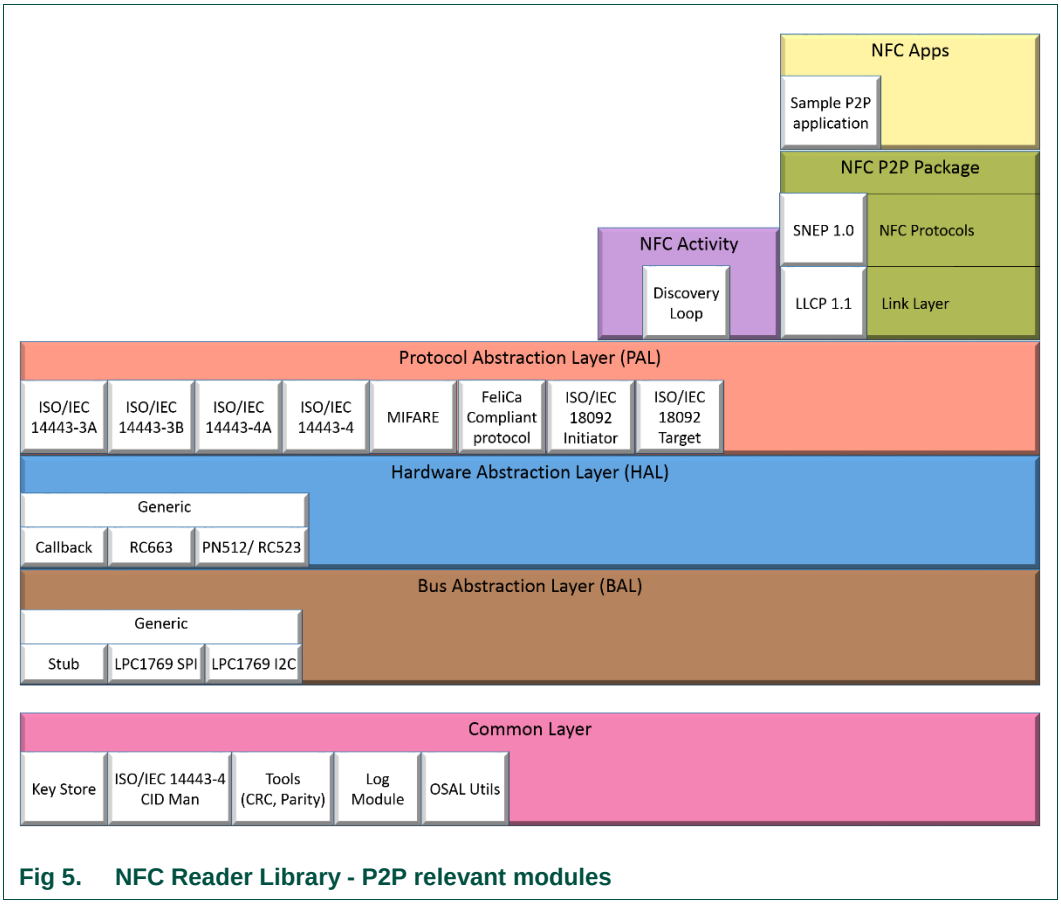


Fig 5. NFC Reader Library - P2P relevant modules

3.3.3 Card Emulation

The Card Emulation mode allows a NFC reader IC to emulate the behaviour of a contactless card or tag. The card emulation functionalities will be available in next releases of the NFC Reader Library.

3.4 NXP Export Controlled Reader Library

The Export Controlled version of the Reader Library [2] is an extension of the NXP Reader Library which provides full support for MIFARE Plus and MIFARE DESFire cards and enables the usage of Secure Application Module (SAM), designed to support secure storage of cryptographic keys and the implementation of cryptographic functions in the transactions between the contactless smart card and the contactless reader.

The distribution of the Export Controlled Reader Library software is subject to the signature of a NDA with NXP since some modules are bound to export control regulations. In order to sign a NDA with NXP please contact your NXP representative. The NXP Export Controlled Reader Library can be downloaded from DocStore [30].

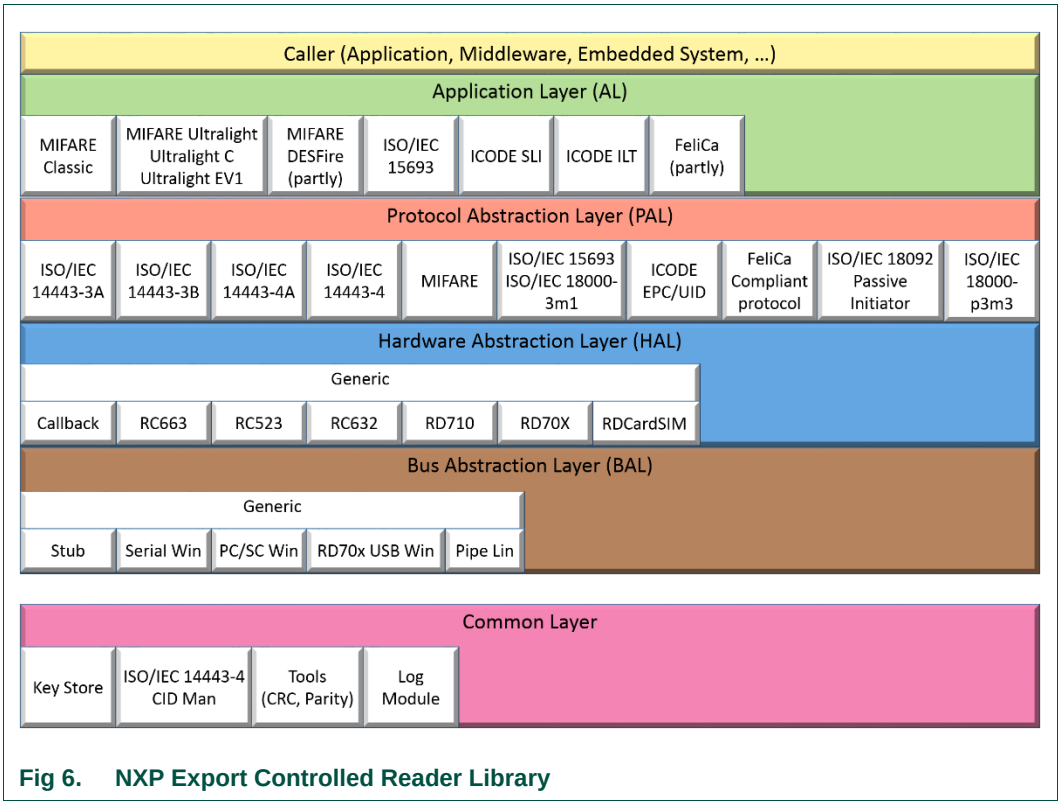


Fig 6. NXP Export Controlled Reader Library

4. NFC Reader Library API: Protocol Abstraction Layer (PAL)

In this section, the ISO/IEC 14443, MIFARE, FeliCa and ISO/IEC 18092 Initiator and Target components defined in the Protocol Abstraction Layer (PAL) of the NFC Reader Library are explained in depth.

4.1 ISO/IEC 14443

The ISO/IEC 14443 is the proximity contactless smartcard standard which describes the communication between a proximity reader IC and a contactless smartcard. In particular, it describes the physical characteristics of the cards, as well as the RF communication parameters and the contactless protocol to setup the communication and allow the exchange of data from the application layer.

The standard is divided into four different parts: Part 1 describes the physical characteristics of the proximity contactless cards and the antennas. Part 2 specifies the power, frequency and modulation of the RF field as well as the coding of the bits for the communication. Part 3 explains how the communication between the reader IC and the contactless smartcard is established. Finally, part 4 specifies the protocol for the application layer to exchange data once the communication has been setup.

This standard provides two different flavours to perform the communication, which are known as Type A and Type B. Although both types work at the same frequency and

power range, they use different kinds of bit coding and modulation techniques, as well as a different protocol to establish the communication.

Both NXP CLRC 663 reader IC and NXP PN512 reader IC support Type A and Type B communication.

In order to get familiar with the lower layers of the communication between a proximity reader and a contactless smartcard, a brief introduction will be given to the parts 3 and 4 of this standard on the following subsections, as well as a description of the corresponding functions from the library.

4.1.1 ISO/IEC 14443-3A

In this part of the standard, the Type A communication initialization is described. This initialization consists of three steps. First, the reader IC checks if there are contactless smartcards within its RF field. Second, if there is more than one contactless smartcard, the anticollision algorithm is performed. And finally, the desired contactless smartcard is selected.

The reader IC is the responsible for managing the different states of the contactless smartcard. The state diagram of the contactless smartcard is illustrated in Fig 7,

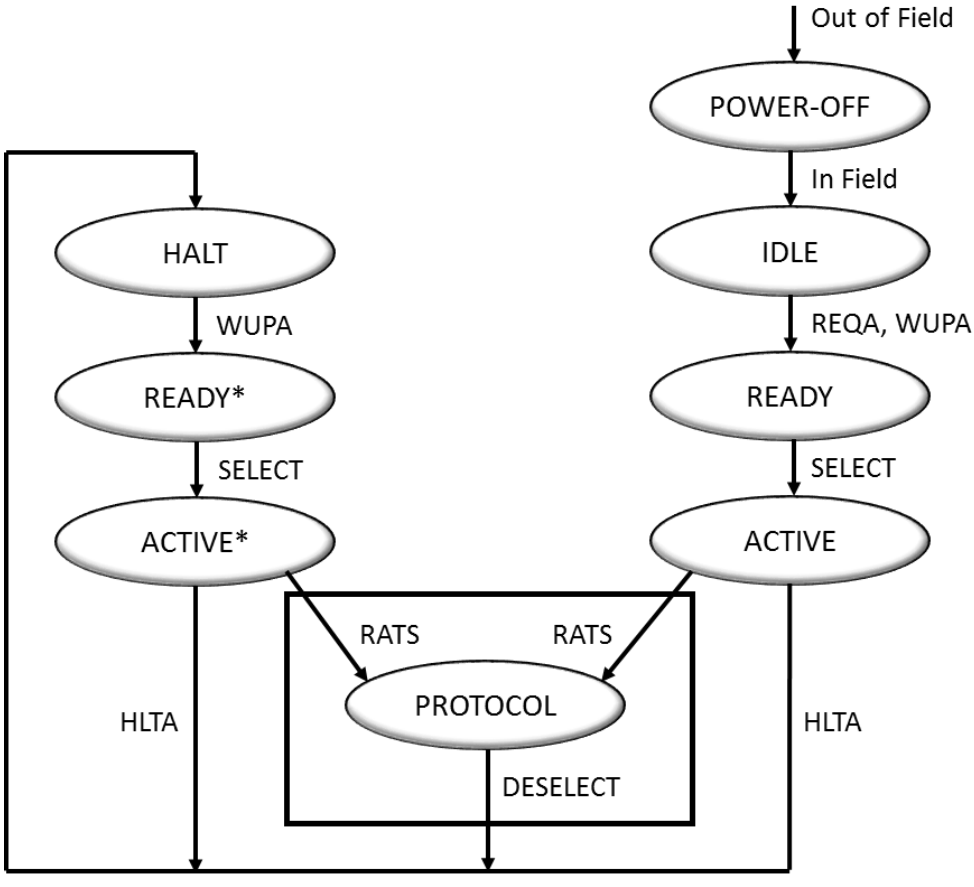


Fig 7. Type A proximity contactless smartcard state diagram

If the card is not in the reader RF field, it is in the POWER-OFF state. Once the card is powered, it enters the IDLE state. It stays in this state until it receives a REQA or WUPA command from the reader IC, to which it shall answer with an ATQA command and switches to READY state. In the READY state, the anticollision algorithm is performed. This algorithm starts with the reader IC sending an anticollision command. All the cards shall respond to this command with their UID (Unique Identifier). As different cards have different UIDs, if there are more than one card answering at the same time, the reader IC will be able to detect the collision and differentiate each card through their UID. The UID of a card may consist of 4, 7 or 10 bytes, and therefore, the anticollision method is an iterative process in which the number of iterations depends on the length of the UID of the card selected.

Once a card is chosen, it is selected through the SELECT command. The card answers with a SAK command, indicating if it is compliant with the ISO/IEC 14443-4 part of the standard or if it supports other proprietary higher layer protocols, and enters the ACTIVE state. In case the card is compliant with the ISO/IEC 14443-4A, the reader IC shall send a RATS command (see 4.1.2.4). Otherwise, it sends another proprietary command to begin with the higher layer dialog. The reader IC can also send a HLTA command, making the card enter the HALT state. This state is similar to the IDLE state, with the difference that in this state, the card ignores the REQA command and can only be driven to the READY state through a WUPA command from the reader.

An example of an initialization dialog between a reader and a Type A card with a 7 bytes UID is shown below:

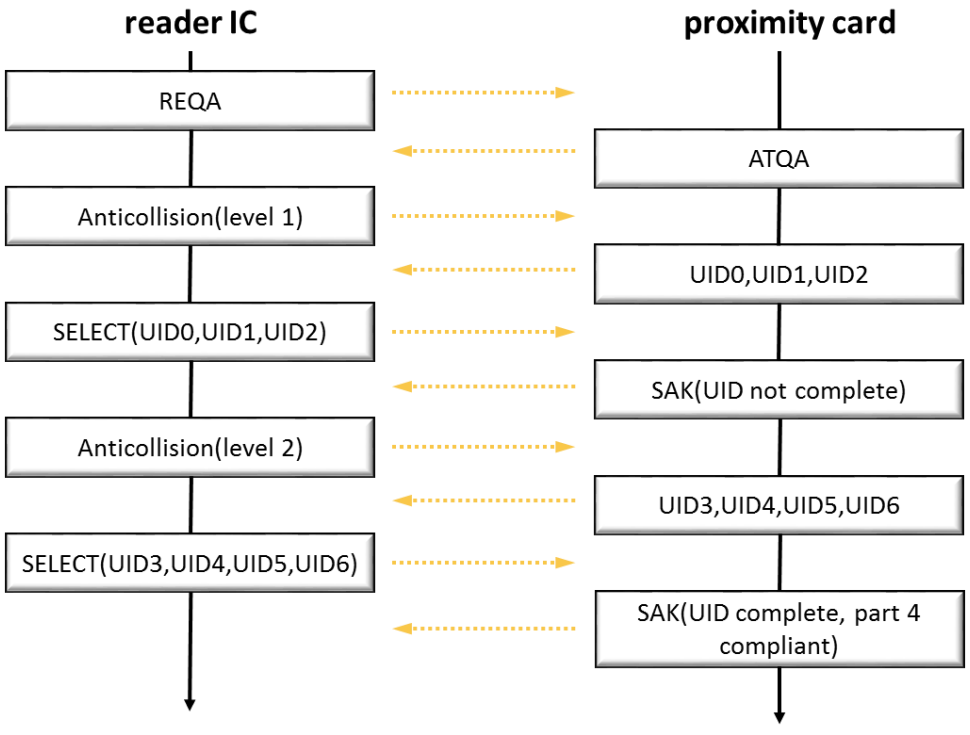


Fig 8. ISO/IEC 14443-3A Operation Flow

For further details, please refer to the ISO/IEC 14443-3A standard [18].

The NFC Reader Library implementation of the ISO/IEC14443-3A is described below.

4.1.1.1 ISO/IEC 14443-3A Data Parameter Structure

A special structure has been defined in the NFC Reader Library in order to store the parameters related to the ISO/IEC 14443-3A standard. This structure has been called phpalI14443p3a_Sw_DataParams_t:

```
typedef struct{
   void  * pHalDataParams;
   uint8_t abUid[10];
   uint8_t bUidLength;
   uint8_t bUidComplete;
} phpalI14443p3a_Sw_DataParams_t;
```

***pHalDataParams**: Pointer to the underlying HAL parameter structure.

abUid[10]: Array holding the UID of the card in the ACTIVE state.

bUidLength: Length of the UID stored in abUid[]. Depending on the card, it may be 4, 7 or 10 bytes.

bUidComplete: If this variable value is 1 indicates that the UID is complete, a value of 0 means it is incomplete

4.1.1.2 Initialization ISO/IEC 14443-3a

This function initiates the PAL ISO/IEC 14443-3A component.

```
phStatus_t phpalI14443p3a_Sw_Init(
   phpalI14443p3a_Sw_DataParams_t * pDataParams,      [In]
   uint16_t wSizeOfDataParams,                        [In]
   void * pHalDataParams );                           [In]
```

***pDataParams:** Pointer to the PAL layer data parameter component.

wSizeOfDataParams: Specifies the size of the data parameter structure. It is recommended to pass sizeof(phalMfc_Sw_DataParams_t).

***pHalDataParams:** Pointer to the corresponding underlying HAL parameter component, depending on the used reader.

This function may return the following values:

PH_ERR_SUCCESS: Operation successful.

PH_ERR_INVALID_DATA_PARAMS: wSizeOfDataParams does not match with the defined size of the PAL phpalI14443p3a_Sw_DataParams_t structure.

4.1.1.3 Activate Card

This function changes the card status to ACTIVE state, whether it was in the HALT state or it had not been activated yet. It performs the whole activation procedure, covering all possible states and situations, as specified in ISO/IEC 14443-3A standard. Therefore, most of the other functions from this module are called by this function, allowing it to send different commands such as REQA, WUPA, Anticollision or SELECT.

If the activation is successful, the complete UID of the activated card is acquired. Even when there are more cards present in the RF field, the activation function ensures that just one UID is captured (and therefore, only one card is activated).

```
phStatus_t phpalI14443p3a_ActivateCard(
    void * pDataParams,                 [In]
    uint8_t * pUidIn,                   [In]
    uint8_t bLenUidIn,                  [In]
    uint8_t * pUidOut,                  [Out]
    uint8_t * pLenUidOut,               [Out]
    uint8_t * pSak,                     [Out]
    uint8_t * pMoreCardsAvailable );    [Out]
```

***pDataParams:** Pointer to the PAL layer data parameter component. UidLength and Uidab attributes are changed during the execution of this function.

***pUidIn:** Pointer to the UID of the card to be activated. If this variable is NULL and there is at least one card in the reader RF field, it will be activated and the card UID will be read.

bLenUidIn: Number of relevant bytes of the pUidIn array. It can take the values 0, 4, 7 and 10:

0 – It means that the UID is unknown. Therefore, the function begins with a REQA command. At the end of the function, the complete UID of the card shall be captured.

4, 7, 10 – The function begins with a WUPA command.

***pUidOut:** Pointer to the complete UID of the activated card.

***pLenUidOut:** Length of pUidOut. Only values 4, 7 and 10 are possible.

***pSak:** SAK command received from the card. It is one byte long and specifies the type of card.

***pMoreCardsAvailable:** Indicates whether one or more cards are within the reader RF field at the same time.

PH_ON: More than one card is in the RF field. A collision occurred.

PH_OFF: Only one card in the reader RF.

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

PH_ERR_INVALID_PARAMETER: bLenUidIn is not equal to 0, 4, 7 or 10.

Other: Value returned by the underlying component.

4.1.1.4 Request A

This function transmits a request Type A command (REQA) and waits to receive an answer to that request (ATQA). The data rate is automatically set to 106 kbit/s for both receiver and transmitter. During this operation, the CRC module of the reader chip is turned off for both reception and transmission signal. After the REQA command is transmitted, the routine waits for any answer until a timeout event (timers T1, T0) occurs.

```
phStatus_t phpalI14443p3a_RequestA(
    void * pDataParams,                 [In]
    uint8_t * pAtqa );                  [Out]
```

***pDataParams:** Pointer to the PAL layer data parameter component.

***pAtqa:** Pointer to ATQA. If the request process is successful, then the 2 bytes from the ATQA received are written in this variable.

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

PH_ERR_PROTOCOL_ERROR: Invalid response received.

Other: Value returned by the underlying component.

Note: If no answer to the REQA command is received nor any changes in the RF field are detected by the reader before the timeout, the function terminates with a timeout error. There are two time constants defined in the *phpalI14443p3a_Sw_Int.h* file that determine the ATQA timeout: PHPAL_I14443P3A_EXT_TIME_US, PHPAL_I14443P3A_SELECTION_TIME_US.

The resulting waiting time is the sum of the both values in microseconds.

4.1.1.5 Wake Up A

This function changes to ACTIVE state one card that is in HALT state.

```
phStatus_t phpalI14443p3a_WakeUpA(
    void * pDataParams,                 [In]
    uint8_t * pAtqa );                  [Out]
```

***pDataParams:** Pointer to the PAL layer data parameter component.

***pAtqa** Pointer to ATQA. If the wake up process is successful, then the 2 bytes from the ATQA received are written in this variable.

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

PH_ERR_PROTOCOL_ERROR: Invalid response code received.

Other: Value returned by the underlying component.

4.1.1.6 Anticollision

This function is responsible for performing the Type A anticollision procedure. This is performed through a process at which the reader IC obtains the whole UID of one of the cards. The anticollision routine may perform 1 to 3 loops depending on the length of the UID of the card chosen. The anticollision procedure is mandatory for ISO/IEC 14443A compliant products, and all the NXP MIFARE products support this anticollision procedure.

```
phStatus_t phpalI14443p3a_Anticollision(
    void * pDataParams,                 [In]
    uint8_t bCascadeLevel,              [In]
    uint8_t * pUidIn,                   [In]
    uint8_t bNvbUidIn,                  [In]
    uint8_t * pUidOut,                  [Out]
    uint8_t * pNvbUidOut );             [Out]
```

***pDataParams:** Pointer to the PAL layer data parameter component.

bCascadeLevel: Number of loop in which the anticollision procedure is at this moment. This parameter may take three values:

#define PHPAL_I14443P3A_CASCADE_LEVEL_1 0x93

#define PHPAL_I14443P3A_CASCADE_LEVEL_2 0x95

#define PHPAL_I14443P3A_CASCADE_LEVEL_3 0x97

The reader transmits the current cascade level in the anticollision command. If this value differs from the three values above, the command is invalid.

***pUidIn:** Pointer to the UID of the card.

bNvbUidIn: Number of valid bits in the UID of the card currently processed by the anticollision procedure. This variable consists of two parts: the four MSB (Most Significat bit) keep the information of the number of complete valid bytes, and the four LSB (Least Signicant bit) keep the number of remaining valid bits.

***pUidOut:** Pointer to the array where the updated UID of the card is loaded. During the operation of this function, the first byte of the UID may equal PHPAL_I14443P3A_CASCADE_TAG, meaning that the UID is not complete yet, and that another anticollision loop will be required.

***pNvbUidOut:** Length of the UID array. It specifies how many bytes of the UID are currently relevant.

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

PH_ERR_INVALID_PARAMETER: Invalid `bCascadeLevel` or invalid `bNvbUidIn`.

PH_ERR_PROTOCOL_ERROR: Invalid response received.

Other: Value returned by the underlying component.

4.1.1.7 Selection

This function allows the reader to send a SELECT command. This function and phpalI14443p3a_Anticollision() are both together implemented within the phpalI14443p3a_ActivateCard() function.

After the SELECT command has been successfully sent, the card specified through the UID responds with a SAK command indicating the card type (MIFARE Classis 1k, 4k, MIFARE DESFire, etc.).

```
phStatus_t phpalI14443p3a_Select(
    void * pDataParams,                [In]
    uint8_t bCascadeLevel,             [In]
    uint8_t * pUidIn,                  [In]
    uint8_t * pSak );                  [Out]
```

***pDataParams:** Pointer to the PAL layer data parameter component. This function sets the `pDataParam->UidComplete` flag to 1 when the acquisition of the UID is completed successfully.

bCascadeLevel: Number of loop in which the anticollision procedure is. This parameter may take three values:

#define PHPAL_I14443P3A_CASCADE_LEVEL_1 0x93

```
#define PHPAL_I14443P3A_CASCADE_LEVEL_2     0x95
#define PHPAL_I14443P3A_CASCADE_LEVEL_3     0x97
```

***pUidIn:** UID of the card to be selected. This value should not be NULL, as the function needs to know the whole UID of the card to be selected (or the part of it corresponding to the current cascade level).

***pSak:** The SAK value of the card. . In the special case of receiving a SAK value of 0x04, this means that the UID of the selected card is not completed yet and a new loop in the anticollision procedure is required.

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

PH_ERR_PROTOCOL_ERROR: Mismatch between the first byte of the UID and the SAK. If the first byte of the UID is pUidIn[0] = PHPAL_I14443P3A_CASCADE_TAG, the SAK received must be SAK = 0x04, and vice versa.

Other: Value returned by the underlying component.

4.1.1.8 Halt A

After a card has been activated, the reader can make the card enter into the HALT state. The card can be later reactivated through a WUPA command, or using the phpalI14443p3a_ActivateCard() function.

```
phStatus_t phpalI14443p3a_HaltA(
    void * pDataParams );                    [In]
```

***pDataParams:** Pointer to the PAL layer data parameter component.

The values returned by the function can be:

PH_ERR_SUCCESS: The card has entered the HALT state successfully.

PH_ERR_PROTOCOL_ERROR: A protocol error has occurred and the card has not entered the HALT state.

Other: Value returned by the underlying component.

4.1.1.9 Exchange

Most of the ISO/IEC 14443-3A related functions are based on a half-duplex bidirectional communication between the reader IC and the card, in which the reader IC sends a command and waits for a response from the card. This function gives the possibility to the developer to send an array of bytes to the card and read the corresponding response.

```
phStatus_t phpalI14443p3a_Exchange(
    void * pDataParams,                      [In]
    uint16_t wOption,                        [In]
    uint8_t * pTxBuffer,                     [In]
    uint16_t wTxLength,                      [In]
    uint8_t ** ppRxBuffer,                   [Out]
    uint16_t * pRxLength );                  [Out]
```

***pDataParams:** Pointer to the PAL layer data parameter component.

wOption: All ISO/IEC 14443-3 functions pass the value EXCHANGE_DEFAULT as the default parameter.

***pTxBuffer:** Pointer to the array of data to be transmitted. This array actually contains the reader command defined by its byte code and the corresponding data.

wTxLength: Number of bytes to be transmitted.

****ppRxBuffer:** Pointer to the received array of data.

***pRxLength:** Pointer to the address where the information about the received data is.

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

Other*:* Value returned by the underlying component.

4.1.2 ISO/IEC 14443-4A

This part of the standard performs the transmission protocol negotiation for the Type A communication. In this negotiation, both parties can increase the bit rate of the communication (if supported by the card). Besides, both the reader IC and the card can indicate the maximum frame size they accept, and the card can also inform the reader about other parameters, such as the frame waiting time (time within the card shall start the response frame after the end of the corresponding reader frame), or its compliancy with CID (The CID is a logical address that allows the reader to keep up to 15 cards activated at the same time, and to communicate with them without the need of sending the others to the HALT state).

The communication negotiation is as follows. Once a Type A card is in the ACTIVE state (and assuming that is ISO/IEC 14443-4 compliant) the reader sends a RATS (Request Answer to Select) command indicating the maximum frame size that it accepts and the CID assigned to the card. The card answers with an ATS (Answer To Select) command, which may contain, optionally, its maximum frame size, the bit rates that it supports (106, 212, 424 or 848 Kbit/s), parameters related to the time between frames, indicators of whether the card supports CID and NAD or not and other optional information.

If the card supports higher bit rates than 106 Kbit/s, the reader can send a Protocol Parameter Selection (PPS) request indicating the speed at which the rest of the communication will be performed. Once the card has answered to this command with a PPS response, the communication is completely set up

For further details, please refer to the ISO/IEC 14443-4 standard [18].

The NFC Reader Library implementation regarding the part four of the ISO/IEC standard is described below

4.1.2.1 ISO/IEC 14443-4A Data Parameter Structure

A special structure has been defined in the NFC Reader Library in order to store the parameters related to the ISO/IEC 14443-4A standard. This structure has been called phpalI14443p4a_Sw_DataParams_t:

```
typedef struct{
    void  * pHalDataParams;
    uint8_t bCidSupported;
    uint8_t bNadSupported;
```

```
    uint8_t bCid;
    uint8_t bBitRateCaps;
    uint8_t bFwi;
    uint8_t bFsci;
    uint8_t bFsdi;
    uint8_t bDri;
    uint8_t bDsi;
} phpalI14443p4a_Sw_DataParams_t;
```

***pHalDataParams**: Pointer to the underlying HAL layer data parameter component. This attribute can only be assigned by phpalI14443p4_Sw_Init() (see 4.1.2.2).

bCidSupported: Card Identifier (CID) support flag. If the variable is non-zero, CID is supported.

bNadSupported: NAD support flag. If the variable is non-zero, NAD is supported.

bCid: Card Identifier value assigned to the card. It is ignored if bCidSupported is zero. The possible values it can take are in the range from 0 to 14. When a new CID is required, the allocation function phCidManager_GetFreeCid() should be called. When a card is deselected, the CID assigned to that card shall be released using the phCidManager_FreeCid() function.

bBitRateCaps: Raw TA(1) byte of the ATS received. This byte contains the values (known as divisors) with the information about the different bit rates that the card is able to handle. No API function has been implemented in the NFC Reader Library to parse this byte.

bFwi: Frame Waiting time Integer. Byte code that determines the Frame Waiting Time: the time within the card shall start the response frame after the end of the corresponding reader frame. The FWT is calculated by the following formula:

$$FWT = (256 \times 16 / 13.56MHz) \times 2^{bFwi}$$

The FWI can take values from 0 to 14. The reader waits for a response for a time of FWT + 60us. If no answer has been received within that time, the reader drops the communication with the card.

bFsci: Frame Size for proximity Card Integer. It contains information about the maximal number of bytes that the card is able to receive in a single frame. It can take values from 0 to 8. The corresponding maximum frame size is obtained according to the table below:

Table 1. bFsdi (bFsci) to FSD (FSC) conversion

bFsdi	0	1	2	3	4	5	6	7	8
FSD (bytes)	16	24	32	40	48	64	96	128	256

bFsdi: Frame Size for proximity coupling Device (Reader) Integer. It contains information about the maximal number of bytes that the reader is able to receive in a single frame. It can take values from 0 to 8. The corresponding maximum frame size is obtained according to the table above.

bDri: Divisor for the Reader to card communication. This parameter acts as a divisor when referring to the bit duration. When referring to the bit rate, it behaves as a multiplier. There are 4 legal values (listed in Table 2) defined for this parameter, each representing a different bit rate.

bDsi: Divisor for the card to reader communication. This parameter acts as a divisor when referring to the bit duration. When referring to the bit rate, it behaves as a

multiplier. There are 4 legal values (listed in Table 2) defined for this parameter, each representing a different bit rate.

4.1.2.2 Initialization ISO/IEC 14443-4A Parameter Component

This function is used to assign initial values to the attributes of the phpalI14443p4a_Sw_DataParams_t parameter component. The pointer to the underlying HAL is the only attribute that can be assigned as an input argument. The values of the corresponding ISO/IEC 14443-4 parameters are set during the RATS – ATS command exchange procedure (see 4.1.2.4).

```
phStatus_t phpalI14443p4a_Sw_Init(
    phpalI14443p4a_Sw_DataParams_t * pDataParams,      [In]
    uint16_t wSizeOfDataParams,                        [In]
    void * pHalDataParams );                           [In]
```

***pDataParams**: Pointer to the phpalI14443p4a_Sw_DataParams_t parameter component.

wSizeOfDataParams: Specifies the size of the data parameter structure. It is recommended to pass sizeof(phpalI14443p4a_Sw_DataParams_t).

***pHalDataParams**: Pointer to the underlying HAL layer data parameter component, .

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

PH_ERR_INVALID_DATA_PARAMS: wSizeOfDataParams does not match with the defined size of the PAL phpalI14443p4_Sw_DataParams_t structure.

4.1.2.3 Activate Card

This function sends a RATS command (phpalI14443p3a_Rats()) followed by a PPS command (phpalI14443p3a_Pps()) when appropriate.

If the reader does not want to increase the 106Kbit/s bit rate to a faster one, the PPS sequence is not performed. The verification whether the card supports a different bit rate for each direction has not been implemented. In the case this needs to be checked, the whole TA(1) byte received is stored in phpalI14443p4a_Sw_DataParams_t->bBitRateCaps.

As specified in the ISO/IEC 14443 standard, the RATS command can be sent after a successful selection of the card.

```
phStatus_t phpalI14443p4a_ActivateCard(
    void * pDataParams,                    [In]
    uint8_t bFsdi,                         [In]
    uint8_t bCid,                          [In]
    uint8_t bDri,                          [In]
    uint8_t bDsi,                          [In]
    uint8_t * pAts );                      [Out]
```

***pDataParams**: Pointer to the phpalI14443p4a_Sw_DataParams_t parameter component.

bFsdi: Frame Size for proximity coupling Device (Reader) Integer. It contains information about the maximal number of bytes that the reader is able to receive in a single frame. It

can take values from 0 to 8. The corresponding maximum frame size is obtained according to Table 1.

bCid: Card Identifier number.

bDri: Divisor for the reader to card communication. This parameter acts as a divisor when referring to the bit duration. When referring to the bit rate, it behaves as a multiplier. There are 4 legal values (listed in Table 2) defined for this parameter, each representing a different bit rate.

bDsi: Divisor for the card to reader communication. This parameter acts as a divisor when referring to the bit duration. When referring to the bit rate, it behaves as a multiplier. There are 4 legal values (listed in Table 2) defined for this parameter, each representing a different bit rate.

***pAts**: Pointer to the buffer where the received ATS response is stored. Its content is parsed by the function and stored in the data parameter component.

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

PH_ERR_INVALID_PARAMETER: Invalid value of bDri or bDsi (see Table 2).

PH_ERR_PROTOCOL_ERROR: Invalid response received.

Other: Value returned by the underlying component.

4.1.2.4 RATS

This function stores all the parameters obtained during the RATS – ATS command exchange procedure. As specified in the ISO/IEC 14443 standard, the RATS command (and consequently this API function) can be sent after a successful selection of the card has been performed using the functions phpalI14443p3a_Select() or phpalI14443p3a_ActivateCard() (see sections 4.1.1.7 or 4.1.1.3). As an alternative to this function, see phpalI14443p4a_ActivateCard() in section 4.1.2.3, which also implements the PPS procedure.

When receiving the ATS answer from the card, all the relevant information is parsed and stored in the corresponding phpalI14443p4a_Sw_DataParams_t parameter component in accordance with the ISO/IEC 14443-4 standard. Therefore, each attribute of the parameter component will contain the information obtained from the ATS received, or its default value, in case that parameter was not included in the ATS. By default, the communication bit rate is set to 106kbit/s, and can be later renegotiated through the phpalI14443p4a_Pps() function (see 4.1.2.5).

When this function is executed, the reader IC performs the following steps:

First, it sends a RATS command containing FSDI and CID. CID should be obtained from the CID manager by calling the phCidManager_FreeCid() function.

Then, it waits for the ATS response from the card, or for the timeout. The timeout occurs 4833 + 60 (extension) microseconds after the end of the frame sent.

Finally, if the ATS is received, it parses and stores the relevant information in the corresponding phpalI14443p4a_Sw_DataParams_t parameter component. The rules for using the default values are implemented as well. If there is a disagreement with the ISO/IEC 14443-4A part of the standard in the ATS received, the function returns a protocol error.

The TA(1) byte is not parsed nor tested its compliance with the ISO/IEC standard. Its raw content is directly stored in the bBitRateCaps attribute of the phpalI14443p4a_Sw_DataParams_t parameter component.

The function waits for a SFGT (Start-up Frame Guard Time) amount of time after having received the ATS frame.

The frame waiting time is calculated from the FWI parameter, and the reader timeout is set to this value.

If the card does not respond with a valid ATS frame, the reader sends a DESELECT request command (repeatedly, as described in section 4.1.4.9). If this also fails, the reader sends a HLTA command.

```
phStatus_t phpalI14443p4a_Rats(
    void * pDataParams:          [In]
    uint8_t bFsdi:               [In]
    uint8_t bCid                 [In]
    uint8_t * pAts );            [Out]
```

***pDataParams**: Pointer to the phpalI14443p4a_Sw_DataParams_t parameter component.

bFsdi: Frame Size for proximity coupling Device (Reader) Integer. It contains information about the maximal number of bytes that the reader is able to receive in a single frame. It can take values from 0 to 8. The corresponding maximum frame size is obtained according to Table 1.

bCid: Card Identifier value. The possible values it can take are those in the range from 0 to 14. When a new CID is required, the allocation function phCidManager_GetFreeCid() should be called.

***pAts**: Pointer to the buffer where the received ATS command is stored. Its content is parsed by the function and stored in the data parameter component.

The values returned by the function can be:

PH_ERR_SUCCESS; Operation successful.

PH_ERR_PROTOCOL_ERROR; Invalid response received.

PH_ERR_INVALID_PARAMETER: bFsdi greater than 8 or bCid greater than 14.

Other; Value returned by the underlying component.

4.1.2.5 Protocol and Parameter Selection

This function performs the ISO/IEC 14443-4 PPS request command. For the moment, this command can only be used to change the bit rate of the communication.

The reader IC sends the desired bit rate to the card and waits for a response. Once the confirmation response from the card is received, both reader IC and card are configured to work at the new agreed bit rate. According to the ISO/IEC 14443 standard, a card can be configured using a PPS command just subsequently after a RATS command (phpalI14443p4a_Rats()). The verification whether the card supports a different bit rate for each direction has not been implemented. In the case this needs to be checked, the whole TA(1) byte received is stored in `phpalI14443p4a_Sw_DataParams_t->bBitRateCaps`.

```
phStatus_t phpalI14443p4a_Pps(
    void * pDataParams,          [In]
    uint8_t bDri,                [In]
    uint8_t bDsi);               [In]
```

***pDataParams**: Pointer to the phpalI14443p4a_Sw_DataParams_t parameter component. Its member bCid identifies the card.

bDri: Divisor for the reader to card communication. This parameter acts as a divisor when referring to the bit duration. When referring to the bit rate, it behaves as a multiplier. There are 4 legal values (listed in Table 2) defined for this parameter, each representing a different bit rate.

bDsi: Divisor for the card to reader communication. This parameter acts as a divisor when referring to the bit duration. When referring to the bit rate, it behaves as a multiplier. There are 4 legal values (listed in Table 2) defined for this parameter, each representing a different bit rate.

Table 2. DRI and DSI identifiers

DRI or DSI identifier	Bit rate
PHPAL_I14443P4A_DATARATE_106	106 kbit/s
PHPAL_I14443P4A_DATARATE_212	212 kbit/s
PHPAL_I14443P4A_DATARATE_424	424 kbit/s
PHPAL_I14443P4A_DATARATE_848	848 kbit/s

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

PH_ERR_INVALID_PARAMETER: Invalid value for bDri or bDsi used.

PH_ERR_PROTOCOL_ERROR: Invalid response received.

Other: Value returned by the underlying component.

4.1.2.6 Get ISO/IEC 14443-4A Parameters

This function returns all the attributes of a phpalI14443p4a_Sw_DataParams_t parameter component.

```
phStatus_t phpalI14443p4a_GetProtocolParams(
    void * pDataParams,          [In]
    uint8_t * pCidEnabled,       [Out]
    uint8_t * pCid,              [Out]
    uint8_t * pNadSupported,     [Out]
    uint8_t * pFwi,              [Out]
    uint8_t * pFsdi,             [Out]
    uint8_t * pFsci );           [Out]
```

***pDataParams**: Pointer to the phpalI14443p4a_Sw_DataParams_t parameter component.

***pCidEnabled**: Pointer to the CID enabling flag. If it is non-zero, it means that the CID is enabled.

***pCid**: Pointer to the CID value. The possible values it can take are those in the range from 0 to 14.

***pNadSupported**: Pointer to the NAD support flag. If it is non-zero, it means that the CID is enabled.

***pFwi**: Frame Waiting time Integer. Byte code that determines the Frame Waiting Time: the time within which the card shall start the response frame after the end of the corresponding reader frame. The FWI can take values from 0 to 14.

***pFsdi**: Frame Size for proximity coupling Device Integer. It contains information about the maximal number of bytes that the reader is able to receive in a single frame. It can take values from 0 to 8. The corresponding maximum frame size is obtained according to Table 1.

***pFsci**: Frame Size for proximity Card Integer. It contains information about the maximal number of bytes that the card is able to receive in a single frame. It can take values from 0 to 8. The corresponding maximum frame size is obtained according to Table 1.

The values returned by the function can be:

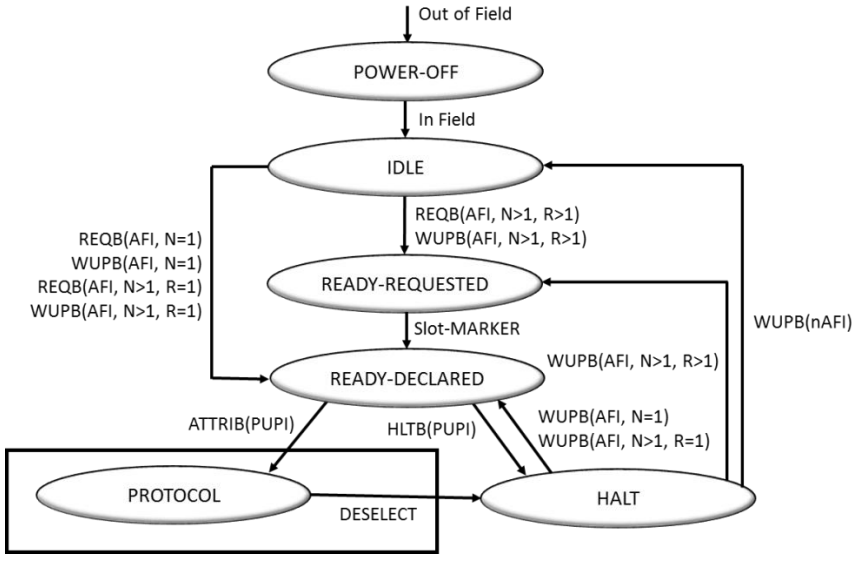
PH_ERR_SUCCESS: Operation successful.

Other: Value returned by the underlying component.

4.1.3 ISO/IEC 14443-3B

In this part of the standard, the initialization of a Type B communication is described. This initialization begins with an anticollision procedure in which the reader IC asks for the UID of the different cards under its RF field at different randomly assigned time slots. Once the reader IC has received the card's UID, it can select it and begin with the higher layer protocol communication, or it can send it to the HALT state.

The state diagram of the Type B card initialization is shown below:



Type B proximity card state diagram

When the card is out of the RF field generated by the reader IC, it is in the POWER-OFF state. As soon as it is powered, it enters the IDLE state, where it waits for REQB or

WUPB commands with an adequate AFI (Application Family Identifier, allows the reader IC to wake up only the cards from a certain application or family of applications). This command includes an N parameter indicating the number of time slots in the anticollision procedure. The card generates a random R number indicating the time slot at which the card will answer. If this number is 1, the card enters the READY-DECLARED state and sends an ATQB frame to the reader IC that includes its PUPI (a random identifier) and other protocol information. If this is not the case, it enters the READY-REQUESTED state. In this state, it waits for the reader IC to send a slot-MARKER with its R number, to enter the READY-DECLARED state and send an ATQB frame.

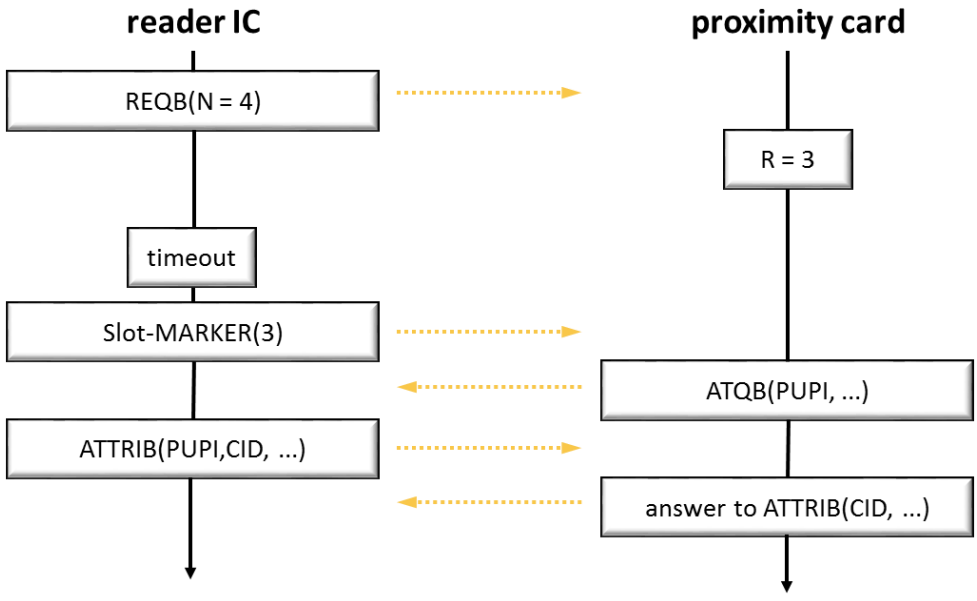
If the reader IC wants to select a card in the READY-DECLARED state, it shall send an ATTRIB command with its PUPI on it. This command may include, besides other protocol information, information from the higher layer application as well. When receiving this command, the card responds with an answer to ATTRIB command, which may also contain information from the higher layer application. Once this frame is received by the reader IC, the exchange of information from the higher layer starts.

The reader IC can also send a card in the READY-DECLARED state to the HALT state through a HLTB command. This state is similar to the IDLE state, with the difference that in this state the REQB command is ignored.

As we said before, the ATQB frame contains some protocol information about the card. The most important parameter included is the PUPI, a 4-byte value that identifies the card. It also informs about the bit rates at which the card is able to work (106, 212, 424 or 848 Kbit/s), the frame waiting time (time within the card shall start the response frame after the end of the corresponding reader IC frame), the maximum frame size that it admits on reception, its compliancy with the ISO/IEC 14443-4 part of the standard, or if it supports CID or NAD addressing (The CID is a logical address that allows the reader IC to keep up to 15 cards activated at the same time, and to communicate with them without the need of sending the others to the HALT state).

On the other hand, the ATTRIB command selects the card through its PUPI, and includes some protocol information about the reader IC. This information includes the bit rates, at which the rest of the communication will be performed, the maximum frame size that the reader IC admits on reception, the CID assigned to the card, and some information about the frame timing.

An example of the Type B card initialization is shown below:



ISO/IEC 14443-3B Operation Flow

As we said before, the value of R is randomly chosen by the card, and shall always be between 1 and N. As we can see in the figure above, the Slot-Marker commands do not need to be called sequentially with incremental slot numbers.

The NFC Reader Library implementation of the ISO/IEC 14443-3B part is described below.

4.1.3.1 ISO/IEC 14443-3B Data Parameter Structure

A special structure has been defined in the NFC Reader Library in order to store the parameters related to the ISO/IEC 14443-3B standard. This structure has been called `phpalI14443p3b_Sw_DataParams_t`.

```
typedef struct{
    void  * pHalDataParams;
    uint8_t bExtAtqb;
    uint8_t pPupi[4];
    uint8_t pPupiValid;
    uint8_t bCidSupported;
    uint8_t bNadSupported;
    uint8_t bCid;
    uint8_t bFwi;
    uint8_t bFsci;
    uint8_t bFsdi;
    uint8_t bDri;
    uint8_t bDsi;
    uint8_t bAttribParam1;
    uint8_t *pHigherLayerInf;
    uint8_t wHigherLayerInfLen;
    uint8_t *pHigherLayerResp;
    uint8_t wHigherLayerRespSize;
    uint8_t wHigherLayerRespLen;
} phpalI14443p3b_Sw_DataParams_t;
```

***pHalDataParams**: Pointer to the underlying HAL parameter structure.

bExtAtqb: Flag indicating whether the last ATQB received has the extended format.

pPupi[4]: Array containing the PUPI of the card.

pPupiValid: Flag indicating whether the stored PUPI is valid or not (0 if it is not valid).

bCidSupported: CID support flag. If it is non-zero, it means that the CID is enabled

bNadSupported: NAD support flag. If it is non-zero, it means that the NAD is enabled.

bCid: Card Identifier value. It is ignored if bCidSupported is zero. The possible values it can take are those in the range from 0 to 14. When a new CID is required, the allocation function phCidManager_GetFreeCid() should be called. When a card is deselected, the CID assigned to that card shall be released using the phCidManager_FreeCid() function.

bFwi: Frame Waiting time Integer. Byte code that determines the Frame Waiting Time: It is the time in which the card shall start the response frame after the end of the corresponding reader frame. The FWT is calculated by the following formula:

$$FWT = (256 \times 16 / 13.56MHz) \times 2^{bFwi}$$

The FWI can take values from 0 to 14. The reader waits for a response for a time of FWT + 60us. If no answer has been received within that time, the reader drops the communication with the card.

bFsci: Frame Size for proximity Card Integer. It contains information about the maximal number of bytes that the card is able to receive in a single frame. It can take values from 0 to 8. The corresponding maximum frame size is obtained according to the table below:

Table 3. bFsdi (bFsci) to FSD (FSC) conversion

bFsdi	0	1	2	3	4	5	6	7	8
FSD (bytes)	16	24	32	40	48	64	96	128	256

bFsdi: Frame Size for proximity coupling Device (Reader) Integer. It contains information about the maximal number of bytes that the reader is able to receive in a single frame. It can take values from 0 to 8. The corresponding maximum frame size is obtained according to the table above.

bDri: Divisor for the reader to card communication. This parameter acts as a divisor when referring to the bit duration. When referring to the bit rate, it behaves as a multiplier. There are 4 legal values (listed in Table 2) defined for this parameter, each representing a different bit rate.

bDsi: Divisor for the card to reader communication. This parameter acts as a divisor when referring to the bit duration. When referring to the bit rate, it behaves as a multiplier. There are 4 legal values (listed in Table 2) defined for this parameter, each representing a different bit rate.

bAttribParam1: Raw Param1 byte of the ATTRIB command. This byte contains information about the frame timing between the reader and the card. No API function has been implemented in the NFC Reader Library to parse this byte.

***pHigherLayerInf**: Pointer to the higher layer information to be sent in the ATTRIB command.

wHigherLayerInfLen: Length of the higher layer information to be sent in the ATTRIB command.

***pHigherLayerResp**: Pointer to the higher layer response received in the answer to ATTRIB command.

wHigherLayerRespSize: Size of the buffer reserved for the higher layer response from the answer to ATTRIB command.

wHigherLayerRespLen: Length of the higher layer response received in the answer to ATTRIB command.

4.1.3.2 Initialization ISO/IEC 14443-3B Parameter Component

This function initializes the phpalI14443p4b_Sw_DataParams_t parameter component.

```
phStatus_t phpalI14443p3b_Sw_Init(
    phpalI14443p3b_Sw_DataParams_t * pDataParams,        [In]
    uint16_t wSizeOfDataParams,                          [In]
    void * pHalDataParams );                             [In]
```

***pDataParams:** Pointer to the `phpalI14443p4b_Sw_DataParams_t` parameter component.

wSizeOfDataParams: Specifies the size of the data parameter structure. It is recommended to pass sizeof `phpalI14443p4b_Sw_DataParams_t`).

***pHalDataParams:** Pointer to the corresponding underlying HAL parameter component.

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

PH_ERR_INVALID_DATA_PARAMS: `wSizeOfDataParams` does not match with the defined size of the PAL phpalI14443p3a_Sw_DataParams_t structure.

4.1.3.3 Get ISO/IEC 14443-3B Parameters

This function returns the main attributes of a phpalI14443p3b_Sw_DataParams_t parameter component.

```
phStatus_t phpalI14443p3b_GetProtocolParams(
    void * pDataParams,                  [In]
    uint8_t * pCidEnabled,               [Out]
    uint8_t * pCid,                      [Out]
    uint8_t * pNadSupported,             [Out]
    uint8_t * pFwi,                      [Out]
    uint8_t * pFsdi,                     [Out]
    uint8_t * pFsci );                   [Out]
```

***pDataParams**: Pointer to the phpalI14443p3b_Sw_DataParams_t parameter component.

***pCidEnabled**: Pointer to the CID enable flag. If it is non-zero, it means that the CID is enabled.

***pCid**: Pointer to the Card Identifier. The possible values it can take are those in the range from 0 to 14.

***pNadSupported**: Pointer to the NAD support flag. If it is non-zero, it means that the NAD is enabled .

***pFwi**: Frame Waiting time Integer. Byte code that determines the Frame Waiting Time: the time within which the card shall start the response frame after the end of the corresponding reader frame. The FWI can take values from 0 to 14.

***pFsdi**: Frame Size for proximity coupling Device Integer. It contains information about the maximal number of bytes that the reader is able to receive in a single frame. It can take values from 0 to 8. The corresponding maximum frame size is obtained according to Table 1.

***pFsci**: Frame Size for proximity Card Integer. It contains information about the maximal number of bytes that the card is able to receive in a single frame. It can take values from 0 to 8. The corresponding maximum frame size is obtained according to Table 1.

The values returned by the function can be:

`PH_ERR_SUCCESS`: Operation successful.

4.1.3.4 Set Config ISO/IEC 14443-3B

This function is used to set the value of a certain attribute from the phpalI14443p3b_Sw_DataParams_t data parameter component. Currently, only the value of the bAttribParam1 attribute can be set. This attribute shall be set before executing the phpalI14443p3b_Attrib function (see 4.1.3.12).

```
phStatus_t phpalI14443p3b_SetConfig(
    void * pDataParams,          [In]
    uint16_t wConfig,            [In]
    uint16_t wValue );           [In]
```

***pDataParams**: Pointer to the phpalI14443p3b_Sw_DataParams_t parameter component.

wConfig: Configuration identifier that represents the desired attribute to be set. Currently, the only admitted value is PHPAL_I14443P3B_CONFIG_ATTRIB_PARAM1.

wValue: Configuration value to which the wConfig identifier shall be set.

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

PH_ERR_PROTOCOL_ERROR: Invalid response received.

Other: Value returned by the underlying component.

4.1.3.5 Get Config ISO/IEC 14443-3B

This function is used to get the value of a certain attribute from the phpalI14443p3b_Sw_DataParams_t data parameter component. Currently, only the value of the bAttribParam1 attribute can be acquired.

```
phStatus_t phpalI14443p3b_GetConfig(
    void * pDataParams,          [In]
    uint16_t wConfig,            [In]
    uint16_t * pValue );         [In]
```

***pDataParams**: Pointer to the phpalI14443p3b_Sw_DataParams_t parameter component.

wConfig: Configuration identifier that represents the desired attribute to be acquired. Currently, the only admitted value is PHPAL_I14443P3B_CONFIG_ATTRIB_PARAM1.

***pValue**: Pointer to the variable where the value of the attribute is returned.

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

PH_ERR_PROTOCOL_ERROR: Invalid response received.

Other: Value returned by the underlying component.

4.1.3.6 Set Higher Layer Inf ISO/IEC 14443-3B

This function is used to set the higher layer information field of the ATTRIB command. This attribute shall be set before executing the phpalI14443p3b_Attrib function (see 4.1.3.12).

```
phStatus_t phpalI14443p3b_SetHigherLayerInf(
    void * pDataParams,                 [In]
    uint16_t * pTxBuffer,               [In]
    uint16_t wTxLength,                 [In]
    uint16_t * pRxBuffer,               [In]
    uint16_t wRxBufSize );              [In]
```

***pDataParams**: Pointer to the phpalI14443p3b_Sw_DataParams_t parameter component.

***pTxBuffer**: Pointer to the higher layer information to be sent in the ATTRIB command. It can be NULL if wTxLength is 0.

wTxLength: Length of the higher layer information to be sent in the ATTRIB command. It can be 0.

***pRxBuffer**: Pointer to the buffer reserved for the higher layer response from the answer to ATTRIB command.

wRxLength: Size of the buffer reserved for the higher layer response from the answer to ATTRIB command.

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

PH_ERR_USE_CONDITION: Feature not available.

4.1.3.7 Get Higher Layer Resp ISO/IEC 14443-3B

This function is used to get the higher layer response field from the answer to ATTRIB command received.

```
phStatus_t phpalI14443p3b_GetHigherlayerResp(
    void * pDataParams,                 [In]
    uint16_t ** ppRxBuffer,             [Out]
    uint16_t * pRxLength );             [Out]
```

***pDataParams**: Pointer to the phpalI14443p3b_Sw_DataParams_t parameter component.

****ppRxBuffer**: Pointer to the higher layer response received in the answer to ATTRIB command.

***pRxLength**: Pointer to the length of the higher layer response received in the answer to ATTRIB command.

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

PH_ERR_USE_CONDITION: Feature not available.

4.1.3.8 Activate Card

This function performs the whole Type B card initialization process , from its waking up with a REQB or a WUPB command, until its selection through an ATTRIB command, including the anticollision procedure, as specified in ISO/IEC 14443-3B.

If no PUPI is provided to the function, a REQB command is performed. If the PUPI is given, a WUPB command is sent. If the activation is successful, the PUPI of the activated card is acquired. Even when there are more cards present in the reader IC RF field, the activation function ensures that just one card is selected.

```
phStatus_t phpalI14443p3b_ActivateCard(
    void * pDataParams,                 [In]
    uint8_t * pPupi,                    [In]
    uint8_t bPupiLength,                [In]
    uint8_t bNumSlots,                  [In]
    uint8_t bAfi,                       [In]
    uint8_t bExtAtqb,                   [In]
    uint8_t bFsdi,                      [In]
    uint8_t bCid,                       [In]
    uint8_t bDri,                       [In]
    uint8_t bDsi,                       [In]
    uint8_t * pAtqb,                    [Out]
    uint8_t * pAtqbLen,                 [Out]
    uint8_t * pMbli,                    [Out]
    uint8_t * pMoreCardsAvailable );    [Out]
```

***pDataParams:** Pointer to the phpalI14443p3b_Sw_DataParams_t parameter component.

***pPupi:** Pointer to the PUPI of the card to be activated.

pPupiLength: Length of the given PUPI. It can take the values 0 and 4:

0 – means that the PUPI is unknown. Therefore, the function begins with a REQB command. At the end of the function the PUPI of the card is captured.

4 – the function begins with a WUPB command.

bNumSlots: Parameter N from the REQB or ATQB command. It contains information about the number of slots in the anticollision procedure. The number of slots is calculated as Num_of_slots = 2^N. The bNumSlots parameter can take values from 0 to 4.

bAfi: Application Family Identifier. To ignore this field, it shall be set to 0.

bExtAtqb: Flag to enable the extended format of the ATQB command.

bFsdi: Frame Size for proximity coupling Device Integer. It contains information about the maximal number of bytes that the reader IC is able to receive in a single frame. It can take values from 0 to 8. The corresponding maximum frame size is obtained according to Table 1.

bCid: Card Identifier.

bDri: Divisor for the reader to card communication. This parameter acts as a divisor when referring to the bit duration. When referring to the bit rate, it behaves as a multiplier. There are 4 legal values (listed in Table 2) defined for this parameter, each representing a different bit rate.

bDsi: Divisor for the card to reader communication. This parameter acts as a divisor when referring to the bit duration. When referring to the bit rate, it behaves as a multiplier. There are 4 legal values (listed in Table 2) defined for this parameter, each representing a different bit rate.

***pAtqb**: Pointer to the buffer where the received ATQB response is stored. Its content is parsed by the function and stored in the data parameter component.

***pAtqbLen**: Pointer to the length of the ATQB response received.

***pMbli**: Pointer to the Maximum Buffer Length Index. It contains information about the Maximum Buffer Length of the card.

***pMoreCardsAvailable:** Indicates whether one or more cards are under the Reader RF field at the same time.

PH_ON: More cards available. A collision occurred.

PH_OFF: Just one card under reader IC RF field.

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

PH_ERR_PROTOCOL_ERROR: Invalid response received.

Other: Value returned by the underlying component.

4.1.3.9 Request B

This function transmits a request Type B command (REQB) and waits to receive an answer to that request (ATQB), or for a timeout, which would mean that none of the cards had chosen the first slot in the anticollision procedure. The data rate is automatically set to 106 kbit/s for both receiver and transmitter.

```
phStatus_t phpalI14443p3b_RequestB(
    void * pDataParams,                 [In]
    uint8_t bNumSlots,                  [In]
    uint8_t bAfi,                       [In]
    uint8_t bExtAtqb,                   [In]
    uint8_t * pAtqb,                    [Out]
    uint8_t * pAtqbLen );               [Out]
```

***pDataParams:** Pointer to the phpalI14443p3b_Sw_DataParams_t parameter component.

bNumSlots: Parameter N from the REQB or ATQB command. It contains information about the number of slots in the anticollision procedure. The number of slots is calculated as Num_of_slots = 2^N. The bNumSlots parameter can take values from 0 to 4.

bAfi: Application Family Identifier. To ignore this field, it shall be set to 0.

bExtAtqb: Flag to enable the extended format of the ATQB command.

***pAtqb**: Pointer to the buffer where the received ATQB response is stored. Its content is parsed by the function and stored in the data parameter component.

***pAtqbLen**: Pointer to the length of the ATQB response received.

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

PH_ERR_PROTOCOL_ERROR: Invalid response received.

Other: Value returned by the underlying component.

4.1.3.10 Wake Up B

This function activates a card that was pushed to the HALT state previously.

```
phStatus_t phpalI14443p3b_WakeUpB(
    void * pDataParams,                 [In]
    uint8_t bNumSlots,                  [In]
    uint8_t bAfi,                       [In]
    uint8_t bExtAtqb,                   [In]
    uint8_t * pAtqb,                    [Out]
    uint8_t * pAtqbLen );               [Out]
```

***pDataParams:** Pointer to the phpalI14443p3b_Sw_DataParams_t parameter component.

bNumSlots: Parameter N from the REQB or ATQB command. It contains information about the number of slots in the anticollision procedure. The number of slots is calculated as Num_of_slots = 2^N. The bNumSlots parameter can take values from 0 to 4.

bAfi: Application Family Identifier. To ignore this field, it shall be set to 0.

bExtAtqb: Flag to enable the extended format of the ATQB command.

***pAtqb**: Pointer to the buffer where the received ATQB response is stored. Its content is parsed by the function and stored in the data parameter component.

***pAtqbLen**: Pointer to the length of the ATQB response received.

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

PH_ERR_PROTOCOL_ERROR: Invalid response received.

Other: Value returned by the underlying component.

4.1.3.11 Slot Marker

This function sends a Slot-MARKER command and stores the ATQB frame received.

```
phStatus_t phpalI14443p3b_SlotMarker(
    void * pDataParams,                 [In]
    uint8_t bSlotNumber,                [In]
    uint8_t * pAtqb,                    [Out]
    uint8_t * pAtqbLen );               [Out]
```

***pDataParams:** Pointer to the phpalI14443p3b_Sw_DataParams_t parameter component.

bSlotNumber: Slot number . The possible values are those in the range from 1 to 15 (which represent the slots from 2 to 16).

***pAtqb**: Pointer to the buffer where the received ATQB response is stored. Its content is parsed by the function and stored in the data parameter component.

***pAtqbLen**: Pointer to the length of the ATQB response received.

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

PH_ERR_PROTOCOL_ERROR: Invalid response received.

Other: Value returned by the underlying component.

4.1.3.12 Attrib

This function sends an ATTRIB command to the chosen card in order to select this card and begin with the data exchange. It also sets some communication parameters, such as the bit rates.

If the developer wants to use frame timing parameters different to those by default, they shall set the bAttribParam1 attribute from the parameter component to the proper value, as defined in ISO/IEC 14443-3B. This can be done through the phpalI14443p3b_SetConfig function (see 4.1.3.4). Similarly, the higher layer information field can be set to a certain value through the phpalI14443p3b_SetHigherLayerInf() function (see 4.1.3.6).

```
phStatus_t phpalI14443p3b_Attrib(
    void * pDataParams,              [In]
    uint8_t * pAtqb,                 [In]
    uint8_t bAtqbLen,                [In]
    uint8_t bFsdi,                   [In]
    uint8_t bCid,                    [In]
    uint8_t bDri,                    [In]
    uint8_t bDsi,                    [In]
    uint8_t * pMbli );               [Out]
```

***pDataParams:** Pointer to the phpalI14443p3b_Sw_DataParams_t parameter component.

***pAtqb**: Pointer to the buffer where the received ATQB response has been stored.

bAtqbLen: Length of the ATQB response received.

bFsdi: Frame Size for proximity coupling Device Integer. It contains information about the maximal number of bytes that the reader IC is able to receive in a single frame. It can take values from 0 to 8. The corresponding maximum frame size is obtained according to Table 1.

bCid: Card Identifier.

bDri: Divisor for the reader to card communication. This parameter acts as a divisor when referring to the bit duration. When referring to the bit rate, it behaves as a multiplier. There are 4 legal values (listed in Table 2) defined for this parameter, each representing a different bit rate.

bDsi: Divisor for the card to reader communication. This parameter acts as a divisor when referring to the bit duration. When referring to the bit rate, it behaves as a multiplier. There are 4 legal values (listed in Table 2) defined for this parameter, each representing a different bit rate.

***pMbli**: Pointer to the Maximum Buffer Length Index. It contains information about the Maximum Buffer Length of the card.

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

PH_ERR_PROTOCOL_ERROR: Invalid response received.

Other: Value returned by the underlying component.

4.1.3.13 Halt B

After a card has been sent to the READY-DECLARED state, the reader IC can make the card enter the HALT state. The card can be later reactivated through a WUPB command, or using the phpalI14443p3b_ActivateCard() function.

```
phStatus_t phpalI14443p3a_HaltB(
    void * pDataParams );                    [In]
```

***pDataParams:** Pointer to the phpalI14443p3b_Sw_DataParams_t parameter component.

The values returned by the function can be:

PH_ERR_SUCCESS: The card has entered the HALT state successfully.

PH_ERR_PROTOCOL_ERROR: Invalid response received.

Other: Value returned by the underlying component.

4.1.3.14 Exchange

Most of the ISO/IEC 14443-3A related functions are based on a half-duplex bidirectional communication between the reader IC and the card, in which the reader IC sends a command and waits for a response from the card. This function gives the possibility to the developer to send an array of bytes to the card and read the corresponding response.

```
phStatus_t phpalI14443p3b_Exchange(
    void * pDataParams,                      [In]
    uint16_t wOption,                        [In]
    uint8_t * pTxBuffer,                     [In]
    uint16_t wTxLength,                      [In]
    uint8_t ** ppRxBuffer,                   [Out]
    uint16_t * pRxLength );                  [Out]
```

***pDataParams:** Pointer to the phpalI14443p3b_Sw_DataParams_t parameter component.

wOption: All ISO/IEC 14443-3 functions pass the value EXCHANGE_DEFAULT as the default parameter.

***pTxBuffer:** Pointer to the array of data to be transmitted. This array actually contains is the reader IC command defined by its byte code and the corresponding data.

wTxLength: Number of bytes to be transmitted.

****ppRxBuffer:** Pointer to the received array of data.

***pRxLength:** Pointer to the address where the information about the received data is.

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

Other: Value returned by the underlying component.

4.1.4 ISO/IEC 14443-4

This part of the standard provides a means for the higher layer protocols of the card and the reader to communicate one with each other, once the communication has been initialized. All the block formats and the operations described are common for both Type A and Type B communication, so the higher layer applications do not need to know what protocols are being used underneath.

This part of the standard, besides describing the information blocks which allow higher layer applications to exchange data, it defines other control blocks that allow the protocol to perform other tasks such as confirming the correct reception of a frame, checking the presence of a card, requesting an extension of the frame waiting time, or finishing the communication and sending the card to the HALT state.

Furthermore, it includes other functionalities such as multi-activation (the reader is able to hold several cards in the ACTIVE state simultaneously thanks to the CID field), chaining (the reader and card are able to transmit information longer than the frame sizes defined, by dividing it into several blocks), or power level indication (this functionality has not been implemented in the Reader Library).

For further details on the ISO/IEC 14443-4, please refer to the standard [18].

The NFC Reader Library implementation of the ISO/IEC 14443-4 is described below.

4.1.4.1 ISO/IEC 14443-4 Data Parameter Structure

A special structure has been defined in the NFC Reader Library in order to store the parameters related to the ISO/IEC 14443-4 standard. This structure has been called phpalI14443p4a_Sw_DataParams_t. Besides the parameters defined in the standard, there are also some state variables included in the structure for internal management and advanced handling of the functions of the module. The values of the attributes can be modified using the `phpalI14443p4_SetConfig()` function, or reset using the `phpalI14443p4_ResetProtocol()` function (see 4.1.4.5 or 4.1.4.3 respectively).

```
typedef struct{
    void  * pHalDataParams;
    uint8_t   bStateNow;
    uint8_t   bCidEnabled;
    uint8_t   bCid;
    uint8_t   bNadEnabled;
    uint8_t   bNad;
    uint8_t   bFwi;
    uint8_t   bFsdi;
    uint8_t   bFsci;
    uint8_t   bPcbBlockNum;
    uint8_t   bMaxRetryCount;
} phpalI14443p4_Sw_DataParams_t;
```

***pHalDataParams**: Pointer to the underlying HAL layer data parameter component. This attribute can only be assigned by phpalI14443p4_Sw_Init() (see 4.1.4.2).

bCidEnabled: Pointer to the CID enabling flag. If it is non-zero, it means that the CID is enabled.

bCid: 4 bit Card Identifier. The possible values it can take are those in the range from 0 to 14. Ignored if bCidSupported is zero.

bNadEnabled: Node Address enable flag. Nonzero means NAD is enabled.

bNad: Node Address. Ignored if bNadEnabled is zero.

bFwi: Frame Waiting time Integer. Byte code that determines the Frame Waiting Time: the time within which the card shall start the response frame after the end of the corresponding reader frame. The FWT is calculated by the following formula:

$$FWT = (256 \times 16 / 13.56MHz) \times 2^{bFwi}$$

The FWI can take values from 0 to 14. The reader waits for a response for a time of FWT + 60us. If no answer has been received within that time, the reader drops the communication with the card.
The FWI value is obtained at the initialization process from the ATS frame received when using the phpalI14443p4a_Rats() function. The value is automatically stored in the corresponding phpalI14443p4a_Sw_DataParams_t parameter component (see 4.1.2.1). This value needs to be copied in the bFwi attribute of the phpalI14443p4_Sw_DataParams_t parameter component (see 4.1.4.1), so that afterwards phpalI14443p4_Exchange() and phpalI14443p4_PressCheck() functions are appropriately performed.

bFsdi: Frame Size for proximity coupling Device Integer. It contains information about the maximal number of bytes that the reader is able to receive in a single frame. It can take values from 0 to 8. The corresponding maximum frame size is obtained according to Table 1. The actual limit of the information field is smaller due to the prologue and epilogue fields of the block.

bFsci: Frame Size for proximity Card Integer. It contains information about the maximal number of bytes that the card is able to receive in a single frame. It can take values from 0 to 8. The corresponding maximum frame size is obtained according to Table 1. The actual limit of the information field is smaller due to the prologue and epilogue fields of the block.

bMaxRetryCount: When executing the phpalI14443p4_Exchange() or phpalI14443p4_Deselect() functions a violation of the ISO/IEC 14443-4 protocol occurs, a retransmission is performed. The number of retransmissions is restricted by this attribute.

bStateNow: Current exchange data state. This is a state variable necessary for the internal management, and should not be modified by the developer.

bPcbBlockNum: Block number of the current information block. This parameter is just for internal management, and should not be modified by the developer, although the phpalI14443p4_SetConfig() function (see 4.1.4.5) provides this option.

4.1.4.2 Init ISO/IEC 14443-4 Parameter Component

This function fills a given data parameter component with its default values.

```
phStatus_t phpalI14443p4_Sw_Init(
    phpalI14443p4_Sw_DataParams_t * pDataParams,          [In]
```

uint16_t wSizeOfDataParams, [In]
void * pHalDataParams); [In]

*__pDataParams__: Pointer to the phpalI14443p4_Sw_DataParams_t parameter component.

wSizeOfDataParams: Specifies the size of the data parameter structure. It is recommended to pass sizeof(phpalI14443p4_Sw_DataParams_t).

***pHalDataParams:** Pointer to the underlying HAL layer data parameter component, depending on the used reader.

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

PH_ERR_INVALID_DATA_PARAMS: wSizeOfDataParams does not match with the defined size of the PAL phpalI14443p4_Sw_DataParams_t structure.

4.1.4.3 Reset Protocol ISO/IEC 14443-4

This function sets all the attributes from the phpalI14443p4_Sw_DataParams_t parameter component to zero, or to their initial or default values as defined in the ISO/IEC 14443-4 standard (see the table below).

phStatus_t phpalI14443p4_ResetProtocol(
void * pDataParams); [In]

*__pDataParams__: Pointer to the phpalI14443p4_Sw_DataParams_t parameter component.

Table 4. Values after reset

attribute	value	Meaning
bCidEnabled	NULL	not enabled
bCid	NULL	0
bNadEnabled	NULL	not enabled
bNad	NULL	0
bFwi	4	4.8ms
bFsdi	0	16 bytes
bFsci	2	32 bytes
bMaxRetryCount	2	2

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

4.1.4.4 Set Protocol ISO/IEC 14443-4

This function sets most of the attributes from the phpalI14443p4_Sw_DataParams_t parameter component. Some internal parameters such as state variables stay untouched.

Once the initialization of the communication has been done (and after the phpalI14443p4a_Rats() function has been executed see 4.1.2.4), it is recommended to pass the output values of the phpalI14443p4a_GetProtocolParams() function (see 4.1.2.6) as the input parameters to this function.

If the developer needs to set just one attribute he should consider using the phpalI14443p4_SetConfig() function (see section 4.1.4.5).

```
phStatus_t phpalI14443p4_SetProtocol(
    void * pDataParams,                [In]
    uint8_t bCidEnable,                [In]
    uint8_t bCid,                      [In]
    uint8_t bNadEnable,                [In]
    uint8_t bNad,                      [In]
    uint8_t bFwi,                      [In]
    uint8_t bFsdi,                     [In]
    uint8_t bFsci );                   [In]
```

***pDataParams**: Pointer to the phpalI14443p4_Sw_DataParams_t parameter component.

bCidEnable: Pointer to the CID enabling flag. If it is non-zero, it means that the CID is enabled.

bCid: 4 bit Card Identifier. The possible values it can take are those in the range from 0 to 14. Ignored if bCidSupported is zero.

bNadEnable: Node Address enable flag. Nonzero means CID is enabled.

bNad: Node Address. Ignored if bNadEnabled is zero.

bFwi: Frame Waiting time Integer. Byte code that determines the Frame Waiting Time: the time within which the card shall start the response frame after the end of the corresponding reader frame. The FWT is calculated by the following formula:

$$FWT = (256 \times 16 / 13.56MHz) \times 2^{bFwi}$$

The FWI can take values from 0 to 14. The reader waits for a response for a time of FWT + 60us. If no answer has been received within that time, the reader drops the communication with the card.

The FWI value is obtained at the initialization process from the ATS frame received when using the phpalI14443p4a_Rats() function. The value is automatically stored in the corresponding phpalI14443p4a_Sw_DataParams_t parameter component (see 4.2.1). This value needs to be copied in the bFwi attribute of the phpalI14443p4_Sw_DataParams_t parameter component (see 4.1.4.1).

bFsdi Frame Size for proximity coupling Device Integer. It contains information about the maximal number of bytes that the reader is able to receive in a single frame. It can take values from 0 to 8. The corresponding maximum frame size is obtained according to Table 1.

bFsci: Frame Size for proximity Card Integer. It contains information about the maximal number of bytes that the card is able to receive in a single frame. It can take values from 0 to 8. The corresponding maximum frame size is obtained according to Table 1.

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

PH_ERR_INVALID_PARAMETER: At least one of the input attributes (among bCid, bFwi, bFsdi and bFsci) has an invalid value.

4.1.4.5 Set Config ISO/IEC 14443-4

This function is used to set a single attribute from the phpalI14443p4_Sw_DataParams_t data parameter component. If the developer needs to set more than one attribute at the same time he should consider using the phpalI14443p4_SetProtocol() function (see section 4.1.4.4).

```
phStatus_t phpalI14443p4_SetConfig(
    void * pDataParams,          [In]
    uint16_t wConfig,            [In]
    uint16_t wValue );           [In]
```

***pDataParams**: Pointer to the phpalI14443p4_Sw_DataParams_t parameter component.

wConfig: Configuration identifier that represents the desired attribute to be set (see the second column of Table 5).

wValue: Configuration value that the chosen attribute shall be set to (see the third column of Table 5).

Table 5. Identifiers of attributes of `phpalI14443p4_Sw_DataParams_t`

The first column contains the attributes of the `phpalI14443p4_Sw_DataParams_t` parameter structure as presented in section 4.1.4.1.
In the second column the identifiers used as the second input argument to point the attribute that shall be set (or got) are shown
In the third column the legal values that can be assigned to each particular parameter are indicated

Attribute	configuration identifier	configuration value
bPcbBlockNum	PHPAL_I14443P4_CONFIG_BLOCKNO	Only 0 or 1 are legal values
bCidEnabled, bCid	PHPAL_I14443P4_CONFIG_CID	CID enabler << 8 | CID; CID from 0 to 14
bNadEnabled, bNad	PHPAL_I14443P4_CONFIG_NAD	NAD enabler << 8 | NAD
bFwi	PHPAL_I14443P4_CONFIG_FWI	From 0 to 14
bFsdi, bFsci	PHPAL_I14443P4_CONFIG_FSI	FSDI << 8 | FSCI
bMaxRetryCount	PHPAL_I14443P4_CONFIG_MAXRETRYCOUNT	from 0 to 255

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

PH_ERR_INVALID_PARAMETER: The given configuration value is illegal for the parameter indicated.

PH_ERR_UNSUPPORTED_PARAMETER: The given configuration is unknown.

4.1.4.6 Get Config ISO/IEC 14443-4

This function is used to get the value of a certain attribute from the phpalI14443p4_Sw_DataParams_t data parameter component.

```
phStatus_t phpalI14443p4_GetConfig(
    void * pDataParams,          [In]
```

```
    uint16_t wConfig,           [In]
    uint16_t *wValue );         [In]
```

***pDataParams**: Pointer to the phpalI14443p4_Sw_DataParams_t parameter component.

wConfig: Configuration identifier that represents the desired attribute to be set (see the second column of Table 5).

***wValue**: Pointer to the variable where the value of the attribute is returned, coded in the format shown in the third column of Table 5.

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

PH_ERR_UNSUPPORTED_PARAMETER: The given configuration is unknown.

4.1.4.7 Exchange

This is the function responsible for the exchange of the information of the upper layer applications. It is able to handle all the events related to the ISO/IEC 14443-4 data communication:

- Encapsulating the data for transmission into one or more I-blocks (depending on the value of FSC and the length of the message).
- Switching the block number.
- Handling the received acknowledgement blocks (both R(ACK) and R(NAK)).
- Handling unanswered frames and frame errors.

The developer just needs to process the data to be transmitted and the data received from the upper layer, no knowledge about the structures of the frames or the operation of the protocol is required.

MIFARE DESFire commands and other products compliant with part 4 of the ISO 14443 standard, should be transferred using this function.

```
phStatus_t phpalI14443p4_Exchange(
    void * pDataParams,             [In]
    uint16_t wOption,               [In]
    uint8_t * pTxBuffer,            [In]
    uint16_t wTxLength,             [In]
    uint8_t ** ppRxBuffer,          [Out]
    uint16_t * pRxLength );         [Out]
```

***pDataParams**: Pointer to the phpalI14443p4_Sw_DataParams_t parameter component.

wOption: Option parameter. The possible values are described in the *src\NxpRdLib_PublicRelease\intfs0\phpalI14443p4.h* file. In the case of a communication with a MIFARE DESFire card, EXCHANGE_DEFAULT option can be used.

***pTxBuffer**: Pointer to the application data to be transmitted.

wTxLength: Length of the application data to be transmitted.

****ppRxBuffer**: Pointer to the buffer where the received application data from the card will be stored.

***pRxLength**: Pointer to the length of the application data received.

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

PHPAL_I14443P4_ERR_RECOVERY_FAILED: Expected frames from the card not received (after the retransmissions).

PH_ERR_SUCCESS_CHAINING: Reception successful, but the receive buffer from the card is full. The function needs to be called once again with wOption = PH_EXCHANGE_RXCHAINING_BUFSIZE.

Other: Value returned by the underlying component.

4.1.4.8 Presence Check

This function performs the presence check procedure as described in ISO/IEC 14443-4. The Reader sends a R(NAK) block to the card and waits for a R(ACK) response from it. If a timeout error occurs, it means that the card is not there anymore. If, instead, the reader receives a R(ACK) block, means that the card is still in its RF field.

```
phStatus_t phpalI14443p4_PresCheck(
    void * pDataParams );                    [In]
```

*__pDataParams__: Pointer to the phpalI14443p4_Sw_DataParams_t parameter component.

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

PH_ERR_PROTOCOL_ERROR: Incomplete block received or non-R(ACK) response received.

Other: Value returned by the underlying component.

4.1.4.9 Deselect

This function is the responsible for deselecting a card that has been previously activated through the phpalI14443p4a_Rats() function (see 4.1.2.4), as described in the ISO/IEC 14443-4 standard.

If a card has been deselected successfully, the exchange and presence check procedures from this part of the protocol will not work with it until it is reactivated, using the phpalI14443p4a_Rats() or phpalI14443p4a_ActivateCard() functions (see 4.1.2.4 or 4.1.2.3 respectively).

If the Deselect request is not answered by an error free Deselect response, it is automatically retransmitted. The number of retransmissions is determined by the bMaxRetryCount parameter (see 4.1.4.1). The developer shall take into account that a successful deselect procedure shall be immediately followed by the execution of the phCidManager_FreeCid() function, in order to release the CID number that had been assigned to the card.

Once the card has been deselected, it enters de HALT state, which means that for waking it up, the phpalI14443p3a_WakeUpA() function (see 4.1.1.5) needs to be executed.

```
phStatus_t phpalI14443p4_Deselect(
    void * pDataParams );                    [In]
```

*__pDataParams__: Pointer to the phpalI14443p4_Sw_DataParams_t parameter component.

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

PH_ERR_PROTOCOL_ERROR: No correct deselect response received when not performing retransmissions (bMaxRetryCount=1).

PHPAL_I14443P4_ERR_RECOVERY_FAILED: No correct deselect response received after multiple deselect attempts.

Other: Value returned by the underlying component.

4.2 MIFARE

4.2.1 Technical Introduction

MIFARE is NXP's well-known brand for a wide range of contactless IC products such as MIFARE Classic, MIFARE Ultralight, MIFARE DESFire and MIFARE Plus. MIFARE technology products are compliant with the international standard ISO/IEC 14443-A (see Section 4.1.1). MIFARE technologies are widely used in more than 40 different applications worldwide: transportation, access control, couponing …

The MIFARE component in the Protocol Abstraction Layer of the NFC Reader Library defines the interface for MIFARE protocols in the Application Layer that are not part of the ISO/IEC 14443 standard (MIFARE Classic, MIFARE Ultralight). These operations include the authentication and the data exchange among others.

4.2.2 Parameter Structure

The MIFARE component parameter structure stores component parameters associated to the layers with which the component interacts.

The MIFARE component takes the ISO/IEC 14443-4 component as input parameter in order to provide support for both ISO/IEC 14443-3 compliant products such as MIFARE Classic and MIFARE Ultralight and ISO/IEC 14443-4 compliant products such as MIFARE Plus and MIFARE DESFire. It also takes the HAL component as input parameter as it relies in the hardware for MIFARE specific operations that cannot be performed by the software such as MIFARE Classic authentication.

```
typedef struct {
    void    * pHalDataParams;
    void    * pPalI14443p4DataParams;
} phpalMifare_Sw_DataParams_t;
```

*** pHalDataParams:** Pointer to the parameter structure of the underlying HAL layer component.

*** pPalI14443p4DataParams:** Pointer to the ISO/IEC 14443-4 parameter structure.

4.2.3 Component Initialization

The MIFARE component is initialized calling the `phpalMifare_Sw_Init()` function. This function takes the underlying HAL parameter component and the ISO/IEC 14443-4 component for the initialization of its parameter structure.

Once the component has successfully been initialized the rest of the MIFARE component API functions can be called.

```
phStatus_t phpalMifare_Sw_Init(
```

```
    phpalMifare_Sw_DataParams_t *pDataParams,      [In]
    uint16_t wSizeOfDataParams,                    [In]
    void  * pHalDataParams,                        [In]
    void  * pPalI14443p4DataParams);               [In]
```

*** pDataParams**: Pointer to the MIFARE parameter component.

wSizeOfDataParams: Size of the `phpalMifare_Sw_DataParams_t parameter component.`

*** pHalDataParams**: Pointer to the underlying HAL parameter component.

*** pPalI14443p4DataParams**: Pointer to the ISO/IEC 14443-4 parameter component.

The values returned by the function can be:

PH_ERR_SUCCESS Operation successful.

Other: Value returned by the underlying component.

4.2.4 MIFARE API

The MIFARE API provides services to upper MIFARE-based application layer components that are not covered by the ISO/IEC 14443 component.

4.2.4.1 ISO/IEC 14443-3 Data Exchange

This command is used for the data exchange between the contactless reader IC and the ISO/IEC 14443-3 compliant MIFARE card. This command should not be directly called by the developer since it is called internally by upper layer MIFARE product components.

When a Read command is executed, the reception buffer should be checked and when a Write command is executed, the transmission buffer should be specified.

```
phStatus_t phpalMifare_ExchangeL3(
    void * pDataParams,                            [In]
    uint16_t wOption,                              [In]
    uint8_t * pTxBuffer,                           [In]
    uint16_t wTxLength,                            [In]
    uint8_t ** ppRxBuffer,                         [Out]
    uint16_t * pRxLength );                        [Out]
```

*** pDataParams:** `Pointer to the MIFARE parameter component phpalMifare_Sw_DataParams_t.`

wOption: It indicates in which part of the stream is located the given data: in the first part (`PH_EXCHANGE_BUFFER_FIRST`), in the middle part (`PH_EXCHANGE_BUFFER_CONT`) or in the last part (`PH_EXCHANGE_BUFFER_LAST`).

*** pTxBuffer:** Data to be transmitted.

wTxLength: Length of the data to be transmitted.

**** ppRxBuffer:** Pointer to the received data.

*** pRxLength**: Length of the received data.

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

Other: Value returned by the underlying component.

4.2.4.2 ISO/IEC 14443-4 Data Exchange

This command is used for the data exchange between the contactless reader IC and the ISO/IEC 14443-4 compliant MIFARE card.

```
phStatus_t phpalMifare_ExchangeL4(
    void * pDataParams,                 [In]
    uint16_t wOption,                   [In]
    uint8_t * pTxBuffer,                [In]
    uint16_t wTxLength,                 [In]
    uint8_t ** ppRxBuffer,              [Out]
    uint16_t * pRxLength );             Out]
```

*** pDataParams:** Pointer to the MIFARE parameter component phpalMifare_Sw_DataParams_t.

wOption: It indicates in which part of the stream is located the given data: in the first part (`PH_EXCHANGE_BUFFER_FIRST`), in the middle part (`PH_EXCHANGE_BUFFER_CONT`) or in the last part (`PH_EXCHANGE_BUFFER_LAST`).

*** pTxBuffer:** Data to be transmitted.

wTxLength: Length of the data to be transmitted.

**** ppRxBuffer:** Pointer to the received data.

*** pRxLength**: Length of the received data.

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

Other: Value returned by the underlying component.

4.2.4.3 MIFARE Proximity Check

The proximity check function is used in order to verify that the MIFARE IC is in close proximity to the contactless reader. This functionality can be used to effectively prevent relay attacks. The proximity check is based on a precise time measurement of challenge-response pairs in combination with cryptographic methods.

This functionality is only available in MIFARE Plus cards configured in security level 3.

```
phStatus_t phpalMifare_ExchangePc(
    void * pDataParams,                 [In]
    uint16_t wOption,                   [In]
    uint8_t * pTxBuffer,                [In]
    uint16_t wTxLength,                 [In]
    uint8_t ** ppRxBuffer,              [Out]
    uint16_t * pRxLength );             [Out]
```

*** pDataParams:** Pointer to the MIFARE parameter component phpalMifare_Sw_DataParams_t.

wOption: It indicates in which part of the stream is located the given data: in the first part (`PH_EXCHANGE_BUFFER_FIRST`), in the middle part (`PH_EXCHANGE_BUFFER_CONT`) or in the last part (`PH_EXCHANGE_BUFFER_LAST`).

*** pTxBuffer:** Data to be transmitted.

wTxLength: Length of the data to be transmitted.

**** ppRxBuffer:** Pointer to the received data.

*** pRxLength**: Length of the received data.

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

Other: Value returned by the underlying component.

4.2.4.4 Set Minimum FDT for Proximity Check

The Frame Delay Time (FDT), is defined as the time between the final pause transmitted by the reader at the end of a message and the leading edge of the modulation pulse for the start bit transmitted by the card.

This function allows the developer to set or reset the minimum FDT time used by the reader for the proximity check functionality. The call to this function sets the FDT configuration of the underlying HAL component.

```
phStatus_t phpalMifare_SetMinFdtPc(
    void * pDataParams,                 [In]
    uint16_t wValue );                  [In]
```

*** pDataParams:** Pointer to the MIFARE parameter component phpalMifare_Sw_DataParams_t.

wValue: Option parameter for setting – 1 – or resetting – 0 – the FDT.

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

PH_ERR_IO_TIMEOUT: Timeout error.

PH_ERR_AUTH_ERROR: Authentication error.

Other: Value returned by the underlying component.

4.2.4.5 MIFARE Exchange Raw

This function is used to exchange raw data with MIFARE cards, which means that no CRC and no parity bytes are exchanged.

```
phStatus_t phpalMifare_ExchangeRaw(
    void * pDataParams,                 [In]
    uint16_t wOption,                   [In]
    uint8_t * pTxBuffer,                [In]
    uint16_t wTxLength,                 [In]
    uint8_t bTxLastBits,                [In]
    uint8_t ** ppRxBuffer,              [Out]
    uint16_t * pRxLength,               [Out]
    uint8_t * pRxLastBits );            [Out]
```

*** pDataParams:** Pointer to the MIFARE parameter component phpalMifare_Sw_DataParams_t.

wOption: It indicates in which part of the stream is located the given data: in the first part (`PH_EXCHANGE_BUFFER_FIRST`), in the middle part (`PH_EXCHANGE_BUFFER_CONT`) or in the last part (`PH_EXCHANGE_BUFFER_LAST`).

*** pTxBuffer:** Data to be transmitted.

wTxLength: Length of the data to be transmitted.

bTxLastBits: Number of valid bits of the last byte received.

**** ppRxBuffer:** Pointer to the received data.

*** pRxLength**: Length of the received data.

bRxLastBits: Number of valid bits of the last byte transmitted.

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

Other: Value returned by the underlying component.

4.2.4.6 MIFARE Classic Authentication with key number

This function allows to complete a MIFARE Classic authentication using key stored in the Key Store module (see 8.1). The key to be used is identified by the key number and key version input parameters.

```
phStatus_t phpalMifare_MfcAuthenticateKeyNo(
    void * pDataParams,                 [In]
    uint8_t bBlockNo,                   [In]
    uint8_t bKeyType,                   [In]
    uint16_t wKeyNo,                    [In]
    uint16_t wKeyVersion,               [In]
    uint8_t * pUid );                   [In]
```

*** pDataParams:** Pointer to the MIFARE parameter component `phpalMifare_Sw_DataParams_t`.

bBlockNo: MIFARE Classic memory block number to authenticate against.

bKeyType: MIFARE Classic key type to use for the authentication: `#PHPAL_MIFARE_KEYA` or `#PHPAL_MIFARE_KEYB`.

wKeyNo: Number identifier of the key to use for the authentication.

wKeyVersion: Version of the key use for the authentication.

*** pUid**: UID of the MIFARE Classic card to authenticate with.

4.2.4.7 MIFARE Classic Authentication with input key

This function allows to complete MIFARE Classic authentication using a key passed as an input parameter to the function.

```
phStatus_t phpalMifare_MfcAuthenticate(
    void * pDataParams,                 [In]
    uint8_t bBlockNo,                   [In]
    uint8_t bKeyType,                   [In]
    uint8_t * pKey,                     [In]
    uint8_t * pUid );                   [In]
```

*** pDataParams:** Pointer to the MIFARE parameter component `phpalMifare_Sw_DataParams_t`.

bBlockNo: MIFARE Classic memory block number to authenticate against.

bKeyType: MIFARE Classic key type to use for the authentication: `#PHPAL_MIFARE_KEYA` or `#PHPAL_MIFARE_KEYB`.

*** Key:** MIFARE Classic key to use for the authentication.

*** pUid**: UID of the MIFARE Classic card to authenticate with.

4.3 FeliCa PAL

4.3.1 Technical Introduction

The FeliCa component in the Protocol Abstraction Layer implements the initialization and the anticollision procedures according to the JIS X 6319-4 specification.

FeliCa cards are identified by an 8-byte identifier called Manufacture ID (IDm). This IDm parameter defines both the Manufacturer Code (2 bytes) and the Card Identification Number (6 bytes). Additionally, FeliCa defines the 8 byte Manufacture Parameter (PMm) value, which defines the maximum response time parameter. This parameter is used by the reader IC to determine the command response timeout for a particular card.

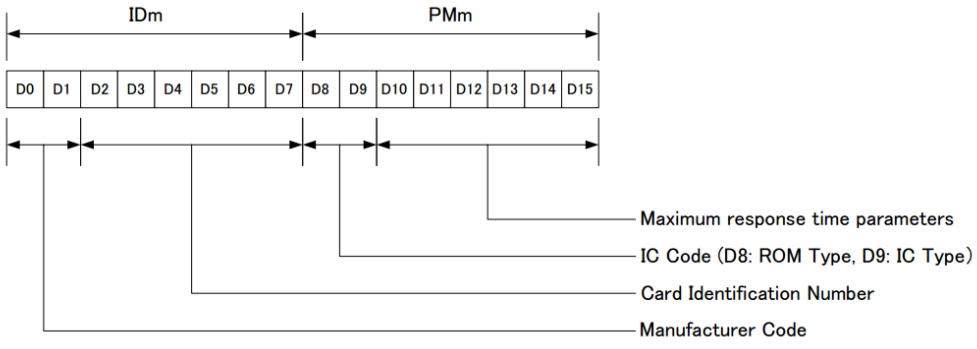
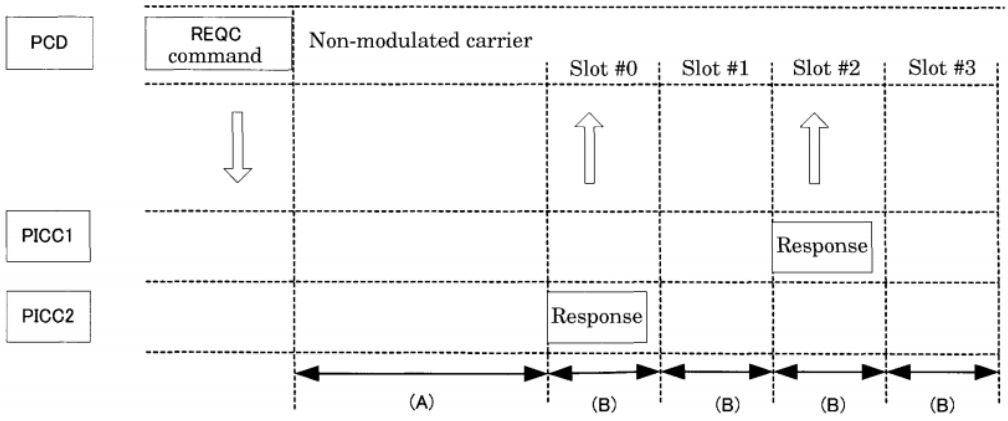


Fig 9. FeliCa IDm and PMm

A contactless reader IC communicating with FeliCa cards periodically sends ReqC commands. FeliCa cards within the reader IC RF field shall respond to this request with an ATQC message indicating the possibility to start the communication.

In order to avoid collisions when more than one card responds at the same time, FeliCa implements a time slot based mechanism. The first time slot shall start after 512 x 64/fc (approximately 2.417 ms) from the completion of the ReqC command. Each time slot shall last 256 x 64/fc (approximately 1.208 ms). The number of time slots to sense during the anticollision procedure is configurable. FeliCa cards on the field respond to the ReqC command by sending an ATQC response at the beginning of the time slot. The time slot on which the response message is transmitted is randomly selected. It is responsibility of the reader IC to select the FeliCa card to communicate with.

The Fig 10 shows an anticollision procedure example where four time slots are sensed and two cards respond.



(A) Card processing time (approximately 2.417 ms)

(B) Time Slot duration (approximately 1.208 ms)

Fig 10. FeliCa anticollision example

4.3.2 Parameter Structure

The FeliCa PAL component defines its own parameter structure, which is used mainly to store parameters associated to the FeliCa card with which the communication is being performed.

```
typedef struct {
    void    * pHalDataParams;
    uint8_t   aIDmPMm[PHPAL_FELICA_IDM_LENGTH + PHPAL_FELICA_PMM_LENGTH];
    uint8_t   bIDmPMmValid;
    uint8_t   bLength;
} phpalFelica_Sw_DataParams_t;
```

*** pHalDataParams:** Pointer to the parameter structure of the underlying HAL layer component.

aIDmPMm: Manufacture ID (IDm) and Manufacture Parameters (PMm) of the FeliCa card (8 bytes + 8 bytes).

bIDmPMmValid: Indicates whether the stored IDm and PMm are valid, 1, or not, 0.

bLength: Current negotiated data length, which is used internally as a parameter for the `phpalFelica_Exchange()` function.

4.3.3 Component Initialization

The FeliCa PAL component can be initialized using the `phpalFelica_Sw_Init()` function. This function sets the pointer to the underlying HAL component on top of which the FeliCa PAL component runs and initializes the FeliCa cards associated values to their default values. The FeliCa PAL component must be initialized before the rest of the API can be used for the detection, initialization and data exchange operations with FeliCa cards.

```
phStatus_t phpalFelica_Sw_Init(
    phpalFelica_Sw_DataParams_t * pDataParams,      [In]
    uint16_t wSizeOfDataParams,                     [In]
    void * pHalDataParams );                        [In]
```

*** pDataParams**: Pointer to the FeliCa PAL parameter component.

wSizeOfDataParams: Size of the `phpalFelica_Sw_DataParams_t parameter component.`

*** pHalDataParams**: Pointer to the underlying HAL parameter component.

The values returned by the function can be:

PH_ERR_SUCCESS Operation successful.

Other: Value returned by the underlying component.

4.3.4 FeliCa PAL API

This section details the set of FeliCa PAL functions dedicated to the protocol initialization setup and the anticollision procedure.

4.3.4.1 RequestC

RequestC is the command used by the contactless reader IC in order to detect whether FeliCa cards exist in its RF field.

The reader IC waits until all FeliCa cards in all timeslots have had enough time to respond with their ATQC response. The number of time slots to be sensed during the anticollision procedure is defined by `bNumTimeSlots` argument.

Note: The function only returns the first response received.

```
phStatus_t phpalFelica_ReqC (
    void * pDataParams,             [In]
    uint8_t * pSystemCode,          [In]
    uint8_t bNumTimeSlots,          [In]
    uint8_t * pRxBuffer );          [Out]
```

*** pDataParams**: Pointer to the FeliCa parameter component `phpalFelica_Sw_DataParams_t`.

*** pSystemCode:** The system code is used to specify the application by the reader IC and is used to select the FeliCa card before sending the ATQC command. All FeliCa cards should respond when the system code 0xFFFF is transmitted.

BNumTimeSlots: Number of timeslots to use for cards detection. see Table 6.

Table 6. Number of time slots to be used during the anticollision procedure

JIS X 6319-4 value	Number of timeslots	NFC Reader Library identifier
0x00	1 Time Slot	`PHPAL_FELICA_NUMSLOTS_1`
0x01	2 Time Slots	`PHPAL_FELICA_NUMSLOTS_2`
0x03	4 Time Slots	`PHPAL_FELICA_NUMSLOTS_4`
0x07	8 Time Slots	`PHPAL_FELICA_NUMSLOTS_8`
0x0F	16 Time Slots	`PHPAL_FELICA_NUMSLOTS_16`

*** pRxBuffer**: Identifier of the FeliCa card detected. (8 bytes IDm + 8 bytes PMm)

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

PH_ERR_INVALID_PARAMETER: Invalid code for the number of time slots passed.

Other: Value returned by the underlying component.

4.3.4.2 Card Activation

The `phpalFelica_ActivateCard()` command is used in order to complete the activation of a FeliCa card. This command internally calls the `phpalFelica_ReqC()` function for the detection of FeliCa cards in the reader IC RF field. Additionally, the card activation can be performed at the system level (see section 5.4) by indicating the service code to address.

If a valid IDm is passed to this function, this value is stored as the current IDm and, therefore, no real card activation is completed.

```
phStatus_t phpalFelica_ActivateCard(
    void* pDataParams,                  [In]
    uint8_t* pIDmPMm,                   [In]
    uint8_t bIDmPMmLength,              [In]
    uint8_t * pSystemCode,              [In]
    uint8_t bNumTimeSlots,              [In]
    uint8_t * pRxBuffer,                [Out]
    uint8_t * pRxLength,                [Out]
    uint8_t * pMoreCardsAvailable );    [Out]
```

*** pDataParams**: Pointer to the FeliCa parameter component phpalFelica_Sw_DataParams_t.

*** pIDmPMm**: IDm followed by PMm. If this parameter is supplied then it is stored and no real activation is performed.

BIDmPMmLength: `pIDmPMm` length. 16 bytes if IDm and PMm are supplied; 0 if not.

*** pSystemCode**: The system code is used to specify the application by the reader and is used to select the FeliCa card before sending the ATQC command. All FeliCa cards should respond when the system code 0xFFFF is transmitted.

BNumTimeSlots: Number of timeslots to use for cards detection. see Table 6.

*** pRxBuffer**: 8 bytes IDm + 8 bytes PMm

*** pRxLength**: Length of received data. 0 or 16

*** pMoreCardsAvailable**: Indicates whether there are more cards in the field or not.

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

PH_ERR_INVALID_PARAMETER: pIDmPmm length is not 16 bytes.

Other: Value returned by the underlying component.

4.3.4.3 Exchange

The Exchange command is used for the data exchange between the contactless reader IC and the FeliCa card. This command should not be directly called by the developer since it is called internally by FeliCa Application Layer functions.

When a Read command is executed, the reception buffer should be checked and when a Write command is executed, the transmission buffer should be specified.

```
phStatus_t phpalFelica_Exchange(
    void * pDataParams,                 [In]
    uint16_t wOption,                   [In]
    uint16_t wN,                        [In]
    uint8_t* pTxBuffer,                 [In]
    uint16_t wTxLength,                 [In]
    uint8_t ** ppRxBuffer,              [Out]
    uint16_t * pRxLength );             [Out]
```

*** pDataParams**: Pointer to the FeliCa parameter component `phpalFelica_Sw_DataParams_t`.

WOption: It indicates in which part of the stream is located the given data: in the first part (`PH_EXCHANGE_BUFFER_FIRST`), in the middle part (`PH_EXCHANGE_BUFFER_CONT`) or in the last part (`PH_EXCHANGE_BUFFER_LAST`).

Wn: Number of blocks to exchange. This value is used to calculate the response timeout.

pTxBuffer: Data to be transmitted. The length and IDm values are automatically added by the NFC Reader Library implementation.

wTxLength: Length of the data to be transmitted.

**** ppRxBuffer**: Pointer to the received data. The response code, length and IDm values are automatically removed by the NFC Reader implementation.

*** pRxLength**: Length of the received data.

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

PH_ERR_INVALID_PARAMETER: `wTxLength` exceeds the maximum allowed length of data to be transmitted.

Other: Value returned by the underlying component.

4.3.4.4 Get Serial Number

This function is used to retrieve IDm and PMm values from a specific card.

```
phStatus_t phpalFelica_GetSerialNo(
    void * pDataParams,                 [In]
    uint8_t * pUidOut,                  [Out]
    uint8_t * pLenUidOut );             [Out]
```

*** pDataParams**: Pointer to the FeliCa parameter component `phpalFelica_Sw_DataParams_t`.

*** pUidOut**: IDm and PMm values of the FeliCa card.

*** pLenUidOut**: Length of `pUidOut` value. 16 bytes if a valid value is received; 0 if not.

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

PH_ERR_USE_CONDITION: No Serial number available at the moment.

Other: Value returned by the underlying component.

4.4 ISO/IEC 18092

4.4.1 Technical Introduction

4.4.1.1 ISO/IEC 18092 Standard

The ISO/IEC 18092 standard [19], also known as NFCIP-1, defines the interface and protocol for simple wireless communication between two NFC devices. The RF layer used in the ISO/IEC 18092 standard inherits from previous proximity contactless technologies, more specifically from the ISO/IEC 14443-A protocol and Sony FeliCa (JIS-6319-4).

According to the specification, a NFC reader can communicate with bitrates of 106 Kbps (ISO/IEC 14443A modulation), 212 Kbps and 424 Kbps (FeliCa modulation).

The ISO/IEC 18092 component is responsible for encapsulating packets coming from upper protocols, into final binary format that is used for the transmission on the RF field. Since LLCP defines a bidirectional balanced communication mechanism where just one packet can be transmitted per peer at a time, each time a packet is transmitted the ISO/IEC 18092 component will wait for a response. If none response is received in the negotiated period of time, the link will be considered broken.

NFCIP-1 defines two new terms to identify devices involved in the communication: Initiator and Target (see section 4.4.1.2); and two new communication modes: Active and Passive (see section 4.4.1.2). Therefore, NFCIP-1 covers four possible combinations: Passive Initiator, Active Initiator, Passive Target and Active Target. The NFC Reader Library implements the four possibilities. These new terms are further explained in the following sections.

4.4.1.2 NFCIP-1 Devices

NFCIP-1 Communication Roles

The Peer-to-Peer operating mode, specified in the NFCIP-1 standard, defines a new bidirectional communication scheme. Unlike in traditional smart card scenarios, both devices are able to ask and respond. Therefore, NFCIP-1 mode defines the Initiator and Target concepts.

Initiator

The device starting the communication and generating the RF field at 13.56 MHz.

Target

The device responding to the initiator communication establishment request either using its own generated RF field or modulating the RF generated by the initiator.

NFCIP-1 Communication Modes

In the traditional smart card scenario, the reader is always supplying the power to the smartcard. The Peer-to-Peer operating mode defines a new communication mode, the

active communication mode, where both the Initiator and the Target device have the possibility to generate their own RF field to communicate.

Active communication

In the active communication mode, both the initiator and the target devices generate their own RF field to transmit data. Once the data has been transmitted, the RF field is switched off in order not to interfere with the RF field generated by the other peer.

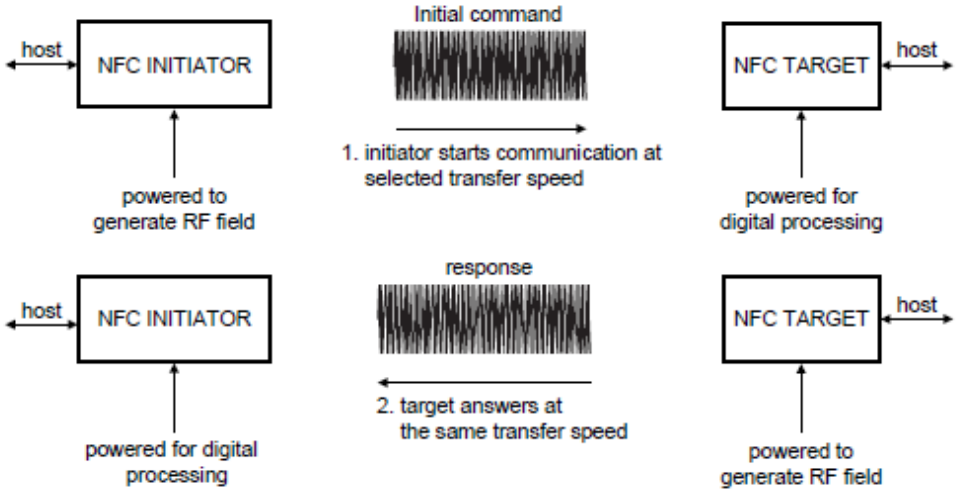


Fig 11. Active communication scheme

The main advantage of implementing the active communication, compared to the passive communication, is the larger coverage distance in case of the same bit rate, or the higher bit rate in case of the same coverage distance. According to the ISO/IEC 18092 standard, higher bit rates than 424 kbps are expected for this communicating mode in future standard updates.

Passive communication

In the passive communication mode, the initiator starts the communication and generates its own RF field, which will be maintained until the communication is closed. This field is used by the target as a mean to obtain energy and to transmit data using a load modulation scheme.

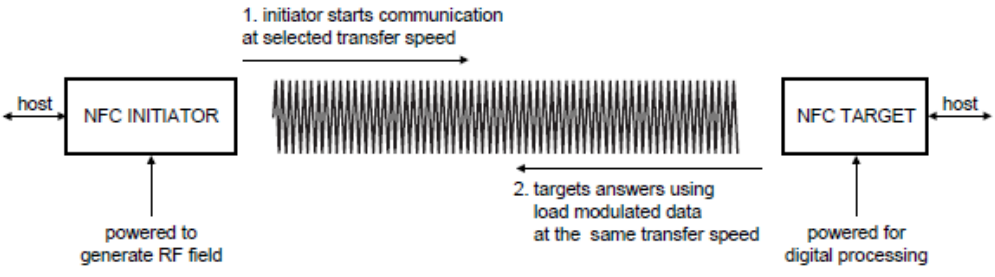


Fig 12. Passive communication scheme

The passive communication mode inherits from the traditional smart card scenario. This is the reason why some readers in the market only support this configuration.

NFCIP-1 Devices vs Reader ICs

The NFC Reader Library supports the four communication configurations defined by the ISO/IEC 18092 standard: Passive Initiator, Active Initiator, Passive Target and Active Target. However, not all the reader ICs available on the Hardware Abstraction Layer support all the communication modes and roles.

Fig 13 depicts communication modes supported by NXPCLRC 663 reader IC and NXP PN512 reader IC.

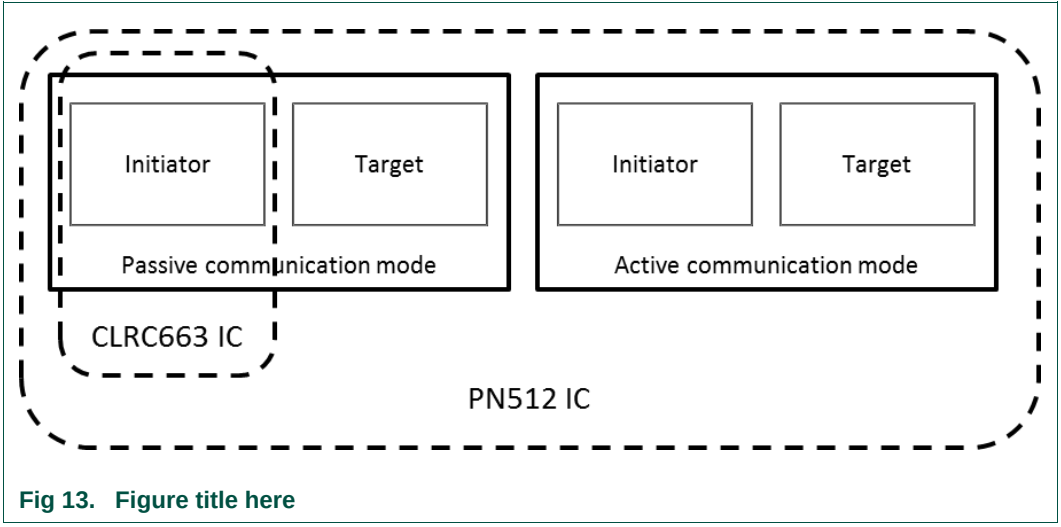


Fig 13. Figure title here

The PN512 reader IC supports all possible NFCIP-1 configurations whereas CLRC663 reader IC only provides support for Passive Initiator configuration.

4.4.1.3 ISO/IEC 18092 API Communication Flow

Two NFCIP-1 devices communicating with each other shall follow a well-established flow of communication in order to succeed. The proposed communication flow that ensure the correct functioning of the standard is shown in Fig 14.

For further details about each function, please consult the following sections where both ISO/IEC 18092 Initiator API and ISO/IEC 18092 Target API are explained in detail.

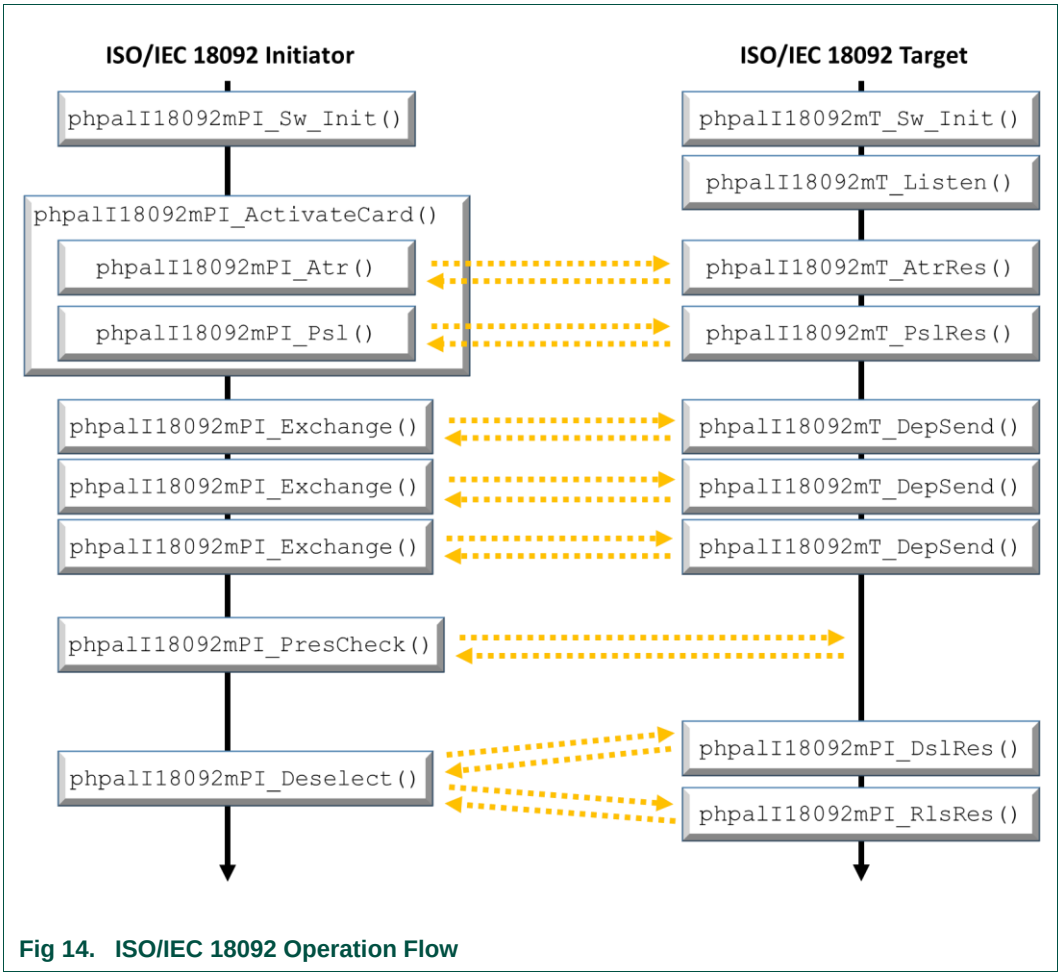


Fig 14. ISO/IEC 18092 Operation Flow

It is important to note that a developer may not need to make use of all these functions since some of them are internally called by upper layer protocols such as LLCP, SNEP, and others.

4.4.2 ISO/IEC 18092 Initiator

A NFC Reader configured as an Initiator is responsible for the 13.56 MHz RF field generation and for initializing the communication. In order to initialize the communication, the initiator sends ATR_REQ commands periodically, waiting for an ATR_RES response from a target to complete the establishment of the communication.

It is during the communication setup when the reader IC indicates the communication mode that is going to operate in: Active communication mode, Passive communication mode or both. Available configurations in the NFC Reader Library can be found in the header file: *NxpRdLib_PublicRelease/intfs/phacDiscLoop.h* file.

Table 7. ISO18092 PAL communication mode configuration

Parameter	Description	Value
PHAC_DISCLOOP_CON_POLL_A	ISO/IEC 14443A Passive mode	0x01U
PHAC_DISCLOOP_CON_POLL_B	ISO/IEC 14443B Passive mode	0x02U
PHAC_DISCLOOP_CON_POLL_F	FeliCa Passive mode	0x04U

Parameter	Description	Value
PHAC_DISCLOOP_CON_POLL_ACTIVE	ISO/IEC 18092 P2P Active mode	0x08U

ISO18092mPI PAL layer component defines its own `phpalI18092mPI_Sw_DataParams_t` parameter structure in order to store the ISO/IEC 18092 protocol attributes that configures the communication. These parameters are listed in the following table.

Table 8. Parameters from ISO18092 Initiator Pal component

On the right column there are default values given by the Initialization function (see section 4.4.2.1). Apart from these parameters, there are few other for internal management.

parameter	description	default init value
pHalDataParams	Pointer to the parameter structure of the underlying HAL layer.	input parameter Init
bNfcIdValid	Whether current NfcID is valid or not.	PH_OFF
aNfcid3i	NFC ID 10 bytes long identifier	input parameter ATR_REQ (section 4.4.2.3)
bStateNow	Current exchange state	0
bDID	Device identifier	NULL
bNADenabled	NAD enabler	PH_OFF
bNAD	Node Address	NULL
bWT	Waiting timeout for a target	14 ()
bFSL	Frame length	0 (means 64 bytes)
bPNI	Packet number	NULL
bDSi	Divisor Send from Parameter Select Request (Initiator to target)	NULL (106kbit/s)
bDRi	Divisor Receive initiator	NULL (106kbit/s)
bMaxRetryCount	Maximum number of attempts to send a Request looking for a valid Target.	2
bAtnDisabled	ATN Disabler	0
bActiveMode	Active mode configuration bit	PH_OFF

4.4.2.1 Protocol Initialization

The first step to complete is the initialization of the component and the setup of the `phpalI18092mPI_Sw_DataParams_t` parameter component. This function automatically calls the `phpalI18092mPI_ResetProtocol()` function in order to set default values of the structure and registers a pointer to the component of the underlying HAL layer.

```
phStatus_t phpalI18092mPI_Sw_Init(
    phpalI18092mPI_Sw_DataParams_t * pDataParams,     [In]
    uint16_t wSizeOfDataParams,                       [In]
    void * pHalDataParams );                          [In]
```

***pDataParams:** Pointer to the `phpalI18092mPI_Sw_DataParams_t` parameter structure.

wSizeOfDataParams: Size of the `phpalI18092mPI_Sw_DataParams_t` parameter component.

***pHalDataParams:** Pointer to the underlying HAL component.

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

PH_ERR_INVALID_DATA_PARAMS: Operation failed, invalid data parameters.

4.4.2.2 Reset Protocol

This function resets the values of given ISO18092mPI PAL structure parameter component to the default values defined in Table 8.

```
phStatus_t phpalI18092mPI_ResetProtocol(
    void * pDataParams );                    [In]
```

***pDataParams**: Pointer to the `phpalI18092mPI_Sw_DataParams_t` parameter component.

The value returned by the function is:

PH_ERR_SUCCESS: Operation successful.

4.4.2.3 Attribute Request

This function initializes the communication setup. It sends an ISO/IEC 18092 Attribute Request command and listens for Attribute Responses coming from targets.

DID and NAD values received from Target's Attribute Response are verified in order to ensure that they match with the Initiator defined values. The Initiator timeout value is set according to TO Timeout value defined in the Attribute Response command.

In case the whole communication setup procedure is successfully completed, the ISO18092 PAL structure parameter will be filled in accordance with the negotiated communication values.

Note: This function does not allow setting either BSi or BRi attributes of the ISO18092 communication protocol since those are not supported by the passive communication mode. The send and receive rates are set by the `phpalI18092mPI_Psl()` function.

```
phStatus_t phpalI18092mPI_Atr(
                  void * pDataParams,      [In]
                  uint8_t *pNfcid3i,       [In]
                  uint8_t bDid,            [In]
                  uint8_t bLri,            [In]
                  uint8_t bNadEnable,      [In]
                  uint8_t bNad,            [In]
                  uint8_t *pGi,            [In]
                  uint8_t bGiLength,       [In]
                  uint8_t *pAtrRes,        [Out]
                  uint8_t *pAtrResLength );                              [Out]
```

***pDataParams**: Pointer to the `phpalI18092mPI_Sw_DataParams_t` parameter component.

***pNfcid3i**: The application randomly generates 10 bytes value for the initiator. For Passive communication mode 212 and 424 kbps the NFCID3i shall be replaced by NFCID2t. In the Discovery Loop there is UID from `phacDiscLoop_Sw_Int_DetectA()` or ID+PM from `phacDiscLoop_Sw_Int_DetectF()` used as NFCID2.

bDid: Device Identifier used for multiple data transport protocol activation with more than one target. Value must be in range from 0 to 14. Zero disables DID usage.

bLri: Length Reduction of the Transport Data on the Initiator side for the supported transmission rates. If this Lri value differs from the one received from the Target Attribute Response, then the smaller value is selected for the communication. The negotiated

Length Reduction value is retained as FSL value in the `phpalI18092mPI_Sw_DataParams_t` structure parameter.

Table 9. Table of Length Reduction values

The first column refers to LRi bits and how they are placed in the Protocol Parameter Initiator byte or FSL byte defined by ISO18092 standard.
The second column refers to the number of bytes that the Initiator shall send in the Transport Data field within DEP.
On the third column there are identifiers from the NFC Reader Library similar to the previous columns.

LRi bits ISO18092	Max byte count in the DEP Transport Data	NFC Reader Library identifier
00	64 bytes	`PHPAL_I18092MPI_FRAMESIZE_64`
01	128 bytes	`PHPAL_I18092MPI_FRAMESIZE_128`
10	192 bytes	`PHPAL_I18092MPI_FRAMESIZE_192`
11	254 bytes	`PHPAL_I18092MPI_FRAMESIZE_254`

bNadEnable: Node Address enabler. Zero or `PH_OFF` disables NAD usage. `PH_ON` or any nonzero value enables NAD.

bNad: Node Address used in DEP for logical addressing. This parameter is ignored if `bNadEnabled` is equal to zero.

***pGi**: Optional General Information bytes sent by the Initiator.

bGiLength: Number of General Information bytes sent by the Initiator.

***pAtrRes**: Pointer to the Attribute Response command received from the Target.

***pAtrResLength**: Attribute Response length.

The values returned by the function can be:

PH_ERR_INVALID_PARAMETER: bDid, bLri or bGiLength value out of .valid range.

PH_ERR_PROTOCOL_ERROR: The received response is not ISO/IEC 18092 compliant.

PH_ERR_IO_TIMEOUT: Timeout for reply expired, e.g. target removal.

PH_ERR_SUCCESS: Operation successful.

Other: Value returned by the underlying component.

4.4.2.4 Parameter Selection

This function provides the ISO/IEC 18092 initiator defined Parameter Selection Request used to modify communication parameters such as the bit rate. After the command is transmitted, the initiator listens for the Parameter Selection Response from the Target.

```
phStatus_t phpalI18092mPI_Psl(
    void * pDataParams,          [In]
    uint8_t bDsi,                [In]
    uint8_t bDri,                [In]
    uint8_t bFsl );              [In]
```

***pDataParams**: Pointer to the `phpalI18092mPI_Sw_DataParams_t` parameter component.

bDsi: Divisor value for the initiator to target transmission data rate. Only the values from Table 10 (column *NFC Reader Library identifier*) are accepted.

Table 10. Table of Divisor Send/Receive

ISO18092	ISO18092 bit duration Divisor D	Bit rate Kbit/s	ISO18092 Divisor D	NFC Reader Library identifier
0	1	106	1	PHPAL_I18092MPI_DATARATE_106
1	2	212	2	PHPAL_I18092MPI_DATARATE_212
2	4	424	4	PHPAL_I18092MPI_DATARATE_424

bDri: Divisor value for the target to initiator transmission data rate. Only the values defined in Table 10 (column *NFC Reader Library identifier*) are accepted.

bFsl: maximum value for the Frame Length. Valid values defined in Table 11.

Table 11. Table of valid Length Reduction values

LR	Maximum Length
00	Only Byte 0 to Byte 63 is valid in the Transport Data
01	Only Byte 0 to Byte 127 is valid in the Transport Data
10	Only Byte 0 to Byte 191 is valid in the Transport Data
11	Only Byte 0 to Byte 255 is valid in the Transport Data

The values returned by the function can be:

PH_ERR_INVALID_PARAMETER: bDsi, bDri or bFsl value out of valid range.

PH_ERR_PROTOCOL_ERROR: Received response is not ISO/IEC 18092 compliant.

PH_ERR_IO_TIMEOUT: Timeout for reply expired, e.g. target removal.

PH_ERR_SUCCESS: Operation successful.

Other: Value returned by the underlying component.

4.4.2.5 Activate Card

This function integrates both the Attribute request and the Parameter Selection commands in a single command.

```
phStatus_t phpalI18092mPI_Sw_ActivateCard(
    void * pDataParams,                 [In]
    uint8_t * pNfcid3i,                 [In]
    uint8_t bDid,                       [In]
    uint8_t bNadEnable,                 [In]
    uint8_t bNad,                       [In]
    uint8_t bDsi,                       [In]
    uint8_t bDri,                       [In]
    uint8_t bFsl,                       [In]
    uint8_t * pGi,                      [In]
    uint8_t bGiLength,                  [In]
    uint8_t * pAtrRes,                  [Out]
    uint8_t * pAtrResLength );          [Out]
```

***pDataParams**: Pointer to the `phpalI18092mPI_Sw_DataParams_t` parameter component.

***pNfcid3i**: The application randomly generates 10 bytes value for the initiator. For Passive communication mode 212 and 424 kbps the NFCID3i shall be replaced by

NFCID2t. In the Discovery Loop there is UID from `phacDiscLoop_Sw_Int_DetectA()` or ID+PM from `phacDiscLoop_Sw_Int_DetectF()` used as NFCID2.

bDid: Device Identifier used for multiple data transport protocol activation with more than one target. Value must be in range from 0 to 14. Zero disables DID usage.

bNadEnable: Node Address enabler. Zero or `PH_OFF` disables NAD usage. `PH_ON` or any nonzero value enables NAD.

bNad: Node Address used in DEP for logical addressing. This parameter is ignored if `bNadEnabled` is equal to zero.

bDsi: Divisor value for the initiator to target transmission data rate. Only the values from Table 10 (column *NFC Reader Library identifier*) are accepted.

bDri: Divisor value for the target to initiator transmission data rate. Only the values defined in Table 10 (column *NFC Reader Library identifier*) are accepted.

bFsl: maximum value for the Frame Length. Valid values defined in Table 11.

***pGi**: Optional General Information bytes sent by the Initiator.

bGiLength: Number of General Information bytes sent by the Initiator.

***pAtrRes**: Pointer to the Attribute Response command received from the Target.

***pAtrResLength**: Attribute Response length.

The values returned by the function can be:

PH_ERR_INVALID_PARAMETER: bDid, bLri or bGiLength value out of .valid range.

PH_ERR_PROTOCOL_ERROR: The received response is not ISO/IEC 18092 compliant.

PH_ERR_IO_TIMEOUT: Timeout for reply expired, e.g. target removal.

PH_ERR_SUCCESS: Operation successful.

Other: Value returned by the underlying component.

4.4.2.6 Deselect

This function sends an ISO/IEC 18092 command with either Deselect Request or Release Request, and then waits for either Deselect Response or Release Response.

These Deselect and Release commands may be useful if more than one Target device is managed by the Initiator so that other Targets can be initialized.

```
phStatus_t phpalI18092mPI_Deselect(
    void * pDataParams,                    [In]
    uint8_t bDeselectCommand );            [In]
```

***pDataParams**: Pointer to the `phpalI18092mPI_Sw_DataParams_t` parameter component.

bDeselectCommand: Indicates whether Deselect or Release command is indicated. `PHPAL_I18092MPI_DESELECT_DSL` for Deselect and `PHPAL_I18092MPI_DESELECT_RLS` for Release.

The values returned by the function can be:

PH_ERR_PROTOCOL_ERROR: Received response is not ISO/IEC 18092 compliant.

PH_ERR_IO_TIMEOUT: Timeout for reply expired, e.g. target removal.

PH_ERR_SUCCESS: Operation successful.

Other: Value returned by the underlying component.

4.4.2.7 Exchange Data

This function sends the ISO/IEC 18092 defined Data Exchange Protocol Request commands in order to transmit data to the target, and then waits for the Data Exchange Response command. All the upper protocols and the packets transmitted between the initiator and the target leverage on this function.

```
phStatus_t phpalI18092mPI_Exchange(
    void * pDataParams,                 [In]
    uint16_t wOption,                   [In]
    uint8_t * pTxBuffer,                [In]
    uint16_t wTxLength,                 [In]
    uint8_t ** ppRxBuffer,              [Out]
    uint16_t * pRxLength );             [Out]
```

***pDataParams**: Pointer to the `phpalI18092mPI_Sw_DataParams_t` parameter component.

wOption: Option parameter indicating how to send the DEP frame sequence according to Table 12.

Table 12. Exchange options

NFC Reader Library identifier	Functioning
`PH_EXCHANGE_DEFAULT`	Default exchange mode. Sufficient to perform NFC P2P correctly
`PH_EXCHANGE_BUFFERED_BIT`, `PH_EXCHANGE_LEAVE_BUFFER_BIT`	Advanced buffer methods related to HAL buffer.
`PH_EXCHANGE_TXCHAINING`, `PH_EXCHANGE_RXCHAINING`, `PH_EXCHANGE_RXCHAINING_BUFSIZE`	ISO18092 frame chaining. They don´t need to be set. If the data to be exchanged is more than Frame Size configured by `phpalI18092mPI_Atr()` Length Reduction or `phpalI18092mPI_Psl()`, this function performs the chaining automatically. In result frame of any size is transmitted correctly.

***pTxBuffer**: Data to be transmitted from the initiator to the target.

wTxLength: Length of the data to be transmitted.

****ppRxBuffer**: Pointer to the received data.

***pRxLength**: Number of received data bytes.

The values returned by the function can be:

PH_ERR_INVALID_PARAMETER: Invalid flag bit in xOption used.

PH_ERR_PROTOCOL_ERROR: Received response is not ISO/IEC 18092 compliant.

PH_ERR_IO_TIMEOUT: Timeout for reply expired, e.g. target removal.

PHPAL_I18092MPI_ERR_RECOVERY_FAILED: Recovery failed, target does not respond any more.

PH_ERR_SUCCESS: Operation successful.

Other: Value returned by the underlying component.

4.4.2.8 Presence Check

This function performs a presence check in order to determine whether the current target is still in the RF field range or not. The reader transmits an ISO DEP packet marked as

Supervisory Attention PDU, and then listens for the Supervisory Attention Response from the other device.

phStatus_t phpalI18092mPI_PresCheck(
    void * pDataParams); [In]

***pDataParams**: Pointer to the `phpalI18092mPI_Sw_DataParams_t` parameter component.

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

PH_ERR_PROTOCOL_ERROR: Received response is not ISO/IEC 18092 compliant.

PH_ERR_IO_TIMEOUT: Timeout for reply expired, e.g. target removal.

PHPAL_I18092MPI_ERR_RECOVERY_FAILED: Recovery failed, target does not respond any more.

PH_ERR_SUCCESS: Operation successful.

Other: Value returned by the underlying component.

4.4.3 ISO/IEC 18092 Target

A NFC Reader running in target configuration will respond to the ATR_REQ command sent by the initiator with an ATR_RES response to complete the communication setup. This response will be transmitted using its own generated RF field in case of active communication or modulating the initiator generated RF field in case of passive communication.

In order to set the NFC Reader in target mode, the DISCOVERY_MODE tag of the Discovery Loop has to be set to `PHAC_DISCLOOP_SET_LISTEN_MODE` value. Information regarding the Discovery Loop is provided in Section 6.

The `phpalI18092mT_Sw_DataParams_t` parameter structure of the ISO18092mT PAL used to configure the reader in target mode consists of the following parameters.

Table 13. Parameters from ISO18092 Target Pal component

The right column contains the default values given by the Initialization function (see section 4.4.3.1). Besides these parameters, there are several more for internal management.

parameter	description	default init value
pHalDataParams	Pointer to the parameter structure of the underlying HAL layer.	input parameter Init
bNfcIdValid	Whether current NfcID is valid or not.	PH_OFF
aNfcid3i	NFC ID 10 bytes long identifier	input parameter ATR_REQ (section 4.4.2.3)
aNfcid3t	NFC ID 10 bytes long identifier of the initiator	
bStateNow	Current exchange state	0
bDid	Device identifier	NULL
bNADEnabled	NAD enabler	PH_OFF
bNad	Node Address	NULL
BWt	Waiting timeout for a target	14 ()
bFsl	Frame length	0 (means 64 bytes)
bPni	Packet number	NULL

parameter	description	default init value
bDst	Sending divisor	NULL (106kbit/s)
bDrt	Receiving divisor	NULL (106kbit/s)
bBsi	Sending bit rate supported by the initiator	0
bBri	Receiving bit rate supported by the initiator	0
BLri	Length reduction value for the initiator	0 (means 64 bytes)
bBst	Sending bit rate supported by the target	0
bBrt	Receiving bit rate supported by the target	0
bLrt	Length reduction value for the target	0 (means 64 bytes)
MaxRetryCount	Maximum number of attempts to send a Request looking for a valid Target.	2
pGt	Optional General Information bytes for the target.	NULL
bGtLength	Number of General Information bytes for the target	0
bTo	Timeout value	0
bTargetState	Target state	0 (State none)
bRtoxDisabled	Rtox disabler	PH_OFF
bRtox	Response timeout extension value	01 (Min Rtox value)
pSet_Interrupt	Callback to interrupt function	NULL
ovrTask	Pointer to the upper associated task component	NULL
bCmdtype	Command type	0 (Cmd Atttribute Reques)
phOsal	Pointer to the associated OSAL component	NULL
dwTimerId	Timer ID for Rtox management	0xFFFF (Invalid timer)
rtoxStatus	Rtox status	0
bActiveMode	Active mode configuration bit	PH_OFF

During the following subsections, the different functions used for the target device management are presented.

4.4.3.1 Protocol Initialization

The ISO18092mT component must be initialized by calling its Init function in order to setup the `phpalI18092mT_Sw_DataParams_t` parameter component. This function automatically calls the `phpalI18092mT_ResetProtocol()` function in order to set the default values of the structure and simultaneously registers a pointer to the component of the underlying HAL layer.

The target configuration registers a callback to its enabling function.

```
phStatus_t phpalI18092mT_Sw_Init(
    phpalI18092mT_Sw_DataParams_t * pDataParams,      [In]
    uint16_t wSizeOfDataParams,                       [In]
    void * pHalDataParams,                            [In]
    pInterruptSetCallback pSetInterrupt);             [In]
```

***pDataParams:** Pointer to the `phpalI18092mT_Sw_DataParams_t` parameter structure.

wSizeOfDataParams: Size of the `phpalI18092mT_Sw_DataParams_t` parameter component.

***pHalDataParams:** Pointer to the underlying HAL component.

pSetInterrupt: Pointer to the function that enables the callback interruption associated to the ISO18092mT component.

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

PH_ERR_INVALID_DATA_PARAMS: Operation failed, invalid data parameters.

4.4.3.2 Reset Protocol

This function reset values of given ISO18092mT PAL structure parameter component to the default values defined in Table 13.

```
phStatus_t phpalI18092mT_ResetProtocol(
    void * pDataParams );                    [In]
```

***pDataParams**: Pointer to the `phpalI18092mT_Sw_DataParams_t` parameter component.

The values returned by the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_ERR_PROTOCOL_ERROR`: Received response is not ISO/IEC 18092 compliant.

`PH_ERR_IO_TIMEOUT`: Timeout for reply expired, e.g. target removal.

4.4.3.3 RF Field Listening

A NFC Reader configured to act as a target device for communication shall sense the RF field in order to listen for commands from the other peer. This function configures how the listening is carried out on the target device.

This function initializes the buffer to be used according to the command expected to be received. Therefore, the reception buffer will be configured based on the `phpalI18092mT_Sw_DataParams_t` structure and `bCmdType` value at the particular moment when the function is called.

```
phStatus_t phpalI18092mT_Listen (
    void * pDataParams,                      [In]
    uint16_t wOption                         [In]
    uint8_t ** ppRxBuffer,                   [Out]
    Uint16_t * pRxLength,                    [Out]
    void* context);                          [In]
```

***pDataParams**: Pointer to the `phpalI18092mT_Sw_DataParams_t` parameter component.

wOption: Parameter indicating how to the send DEP frame sequence according to Table 12.

***ppRxBuffer**: Pointer to the reception buffer to be initialized by the library core.

***pRxLength**: Length of the reception buffer.

***context**: Input context for the upper component defined in `ovrTask` value. It may be NULL if it is not used.

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

PH_ERR_PROTOCOL_ERROR: Received response is not ISO/IEC 18092 compliant.

PH_ERR_IO_TIMEOUT: Timeout for reply expired, e.g. target removal.

Other: Value returned by the underlying component.

4.4.3.4 Attribute Response

A NFC Reader configured in target mode is periodically sensing the RF field looking for ATR_REQ commands from an initiator trying to setup a communication.

When the target receives the ATR_REQ command from the initiator, it checks the validity of the values received such as supported bit rates or the right configuration of DID and NAD fields. If the request is considered valid, the target configures its own structure and sends the ATR_RES response containing all the parameters that define the communication channel.

The ATR_RES response to be sent to the initiator should have previously been defined as it is explained in 4.4.2.3.

After the target device responds with the ATR_RES sequence, the communication channel is considered established for the transmission of upper link layer packets.

```
phStatus_t phpalI18092mT_AtrRes (
    void * pDataParams,                 [In]
    uint8_t *pAtr,                      [In]
    Uint16_t wAtrLength);               [In]
```

***pDataParams**: Pointer to the `phpalI18092mT_Sw_DataParams_t` parameter component.

***pAtr:** Received Attribute Request bytes coming from the initiator.

wAtrLength: Number of bytes of the received Attribute Request.

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

PH_ERR_PROTOCOL_ERROR: The received response is not ISO/IEC 18092 compliant.

PH_ERR_IO_TIMEOUT: Timeout for reply expired, e.g. target removal.

Other: Value returned by the underlying component.

4.4.3.5 Set Attribute Response

A target receiving an ATR_REQ command will respond with an ATR_RES in order to setup the communication channel. The ATR_RES shall be configured with the selected parameters for the communication. This function sets the ATR_RES according to the parameters defined by the developer.

```
phStatus_t phpalI18092mT_SetAtrRes (
    phpalI18092mT_Sw_DataParams_t * pDataParams,    [In]
    uint8_t *pNfcid3t,                              [In]
    uint8_t bLrt,                                   [In]
    uint8_t bNadEnable,                             [In]
    uint8_t bBst,                                   [In]
```

```
uint8_t bBrt,                                   [In]
uint8_t bBTo,                                   [In]
uint8_t *pGt,                                   [In]
uint8_t bGtLength;                              [In]
```

***pDataParams**: Pointer to the `phpalI18092mT_Sw_DataParams_t` parameter component.

***pNfcid3t**: Randomly generated 10 bytes value by the application for the target device. For Passive communication mode 212 and 424 kbps the NFCID3i shall be replaced by NFCID2t. In the Discovery Loop there is UID from `phacDiscLoop_Sw_Int_DetectA()` or ID+PM from `phacDiscLoop_Sw_Int_DetectF()` used as NFCID2.

bLrt: Length Reduction of the Transport Data on the target side for the supported transmission rates indication. Negotiated Length Reduction value is retained as FSL value in the `phpalI18092mT_Sw_DataParams_t` structure parameter. Available values are included in Table 9.

bNadEnable: Node Address enabler. Zero or `PH_OFF` disables NAD usage. `PH_ON` or any nonzero value enables NAD.

bBst: Bit rates supported by the target in sending direction.

bBrt: Bit rates supported by the target in receiving direction.

bTo: Timeout value used to code the Response Waiting Time.

***pGt**: Optional General Information bytes sent by the target.

bGtLength: Number of General Information bytes sent by the target.

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

PH_ERR_PROTOCOL_ERROR: Received response is not ISO/IEC 18092 compliant.

PH_ERR_IO_TIMEOUT: Timeout for reply expired, e.g. target removal.

Other: Value returned by the underlying component.

4.4.3.6 Parameter Selection Response

This function responds to the ISO18092 Parameter Selection Request sent by the initiator in order to trigger the modification of the communication parameters such as the transmission bit rate.

The target validates the parameters proposed by the initiator and sends the response indicating the acceptance and the new values for the communication parameters. The list of expected values can be found in Table 10 and Table 11.

```
phStatus_t phpalI18092mT_PslRes(
    void * pDataParams,                         [In]
    uint8_t *pPslReq,                           [In]
    uint16_t wPslReqLength);                    [In]
```

***pDataParams**: Pointer to the `phpalI18092mT_Sw_DataParams_t` parameter component.

***pPslReq:** Received Parameter Selection Request bytes from the initiator.

wPslReqLength: Number of bytes of the received Parameter Selection Request.

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

PH_ERR_PROTOCOL_ERROR: Received response is not ISO/IEC 18092 compliant.

PH_ERR_IO_TIMEOUT: Timeout for reply expired, e.g. target removal.

Other: Value returned by the underlying component.

4.4.3.7 Deselect Response

This function responds to the ISO18092 Deselect Request sent by the initiator. After the correct completion of the process, the initiator releases the DID assigned to the target; and the target returns to the initially chosen state and enables the default bit rate for the communication.

```
phStatus_t phpalI18092mT_DslRes(
    void * pDataParams,                 [In]
    uint8_t *pDslReq,                   [In]
    uint16_t wDslReqLength);            [In]
```

***pDataParams**: Pointer to the `phpalI18092mT_Sw_DataParams_t` parameter component.

***pDslReq:** Received Deselect Request bytes coming from the initiator.

wDslReqLength: Number of bytes of the received Deselect Request.

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

PH_ERR_PROTOCOL_ERROR: Received response is not ISO/IEC 18092 compliant.

PH_ERR_IO_TIMEOUT: Timeout for reply expired, e.g. target removal.

Other: Value returned by the underlying component.

4.4.3.8 Release Response

This function responds to the ISO18092 Release Request sent by the initiator. Once the target has been successfully released, both the initiator and the target return to their initial state.

```
phStatus_t phpalI18092mT_RlsRes(
    void * pDataParams,                 [In]
    uint8_t *pRslReq,                   [In]
    uint16_t wRslReqLength);            [In]
```

***pDataParams**: Pointer to the `phpalI18092mT_Sw_DataParams_t` parameter component.

***pRslReq:** Received Release Request bytes coming from the initiator

wRslReqLength: Number of bytes of the received Release Request

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

PH_ERR_PROTOCOL_ERROR: Received response is not ISO/IEC 18092 compliant.

PH_ERR_IO_TIMEOUT: Timeout for reply expired, e.g. target removal.

Other: Value returned by the underlying component.

4.4.3.9 Exchange Data Response

This function responds to the Data Exchange Protocol Request received from the initiator. The target should process the received command and respond accordingly within the defined timeout period. If the timeout period is not sufficient to complete the task, it can ask for a Reception Timeout Extension time.

Since the name could lead to a misunderstanding, it is important to remark that the communication is not a traditional request / response communication where the reader asks and the smart card responds. Peer-to-Peer communication mode defines a bidirectional channel where both the initiator and target are able send commands and receive responses.

```
phStatus_t phpalI18092mT_DepSend(
    void * pDataParams,                 [In]
    uint16_t wOption,                   [In]
    uint8_t * pTxBuffer,                [In]
    uint16_t wTxLength);                [In]
```

***pDataParams**: Pointer to the `phpalI18092mT_Sw_DataParams_t` parameter component.

wOption: Option parameter indicating how to send the DEP frame sequence according to Table 12.

***pTxBuffer**: Data to be transmitted from the target to the initiator.

wTxLength: Length of the data to be transmitted.

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

PH_ERR_PROTOCOL_ERROR: Received response is not ISO/IEC 18092 compliant.

PH_ERR_IO_TIMEOUT: Timeout for reply expired, e.g. target removal.

Other: Value returned by the underlying component.

5. NFC Reader Library API: Application Layer (AL)

In this section, the MIFARE Classic, MIFARE Ultralight, MIFARE DESFire, FeliCa, Jewel Topaz and NFC Forum Tag Types operation components defined in the Application Layer (AL) of the NFC Reader Library are explained in depth.

5.1 MIFARE Classic

5.1.1 Technical Introduction

The MIFARE Classic card was launched in 1994 and is the most widely used contactless smart card IC in world. MIFARE Classic is compliant with the ISO/IEC 14443-3A standard except for the authentication and encryption protocols that are based on the Crypto-1 algorithm that (NXP proprietary). MIFARE Classic is widely used in -contactless services such as public transport, access management, loyalty programs and many other applications.

The MIFARE Classic memory is divided into 16 bytes data blocks, grouped together to form sectors. MIFARE Classic is available in 1kbyte and 4Kbyte card ICs. The MIFARE Classic 1KB product is divided into 16 sectors of 4 data blocks each.

The MIFARE Classic 4KB product is divided in forty sectors. The first 32 sectors contain 4 data blocks and the 8 remaining sectors contain 16 data blocks.

The Fig 15 shows the memory map structure of the MIFARE Classic 1K chip.

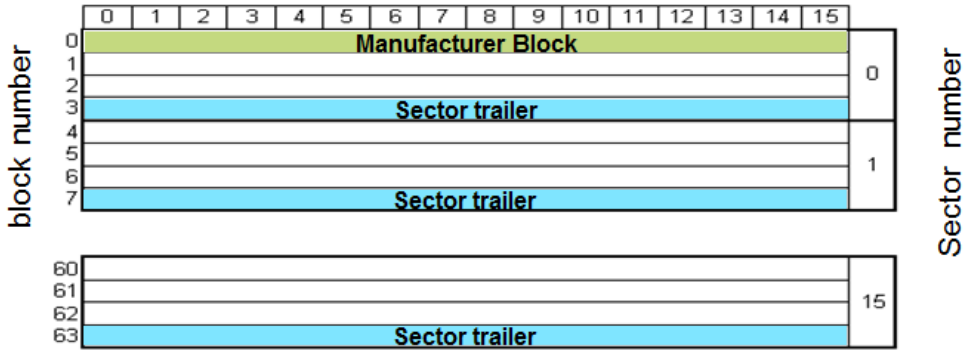


Fig 15. MIFARE Classic 1KB memory map

Block 0 of sector 0 is called the Manufacturer Block and contains the IC manufacturer data. This block is programmed during production and it cannot be changed afterwards.

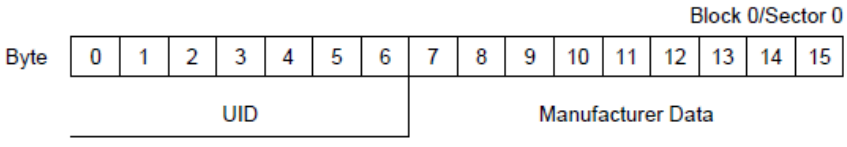


Fig 16. MIFARE Classic 7 Bytes UID Manufacturer Block

The last block of every sector is known as the sector trailer. The sector trailer defines the keys and access conditions of this sector. It holds the Secret Key A (bytes 0 to 5), the access conditions (bytes 6 to 9) and Secret Key B (optional) from bytes 10 to 15. Before any memory operation is performed in one block, the reader IC shall authenticate against its sector.

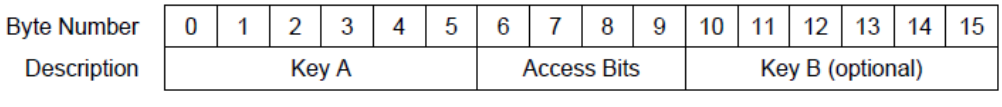


Fig 17. MIFARE Classic Sector Trailer

5.1.2 MIFARE Classic Parameter Structure

The NFC Reader Library defines a structure in order to store the parameters related to the MIFARE Classic operation. This structure is called `phalMfc_Sw_DataParams_t`.

```
typedef struct{
    void * pPalMifareDataParams;
    void * pKeyStoreDataParams;
} phalMfc_Sw_DataParams_t;
```

*** pPalMifareDataParams:** Pointer to the MIFARE parameter structure on the PAL layer.

***pKeyStoreDataParams:** Pointer to the Key Store parameter structure.

5.1.3 MIFARE Classic Component Initialization

This function initializes the MIFARE Classic component. It takes as inputs the MIFARE component that provides ISO/IEC 14443-3A and MIFARE specific services and the key store component that is needed for cryptographic operations.

```
phStatus_t phalMfc_Sw_Init(
                    phalMfc_Sw_DataParams_t * pDataParams,      [In]
                    uint16_t wSizeOfDataParams,                 [In]
                    void * pPalMifareDataParams,                [In]
                    void * pKeyStoreDataParams );               [In]
```

***pDataParams:** `Pointer to the phalMfc_Sw_DataParams_t parameter component.`

wSizeOfDataParams: `Size of the phalMfc_Sw_DataParams_t parameter component.`

***pPalMifareDataParams**: `Pointer to the underlying MIFARE PAL parameter component.`

***pKeyStoreDataParams:** `Pointer to the Key Store parameter structure.`

The values returned by the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_ERR_INVALID_DATA_PARAMS: wSizeOfDataParams does not agree with the defined size of A MIFARE Classic component.`

5.1.4 MIFARE Classic Authentication

This function authenticates the reader IC against a particular block of the MIFARE Classic IC. The cryptographic algorithm used to complete this authentication is the Crypto1, which is NXP proprietary. The algorithm shall be implemented on the underlying hardware component in the HAL layer.

Once authenticated, any subsequent operation on the blocks within the same sector is allowed.

```
phStatus_t phalMfc_Authenticate(
                    void * pDataParams,     [In]
                    uint8_t bBlockNo,       [In]
                    uint8_t bKeyType,       [In]
                    uint16_t wKeyNumber,    [In]
                    uint16_t wKeyVersion,   [In]
                    uint8_t *pUid,          [In]
                    uint8_t bUidLength );   [In]
```

***pDataParams:** Pointer to the MIFARE Classic AL layer data parameter structure.

bBlockNo: Block number to authenticate against.

bKeyType: Can be either `PHAL_MFC_KEYA` or `PHAL_MFC_KEYB`.

wKeyNumber: Key number used for authentication (position of the key in the Key Store)

***pUid:** Pointer to the card UID to authenticate against.

bUidLength: UID length. Only lengths 4, 7 or 10 are valid.

The values returned by the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_ERR_AUTH_ERROR`: Authentication procedure failed. The key used may not match with the key of a given block.

`PH_ERR_INVALID_PARAMETER`:

- `bKeyType` other than `PHAL_MFC_KEYA` or `PHAL_MFC_KEYB`.
- `wKeyNo` exceeds half of maximum possible number of keys in the EEPROM.

`PH_ERR_IO_TIMEOUT`: Authentication command itself did not succeeded while timeout from timer1 terminated.

5.1.5 PersonalizeUID

This function configures the UID for an specific personalization option, which has direct impact on the behavior during the anticollision and selection process. The execution of this command requires previous authentication against sector 0. Once this function has been executed and accepted by the card, the configuration is automatically locked.

Note: The configuration becomes effective after the card has been unselected or the field is reset.

```
phStatus_t phalMfc_PersonalizeUid(
                    void * pDataParams,     [In]
                    uint8_t bUidType );     [In]
```

***pDataParams:** Pointer to the `phalMfc_Sw_DataParams_t` parameter structure.

bUidType: It specifies the UID type.

- `PHAL_MFC_UID_TYPE_UIDF0`: Anticollision and selection with the double size UID according to ISO/IEC14443-3.
- `PHAL_MFC_UID_TYPE_UIDF1`: Anticollision and selection with the double size UID according to ISO/IEC 14443-3 and optional usage of a selection process shortcut.
- `PHAL_MFC_UID_TYPE_UIDF2`: Anticollision and selection with a single size random ID according to ISO/IEC14443-3.
- `PHAL_MFC_UID_TYPE_UIDF3`: Anticollision and selection with a single size NUID according to ISO/IEC 14443-3 where the NUID is calculated out of the 7-byte UID.

The values returned by the function can be:

`PH_ERR_SUCCESS`: Operation successful.
`Other`: Value returned by the underlying component.

5.1.6 MIFARE Classic Command Set

The command set supported by the MIFARE Classic component is:

5.1.6.1 Read

This function reads a MIFARE Classic data block and returns its 16 bytes of data.

```
phStatus_t phalMfc_ReadValue(
                    void * pDataParams,        [In]
                    uint8_t bBlockNo,          [In]
                    uint8_t * pBlockData);     [Out]
```

***pDataParams:** Pointer to the `phalMfc_Sw_DataParams_t` parameter structure.

bBlockNo: MIFARE Classic block number to be read.

***pBlockDataValue:** Pointer to a16 byte array where the read data is stored.

The returned values from the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_ERR_PROTOCOL_ERROR`:

- Other than 16 bytes are read from `the MIFARE Classic card value block.`
- Data or address bytes within the 16 byte reception buffer do not satisfy MIFARE block data rules.

`Other`: Value returned by the underlying component.

5.1.6.2 Read Value

Value blocks allow to perform electronic purse functions (read, write, increment, decrement, restore, transfer). The value blocks have a fixed data format, which permits error detection, correction and a backup management. A value block can only be generated through a write operation in the value block format.

- *Value*: It is a 4 byte signed value. For data integrity and security, a value is stored three times (twice non-inverted, one inverted).
- *Adr*: It is a 1 byte address, which can be used to save the storage address of a block. The address byte is stored four times, twice inverted and non-inverted.

Byte Number	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Description	value				$\overline{\text{value}}$				value				adr	$\overline{\text{adr}}$	adr	$\overline{\text{adr}}$

Fig 18. MIFARE Classic Value Block

The `phalMfc_ReadValue` function performs the MIFARE Classic Read function and, additionally, it verifies the robustness of the 16 bytes received according to the Value block format `by calling the phalMfc_Int_CheckValueBlockFormat()` function.

```
phStatus_t phalMfc_ReadValue(
                    void * pDataParams,        [In]
                    uint8_t bBlockNo,          [In]
                    uint8_t * pValue,          [Out]
                    uint8_t * pAddrData );     [Out]
```

***pDataParams:** Pointer to the `phalMfc_Sw_DataParams_t` parameter structure.

bBlockNo: Block number to be read.

***pValue:** Pointer to a 4 byte array containing the value read from the data block.

***pAddrData:** Pointer to one byte containing the address read from the data block.

The returned values from the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_ERR_PROTOCOL_ERROR`:

- Other than 16 bytes are read from the `MIFARE Classic card value block.`
- Data or address bytes within the 16 byte reception buffer do not satisfy MIFARE block data rules.

`Other`: Value returned by the underlying component.

5.1.6.3 Write

This function writes 16 bytes of data in a MIFARE Classic data block.

```
phStatus_t phalMfc_WriteValue(
                    void * pDataParams,     [In]
                    uint8_t bBlockNo,       [In]
                    uint8_t * pBlockData);  [In]
```

***pDataParams:** `Pointer to the phalMfc_Sw_DataParams_t parameter structure.`

bBlockNo: `MIFARE Classic block where the data is written.`

*** pBlockData:** `16 byte array of data containing the data to be written into the MIFARE Classic block.`

The returned values from the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`Other`: Value returned by the underlying component.

5.1.6.4 Write Value

The phalMfc_WriteValue function receives the 4 byte input value as an input and it creates the 16 byte formatted Value block structure (Fig 18) by internally calling the phalMfc_Int_CreateValueBlock() function. After that, it performs a Write operation.

```
phStatus_t phalMfc_WriteValue(
                    void * pDataParams,     [In]
                    uint8_t bBlockNo,       [In]
                    uint8_t * pValue,       [In]
                    uint8_t bAddrData );    [In]
```

***pDataParams:** `Pointer to the phalMfc_Sw_DataParams_t parameter structure.`

bBlockNo: `MIFARE Classic block where the data is written`

***pValue:** `4 byte array containing the data to be written.`

bAddrData: `One byte array containing the address to be written.`

The returned values from the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`Other`: Value returned by the underlying component.

5.1.6.5 Increment

The MIFARE Increment operation performs an addition operation on the value store in a certain Value Block and stores the result in a volatile memory.

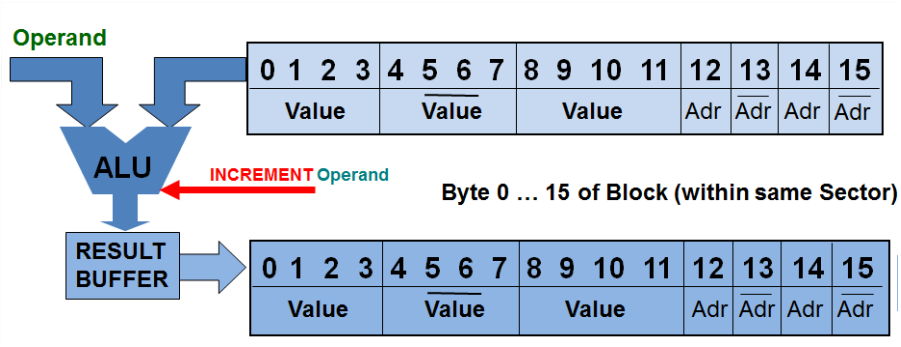


Fig 19. MIFARE Classic Increment operation

In case of loss of energy during one transaction, the value stored in the volatile memory will not be lost and not accessible anymore.

```
phStatus_t phalMfc_Increment(
                    void * pDataParams,     [In]
                    uint8_t bBlockNo,       [In]
                    uint8_t * pValue );     [In]
```

***pDataParams:** Pointer to the `phalMfc_Sw_DataParams_t data` parameter structure.

bBlockNo: Block number to be incremented.

***pValue:** 4 byte array containing the value (LSB first) to be incremented.

The returned values from the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`Other`: Value returned by the underlying component.

5.1.6.6 Decrement

The MIFARE Decrement operation performs a subtraction operation on the value stored in a certain Value Block and stores the result in a volatile memory.

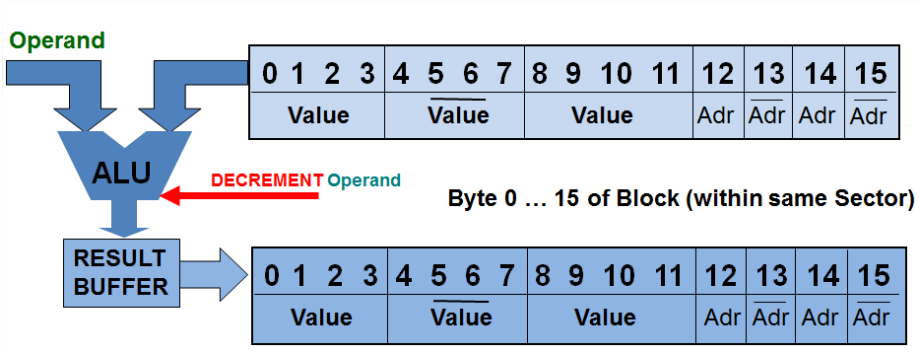


Fig 20. MIFARE Classic Decrement operation

In case of loss of energy during one transaction, the value stored in the volatile memory will not be accessible anymore.

```
phStatus_t phalMfc_Decrement(
                    void * pDataParams,     [In]
                    uint8_t bBlockNo,       [In]
                    uint8_t * pValue );     [In]
```

***pDataParams:** Pointer to `phalMfc_Sw_DataParams_t` parameter structure.

bBlockNo: Block number to be decremented.

***pValue:** 4 byte array containing the value (LSB first) to be decremented.

The values returned by the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`Other`: Value returned by the underlying component.

5.1.6.7 Restore

The Restore function copies the value of a certain Value Block into the volatile memory.

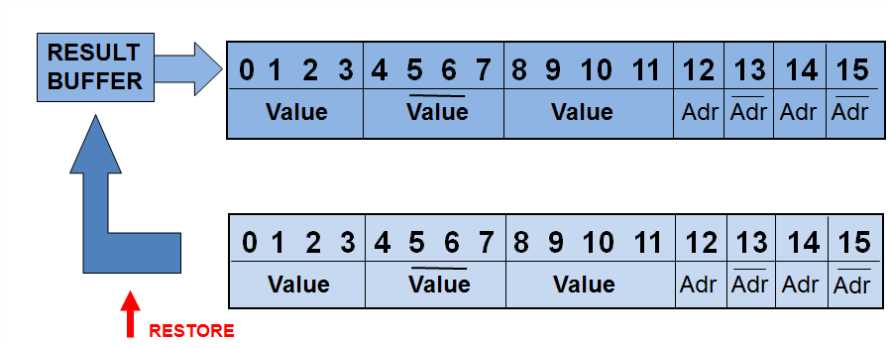


Fig 21. **MIFARE Classic Restore operation**

The function definition is:

```
phStatus_t phalMfc_Restore(
                    void * pDataParams,     [In]
```

uint8_t bBlockNo); [In]

***pDataParams:** Pointer to the `phalMfc_Sw_DataParams_t` parameter structure.

bBlockNo: Block number the transfer buffer shall be restored from.

The returned values from the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`Other`: Value returned by the underlying component.

5.1.6.8 Transfer

The Transfer function writes the value stored in the volatile memory into one MIFARE Classic block.

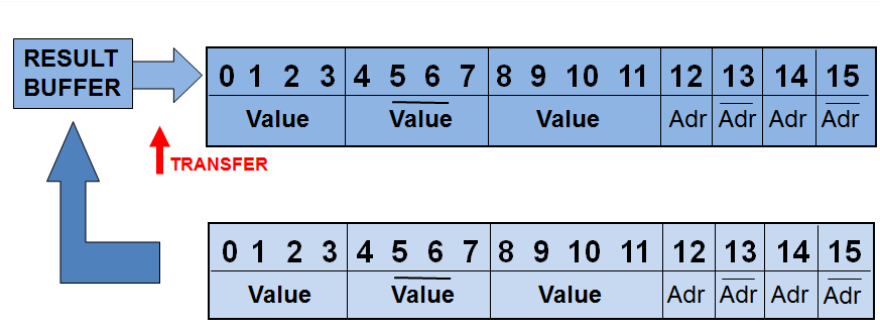


Fig 22. MIFARE Classic Transfer operation

The function definition is:

phStatus_t phalMfc_Transfer(
void * pDataParams, [In]
uint8_t bBlockNo); [In]

***pDataParams:** Pointer to the phalMfc_Sw_DataParams_t parameter structure.

bBlockNo: Bock number where the transfer buffer shall be transferred to.

The returned values from the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`Other`: Value returned by the underlying component.

5.1.6.9 Increment Transfer

This function executes both an Increment and a Transfer command. The value in the source block is copied into the volatile memory of the IC where it is incremented. The obtained value is then transferred to the destination Block.

phStatus_t phalMfc_IncrementTransfer(
*pDataParams, [In]
uint8_t bSrcBlockNo, [In]
uint8_t bDstBlockNo, [In]
uint8_t * pValue); [In]

***pDataParams:** Pointer to the phalMfc_Sw_DataParams_t parameter structure.

bSrcBlockNo: The value in this block is the one to be incremented. The incremented value is stored into this block until it is finally transferred to the destination block.

bDstBlockNo: The destination block number where the incremented value will be stored.

***pValue:** 4 byte array indicating the increment value.

The returned values from the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`Other`: Value returned by the underlying component.

5.1.6.10 Decrement Transfer

This function executes both a Decrement and a Transfer command. The value in the source block is copied into the volatile memory of the IC where it is decremented. The obtained value is then transferred to the destination Block.

```
phStatus_t phalMfc_DecrementTransfer(
                    void * pDataParams,     [In]
                    uint8_t bSrcBlockNo,    [In]
                    uint8_t bDstBlockNo,    [In]
                    uint8_t * pValue );     [In]
```

***pDataParams:** Pointer to the phalMfc_Sw_DataParams_t parameter structure.

bSrcBlockNo: The value in this block is the one to be decremented. The incremented value is stored into this block until it is finally transferred to the destination block.

bDstBlockNo: The destination block number where the decremented value will be stored.

***pValue:** 4 byte value that is value from source block decremented by.

The returned values from the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`Other`: Value returned by the underlying component.

5.1.6.11 Restore Transfer

This function executes both a Restore and a Transfer command respectively. The value in the source block is copied into the volatile memory and then it is stored into the destination Block.

```
phStatus_t phalMfc_RestoreTransfer(
                    void * pDataParams,     [In]
                    uint8_t bSrcBlockNo,    [In]
                    uint8_t bDstBlockNo );  [In]
```

***pDataParams:** Pointer to the MIFARE Classic AL layer data parameter structure.

bSrcBlockNo: Block number to be transferred to the buffer.

bDstBlockNo: Block number where the data is stored.

The returned values from the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`Other`: Value returned by the underlying component.

5.2 MIFARE Ultralight Family

5.2.1 Technical Introduction

5.2.1.1 MIFARE Ultralight

MIFARE Ultralight cards [5] are primarily designed for limited use applications such as public transportation or event ticketing. This product is designed to work on ISO/IEC 14443 Type A compliant environments.

The MIFARE Ultralight memory is organized in pages of 4 bytes. The MIFARE Ultralight card has 7-byte UID and it is stored in the first two pages. Bytes 2 and 3 of page 2 represent the field programmable read-only locking mechanism that allows users to individually lock each page by setting the corresponding bit to logic 1 to prevent further write access. Page 3 is the One-Time-Programmable (OTP) page. These bits can be written just once. The rest of the memory can be used by the users for data storage .

The Fig 23 depicts the reference memory map for MIFARE Ultraligth ICs. As it can be observed, the memory map size for MIFARE Ultralight cards is 64 bytes.

Page	Byte type	0	1	2	3
0	Serial Number	UID0	UID1	UID2	BCC0
1	Serial Number	UID3	UID4	UID5	UID6
2	Internal/lock bytes	BCC1	Internal	Lock0	Lock1
3	OTP bytes	OTP	OTP	OTP	OTP
4	Data read/write	Data	Data	Data	Data
5	Data read/write	Data	Data	Data	Data
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
15	Data read/write	Data	Data	Data	Data

Fig 23. MIFARE Ultralight memory map

MIFARE Ultralight does not implement any security features except the read-only locking mechanism to avoid further writing operations.

5.2.1.2 MIFARE Ultralight EV1

The MIFARE Ultralight EV1 is succeeding the MIFARE Ultralight IC and is fully functional backwards compatible. The MIFARE Ultralight EV1 IC memory size is either 80 bytes or 164 bytes.

MIFARE Ultralight EV1 ICs implement additional functionalities regarding security.

- ECC based originality signature for IC manufacturing check.
- 32-bit password protection to prevent unauthorized access.
- 3 independent 24-bit one-way counters.

For further information regarding these features and how to use them in the NFC Reader Library, please refer to the MIFARE Ultralight EV1 dedicated API in section 5.2.5.

5.2.1.3 MIFARE Ultralight C

MIFARE Ultralight C enhances security features of the MIFARE Ultralight family and is fully functional backwards compatible. The MIFARE Ultralight C IC memory size is 192 bytes.

MIFARE Ultralight C implements an optional 3DES authentication to prevent unauthorized memory operations.

For further information regarding these features and how to use them in the NFC Reader Library, please refer to the MIFARE Ultralight C dedicated API in section 5.2.6.

5.2.2 MIFARE Ultralight Parameter Structure

A special structure is defined in the NFC Reader Library in order to store the parameters related to the MIFARE Ultralight operation. This structure is called phalMful_Sw_DataParams_t.

Note: the same parameter structure is used for both MIFARE Ultralight EV1 and MIFARE Ultralight C.

```
typedef struct{
   void * pPalMifareDataParams;
   void * pKeyStoreDataParams;
   void * pCryptoDataParams;
   void * pCryptoRngDataParams;
} phalMful_Sw_DataParams_t;
```

*** pPalMifareDataParams:** Pointer to the MIFARE parameter structure on the PAL layer.

***pKeyStoreDataParams:** Pointer to the Key Store parameter structure.

***pCryptoDataParams**: Pointer to the phCrypto data parameter component (Only available on the NXP Export Controlled version).

***pCryptoRnqDataParams**: Pointer to the CryptoRng parameter component (Only available on the NXP Export Controlled version).

5.2.3 MIFARE Ultralight Component Initialization

The following function initializes MIFARE Ultralight AL component.

```
phStatus_t phalMfu_Sw_Init(
                    phalMful_Sw_DataParams_t * pDataParams,     [In]
                    uint16_t wSizeOfDataParams,                 [In]
                    void * pPalMifareDataParams,                [In]
                    void * pKeyStoreDataParams,                 [In]
                    void * pCryptoDataParams,                   [In]
                    void * pCryptoRngDataParams );              [In]
```

***pDataParams:** Pointer to the `phalMful_Sw_DataParams_t` parameter structure.

wSizeOfDataParams: Specifies the size parameter structure.

***pPalMifareDataParams:** Pointer to the `palMifare` parameter structure.

***pKeyStoreDataParams:** Pointer to the `phKeystore` parameter structure.

***pCryptoDataParams:** Pointer to the `phCrypto` data parameters structure.

`*`**pCryptoRngDataParams:** Pointer to the parameter structure of the `CryptoRng` layer.

The values returned by the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_ERR_INVALID_DATA_PARAMS`: wSizeOfDataParams does not agree with defined size of MFUL component.

5.2.4 MIFARE Ultralight Command Set

This section explains MIFARE Ultralight operations that can be performed on a MIFARE Ultralight IC memory. Since MIFARE Ultralight EV1 and MIFARE Ultralight C have been designed to be fully backwards compatible, functions included in this section can also be executed on MIFARE Ultralight EV1 and MIFARE Ultralight C cards.

5.2.4.1 Read

The MIFRE Ultralight Read command reads 16 bytes (4 pages) of data starting from the page address passed in the function.

```
phStatus_t phalMful_Read(
                    void * pDataParams,     [In]
                    uint8_t bAddress,       [In]
                    uint8_t * pData );      [Out]
```

***pDataParams:** Pointer to the `phalMful_Sw_DataParams_t data` parameter structure.

bAddress: Indicates the page on the card to start reading from. If it is out of range, MFUL returns NAK.

***pData:** Pointer to 16 byte data array containing the data read from the MIFARE Ultralight card.

The values returned by the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_ERR_PROTOCOL_ERROR`: Number of received data differs from 16 bytes.

`Other`: Value returned by the underlying component.

5.2.4.2 Write

The MIFARE Ultralight Write command writes 4 bytes (1 page) to the addressed memory page.

```
phStatus_t phalMful_Write(
      void * pDataParams,                   [In]
      uint8_t bAddress,                     [In]
      uint8_t * pData );                    [In]
```

***pDataParams:** Pointer the `phalMful_Sw_DataParams_t` parameter structure.

bAddress: Card page to write into.

***pData:** Pointer to 4 byte data array containing data to be written.

The values returned by the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`Other`: Value returned by the underlying component.

5.2.4.3 Compatibility Write

This function performs MIFARE Ultralight Compatibility-Write command. The Compatibility-Write command was implemented to accommodate the established MIFARE reader infrastructure. Even though 16 bytes are transferred to the MIFARE Ultralight, only the least significant 4 bytes (bytes 0 to 3) will be written to the specified address.

```
phStatus_t phalMful_CompatibilityWrite(
      void * pDataParams,                                [In]
      uint8_t bAddress,                                  [In]
      uint8_t * pData );                                 [In]
```

***pDataParams:** Pointer to the `phalMful_Sw_DataParams_t` parameter structure.

bAddress: Page on MIFARE Ultralight to write into.

***pData:** Pointer to 16 byte data array containing data to be written.

The values returned by the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`Other`: Value returned by the underlying component.

5.2.5 MIFARE Ultralight EV1 Command Set

The MIFARE Ultralight EV1 API extends the MIFARE Ultralight API in order to provide the means to handle MIFARE Ultralight EV1 additional features, such as the new security constraints that are explained in section 5.2.1.2.

5.2.5.1 Increment count

This function increments one of the three independent 24-bit one-way counters. These counters are located out of the NVM memory of the MIFARE Ultralight card; therefore they are not writable using MIFARE Ultralight WRITE commands.

The counters can be incremented by an arbitrary value. The increment value is valid immediately and does not require a RF reset or re-activation. Once counter value reaches FFFFFFh and an increment is performed via a valid INC_CNT command, MIFARE Ultralight replies a NAK.

The `phalMful_IncrCnt()` function features anti-tearing support, thus no undefined values originating from interrupted programing cycles are possible. The occurrence of a tearing event can be checked using the `phalMful_ChkTearingEvent()` function.

```
phStatus_t phalMful_IncrCnt(
      void * pDataParams,                                [In]
      uint8_t bCntNum,                                   [In]
      uint8_t * pCnt );                                  [In]
```

***pDataParams:** Pointer to the `phalMful_Sw_DataParams_t` parameter structure.

bCntNum: Identifier of the counter to be incremented. Values from 00 to 02.

*** pCnt**: Increment value (LSB). The input value shall be 4 bytes. However, only the first three data bytes are considered, the fourth byte is ignored.

The values returned by the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`Other`: Value returned by the underlying component.

5.2.5.2 Read Count

This function retrieves the current value of one of the three independent 24-bit one-way counters. These counters are located out of the NVM memory of the MIFARE Ultralight card; therefore they are not readable using MIFARE Ultralight READ commands.

```
phStatus_t phalMful_ReadCnt(
      void * pDataParams,                                [In]
      uint8_t bCntNum,                                   [In]
      uint8_t * pCntValue  );                            [Out]
```

***pDataParams:** Pointer to the `phalMful_Sw_DataParams_t` parameter structure.

bCntNum: Identifier of the counter to be read. Values from 00 to 02.

*** pCnt**: Retrieved counter value (LSB). 3 bytes are received.

The values returned by the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_ERR_PROTOCOL_ERROR`: Length of received data differs from 24 bytes.

`Other`: Value returned by the underlying component.

5.2.5.3 Check Tearing Event

The tearing event command allows the contactless reader to check whether a tearing event happened on a specific counter during its update. If tearing is detected, the developer should process accordingly.

```
phStatus_t phalMful_ChkTearingEvent(
      void * pDataParams,                                [In]
      uint8_t bCntNum,                                   [In]
      uint8_t * pValidFlag );                            [Out]
```

***pDataParams:** Pointer to the `phalMful_Sw_DataParams_t` parameter structure.

bCntNum: Identifier of the counter to be checked. Values from 00 to 02.

***pDataParams:** One byte address containing the valid flag byte.

The values returned by the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`Other`: Value returned by the underlying component.

5.2.5.4 Password Authentication

MFARE Ultralight EV1 defines an optional 32-bit password protection to prevent unauthorized memory operations into configurable parts of the memory. This function

authenticates against the MIFARE Ultralight EV1 card in order to be able to complete read/write operations.

```
phStatus_t phalMful_PwdAuth(
      void * pDataParams,                                [In]
      uint8_t * pPwd,                                    [In]
      uint8_t * pPack );                                 [Out]
```

***pDataParams:** Pointer to the `phalMful_Sw_DataParams_t` parameter structure.

*** pPwd:** 4-byte array containing the password.

*** pPack:** 2-byte array that returns the password acknowledge bytes.

The values returned by the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`Other`: Value returned by the underlying component.

5.2.5.5 Get Version

This function is used for retrieving information regarding MIFARE Ultralight EV1 IC. It provides manufacturer data, product version and the storage size information.

```
phStatus_t phalMful_GetVersion(
      void * pDataParams,                                [In]
      uint8_t * pVersion );                              [Out]
```

***pDataParams:** Pointer to the `phalMful_Sw_DataParams_t` parameter structure.

*** pVersion**: 8-byte array containing manufacturer, product version and storage size information.

The values returned by the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`Other`: Value returned by the underlying component.

5.2.5.6 Fast Read

The Fast Read functionality of MIFARE Ultralight EV1 offers the possibility to read the desired number of pages in the same command.

```
phStatus_t phalMful_FastRead(
      void * pDataParams,                                [In]
      uint8_t  bStartAddr,                               [In]
      uint8_t bEndAddr,                                  [In]
      uint8_t ** pData,                                  [Out]
      uint16_t * wNumBytes );                            [Out]
```

***pDataParams:** Pointer to the `phalMful_Sw_DataParams_t` parameter structure.

bStartAddr: byte address that specifies the first block position to read.

bEndAddr: byte address that specifies the last block position to read.

**** pData:** Pointer to the data read from the card.

*** wNumBytes:** Value that indicates the number of bytes read from the card.

The values returned by the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_ERR_PROTOCOL_ERROR`: Length of received data is not (`bEndAddr - bStartAddr) * 4`.

`Other`: Value returned by the underlying component.

5.2.5.7 Read Signature

MIFARE Ultralight EV1 implements a cryptographically supported originality check which relies on the elliptic curve cryptography (ECC) asymmetric algorithm. With this feature, it is possible to verify with a certain probability, that the ticket is using an NXP Semiconductors manufactured silicon.

This signature can be retrieved using the READ_SIG command and can be verified using the corresponding ECC public key in the PCD.

The originality signature is programmed during chip production and cannot be changed afterwards.

```
phStatus_t phalMful_ReadSign(
      void * pDataParams,                                [In]
      uint8_t bAddr,                                     [In]
      uint8_t ** pSignature );                           [Out]
```

***pDataParams:** Pointer to the `phalMful_Sw_DataParams_t` parameter structure.

bAddr: This value shall be always set to 00.

****pSignature:** 32-byte signature to be used for the originalty check.

The values returned by the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_ERR_PROTOCOL_ERROR`: Length of the retrieved signature is not 32-byte.

`Other`: Value returned by the underlying component.

5.2.6 MIFARE Ultralight C Command Set

The MIFARE Ultralight C API extends the MIFARE Ultralight API in order to provide the means to handle MIFARE Ultralight C additional features, which basically includes the optional 3DES based authentication procedure as is explained in 5.2.1.3.

5.2.6.1 Authenticate

This function executes the MIFARE Ultralight C authentication, which is based on 3DES symmetric cryptography algorithm. Therefore, CRYPTO SYM module must be enabled in order to make use of this function.

The key to be used for the authentication shall have previously been stored on the underlying contactless reader hardware.

Note: this functionality is only available in the NXP Export Controlled version.

```
phStatus_t phalMful_UlcAuthenticate(
      void * pDataParams,                                [In]
      uint16_t wKeyNumber,                               [In]
      uint16_t wKeyVersion );                            [In]
```

***pDataParams:** Pointer to the `phalMful_Sw_DataParams_t` parameter structure.

wKeyNo: `Number identifier of the key to use for the authentication.`

wKeyVersion: `Version of the key used for the authentication.`

The values returned by the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`Other`: Value returned by the underlying component.

5.3 MIFARE DESFire

5.3.1 Technical Introduction

MIFARE DESFire EV1 is a Common Criteria (EAL4+) certified product. MIFARE DESFire [8] is based on open global standards for both air interface and cryptographic methods. It is fully compliant with ISO/IEC 14443A-4 layer and provides the option to use a set of ISO/IEC 7816-4 commands.

MIFARE DESFire EV1 is available in three memory sizes: 2, 4 or 8 Kbytes and can hold up to 28 different applications and 32 files per application. Every application is represented by its 3 bytes Application Identifier (AID) (AID 0x000000 is reserved for the card Master level). Each file ID is represented by one byte, values from 0x00 to 0x1F.

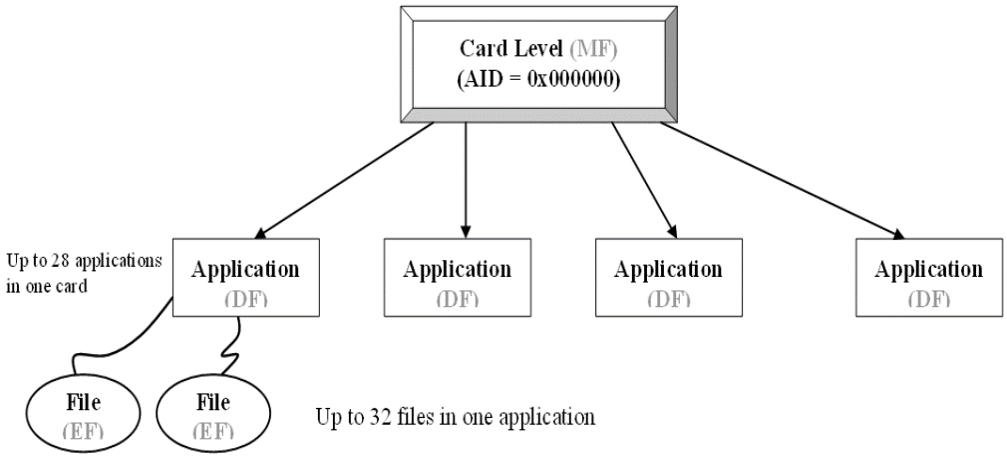


Fig 24. Figure title here

A file can only be selected after the dedicated application where it is hold has been selected. Once within the application, files can be accessed using the files' IDs. Five different types of files can be created within each application:

- Standard Data File: Data file normally used for storing static data (e.g: name).
- Back up Data File: Data file normally used for storing dynamic data.
- Value File: Data file normally used for storing numeric values.
- Linear Record File: Data file normally used for log or record transactions.
- Cyclic Record File: Data file normally used for log or record transactions.

Note: Cyclic record files can store *Total number of records*-1 unique values.

MIFARE DESFire cryptographic methods can be independently attributed to each application. Applications can define up to 14 keys, plus the “free” and “never” key. MIFARE DESFire EV1 supports three types of crypto algorithms:

- TDES (Triple DES, 16-byte key length)
- 3KTDES (Three-key Triple DES, 24-byte key length)
- AES (Standard AES-128, 16-byte key length)

In addition, the communication settings are used to set three different communication modes.

- Plain: No encryption used
- Encrypted: DES, TDES or AES is used to encrypt the transferred data
- MACed: The data is transferred in plain, but a four or eight bytes message authentication code is added to the message. The MAC/CMAC is calculated using the crypto performed in the authentication.

The access rights and communication settings are attributed to the file level. At file creation both the access rights and the communication setting have to be specified. This means, depending on the file type, that a certain operation (e.g: read operation) can be linked to a certain key.

Note: The NFC Reader Library only includes the MIFARE DESFire commands which are not under Export controlled regulations. For the full MIFARE DESFire command set, please refer to the NXP Export Controlled Library [2].

5.3.2 MIFARE DESFire Parameter Structure

A special structure has been defined in the NFC Reader Library in order to store the parameters related to the MIFARE DESFire operation. This structure has been called `phalMfdf_Sw_DataParams_t`.

```
typedef struct{
    void * pPalMifareDataParams;
    void * pKeyStoreDataParams;
    void * pCryptoDataParamsEnc;
    void * pCryptoRngDataParams;
    void * pHalDataParams;
    uint8_t bSessionKey[24];
    uint8_t bKeyNo;
    uint8_t bIv[16];
    uint8_t bAuthMode;
    uint8_t pAid[3];
    uint8_t bCryptoMethod;
    uint8_t bWrappedMode;
    uint16_t wCrc;
    uint32_t dwCrc;
    uint16_t wAdditionalInfo;
    uint16_t wPayLoadLen;
    uint8_t bLastBlockBuffer[16];
    uint8_t bLastBlockIndex;
} phalMfdf_Sw_DataParams_t;
```

*** pPalMifareDataParams:** Pointer to the MIFARE data parameter component on the PAL layer.

***pKeyStoreDataParams:** Pointer to the Key Store data parameter component.

***pCryptoDataParams**: Pointer to the phCrypto data parameter component (Only available on the NXP Export Controlled version).

***pCryptoRnqDataParams**: Pointer to the CryptoRng data parameter component (Only available on the NXP Export Controlled version).

***pHalDataParams**: Pointer to the HAL data parameter component.

bSessionKey[24]: Session key for this authentication

bKeyNo: Key used for the mutual three pass authentication procedure.

bIV: MIFARE DESFire Initialization Vector. Its maximum size is 16-bytes. The Initialization Vector is updated for each transaction.

bAuthMode: Type of authentication used (Authenticate 0x0A, AuthISO 0x1A, AuthAES 0xAA).

pAid: AID of the current selected application

bCryptoMethod: DES , 3DES, 3KDES or AES crypto methods.

bWrappedMode: Wrapped APDU mode. All native commands are wrapped into ISO 7816 APDUs.

wCrc: 2-Byte CRC initial value in Authenticate mode.

dwCrc: 4-Byte CRC initial value in AuthISO and AutheAES mode.

wAdditionalInfo: Specific error codes for MIFARE DESFire functions.

wPayLoadLen: The amount of data to be read. Required to verify the CRC.

bLastBlockBuffer[16]: Buffer used to store the last block of encrypted data in case of chaining.

bLastBlockIndex: Last Block buffer index read during a transaction.

5.3.3 MIFARE DESFire Component Initialization

The following function initializes MIFARE DESFire AL component.

```
phStatus_t phalMfdf_Sw_Init(
      phalMfdf_Sw_DataParams_t * pDataParams,
      uint16_t wSizeOfDataParams,                    [In]
      void * pPalMifareDataParams,                   [In]
      void * pKeyStoreDataParams,                    [In]
      void * pCryptoDataParamsEnc,                   [In]
      void * pCryptoRngDataParams,                   [In]
      void * pHalDataParams );                       [In]
```

***pDataParams**: Pointer to the MIFARE DESFire data parameter component.

wSizeofDataParams: Specifies the size of the data parameter component.

***pPalMifareDataParams**: Pointer to the MIFARE PAL data parameter component.

***pKeyStoreDataParams**: Pointer to the KeyStore data parameter component.

***pCryptoDataParamsEnc**: Pointer to the Crypto Component context for encryption

***pCryptoRnqDataParams**: Pointer to the CryptoRng data parameter component (Only available on the NXP Export Controlled version).

***pHalDataParams**: Pointer to the HAL data parameter component.

The returned values from the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_ERR_INVALID_DATA_PARAMS`: `wSizeOfDataParams` does not agree with defined size of MIFARE DESFire component.

5.3.4 MIFARE DESFire Command Set – Non-export controlled commands.

The MIFARE DESFire commands available in the NFC Reader Library are detailed and explained in this section. For the complete command set, please refer to the NXP Export controlled reference.

5.3.4.1 Create Application

The Create application command allows to create new application on the MIFARE DESFire card. An application is defined by an Application ID (AID), which is implemented as a 3 byte number. The AID 0x000000 is reserved as reference to the card root level.

```
phStatus_t phalMfdf_CreateApplication(
      void * pDataParams,                                [In]
      uint8_t bOption,                                   [In]
      uint8_t * pAid,                                    [In]
      uint8_t bKeySettings1,                             [In]
      uint8_t bKeySettings2,                             [In]
      uint8_t * pISOFileId,                             [In]
      uint8_t * pISODFName,                              [In]
      uint8_t bISODFNameLen );                           [In]
```

***pDataParams**: Pointer to the MIFARE DESFire data parameter component.

bOption: Option field that indicates whether this application has ISO File IDs and DF names.

***pAid**: The Application AID. The AID shall be 3 bytes long.

bKeySettings1: Stores the Application Master Key Settings.

bKeySettings2: Stores and defines several settings such as the number of keys that can be stored within the application for cryptographic purposes, the use or not of ISO Field IDs and the crypto method of the application.

***pISOFileId:** The ISO File ID of the application to be created. The ISO File IDs are used for ISO/IEC 7816-4 file systems. This parameter is used to select the application using the ISO Select command.

***pISODFNAme**: The Dedicated File Name (DF-Name) of the Application. The Application can be referenced using the DF-Name. Any DF name shall be coded on 1 to 16 bytes and each DF name shall be unique within the given card.

bISODFNameLen: The size of the ISO DF-Name.

The returned values from the function can be:

`PH_ERR_SUCCESS`: Operation successful.

PH_ERR_INVALID_DATA_PARAMS: Some of the parameters given to the function do not match with the expected variables.

Other: Value returned by the underlying component.

5.3.4.2 Select Application

The Select Application command allows to select one specific application for further access. If the AID 0x000000 is indicated, the card level is selected.

```
phStatus_t phalMfdf_SelectApplication(
      void * pDataParams,                                [In]
      uint8_t * pAid );                                  [In]
```

***pDataParams**: Pointer to the MIFARE DESFire data parameter component.

***pAid**: The AID of the application to be selected.

The returned values from the function can be:

PH_ERR_SUCCESS: Operation successful.

Other: Value returned by the underlying component.

5.3.4.3 Get Version

The Get Version command returns manufacturing related data of the card. Three frames of manufacturing related data are returned. The first frame contains hardware related information, the second frame contains software related information and the third frame returns the unique serial number, batch number and date of production of the IC

```
phStatus_t phalMfdf_GetVersion(
      void * pDataParams,                                [In]
      uint8_t * pVerInfo );                              [Out]
```

***pDataParams**: Pointer to the MIFARE DESFire data parameter component.

***pVerInfo**: The 28 bytes version data returned by the card.

The returned values from the function can be:

PH_ERR_SUCCESS: Operation successful.

PH_ERR_PROTOCOL_ERROR: The received response violates the contactless protocol.

Other: Value returned by the underlying component.

5.3.4.4 Create Standard Data File

The Create Standard data File command is used to create files for the storage of plain user data within an existing application on the card.

```
phStatus_t phalMfdf_CreateStdDataFile(
      void * pDataParams,                                [In]
      uint8_t bOption,                                   [In]
      uint8_t bFileNo,                                   [In]
      uint8_t * pISOFileId,                              [In]
      uint8_t bCommSett,                                 [In]
      uint8_t * pAccessRights,                           [In]
      uint8_t * pFileSize );                             [In]
```

***pDataParams**: Pointer to the MIFARE DESFire data parameter component.

bOption: Option parameter. The 0x00 value means that pISOFileID is not provided. The 0x01 value means that pISOFileID is provided and valid.

bFileNo: The file number of the new file to be created within the range of 0x00 to 0x1F. If a file with the same specified number already exists within the current application, an error is returned.

***pISOFileId**: The ISO File ID of the file to be created. The ISO File IDs are used for ISO/IEC 7816-4 file systems. This parameter is used to select the file using the ISO Select command.

bCommSett: It defines the communication settings. This settings define the level of security for the communication between the card and the reader IC. There are three options: Plain communication, Plain communication secured by MACing, or Enciphered communication.

***pAccessRights**: Two byte field defining the access rights for the new file (Read, Read&Write or Write access).

***pFileSize**: The size of the file to be created.

The returned values from the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_ERR_INVALID_DATA_PARAMS`: Some of the parameters given to the function do not match with the expected variables.

`Other`: Value returned by the underlying component.

5.3.4.5 Write Data

The Write Data command allows to write data in Standard Data Files and Backup Data files.

```
phStatus_t phalMfdf_WriteData(
      void * pDataParams,                                [In]
      uint8_t bCommOption,                               [In]
      uint8_t bFileNo,                                   [In]
      uint8_t * pOffset,                                 [In]
      uint8_t * pTxData,                                 [In]
      uint8_t * pTxDataLen );                            [In]
```

***pDataParams**: Pointer to the MIFARE DESFire data parameter component.

bCommOption: It defines the communication settings. This settings define the level of security for the communication between the card and the reader IC. There are three options: Plain communication, Plain communication secured by MACing or Enciphered communication.

bFileNo: The file number where the data will be written.

***pOffset**: It specifies a certain offset (bytes) before starting writing the data into the file. This can be used to start writing on a certain point of the file and not to overwrite previous data.

***pTxData**: The data to be written in the file.

***pTxDataLen**: The length of the data to be written in the file.

The returned values from the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_ERR_INVALID_DATA_PARAMS`: Some of the parameters given to the function do not match with the expected variables.

`PH_ERR_UNSOPORTED_PARAMETER`: An introduced parameter is not supported by the function.

`Other`: Value returned by the underlying component.

5.3.4.6 ISO Select File

The ISO Select File selects a certain file in order to perform further operations. The registered ISO AID of the MIFARE DESFire card is “0xD2780000850100”. When selecting this application, the MIFARE DESFire is selected. This command is implemented in compliance with ISO/IEC 7816-4 standard.

```
phStatus_t phalMfdf_IsoSelectFile(
      void * pDataParams,                          [In]
      uint8_t bOption,                             [In]
      uint8_t bSelector,                           [In]
      uint8_t * pFid,                              [In]
      uint8_t * pDFname,                           [In]
      uint8_t bDFnameLen,                          [In]
      uint8_t ** ppFCI,                            [Out]
      uint16_t * pwFCILen );                       [Out]
```

***pDataParams**: Pointer to the MIFARE DESFire data parameter component.

bOption: This variable can take two values. If bOption is equal to 0x00, the FCI is returned. If bOption is equal to 0x0C, the FCI is not returned

bSelector:: ISO Select mechanism. The value 0x00 is the ISO Select by the ISO File ID. The 0x02 value is the ISO Select by EF (Elementary file). The 0x04 value is the ISO Select by DF Name.

***pFid**: The 2 bytes ISO File ID to be selected. The LSB is sent first.

***pDFname**: The Dedicated File Name (DF-Name) of the application to be selected.

bDFnameLen: The length of the DF-name of the application to be selected.

****ppFCI**: File control information. The FCI is the byte string available in the response to the Select command.

***pwCILen:** Length of the file control information returned.

The returned values from the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_ERR_INVALID_DATA_PARAMS`: Some of the parameters given to the function do not match with the expected variables.

`Other`: Value returned by the underlying component.

5.3.4.7 ISO Read Binary

The ISO Read Binary command reads a certain number of bytes from a file. This command is implemented in compliance with ISO/IEC 7816-4 standard.

```
phStatus_t phalMfdf_IsoReadBinary(
      void * pDataParams,                                [In]
      uint16_t wOption,                                  [In]
      uint8_t bOffset,                                   [In]
      uint8_t bSfid,                                     [In]
      uint8_t bBytesToRead,                              [In]
      uint8_t ** ppRxBuffer,                             [Out]
      uint16_t * pBytesRead );                           [Out]
```

***pDataParams**: Pointer to the MIFARE DESFire data parameter component.

wOption: This field allows to set the data exchange mode between the card and the reader IC. It can take two values: `PH_EXCHANGE_DEFAULT` or `PH_EXCHANGE_RXCHAINING`.

bOffset: It specifies a certain offset (bytes) before starting reading the data of the file. This can be used to start reading on a certain point of the file.

bSfid: Short ISO File ID. This field is one byte length and it is used to uniquely identify the Elementary File (EF).

bBytesToread: The number of bytes to be read. If this value is equal to zero, means that the entire file shall be read.

****ppRxBuffer**: The buffer where the read bytes are stored.

***pBytesReader**: Number of bytes that have been read.

The returned values from the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_ERR_INVALID_DATA_PARAMS`: Some of the parameters given to the function do not match with the expected variables.

`Other`: Value returned by the underlying component.

5.3.4.8 ISO Update Binary

The ISO Update Binary command updates the bits already present in a file with the bits given in the `pData` buffer. This command is implemented in compliance with ISO/IEC 7816-4

```
phStatus_t phalMfdf_IsoUpdateBinary(
      void * pDataParams,                                [In]
      uint8_t bOffset,                                   [In]
      uint8_t bSfid,                                     [In]
      uint8_t * pData,                                   [In]
      uint8_t bDataLen );                                [In]
```

***pDataParams**: Pointer to the MIFARE DESFire data parameter component.

bOffset: Specifies a certain offset (bytes) before starting writing the data into the file. This can be used to start writing on a certain point of the file and not to overwrite previous data.

bSfid: Short ISO File ID. This field is one byte length and it is used to uniquely identify the Elementary File (EF).

***pData**: The data to be written into the file.

bDataLen: The length of the data to be written.

The returned values from the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_ERR_INVALID_DATA_PARAMS`: Some of the parameters given to the function do not match with the expected variables.

`Other`: Value returned by the underlying component.

5.4 FeliCa

5.4.1 Technical Introduction

FeliCa is a contactless smart card system developed by Sony. Mainly used in Japan, and other countries such as Singapore. Primarily designed for electronic money cards, FeliCa is nowadays widely used for different contactless services such as transportation, access control and others.

The basic information unit used for the management of FeliCa cards is the block. Each block has a fixed size of 16 bytes, and the total number of blocks on a certain chip depends on the hardware IC. FeliCa supports simultaneous operations of up to 8 blocks.

Blocks are not addressed directly but relative to the service they belong to. Blocks are defined within services that serve as files. These services are organized in areas that serve as logical folders. Finally, these areas are structured in systems, which are the normative unit to be handled and serve as logical cards. More than one system can exist within a card.

Services, areas and systems are identified by their respective service codes, area codes and system codes, which are all 2 bytes codes.

The Fig 25 depicts a sample FeliCa card memory map using DES cryptography.

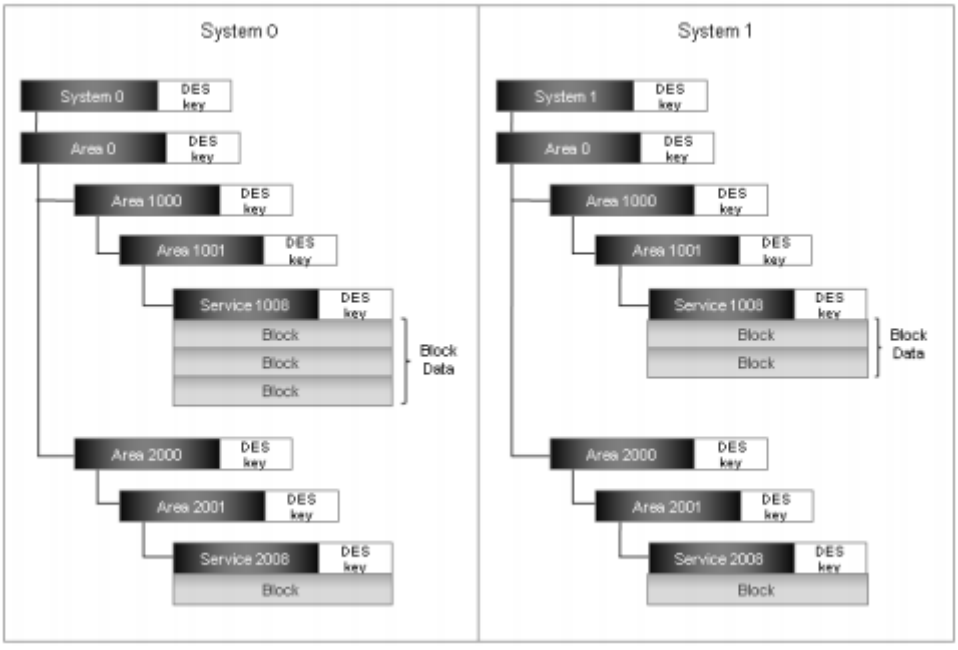


Fig 25. FeliCa memory map

5.4.2 FeliCa Parameter Structure

The FeliCa AL component defines a parameter structure that is used for the exchange of Application level commands with the Felica Card.

```
typedef struct {
      void * pPalFelicaDataParams;
      uint16_t wAdditionalInfo;
} phalFelica_Sw_DataParams_t;
```

*** pPalFelicaDataParams:** Pointer to the FeliCa parameter structure on the PAL layer.

wAdditionalInfo: It stores the last error code received from the FeliCa card.

5.4.3 FeliCa Component Initialization

The FeliCa AL component is initialized using the `phalFelica_Sw_Init()` function. The initialization function takes the FeliCa PAL and FeliCA AL parameter structures as inputs. The FeliCa AL component shall be initialized before its API can be used.

```
phStatus_t phalFelica_Sw_Init(
      phalFelica_Sw_DataParams_t * pDataParams,              [In]
      uint16_t wSizeOfDataParams,                            [In]
      void * pPalFelica_DataParams );                        [In]
```

*** pDataParams**: `Pointer to the FeliCa AL parameter component.`

wSizeOfDataParams: Size of the `phalFelica_Sw_DataParams_t parameter component.`

*** pPalFelica_DataParams**: Pointer to the underlying FeliCa PAL parameter component.

The values returned by the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`Other: Value returned by the underlying component.`

5.4.4 FeliCa Command Set

The JIS X 6319-4 specification defines a set of commands to manage FeliCa cards. Different commands can be executed depending on the mode the card is.

The NFC Reader Library implements the commands defined in mode 0, which is the mode that does not require authentication. For the complete set of commands, please refer to the Export Controller version of the NFC Reader Library [2].

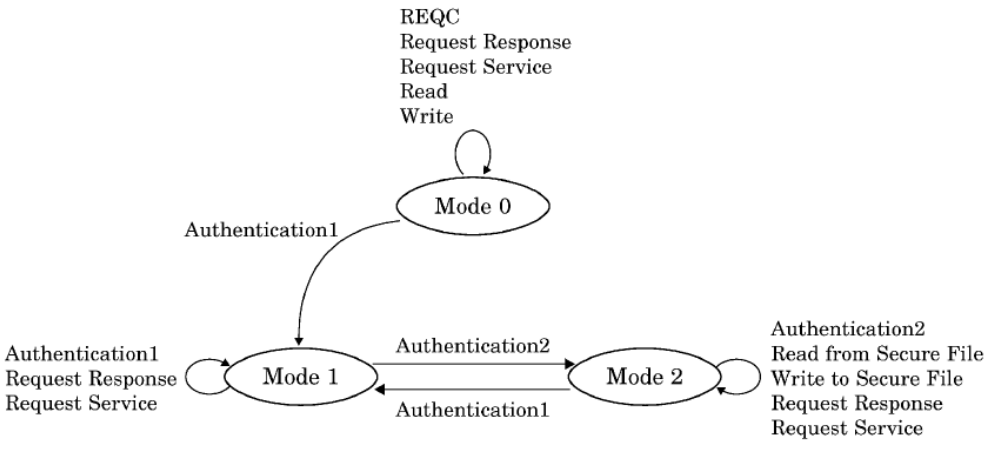


Fig 26. FeliCa commands

5.4.4.1 Request Response

The Request Response command is used to retrieve the current FeliCa card mode.

```
phStatus_t phalFelica_RequestResponse (
      void *pDataParams,                                    [In]
      uint8_t *pMode );                                     [Out]
```

***pDataParams**: Pointer to the FeliCa parameter component phalFelica_Sw_DataParams_t.

***pMode**: Current mode on which the card is running: 0, 1 or 2.

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

Other: Value returned by the underlying component.

5.4.4.2 Request Service

The Request Service command is used to verify the existence of an area or a service. If it exists, the Key Version associated to this area or service is acquired. Up to 16 areas or services can be processed at a time.

If the specified area or service does not exist, the card shall return 0xFFFF.

```
phStatus_t phalFelica_RequestService (
      void *pDataParams,                                    [In]
      uint8_t bTxNumServices,                               [In]
      uint8_t pTxServiceList,                               [In]
      uint8_t *pRxNumServices,                              [Out]
      uint8_t *pRxServiceList );                            [Out]
```

***pDataParams**: Pointer to the FeliCa parameter component phalFelica_Sw_DataParams_t.

bTxNumServices: Number of services or areas for which the key version is being consulted in this command.

*** pTxServiceList**: List with the service codes or area codes for which the key version is being consulted message.

*** pRxNumServices**: Number of the received services or areas.

*** pRxServiceList**: List of the received service Key Versions or area Key Version.

The values returned by the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_ERR_INVALID_PARAMETER`: No service codes supplied.

`Other`: Value returned by the underlying component.

5.4.4.3 Read

The Read command reads a set of services and blocks in a FeliCa card. A maximum number of 8 services and 8 blocks shall be indicated. This Read command can only be executed in non-authentication required services.

```
phStatus_t phalFelica_Read (
      void *pDataParams,                                [In]
      uint8_t bNumServices,                             [In]
      uint8_t *pServiceList,                            [In]
      uint8_t bTxNumBlocks,                             [In]
      uint8_t *pBlockList,                              [In]
      uint8_t bBlockListLength,                         [In]
      uint8_t *pRxNumBlocks,                            [Out]
      uint8_t *pBlockData );                            [Out]
```

***pDataParams**: `Pointer to the FeliCa parameter component phalFelica_Sw_DataParams_t.`

bNumServices: Number of Services in the pServiceList list.

***pServiceList**: List of Services identified by their associated Service Code.

bTxNumBlocks: Number of Blocks to be read.

***pBlockList**: List of Blocks to be read.

bBlockListLength: Length of the list of blocks to be read.

***pRxNumBlocks:** Number of blocks received from the card.

***pBlockData**: Data received from the card – 16 x pRxNumBlocks –.

The values returned by the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_ERR_INVALID_PARAMETER`: No service codes or blocks supplied;

(`bBlockListLength < bTxNumBlocks * 2)` or `(bBlockListLength > bTxNumBlocks * 3`

`bNumBlocks > 8.`

`Other`: Value returned by the underlying component.

5.4.4.4 Write

The Write command stores data in a FeliCa card. This Write command can only be executed in non-authentication required services.

```
phStatus_t phalFelica_Write (
      void *pDataParams,                                [In]
      uint8_t bNumServices,                             [In]
      uint8_t *pServiceList,                            [In]
```

```
    uint8_t bNumBlocks,                                 [In]
    uint8_t *pBlockList,                                [In]
    uint8_t bBlockListLength,                           [In]
    uint8_t *pBlockData );                              [In]
```

***pDataParams**: Pointer to the FeliCa parameter component phalFelica_Sw_DataParams_t.

bNumServices: Number of Services in pServiceList list.

***pServiceList**: List of Services identified by their associated Service Code.

bNumBlocks: Number of blocks to be written.

***pBlockList**: List of Blocks to be written.

bBlockListLength: Length of the list of blocks to be written.

***pBlockData**: Data to be stored on the card – 16 x bBlockListLength –.

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

PH_ERR_INVALID_PARAMETER: No service codes or blocks supplied;

(bBlockListLength < bTxNumBlocks * 2) or (bBlockListLength > bTxNumBlocks * 3); bNumBlocks > 8.

Other: Value returned by the underlying component

5.5 Jewel / Topaz

5.5.1 Technical Introduction

The Topaz IC was developed by Innovision Research & Technology to address NFC and RFID tagging applications, and is therefore compliant with the ISO/IEC 18092, ISO/IEC 21481 and ISO/IEC 14443A standards. The Topaz IC based tag is the one that the NFC Forum has used to define the Type 1 Tag format.

The Topaz Tag utilizes two different memory mapping techniques depending on the memory size of the tag (Static and Dynamic memory structures). The static memory structure applies to a tag with a total physical memory size equal to 120 bytes. The memory availability is 96 bytes for user data. The memory is divided into blocks of 8 bytes numbered from 0 to 14 (Eh). On the other hand, the Dynamic memory structure model applies for tags with a physical memory larger than 120 bytes.

There is an additional 2-byte Header ROM (HR), where HR0=0x11 identifies the tag as a Topaz IC for NFC NDEF data applications. HR1 is reserved for internal use and shall be ignored.

HR0	HR1
11_h	xx_h

EEPROM Memory Map

Type	Block No.	Byte-0 (LSB)	Byte-1	Byte-2	Byte-3	Byte-4	Byte-5	Byte-6	Byte-7 (MSB)	Lockable
UID	0	UID-0	UID-1	UID-2	UID-3	UID-4	UID-5	UID-6		Locked
Data	1	Data0	Data1	Data2	Data3	Data4	Data5	Data6	Data7	Yes
Data	2	Data8	Data9	Data10	Data11	Data12	Data13	Data14	Data15	Yes
Data	3	Data16	Data17	Data18	Data19	Data20	Data21	Data22	Data23	Yes
Data	4	Data24	Data25	Data26	Data27	Data28	Data29	Data30	Data31	Yes
Data	5	Data32	Data33	Data34	Data35	Data36	Data37	Data38	Data39	Yes
Data	6	Data40	Data41	Data42	Data43	Data44	Data45	Data46	Data47	Yes
Data	7	Data48	Data49	Data50	Data51	Data52	Data53	Data54	Data55	Yes
Data	8	Data56	Data57	Data58	Data59	Data60	Data61	Data62	Data63	Yes
Data	9	Data64	Data65	Data66	Data67	Data68	Data69	Data70	Data71	Yes
Data	A	Data72	Data73	Data74	Data75	Data76	Data77	Data78	Data79	Yes
Data	B	Data80	Data81	Data82	Data83	Data84	Data85	Data86	Data87	Yes
Data	C	Data88	Data89	Data90	Data91	Data92	Data93	Data94	Data95	Yes
Reserved	D									
Lock/Reserved	E	LOCK-0	LOCK-1	OTP-0	OTP-1	OTP-2	OTP-3	OTP-4	OTP-5	

Fig 27. Topaz IC static memory map

5.5.2 Jewel/Topaz Parameter Structure

A special structure has been defined in the NFC Reader Library in order to store the parameters related to the Topaz Tag operation. This structure has been called `phalT1T_Sw_DataParams_t`.

```
typedef struct{
      void * pPalI14443p3aDataParams;
      uint8_t abHR[2];
      uint8_t abUid[4];
} phalT1T_Sw_DataParams_t;
```

***pPalI14443p3aDataParams**: Pointer to the ISO/IEC 14443-3A PAL data parameter component.

abHR[2]: Stores the Header ROM bytes (HR0 and HR1).

abUid: Stores the Topaz Tag UID.

5.5.3 Jewel/Topaz Component Initialization

The following function initializes a Topaz component.

```
phStatus_t phalT1T_Sw_Init(
      phalT1T_Sw_DataParams_t * pDataParams,          [In]
      uint16_t wSizeOfDataParams,                     [In]
      void * pPalI14443p3aDataParams );               [In]
```

***pDataParams**: Pointer to the Topaz (T1T) data parameter component.

wSizeofDataParams: Size of the Topaz (T1T) data parameter component.

***pPalI14443p3aDataParams**: Pointer to the ISO/IEC 14443-3A PAL data parameter component.

The values returned by the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_ERR_INVALID_DATA_PARAMS`: Some of the parameters passed to the function do not match with the expected variables.

5.5.4 Jewel/Topaz Command Set

The Jewel/Topaz command set is implemented within the NFC Reader Library. The Jewel/Topaz commands are described and explained in this section.

5.5.4.1 Request A

The function `phalT1T_RequestA()` performs an ISO 14443-3A Request A command.

```
phStatus_t phalT1T_RequestA(
      void * pDataParams,                                [In]
      uint8_t * pAtqa );                                 [Out]
```

***pDataParams**: Pointer to the Topaz (T1T) data parameter component.

***pAtqa**: Stores the Topaz Tag response (ATQA) to the Request A from the reader IC.

The values returned by the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_ERR_PROTOCOL_ERROR`: Invalid response received.

`PH_ERR_FRAMING_ERROR`: Invalid BCC received.

`Other`: Value returned by the underlying component.

5.5.4.2 Read UID

The Read UID command reads the metal-mask header bytes (HR0 and HR1) and the four least significant UID bytes from Block 0.

```
phStatus_t phalT1T_ReadUID(
      void    * pDataParams,                             [In]
      uint8_t * pUid,                                    [Out]
      uint16_t * pLength );                              [Out]
```

***pDataParams**: Pointer to the Topaz (T1T) data parameter component.

***pUid**: The four least significant UID bytes from the Topaz tag.

***pLength**: The number of received data bytes.

The values returned by the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_ERR_PROTOCOL_ERROR`: Invalid response received.

`PH_ERR_FRAMING_ERROR`: Invalid BCC received.

`Other`: Value returned by the underlying component.

5.5.4.3 Read All

The Read All command reads the two Header ROM bytes followed by all the memory blocks from 0x00 to 0x0E.

```
phStatus_t phalT1T_ReadAll(
      void * pDataParams,                                [In]
      uint8_t * pUid,                                    [In]
      uint8_t * pData,                                   [Out]
      uint16_t * pLength );                              [Out]
```

***pDataParams**: Pointer to the Topaz (T1T) data parameter component.

***pUid**: The four least significant UID bytes from the Topaz tag.

***pData**: Buffer that stores all the read data from the tag.

***pLength**: Number of received data bytes.

The values returned by the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_ERR_PROTOCOL_ERROR`: Invalid response received.

`PH_ERR_FRAMING_ERROR`: Invalid BCC received.

`Other`: Value returned by the underlying component.

5.5.4.4 Read Byte

The Read Byte commands reads a single EEPROM memory byte within blocks 0x00 to 0x0E.

```
phStatus_t phalT1T_ReadByte(
      void * pDataParams,                                [In]
      uint8_t * pUid,                                    [In]
      uint8_t   bAddress,                                [In]
      uint8_t * pData,                                   [Out]
      uint16_t * pLength );                              [Out]
```

***pDataParams**: Pointer to the Topaz (T1T) data parameter component.

***pUid**: The four least significant UID bytes from the Topaz tag.

bAddress: Address of the byte to be read.

***pData**: Buffer containing the read data from the tag.

***pLength**: Number of received data bytes.

The values returned by the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_ERR_PROTOCOL_ERROR`: Invalid response received.

`PH_ERR_FRAMING_ERROR`: Invalid BCC received.

`Other`: Value returned by the underlying component.

5.5.4.5 Write Erase Byte

The Write Erase command is used in a static memory structure to write a single memory byte within blocks 0x00 to 0x0E. This command performs the erase-write cycle, which means that it erases the target byte before it writes the new data.

```
phStatus_t phalT1T_WriteEraseByte(
      void * pDataParams,                                [In]
      uint8_t * pUid,                                    [In]
      uint8_t   bAddress,                                [In]
      uint8_t   bTxData,                                 [In]
      uint8_t * pRxData,                                 [Out]
      uint16_t * pLength );                              [Out]
```

***pDataParams**: Pointer to the Topaz (T1T) data parameter component.

***pUid**: The four least significant UID bytes from the Topaz tag.

bAddress: Address of the byte to be written.

bTxData: Buffer containing the data to be written to the tag.

***pRxData**: Buffer that stores the data retrieved from the tag.

***pLength**: Number of received data bytes.

The values returned by the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_ERR_PROTOCOL_ERROR`: Invalid response received.

`PH_ERR_FRAMING_ERROR`: Invalid BCC received.

`Other`: Value returned by the underlying component.

5.5.4.6 Write No Erase Byte

The Write No Erase command is used in static memory structure to write a single memory byte within blocks 0x00 to 0x0E. This command does not erase the target before writing the new data, and its execution time is approximately half of the Write Erase command.

```
phStatus_t phalT1T_WriteNoEraseByte(
      void * pDataParams,                                [In]
      uint8_t * pUid,                                    [In]
      uint8_t   bAddress,                                [In]
      uint8_t   bTxData,                                 [In]
      uint8_t * pRxData,                                 [Out]
      uint16_t * pLength );                              [Out]
```

***pDataParams**: Pointer to the Topaz (T1T) data parameter component.

***pUid**: The four least significant UID bytes from the Topaz tag.

bAddress: Address of the byte to be written.

bTxData: Buffer containing the data to be written to the tag.

***pRxData**: Buffer that stores the data retrieved from the tag.

***pLength**: Number of received data bytes.

The values returned by the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_ERR_PROTOCOL_ERROR`: Invalid response received.

`PH_ERR_FRAMING_ERROR`: Invalid BCC received.

`Other`: Value returned by the underlying component.

5.5.4.7 Read Segment

The Read Segment command reads a complete segment of memory. A segment consists of 16 blocks (128 bytes) of memory.

```
phStatus_t phalT1T_ReadSegment(
      void * pDataParams,                                [In]
      uint8_t * pUid,                                    [In]
      uint8_t   bAddress,                                [In]
      uint8_t * pData,                                   [Out]
      uint16_t * pLength  );                             [Out]
```

***pDataParams**: Pointer to the Topaz (T1T) data parameter component.

***pUid**: The four least significant UID bytes from the Topaz tag.

bAddress: Address from which to start reading the segment.

***pData**: Buffer containing the read data from the tag.

***pLength**: Number of received data bytes.

The values returned by the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_ERR_PROTOCOL_ERROR`: Invalid response received.

`PH_ERR_FRAMING_ERROR`: Invalid BCC received.

`Other`: Value returned by the underlying component.

5.5.4.8 Read Block

The Read Block (or Read 8 bytes) command reads a block of memory.

```
phStatus_t phalT1T_ReadBlock(
      void * pDataParams,                                [In]
      uint8_t * pUid,                                    [In]
      uint8_t   bAddress,                                [In]
      uint8_t * pData,                                   [Out]
      uint16_t * pLength );                              [Out]
```

***pDataParams**: Pointer to the Topaz (T1T) data parameter component.

***pUid**: The four least significant UID bytes from the Topaz tag.

bAddress: Address from which to start writing to the block.

***pData**: Buffer containing the read data from the tag.

***pLength**: Number of received data bytes.

The values returned by the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_ERR_PROTOCOL_ERROR`: Invalid response received.

`PH_ERR_FRAMING_ERROR`: Invalid BCC received.

`Other`: Value returned by the underlying component.

5.5.4.9 Write Erase Block

The Write Erase Block (or Write 8 bytes) command performs an erase-write cycle over a block, which means that it erases the target block before writing the new data.

```
phStatus_t phalT1T_WriteEraseBlock(
      void * pDataParams,                                [In]
      uint8_t * pUid,                                    [In]
      uint8_t   bAddress,                                [In]
      uint8_t * pTxData,                                 [In]
      uint8_t * pRxData,                                 [Out]
      uint16_t * pLength );                              [Out]
```

***pDataParams**: Pointer to the Topaz (T1T) data parameter component.

***pUid**: The four least significant UID bytes from the Topaz tag.

bAddress: Address from which to start writing to the block.

bTxData: Buffer containing the data to be written to the tag.

***pRxData**: Buffer that stores the data retrieved from the tag.

***pLength**: Number of received data bytes.

The values returned by the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_ERR_PROTOCOL_ERROR`: Invalid response received.

`PH_ERR_FRAMING_ERROR`: Invalid BCC received.

`Other`: Value returned by the underlying component.

5.5.4.10 Write No Erase Block

The Write No Erase Block (or Write No Erase 8 bytes) command writes the new data a block without previously erasing the content of the block. Therefore, its executing time is approximately half of the Write Erase command.

```
phStatus_t phalT1T_WriteNoEraseBlock(
      void * pDataParams,                                [In]
      uint8_t * pUid,                                    [In]
      uint8_t   bAddress,                                [In]
      uint8_t * pTxData,                                 [In]
      uint8_t * pRxData,                                 [Out]
      uint16_t * pLength );                              [Out]
```

***pDataParams**: Pointer to the Topaz (T1T) data parameter component.

***pUid**: The four least significant UID bytes from the Topaz tag.

bAddress: Address from which to start writing to the block.

bTxData: Buffer containing the data to be written to the tag.

***pRxData**: Buffer that stores the data retrieved from the tag.

***pLength**: Number of received data bytes.

The values returned by the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_ERR_PROTOCOL_ERROR`: Invalid response received.

`PH_ERR_FRAMING_ERROR`: Invalid BCC received.

`Other`: Value returned by the underlying component.

5.6 NFC Forum Tag Type Operations

5.6.1 Technical Introduction

Service providers developing smart card solutions usually offer different products to provide a more complete solution. Each tag or smart card product have their own memory map and their own set of functions to interact with it, and therefore, solutions that are developed for a particular product are not applicable to others.

The NFC Forum standardization body has released the NFC Forum Platform which consists of four NFC Forum Type Tag specifications [22][23][24][25]. These specifications are independent of both the product and the technology although based on existing contactless card products. The main objective of the NFC Forum Platform is to provide abstraction of the underlying hardware where the data is stored. This way, any NFC device acting in Read&Write mode is able to understand NDEF [26] formatted messages on any NFC Forum tag and to interpret the operation to be completed with the data.

Table 14 depicts available Type Tags and their reference products.

Table 14. NFC Forum Type Tag Platforms

NFC Forum Platform	Reference Products
NFC Forum Type 1 Tag	Innovision Topaz
NFC Forum Type 2 Tag	NXP MIFARE Ultralight family, NXP NTAG family
NFC Forum Type 3 Tag	Sony FeliCa
NFC Forum Type 4 Tag	Tags compliant with ISO-7816 file structure NXP MIFARE DESFire, NXP SmartMX with JCOP, and others.

NFC Forum Type Tag specifications define the data mapping and the way NFC Forum Devices detect, read, and write NDEF data into a particular NFC Forum Tag in order to achieve and maintain interchangeability and interoperability.

5.6.2 NFC Forum Tag Type Operations component

The objective of the NFC Forum Platform is to provide abstraction from the underlying hardware on which the operation is being performed. The NFC Forum Tag Type Operations component of the NFC Reader Library provides an API that allows customers to complete NDEF related operations on NFC Forum Tags. The NFC Reader Library translates these operations to the appropriate Type Tag API functions which implement the specific Type Tag commands. The NFC Forum Tag Type Operations API relies and leverages on the Application Layer components for these operations.

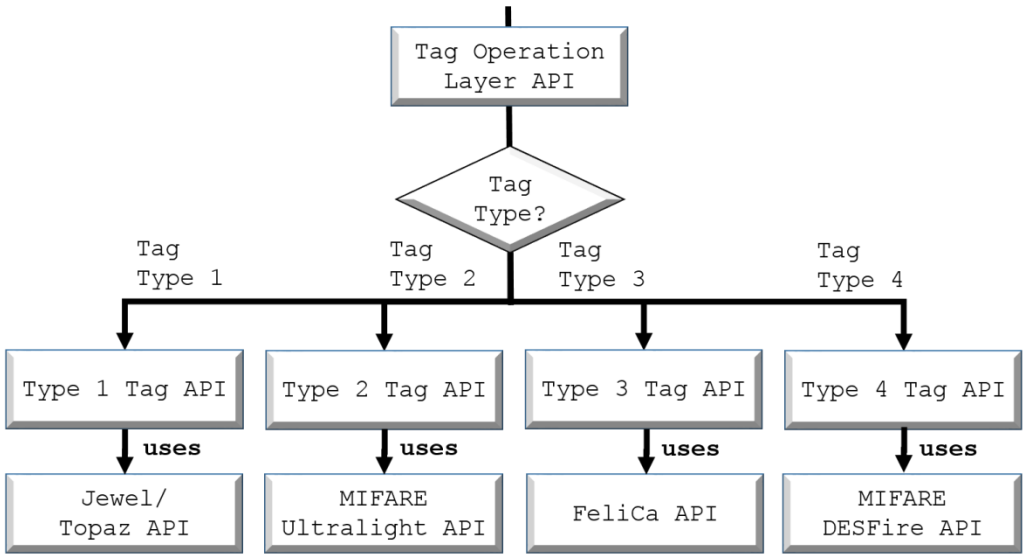


Fig 28. NFC Forum Tag Type Operations component API

For instance, the NFC Forum Tag Type Operations component implements a function to write NDEF messages into a tag (`phalTop_Sw_WriteNdef()`). If we are working with a Type 2 tag, the NFC Reader Library translates this call to the appropriate Type 2 Tag API function (`phalTop_Sw_Int_T2T_WriteNdef()`). Similarly, `phalTop_Sw_Int_T2T_WriteNdef()`, internally makes use of the write function implemented in the MIFARE Ultralight component. This process is performed in a transparent way for the user.

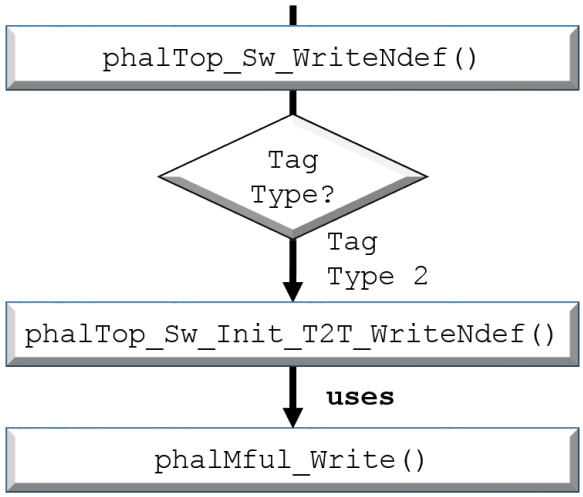


Fig 29. NFC Forum Tag Type Operations API: Write NDEF function for a Type 2 Tag

5.6.3 NFC Forum Tag Type Operations structure

The NFC Forum Tag Type Operations component structure defined by the NFC Reader Library provides abstraction of the underlying hardware where the NDEF operations are being performed. In order to be able to operate with the four NFC Forum defined NFC

tags, the NFC Forum Tag Type Operations component structure stores pointers that reference all the specific NFC Forum Type Tags.

The `bTagType` value that is defined within the phalTop_Sw_DataParams_t structure identifies the detected card technology on which the commands are being executed in order to be able to translate generic NDEF management commands to the card technology specific commands as it is explained in Fig 29.

```
typedef struct  phalTop_Sw_DataParams {
    uint8_t bTagType;
    void * pT1T;
    void * pT2T;
    void * pT3T;
    void * pT4T;
    void * pT5T;
} phalTop_Sw_DataParams_t;
```

***bTagType:** Identifier of the underlying hardware Type Tag technology.

***pT1T:** Pointer to the Type 1 Tag `phalTop_T1T_t` component structure.

***pT2T:** Pointer to the Type 2 Tag `phalTop_T2T_t` component structure.

***pT3T:** Pointer to the Type 3 Tag `phalTop_T3T_t` component structure.

***pT4T:** Pointer to the Type 4 Tag `phalTop_T4T_t` component structure.

***pT5T:** Pointer to the Type 5 Tag `phalTop_T5T_t` component structure (RFU).

The NFC Reader Library defines four structures to manage each NFC Forum Type Tag. The definition of each structure is card technology dependent and therefore the variables that are declared in each structure are completely different.

The explanation of the four Type Tag component structures is out of the scope of this document. The only Type Tag component structure that is explained is the Type 2 Tag, which belongs to the NXP MIFARE Ultralight reference product.

```
typedef struct phalTop_T2T {
    void * phalT2TDataParams;
    uint8_t bNdefPresence;
    uint8_t bVno;
    uint8_t bTms;
    uint8_t bRwa;
    uint16_t wNdefHeaderAddr;
    uint16_t wNdefMsgAddr;
    uint16_t wNdefLength;
    uint16_t wMaxNdefLength;
    uint8_t bMemoryTlvCount;
    phalTop_T2T_MemCtrlTlv_t asMemCtrlTvl[PHAL_TOP_T2T_MAX_MEM_CTRL_TLV];
    uint8_t bLockTlvCount;
    phalTop_T2T_LockCtrlTlv_t asLockCtrlTvl[PHAL_TOP_T2T_MAX_LOCK_CTRL_TLV];
    uint8_t bProprietaryTlvCount;
    phalTop_T2T_ProprietaryTlv_t asPropTvl[PHAL_TOP_T2T_MAX_PROPRIETARY_TLV];
    uint8_t bTerminatorTlvPresence;
    uint8_t bEraseProprietaryTlv;
    uint8_t bNdefFormatted;
    uint8_t bTagState;
    uint8_t bTagMemoryType;
    phalTop_T2T_Sector_t sSector;
}phalTop_T2T_t;
```

*** phalT2TDataParams:** Pointer to the reference application layer component structure (In this case MIFARE Ultralight component).

bNdefPresence: Indicates the presence or absence of a NDEF message in the tag.

bVno: NFC Forum Type Tag specification version number with which the tag is compliant.

bTms: Total data memory size of the tag (calculated as 8 x bTms).

bRwa: Tag read/write access privileges: initialized, read/write, read only.

wNdefHeaderAddr: Header offset of the first NDEF message.

wNdefMsgAddr: Start address of the NDEF message.

wNdefLength: Length of the NDEF message.

wMaxNdefLength: Maximum supported NDEF length depending on the TLV.

bMemoryTlvCount: Number of memory TLV present in the tag.

asMemCtrlTvl: Array of Memory control TLVs.

bLockTlvCount: Number of lock TLV present in the tag.

asLockCtrlTvl: Array of Lock control TLVs.

bProprietaryTlvCount: Number of proprietary TLV present in the tag.

asPropTvl: Array of proprietary TLVs.

bTerminatorTlvPresence: Indicates whether the terminator TLV is present in the tag.

bEraseProprietaryTlv: TLV during write; 1 - erase, 0 - don't erase.

bNdefFormatted: Indicates if the tag is formatted to store NDEF messages, which implies the existence of a well-formatted Capability Container.

bTagState: Current state of the tag: initialized, .

Table 15. NFC Forum Type Tag Platforms

NFC Reader Library state	Tag 2 Type state
`PHAL_TOP_T2T_STATE_UNKNOWN`	Default initial state
`PHAL_TOP_T2T_STATE_INITIALIZED`	Initialized state
`PHAL_TOP_T2T_STATE_READONLY`	Read Only state
`PHAL_TOP_T2T_STATE_READWRITE`	Read/Write state

*** bTagMemoryType:** Indicates the tag memory type: static (memory size equal to 64 bytes) or dynamic (memory size bigger than 64 bytes) according to bTms.

*** sSector:** Configuration details for the current sector.

5.6.4 NFC Forum Tag Type Operations API

The NFC Forum Tag Type Operations component offers a set of functions that allow developers to manage NDEF Formatted data on NFC Forum Tags.

5.6.4.1 Init function

The `phalTop_Sw_Init` function initializes the NFC Forum Tag Type Operations `phalTop_Sw_DataParams_t` component structure by setting the pointers to all NFC Forum

Tag Type components that are passed as input arguments to this function. If a specific Type Tag is not going to be used in the application, the pointer should be set to NULL.

This function calls internally the `phStatus_t phalTop_Reset()` function that is responsible for the initialization of the passed Type Tag structures.

```
phStatus_t phalTop_Sw_Init(
      phalTop_Sw_DataParams_t * pDataParams,                [In]
      uint16_t wSizeOfDataParams,                           [In]
      void * pTopT1T,                                       [In]
      void * pTopT2T,                                       [In]
      void * pTopT3T,                                       [In]
      void * pTopT4T,                                       [In]
      void * pTopT5T);                                      [In]
```

***pDataParams:** Pointer to the `phalTop_Sw_DataParams_t` component.

wSizeofDataParams: Size of the `phalTop_Sw_Init` data parameter component.

***pTopT1T:** Pointer to the Type 1 Tag `phalTop_T1T_t` component structure.

***pTopT2T:** Pointer to the Type 1 Tag `phalTop_T1T_t` component structure.

***pTopT3T:** Pointer to the Type 1 Tag `phalTop_T1T_t` component structure.

***pTopT4T:** Pointer to the Type 1 Tag `phalTop_T1T_t` component structure.

***pTopT5T:** Pointer to the Type 5 Tag `phalTop_T5T_t` component structure (RFU).

5.6.4.2 Reset

The Reset function of the NFC Forum Tag Type Operations component resets the `phalTop_Sw_DataParams_t` parameter structure to its default state by setting all the variables to 0. The current state of all NFC Forum Tag Type Operations structures is set to the default initial state.

This function is automatically called by the NFC Reader Library when the `phalTop_Sw_Init()` function is called by the developer.

```
phStatus_t phalTop_Reset(
    phalTop_Sw_DataParams_t * pDataParams );                [In]
```

***pDataParams:** Pointer to the `phalTop_Sw_DataParams_t` component structure.

The value returned by the function is:

`PH_ERR_SUCCESS` Operation successful.

5.6.4.3 Check NDEF

This function checks whether there is a NDEF message stored in the card or not. The result of this checking is returned to the user in `pNdefPresence` variable.

Before this function can be called by the developer, `phalTop_Sw_DataParams_t` structure shall be updated according to the detected card technology. Function `phalTop_SetConfig()` (see section 5.6.4.8) might be useful for completing this task.

Together with the NDEF presence checking, the function gathers information about card specific configuration such as the specification version number, the total memory size, operation configuration (e.g. read only, read/write) in order to fill in the fields defined at

the Type Tag structure. Therefore, this function shall be called before performing any operation in a tag.

In order to check the presence of the NDEF message, the reader IC looks for the Capability Container existence as it is defined in the corresponding NFC Forum Type Tag specification. In addition, it searches for TLVs stored in the card to retrieve all available information.

```
phStatus_t phalTop_CheckNdef(
    phalTop_Sw_DataParams_t * pDataParams,                  [In]
    uint8_t * pNdefPresence);                               [Out]
```

***pDataParams:** Pointer to the `phalTop_Sw_DataParams_t` component structure.

pNdefPresence: Indicates whether there is a NDEF message stored in the card or not.

- 0: There is no NDEF message stored in the card.
- 1: There is a NDEF message stored in the card.

The value returned by the function is:

`PH_ERR_SUCCESS` Operation successful.

5.6.4.4 Format NDEF

This function configures the tag to store NDEF formatted data. It writes the Capability Container according to the Type Tag specification that addresses the particular technology of the card on which the operation is performed.

This function can be skipped when the card is already formatted for the correct storage of NDEF messages. `phalTop_Sw_CheckNdef()` function, which shall be called before executing this `phStatus_t phalTop_FormatNdef()` function, already checks for the existence of a valid capability container in the tag and it stores the result in the `bNdefFormatted` field of `phalTop_Sw_DataParams_t` structure. In order to get this value, `phalTop_GetConfig()` function (see section 5.6.4.9) and `PHAL_TOP_CONFIG_TAG_FORMATTABLE` tag might be useful.

```
phStatus_t phalTop_FormatNdef(
    phalTop_Sw_DataParams_t * pDataParams );                [In]
```

***pDataParams:** Pointer to the `phalTop_Sw_DataParams_t` component structure.

The values returned by the function can be:

`PH_ERR_SUCCESS` Operation successful.

`PH_ERR_INVALID_PARAMETER`: A parameter value is invalid.

`Other`: Value returned by the underlying component.

5.6.4.5 Read NDEF

This function reads out a NDEF message from a tag. Prior to any tag operation, the `phalTop_CheckNdef()` shall be called to validate if the tag contains a valid NDEF message.

The `phalTop_ReadNdef` function returns the whole NDEF message, including its header and payload.

```
phStatus_t phalTop_ReadNdef(
    phalTop_Sw_DataParams_t * pDataParams,                  [In]
    uint8_t * pData,                                        [Out]
    uint16_t * pLength );                                   [Out]
```

***pDataParams:** Pointer to the `phalTop_Sw_Init` component.

***pData:** The NDEF message read from the tag.

***pLength:** The NDEF message length.

The values returned by the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_ERR_INVALID_PARAMETER`: A parameter value is invalid.

`PH_ERR_PROTOCOL_ERROR`: No valid NDEF present in the tag.

`Other`: Value returned by the underlying component.

5.6.4.6 Write NDEF

This function writes a NDEF message in the tag where the function is executed. Before performing this operation, `phalTop_CheckNdef()` shall be called.

Additionally, if the tag is not already correctly formatted for the storage of NDEF messages, which basically implies the existence of a valid Capability Container, the `phalTop_FormatNdef()` function shall be called. The entire NDEF message should be passed to the function (NDEF header and payload).

```
phStatus_t phalTop_WriteNdef(
    phalTop_Sw_DataParams_t * pDataParams,                  [In]
    uint8_t * pData,                                        [Out]
    uint16_t wLength );                                     [Out]
```

***pDataParams:** Pointer to the `phalTop_Sw_Init` component.

***pData:** The NDEF data to write to the tag.

wLength: The NDEF data length.

The values returned by the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_ERR_INVALID_PARAMETER`: A parameter value is invalid.

`PH_ERR_PROTOCOL_ERR_NOT_FORMATTED`: Non formatted tag.

`Other`: Value returned by the underlying component.

5.6.4.7 Erase NDEF

The calling of this function leads to the removal of a NDEF Message from the tag memory. The memory blocks of the tag where the NDEF message is stored are set to 0. In addition, the Capability Container is erased from the tag (operation available only if the CC block is not OTP).

```
phStatus_t phalTop_EraseNdef(
    phalTop_Sw_DataParams_t * pDataParams );                [In]
```

***pDataParams:** Pointer to the `phalTop_Sw_DataParams_t` component structure.

The values returned by the function can be:

`PH_ERR_SUCCESS` Operation successful.

`Other`: Value returned by the underlying component.

5.6.4.8 Set Config

This function provides a user friendly way to modify some parameters defined in the `phalTop_Sw_DataParams` component structure. This way, developers can make use of tags defined in `/intfs/phalTop.h` file instead of going through the variables in the component structure.

For instance, whenever a new card technology is detected, the developer may call this function indicating `PHAL_TOP_CONFIG_TAG_TYPE` tag and the correct technology, defined in Table 16, instead of setting the `bTagType` variable in `phalTop_Sw_DataParams` structure.

Table 16. NFC Forum Type Tag Platforms

NFC Forum Platform	NFC Reader Configuration Tag
NFC Forum Type 1 Tag	`PHAL_TOP_TAG_TYPE_T3T_TAG`
NFC Forum Type 2 Tag	`PHAL_TOP_TAG_TYPE_T3T_TAG`
NFC Forum Type 3 Tag	`PHAL_TOP_TAG_TYPE_T3T_TAG`
NFC Forum Type 4 Tag	`PHAL_TOP_TAG_TYPE_T3T_TAG`

```
phStatus_t phalTop_SetConfig(
    phalTop_Sw_DataParams_t * pDataParams,                  [In]
    uint16_t wConfig,                                       [In]
    uint16_t wValue );                                      [In]
```

***pDataParams:** Pointer to the `phalTop_Sw_DataParams_t` component structure.

wConfig: Variable identifier to be modified.

wValue: Value for the variable to be modified.

The values returned by the function can be:

`PH_ERR_SUCCESS` Operation successful.

`INVALID_PARAMETER`: Invalid identifier passed.

`Other`: Value returned by the underlying component.

5.6.4.9 Get Config

This function provides a user friendly way to retrieve some parameters defined in the `phalTop_Sw_DataParams` component structure. The value of the variable identified by wConfig value is returned at the wValue pointer.

```
phStatus_t phalTop_GetConfig(
    phalTop_Sw_DataParams_t * pDataParams,                  [In]
    uint16_t wConfig,                                       [In]
    uint16_t * wValue );                                    [Out]
```

***pDataParams:** Pointer to the `phalTop_Sw_DataParams_t` component structure.

wConfig: Identifier of the variable to be retrieved.

wValue: Pointer where the value of the variable is retrieved.

The values returned by the function can be:

`PH_ERR_SUCCESS` Operation successful.

`INVALID_PARAMETER`: Invalid identifier passed

6. NFC Reader Library API: NFC Activity

In this section, the NFC Activity component is explained in depth. How to configure the Discovery loop to detect NFC tags and NFC devices and how to set up the reader IC in different communication and operation modes is presented.

6.1 Discovery Loop

6.1.1 Technical Introduction

NFC technology was approved under the ISO/IEC 18092 international standard and is compatible with both ISO/IEC 14443 Type A and FeliCa contactless protocols. NFC technology added support for ISO/IEC 14443 Type B and ISO/IEC 15693 contactless protocols later on under the ISO/IEC 21481 standard. These contactless technologies are supported differently depending on the operation mode, as it is shown in Fig 30.

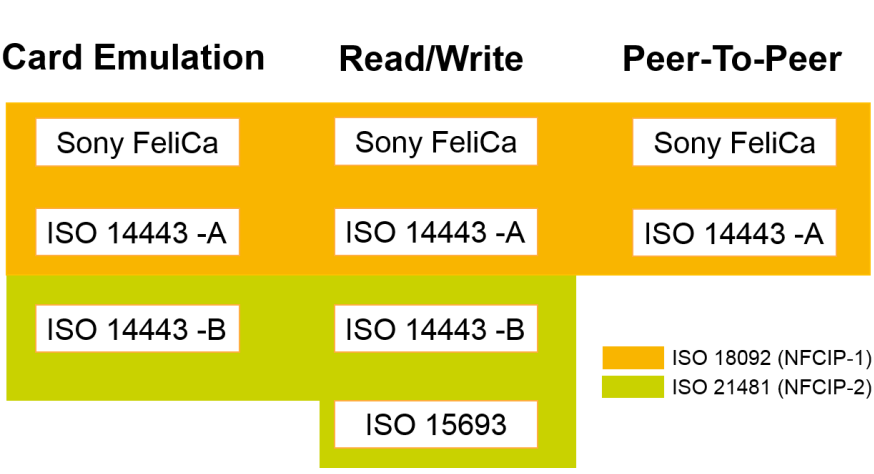


Fig 30. NFC supported contactless standards

Normally, it is not known in advance what kind of contactless smart card or NFC device will be presented to the reader IC and in which mode it will be operating. The Discovery Loop, or also known as polling loop, includes the following functionalities:

- Technology detection: Detects whether there is another device or tag to communicate with and, if so, what technologies it support.
- Collision resolution: Detects the presence of multiple devices or tags and enumerates the different identifiers.
- Device activation: Activates a particular device or tag to establish a communication.

In the NFC Reader Library, the Discovery Loop is implemented as a routine that sequentially sets the reader IC in different functional states and configurations so it can discover the presence of tags or NFC devices in the RF field. This polling loop can be customized by the developer. It can be configured to select which contactless technologies should be scanned in the field and set the guard and pause intervals as well.

The Discovery Loop concept is illustrated graphically in Fig 31. The reader IC is configured to go through several states and sense the field using a specific contactless protocol. The sensing is done by sending initialization request commands looking for other devices or tags to respond. If during the sensing interval there is a device or tag operating in this specific technology within the range, the reader IC will receive the response to that initialization and thus activate the communication channel to start exchanging data.

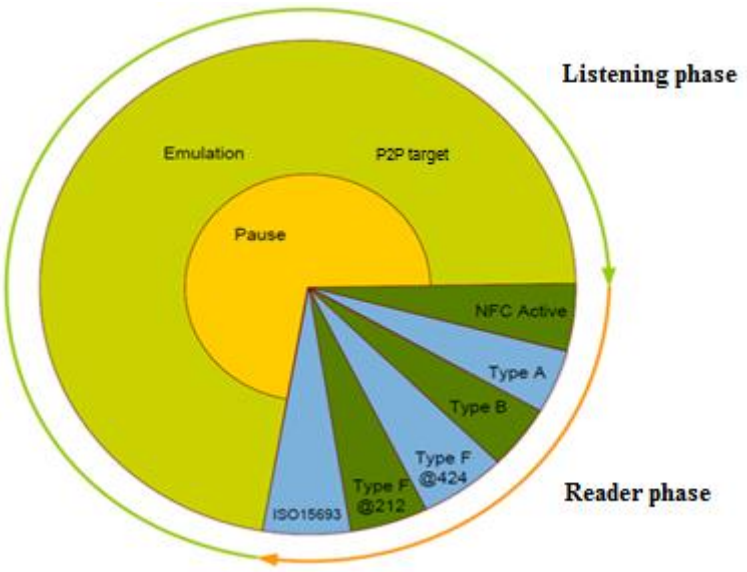


Fig 31. Discovery Loop routine

Unlike HAL, the Discovery Loop uses MCU timers for measuring the guard times, instead of the reader IC timers. Due to the fact the OSAL layer does not provide any thread creation capabilities, the Discovery Loop can only run in the main thread called from the upper application without any possibility of interruption, therefore the upper application is blocked until the Discovery Loop routine exits.

6.1.2 Discovery Loop Data Parameter Structure

A special structure has been defined in the NFC Reader Library in order to store the parameters related to the Discovery Loop. This structure has been called `phacDiscLoop_Sw_DataParams`.

```
typedef struct phacDiscLoop_Sw_DataParams{
    uint8_t bDetectionConfiguration;
    uint16_t wGTa;
    uint16_t wGTb;
    uint16_t wGTf;
    uint16_t wGTbf;
    uint8_t  bGTaUnit;
    uint8_t  bGTbUnit;
    uint8_t  bGTfUnit;
    uint8_t  bGTbfUnit;
    uint8_t  bNumPollLoops;
```

```
    uint8_t  bState;
    uint16_t wTagsFound;
    uint8_t  bBailOut;
    uint8_t  bLoopMode;
    pErrHandlerCallback pErrHandler;
    uint16_t wPausePeriod;
    uint8_t  bPausePeriodUnit;
    uint8_t  bStopFlag;
    uint8_t  bConColl;
    uint8_t  bP2P_ACT_BaudRate;
    void  * pHalDataParams;
    void  * pPal1443p3aDataParams;
    void  * pPal1443p4aDataParams;
    void  * pPalFelicaDataParams;
    void  * pPal1443p3bDataParams;
    void  * pPal18092mPIDataParams;
    void  * pPal18092mTDataParams;
    void  * pAlT1TDataParams;
    void  * pOsal;
    phacDiscLoop_TypeA_Tags_t sTypeATargetInfo;
    phacDiscLoop_TypeF_Tags_t sTypeFTargetInfo;
    phacDiscLoop_TypeB_Tags_t sTypeBTargetInfo;
    phacDiscLoop_TargetParams_t sTargetParams;
} phacDiscLoop_Sw_DataParams_t;
```

bDetectionConfiguration: This variable can be used to configure which `contactless technologies will be scanned for detection. The uint8_t bDetectionConfiguration variable can be set to these values:`

- `PHAC_DISCLOOP_CON_POLL_A:` Flag enabling detection of Type A tags.
- `PHAC_DISCLOOP_CON_POLL_B:` Flag enabling detection of Type B tags.
- `PHAC_DISCLOOP_CON_POLL_F:` Flag enabling detection of Type F tags.
- `PHAC_DISCLOOP_CON_POLL_ACTIVE:` Detection of active initiator mode.

wGTa,wGTb,wGtf and wGTbf: These variables can be used to set up guard time intervals between sending commands for detecting different types of contactless protocols can be configured using these variables.

- `wGTa`: The guard time before detection of Type A tags
- `wGTb`: The guard time before detection of Type B tags.
- `wGtf`: The guard time before detection of Type F tags
- `wGtbf`: The guard time for switching from Type B detection to Type F detection.

bGTaUnit, **bGTbUnit**, **bGTfUnit** and **bGTbfUnit**; These variables are used to set the time unit magnitude) for the guard time and pause interval variables (can be miiliseconds or microseconds):

- `bGTaUnit:` Time units for wGTa.
- `bGTbUnit:` Time units for wGTb.
- `bGTfUnit:` Time units for wGTf.
- `bGTbfUnit:` Time units for wGTbf.

bNumPollLoops; The number of iterations of the Discovery Loop sequence can be configured using the `bNumPollLoop` variable value. Each loop iteration goes through all the sequences defined in the variable `uint8_t bDetectionConfiguration`.

bState; Indicates the current state of the discovery loop.

wTagsFound; The `wTagsFound` variable represents a binary map indicating which tags and NFC devices have been detected during the Discovery Loop. The following bitmasks are defined:

- `PHAC_DISCLOOP_TYPEA_DETECTED_TAG_TYPE1`
- `PHAC_DISCLOOP_TYPEA_DETECTED_TAG_TYPE2`
- `PHAC_DISCLOOP_TYPEA_DETECTED_TAG_TYPE4A`
- `PHAC_DISCLOOP_TYPEA_DETECTED_TAG_NFC_DEP_TYPE4A`
- `PHAC_DISCLOOP_TYPEF_DETECTED_TAG_P2P`
- `PHAC_DISCLOOP_TYPEA_DETECTED_TAG_P2P`
- `PHAC_DISCLOOP_TYPEA_DETECTED`
- `PHAC_DISCLOOP_TYPEB_DETECTED`
- `PHAC_DISCLOOP_TYPEF_DETECTED`

bBailOut; Two bail out flags can be used to exit from the Discovery Loop under the following conditions:

- `PHAC_DISCLOOP_CON_BAIL_OUT_A`: As soon as a Type A tag is detected, it is activated and the Discovery Loop stops without any further scanning for the detection of Type B and F tags.
- `PHAC_DISCLOOP_CON_BAIL_OUT_B`: As soon as a Type B tag is detected, it is activated and the Discovery Loop stops without any further scanning for the detection of Type F tags.

bLoopMode; This variable holds the supported combinations of Poll, Listen and Pause modes for the reader IC configuration. The allowed loop mode combinations are the following:

- `PHAC_DISCLOOP_SET_POLL_MODE | PHAC_DISCLOOP_SET_LISTEN_MODE`
- `PHAC_DISCLOOP_SET_POLL_MODE | PHAC_DISCLOOP_SET_PAUSE_MODE`
- `PHAC_DISCLOOP_SET_LISTEN_MODE`

pErrHandler; Pointer to the user error handler function. The user error handler function can be defined using this definition (`typedef void(*pErrHandlerCallback)(phStatus_t)`.

wPausePeriod: This variable can be used to set up the pause interval. The pause time interval is a period of time where the reader IC is not scanning the field.

bPausePeriodUnit; This variables is used to set the time unit magnitude of the wPausePeriod. . The identifier `PH_OSAL_TIMER_UNIT_US` is used for setting the time unit magnitude to microseconds and the `PH_OSAL_TIMER_UNIT_MS` identifier for setting the time unit magnitude to milliseconds.

bStopFlag; This flag indicates that the discovery loop should exit. `This flag can be enabled by pErrHandlerCallback error handler.`

bConColl: Holds the information whether collision resolution is required or not.

bP2P_ACT_BaudRate: Active Communication P2P Baud Rate.

*** pHalDataParams;** Pointer to the HAL layer parameter component. According to the used reader chip `phhalHw_Rc663_DataParams_t` and `phhalHw_Rc523_DataParams_t`.

*** pPal1443p3aDataParams;** Pointer to the ISO/IEC 14443-3A layer PAL data structure. Required for Type A tag.

*** pPal1443p4aDataParams;** Pointer to the 14443-4A layer PAL data structure. Required for Type A tag.

*** pPalFelicaDataParams**; Pointer to the FeliCa PAL data structure. Required for Type F tag.

*** pPal1443p3bDataParams;** Pointer to the Type B 14443-3B PAL data structure. Required for Type B tag.

*** pPal18092mPIDataParams;** Pointer to the 18092 MPI PAL data structure. Required for Type F tag – passive initiator.

*** pPal18092mTDataParams;** Pointer to the 18092 MT PAL data structure. Required for Type F tag– target.

*** pAlT1TDataParams;** Pointer to T1T AL data parameters.

*** pOsal;** Pointer to the OS layer component.

sTypeATargetInfo; Information gathered from Type A tags in the RF field.

sTypeFTargetInfo; Information gathered from Type F tags in the RF field.

sTypeBTargetInfo; Information gathered from Type B tags in the RF field.

sTargetParams; Information gathered from Type F – target in the RF field.

6.1.3 Discovery Loop Initialization

This function initializes the *phacDiscLoop_Sw_DataParams_t* data structure to the default values. All the components that are passed as input arguments to this function should be initialized before the call to start the Discovery Loop is made.

```
phStatus_t phacDiscLoop_Sw_Init(
      phacDiscLoop_Sw_DataParams_t      *pDataParams,        [In]
      uint16_t                          wSizeOfDataParams,   [In]
      void                              *pHalDataParams,     [In]
      void                              *pOsal );            [In]
```

*pDataParams: Pointer to the `phacDiscLoop_Sw_DataParams_t` parameter component. Pointers to the PAL components are initialized to zero by this function. The developer has to specifically set them.

wSizeOfDataParams: Size of the **`phacDiscLoop_Sw_DataParams_t`** data parameter component.

*__pHalDataParams:__ Pointer to the HAL component according to the used reader chip (i.e. **`phhalHw_Rc523_DataParams_t`**).

***pOsal:** Pointer to the OSAL data parameters. Timers from the MCU are used in delay loops and listening timeouts.

The values returned by the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`Other`: Value returned by the underlying component.

6.1.4 Discovery Loop Set Configuration

The developer can use the `phacDiscLoop_SetConfig()` function to set the configuration parameters defined in the next section (see Section 6.1.6):

```
phStatus_t phacDiscLoop_SetConfig(
      void        *pDataParams,                    [In]
      uint16_t    wConfig,                         [In]
      uint16_t    wValue                           [In]
```

***pDataParams:** Pointer to the **`phacDiscLoop_Sw_DataParams_t`** parameter component.

wConfig: The configuration identifier to be set.

wValue: The configuration value to set.

The values returned by the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_ERR_INVALID_PARAMETER`: Invalid option/response received.

`Other`: Value returned by the underlying component.

All the configuration identifiers are listed in the file *NxpRdLib_PublicRelease/intfs/phacDiscLoop.h*. The developer can recognize them because they use the following naming scheme:

- `PHAC_DISCLOOP_CONFIG_XXX_XXX`

For instance, to set the number of iterations (or loops) of the Discovery Loop to five, this can be configured with:

```
/* Set number of polling loops to 5 */
status = phacDiscLoop_SetConfig(pDataParams, PHAC_DISCLOOP_CONFIG_NUM_POLL_LOOPS,
5);
CHECK_SUCCESS(status);
```

`pDataParams`: Pointer to the `phacDiscLoop_Sw_DataParams_t` parameter component.

`PHAC_DISCLOOP_CONFIG_NUM_POLL_LOOPS`: The configuration identifier for the number of iterations of the Discovery Loop.

`5`: The number of iterations to be set in the Discovery Loop routine.

6.1.5 Discovery Loop Get Configuration

The developer can use this function to get the configuration parameters of the Discovery Loop:

```
phStatus_t phacDiscLoop_GetConfig(
      void        *pDataParams,                    [In]
      uint16_t    wConfig,                         [In]
      uint16_t    *pvalue                          [Out]
```

***pDataParams:** Pointer to the **`phacDiscLoop_Sw_DataParams_t`** parameter component.

wConfig: The configuration identifier.

***pValue:** The returned configuration value.

The values returned by the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_ERR_INVALID_PARAMETER`: Invalid option/response received.

`Other`: Value returned by the underlying component.

For instance, to obtain the tag types detected during one Discovery Loop iteration, this can be done in the following way:

```
/* Get the tag types detected info */
status = phacDiscLoop_GetConfig(pDataParams, PHAC_DISCLOOP_CONFIG_TAGS_DETECTED,
     &wTagsDetected);
```

`pDataParams`: Pointer to the `phacDiscLoop_Sw_DataParams_t` parameter component.

`PHAC_DISCLOOP_CONFIG_TAGS_DETECTED`: The configuration identifier for the tags detected during one iteration of the Discovery Loop.

`wTagsDetected`: The binary map indicating which tag types were found.

It can be checked if one particular Type tag has been detected by using the defined bitmasks. For instance, to inspect if a Type 2 tag has been detected, this can be done in the following way:

```
if (PHAC_DISCLOOP_CHECK_ANDMASK(wTagsDetected, PHAC_DISCLOOP_TYPEA_DETECTED_TAG_TYPE2))
     printf ("Type A T2 tag detected ");
```

6.1.6 Discovery Loop Configurable Parameters

The Discovery Loop allows several configurations and fine tuning that allow developers to set up their required Discovery Loop according their application needs. All the configuration identifiers are listed in the file *NxpRdLib_PublicRelease/intfs/phacDiscLoop.h*. The developer can recognize them because they use the following naming scheme:

- `PHAC_DISCLOOP_CONFIG_XXX_XXX`

The developer can set their discovery loop settings using the `phacDiscLoop_SetConfig()` function. Similarly, the developer can get the current setting using the `phacDiscLoop_GetConfig()` function. The configuration identifier implemented are:

- `#define PHAC_DISCLOOP_CONFIG_GTA_VALUE_US`: Sets the guard time for Type A tag detection in microseconds magnitude.
- `#define PHAC_DISCLOOP_CONFIG_GTB_VALUE_US`: Sets the guard time for Type B tag detection in microsecond magnitude.
- `#define PHAC_DISCLOOP_CONFIG_GTF_VALUE_US`: Sets the guard time for Type F tag detection in microsecond magnitude.
- `#define PHAC_DISCLOOP_CONFIG_GTA_VALUE_MS`: Sets the guard time for Type A tag detection in millisecond magnitude.
- `#define PHAC_DISCLOOP_CONFIG_GTB_VALUE_MS`: Sets the guard time for Type B tag detection in millisecond magnitude.
- `#define PHAC_DISCLOOP_CONFIG_GTF_VALUE_MS`: Sets the guard time for Type F tag detection in millisecond magnitude.
- `#define PHAC_DISCLOOP_CONFIG_MODE`: Sets the polling mode options. The allowed loop mode combinations are the following:
  - `PHAC_DISCLOOP_SET_POLL_MODE | PHAC_DISCLOOP_SET_LISTEN_MODE`
  - `PHAC_DISCLOOP_SET_POLL_MODE | PHAC_DISCLOOP_SET_PAUSE_MODE`

- PHAC_DISCLOOP_SET_LISTEN_MODE

- #define PHAC_DISCLOOP_CONFIG_NUM_POLL_LOOPS: Sets the number of iterations the discovery loop will do.
- #define PHAC_DISCLOOP_CONFIG_TYPEA_POLL_LIMIT: Sets the number of times the polling loop scans for Type A tags.
- #define PHAC_DISCLOOP_CONFIG_TYPEA_DEVICE_LIMIT: Sets the number of Type A tags that can be detected.
- #define PHAC_DISCLOOP_CONFIG_TYPEB_DEVICE_LIMIT: Sets the number of Type B tags that can be detected.
- #define PHAC_DISCLOOP_CONFIG_TYPEF_DEVICE_LIMIT: Sets the number of Type F tags that can be detected.
- #define PHAC_DISCLOOP_CONFIG_TYPEB_POLL_LIMIT: Sets the number of times the polling loop scans for Type B tags.
- #define PHAC_DISCLOOP_CONFIG_TYPEB_AFI_REQ: Sets the AFI to be used during REQB. AFI shoulb be set to zero if it is required that all the cards should respond regardless of AFI.
- #define PHAC_DISCLOOP_CONFIG_TYPEB_EXTATQB: Enables or disables extended ATQB option.
- #define PHAC_DISCLOOP_CONFIG_TYPEB_FSDI: Sets the FSDI for the Type B tags.
- #define PHAC_DISCLOOP_CONFIG_TYPEB_CID: Sets the CID for Type B tags.
- #define PHAC_DISCLOOP_CONFIG_TYPEB_DRI: Sets the DRI for Type B tags.
- #define PHAC_DISCLOOP_CONFIG_TYPEB_DSI: Sets the DSI for Type B tags.
- #define PHAC_DISCLOOP_CONFIG_TYPEF_SYSTEM_CODE: Sets the system code for the selection of FeliCa tags
- #define PHAC_DISCLOOP_CONFIG_TYPEF_TIME_SLOT: Sets the time slot for detecting Type F tags.
- #define PHAC_DISCLOOP_CONFIG_TYPEF_POLL_LIMIT: Sets the number of times the polling loop scans for Type F tags.
- #define PHAC_DISCLOOP_CONFIG_TAGS_DETECTED: Returns the tags that were detected.
- #define PHAC_DISCLOOP_CONFIG_GTA_VALUE: The time units used for GTA timer are returned.
- #define PHAC_DISCLOOP_CONFIG_GTB_VALUE: The time units used for GTB timer are returned.
- #define PHAC_DISCLOOP_CONFIG_GTF_VALUE: The time units used for GTF timer are returned.
- #define PHAC_DISCLOOP_CONFIG_BAIL_OUT: Sets the bail-out parameter.
- #define PHAC_DISCLOOP_CONFIG_TYPEA_NR_TAGS_FOUND: Returns the number of Type A tags found (GET).
- #define PHAC_DISCLOOP_CONFIG_TYPEA_NR_TAGS_ACTIVATED: Returns number of Type A tags that are activated (GET).
- #define PHAC_DISCLOOP_CONFIG_TYPEB_NR_TAGS_FOUND: Returns the number of Type B tags found (GET).

- `#define PHAC_DISCLOOP_CONFIG_TYPEB_NR_TAGS_ACTIVATED`: Returns number of Type B tags that are activated (GET).
- `#define PHAC_DISCLOOP_CONFIG_TYPEF_NR_TAGS_FOUND`: Returns the number of Type F tags found (GET).
- `#define PHAC_DISCLOOP_CONFIG_TYPEF_NR_TAGS_ACTIVATED`: Returns number of Type F tags that are activated (GET).
- `#define PHAC_DISCLOOP_CONFIG_TYPEA_I3P4_FSDI`: Sets the Fsdi for the 14443-4A communication.
- `#define PHAC_DISCLOOP_CONFIG_TYPEA_I3P4_CID`: Sets the CID for the 14443-4A communication.
- `#define PHAC_DISCLOOP_CONFIG_TYPEA_I3P4_DRI`: Sets the Dri for the 14443-4A communication.
- `#define PHAC_DISCLOOP_CONFIG_TYPEA_I3P4_DSI`: Sets the Dsi for the 14443-4A communication.
- `#define PHAC_DISCLOOP_CONFIG_TYPEA_P2P_DID`: Sets DID for Type A P2P device communication.
- `#define PHAC_DISCLOOP_CONFIG_TYPEA_P2P_LRI`: Sets LRi for Type A P2P device communication
- `#define PHAC_DISCLOOP_CONFIG_TYPEA_P2P_NAD_ENABLE`: Enables NAD if wValue = 1. Otherwise, it else disables NAD.
- `#define PHAC_DISCLOOP_CONFIG_TYPEA_P2P_NAD`: Sets the NAD for P2P device communication.
- `#define PHAC_DISCLOOP_CONFIG_TYPEA_P2P_GI_LEN`: Sets the length of the General Bytes.
- `#define PHAC_DISCLOOP_CONFIG_TYPEA_P2P_ATR_RES_LEN`: Sets the Attribute Response length.
- `#define PHAC_DISCLOOP_CONFIG_TYPEF_P2P_DID`: Sets DID for Type F P2P device communication.
- `#define PHAC_DISCLOOP_CONFIG_TYPEF_P2P_LRI`: Sets LRi for Type F P2P device communication.
- `#define PHAC_DISCLOOP_CONFIG_TYPEF_P2P_NAD_ENABLE`: Enables NAD if wValue = 1. Otherwise, it else disables NAD.
- `#define PHAC_DISCLOOP_CONFIG_TYPEF_P2P_NAD`: Sets the NAD for P2P device communication.
- `#define PHAC_DISCLOOP_CONFIG_TYPEF_P2P_GI_LEN`: Sets the length of the General Bytes.
- `#define PHAC_DISCLOOP_CONFIG_TYPEF_P2P_ATR_RES_LEN`: Sets the Attribute Response length.
- `#define PHAC_DISCLOOP_CONFIG_PAUSE_PERIOD`: Sets the pause time interval.
- `#define PHAC_DISCLOOP_CONFIG_PAUSE_PERIOD_MS`: Sets the time magnitude for the pause time interval.
- `#define PHAC_DISCLOOP_CONFIG_DETECT_TAGS`: Specifies the types of tags to be detected.
- `#define PHAC_DISCLOOP_CONFIG_STOP`: Option used to stop the discovery loop.
- `#define PHAC_DISCLOOP_CONFIG_ANTI_COLL`: Option to set anticollsion flag.

- `#define PHAC_DISCLOOP_CONFIG_TYPEF_BAUD:` Sets get baud rate used for FeliCa detection.
- `#define PHAC_DISCLOOP_CONFIG_LISTEN_TIMEOUT:` Sets the listen time in millisecond.
- `#define PHAC_DISCLOOP_CONFIG_ACTIVE_BAUD:` Sets the baud rate used for Active communication.

6.1.7 Discovery Loop Start Routine

The Discovery Loop is started by calling the `phStatus_t phacDiscLoop_Start()` function. This function takes the `bLoopMode` configured in `phacDiscLoop_Sw_DataParams_t` data structure and starts the Discovery Loop accordingly (`PHAC_DISCLOOP_SET_POLL_MODE, PHAC_DISCLOOP_SET_PAUSE_MODE, PHAC_DISCLOOP_SET_LISTEN_MODE)`.All the settings must be done before calling the start function.

```
phStatus_t phacDiscLoop_Start (void * pDataParams)
```

`*pDataParams:` Pointer to the Discovery Loop parameter component.

6.1.8 Discovery Loop - Activate Card

The `phacDiscLoop_Sw_ActivateCard()` function activates the tag type referenced by a given index. This function should follow a previous successful tag detection, passing the tag type as the input argument. In case of a Type A tag according to the detected SAK the tag is additionally activated as ISO14443-4A or ISO18092 for P2P.

Note: This function does not provide FeliCa tags activation since those are automatically activated with the detection.

The `phacDiscLoop_Sw_ActivateCard()` function does not provide anticollision resolution, since anticollision is implemented within the `phacDiscLoop_Start()` function.

```
phStatus_t phacDiscLoop_Sw_ActivateCard(
      phacDiscLoop_Sw_DataParams_t   *pDataParams,          [In]
      uint8_t                        bTagType,              [In]
      uint8_t                        bTagIndex );           [In]
```

***pDataParams:** Pointer to the `phacDiscLoop_Sw_DataParams_t` parameter component. If the tag type is activated successfully, the tag component (i.e. `phacDiscLoop_TypeA_Tags_t` or `phacDiscLoop_TypeB_Tags_t` or `phacDiscLoop_TypeF_Tags_t data structure)` is fulfilled with the parameters of the activated tag.

bTagType: `PHAC_DISCLOOP_TYPEA_ACTIVATE` - activate Type A tags or `PHAC_DISCLOOP_TYPEB_ACTIVATE` – activate Type B tags.

bTagIndex: The tag which has to be activated.

The values returned by the function can be:

`PH_ERR_INVALID_PARAMETER`: Invalid value of `bTagType`

`PH_ERR_INVALID_DATA_PARAMS`: `bTagIndex` is greater than number of previously found (detected) tags.

`PH_ERR_SUCCESS`: Operation successful.

`Other`: Value returned by the underlying component.

7. NFC Reader Library API: NFC P2P Package

In this section, the LLCP and SNEP components are explained in depth. The proper use of the LLCP and SNEP layers will allow the developers to develop an application that can exchange data with another P2P device.

7.1 LLCP

7.1.1 Technical Introduction

The Logical Link Control Protocol (LLCP) NFC Forum's specification [20] provides the procedural means for the transfer of upper layer information units between two NFC Forum Devices.

Located on top of the ISO/IEC 18092 Protocol Layer component, LLCP defines a set of procedures that represent an abstraction of the data link service. This abstraction of the RF field is provided by the LLC Medium Access Control component.

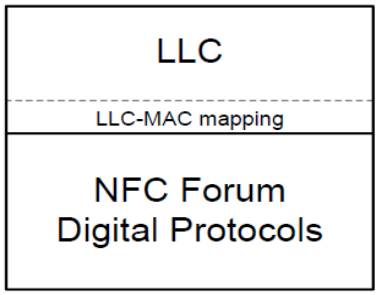


Fig 32. LLCP integration on top of the RF field

Two LLCP components exchange information in the form of LLC Protocol Data Units (PDU). LLC PDUs exchange either LLC management data units such as CONNECT PDU and DISC PDU or upper layer information encapsulated in I PDU and UI PDU.

7.1.1.1 LLCP Functionalities

According to the specification, these are the main functionalities to be implemented by the LLCP protocol:

- *Link Activation, Supervision and Deactivation:* A NFC Device shall recognize compatible LLCP devices that are brought into proximity, shall establish a LLCP Link, shall supervise the connection to the remote peer device, and shall deactivate the link if requested.
- *Asynchronous Balanced Communication:* In order to provide a balanced communication between peers, a NFC device sending a packet to its peer shall wait for its response before the transmission of the next packet.
- *Protocol Multiplexing:* The LLCP link protocol shall be able to accommodate several instances of higher level protocols and at the same time allow multiple logical connections to exist simultaneously. The NFC Reader Library is limited to establish LLCP link connections with a maximum of 5 other devices.

- *Connectionless / Connection-oriented Transport:* Both connectionless (minimal setup with no reliability or flow control guarantees) and connection-oriented (sequenced and guaranteed delivery of service data units) communications shall be supported.

7.1.1.2 LLCP Components

In order to provide the aforementioned functionalities, the LLCP specification defines a set of logical components.

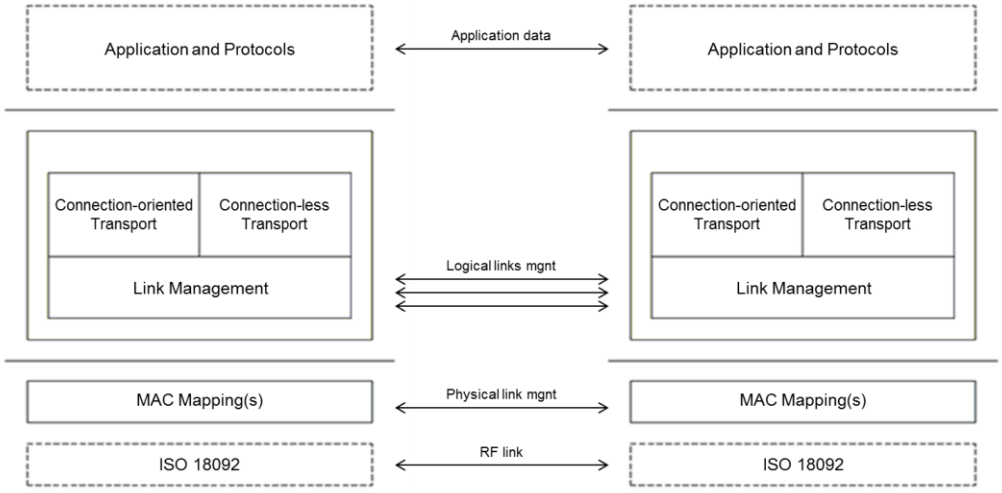


Fig 33. LLCP Components

- *MAC Mapping:* Integrates the LLC layer on top of the ISO/IEC 18092 standard. Guarantees the reliability of the RF link, ensuring that all LLC PDU packets transmitted by a peer are received by the other peer in a sequenced manner.
- *Link Management:* Provides link management from the LLC point of view and handles the logical link communications from upper layer services.
- *Connection-oriented Transport:* A data transmission service with sequenced and guaranteed delivery of service data units that requires connection establishment before the LLCP PDU packets can be transmitted.
- *Connectionless Transport:* An unacknowledged data transmission service with minimal protocol complexity that does not requires connection establishment.

7.1.2 LLCP Link Layer

The LLCP link layer component implemented in the Link Layer of the NFC P2P Package of the NFC Reader Library is fully compliant with the NFC Forum LLCP specification [20].

The LLCP component purpose is to properly create, manage, maintain and disconnect a communication channel between peers for the transfer of upper layer protocols messages.

Note: The structures and functions defined in the LLCP component API include the *FRI* (Forum Reference Implementation) label to indicate compliancy with the NFC Forum specification (i.e. `phlnLlcp_Fri_DataParams_t`)

7.1.2.1 LLCP Structure

The LLCP component defines two main structures for the correct implementation of the functionalities that are associated to it.

LLCP Generic structure

The `phlnLlcp_Fri_DataParams_t` structure contains pointers to buffers and component structures of the underlying components that need to be handled by the LLCP layer. Therefore, the `phlnLlcp_Fri_DataParams_t` structure is the common input parameter for all LLCP API functions.

```
typedef struct {
      phlnLlcp_Fri_t * pLlcp;
      phlnLlcp_sLinkParameters_t * pLinkParams;
      phlnLlcp_Fri_Transport_ * pLlcpSocketTable;
      void * pTxBuffer;
      uint16_t wTxBufferLength;
      void * pRxBuffer;
      uint16_t wRxBufferLength;
      phHal_sRemoteDevInformation_t * pRemoteDevInfo;
      void * pLowerDevice;
} phlnLlcp_Fri_DataParams_t;
```

***pLlcp**: Pointer to the LLCP data parameter component.

***pLinkParams**: Pointer to the phlnLlcp_sLinkParameters_t data parameter component.

***pLlcpSocketTable**: Pointer to the `phlnLlcp_Fri_Transport_t` parameter component.

***pRemoteDeviceInfo**: Pointer to the remote peer device information stored at the `phHal_sRemoteDevInformation_t` parameter component. This information will be obtained from the ATR Response command (see section 4.4.3.4) and should be filled by the developer.

***pTxBuffer**: Pointer to the transmission buffer.

wTxBufferLength: Length of the transmission buffer.

***pRxBuffer**: Pointer to the reception buffer.

wRxBufferLength: Length of the reception buffer.

***pLowerDevice**: Pointer to the underlying `palI18092mPI` or `palI18092mT` PAL data parameter component.

LLCP FRI structure

In addition to the `phlnLlcp_Fri_t` structure containing references to LLCP components, the LLCP implementation defines its own structure in order to store LLCP configuration values associated to the communication channel.

```
typedef struct {
      uint8_t state
      uint8_t nSymmetryCounter
```

```
        uint8_t version
        uint8_t pFrmrInfo [4]
        uint8_t pCtrlTxBuffer [10]
        uint8_t pCtrlTxBufferLength
        uint8_t bDiscPendingFlag
        uint8_t bFrmrPendingFlag
        uint16_t nRxBufferLength
        uint16_t nTxBufferLength
        phlnLlcp_Fri_sPacketHeader_t sFrmrHeader
        phlnLlcp_Fri_Mac_ePeerType_t eRole
        uint32_t hSymmTimer
        phlnLlcp_Fri_Recv_CB_t pfRecvCB
        uint8_t * pRxBuffer
        phlnLlcp_Fri_sPacketHeader_t * psSendHeader
        phlnLlcp_Fri_sPacketSequence_t * psSendSequence
        phNfc_sData_t * psSendInfo
        void * pLinkContext
        void * pChkContext
        void * pSendContext
        void * pRecvContext
        void * osal
        phlnLlcp_Fri_sLinkParameters_t sLocalParams
        phlnLlcp_Fri_sLinkParameters_t sRemoteParams
        phNfc_sData_t sRxBuffer
        phNfc_sData_t sTxBuffer
        phlnLlcp_Fri_LinkStatus_CB_t pfLink_CB
        phlnLlcp_Fri_Check_CB_t pfChk_CB
        phlnLlcp_Fri_Send_CB_t pfSendCB
        phlnLlcp_Fri_Mac_t  MAC
} phlnLlcp_Fri_t;
```

State: Current state of the LLCP component. The Library defined values detailed in Table 17.

Table 17. States of the LLC state machine

Value	NFC Library FRI STATE identifier	Meaning
0	RESET_INIT	Initial state
1	CHECKED	The remote peer device has been checked for LLCP compliance
2	ACTIVATION	The activation phase
3	PAX	Parameter exchange phase
4	OPERATION_RECV	Normal operation phase (ready to receive)
5	OPERATION_SEND	Normal operation phase (ready to send)
6	DEACTIVATION	The deactivation phase

nSymmetryCounter: `Activity counter used to handle symmetry timeout.`

version: Negotiated LLCP Protocol version.

pCtrlTxBuffer: Control frames buffer.

pCtrlTxBufferLength: Size of the control frames buffer.

bDiscPendingFlag: Pending flag of the Disconnect packet.

bFrmrPendingFlag: Pending flag of the Frame Reject packet.

sFrmrHeader: Header of the Frame Reject packet.

pFrmrInfo: Info of the Frame Reject packet.

nRxBufferLength: Actual size of the reception buffer.

nTxBufferLength: Actual size of the transmission buffer.

eRole: Role of the peer in the communication.

hSymmTimer: Identifier of the timer that ensures the symmetry of the link.

***pRxBuffer**: Base reception buffer.

***psSendHeader**: Header of the transmission pending packet.

***psSendSequence**: Sequence of the transmission pending packet.

***psSendInfo**: Data of the transmission pending packet.

***pLinkContext**: Context for the link status notification callback function.

***pChkContext**: Context for the compliance checking callback function.

***pSendContext**: Context for the sending result callback functions.

***pRecvContext**: Context for the reception result callback functions.

***osal**: Pointer to the OSAL component parameter.

sLocalParams: Pointer to the Local parameters.

sRemoteParams: Pointer to the Remote parameters.

sRxBuffer: Internal reception buffer. Should not exceed nRxBufferSize value.

sTxBuffer: Internal transmission buffer. Should not exceed nTxBufferSize value.

pfLink_CB: Pointer to the link status notification callback function.

pfChk_CB: Pointer to the compliance checking callback function.

pfSendCB: Pointer to the sending result callback functions.

pfRecvCB: Pointer to the reception result callback functions.

MAC: Pointer to the MAC Mapping component parameter.

7.1.2.2 Initialization of the LLCP component

The LLCP component must be initialized before it can be used for the link management. The initialization function sets the references to the parameters structures and buffers.

```
phStatus_t phlnLlcp_Fri_Init(
     phlnLlcp_Fri_DataParams_t * pDataParams,                [In]
     uint16_t wSizeOfDataParams,                             [In]
     phlnLlcp_t * pLlcp,                                     [In]
     phlnLlcp_sLinkParameters_t * pLinkParams,               [In]
     phlnLlcp_Transport_t * pLlcpSocketTable,                [In]
     phHal_sRemoteDevInformation_t * pRemoteDevInfo,         [In]
     void * pTxBuffer,                                       [In]
     uint16_t wTxBufferLength,                               [In]
     void * pRxBuffer,                                       [In]
     uint16_t wRxBufferLength,                               [In]
     void * pLowerDevice );                                  [In]
```

***pDataParams**: Pointer to the `phlnLlcp_Fri_DataParams_t`, the parameter component to be initiated.

wSizeOfDataParams: Size of the pDataParams parameter component.

Rest of parameters: These parameters are members of `phlnLlcp_Fri_DataParams_t` (see section 7.1.1.2). Only the content of `*pRemoteDevInfo` and `*pLinkParams` should be manually defined by the developer, the rest of the parameter components are filled by the function during its call.

The value returned by the function is:

`PH_ERR_SUCCESS`: Successful Operation.

Once the initialization has been successfully completed, the LLCP Reset function should be called by the developer in order to initialize the `phlnLlcp_Fri_DataParams_t` structure to the default values for the later LLC Link activation. This function automatically calls the `phlnLlcp_Reset` and `phlnLlcp_Fri_Mac_Reset` functions of the LLC and the MAC Mapping components for their automatic reset.

In addition to the parameters reset, the `phlnLlcp_Reset` function checks the validity of the parameters that have been set during the initialization phase, such as the reception and transmission buffer size depending on the defined Maximum Information Unit (MIU).

This reset function is a prerequisite for the correct execution of `phlnLlcp_Activate()` and `phlnLlcp_Deactivate()`.

```
phStatus_t phlnLlcp_Reset  (
     void *  pDataParams,                                [In]
     phlnLlcp_Fri_LinkStatus_CB_t  pfLink_CB,            [In]
     void *  pContext);                                  [In]
```

***pDataParams**: Pointer to the `phFriNfc_Llcp_t` parameter component.

pfLink_CB: Pointer to the user defined link status notification callback that refers to the activated or deactivated status of the link.

***pContext**: Pointer to the input data to be processed by the callback function.

The values returned by the function can be:

`NFCSTATUS_BUFFER_TOO_SMALL`: Receive buffer is not large enough to support 131 bytes (128 + 2 + 1 == MIU + Header + Sequence) or Transmit buffer is too small to support maximum LLCP frame size (Header + Sequence + MIU)

`NFCSTATUS_INVALID_PARAMETER`: MIU in `psLinkParams` is lower than 128 bytes

`PH_ERR_SUCCESS`: Successful Operation.

`Other`: Value returned by the underlying component.

7.1.3 LLC MAC Mapping Component

The Medium Access Control component is placed on top of the Protocol Abstraction Layer defined by the NFC Reader Library. Its main objective is to provide abstraction of underlying RF standards to the LLC layer. This abstraction covers the functions of data unit fragmentation if required, sequenced and error free delivery of data units, error recovery management and the notification to the upper layer of unrecoverable transmission errors.

The MAC Mapping component is necessary for nearly all the LLCP layer link related functionalities such as reception and transmission of packets and link activation and link deactivation.

According to the NFC Reader Library P2P Package implementation, the LLC link API is responsible for the MAC Mapping API management, and therefore the MAC Mapping functions are not directly called by developers.

MAC Mapping Structure

The MAC layer component defines its own structure, which is responsible for the management of the physical link between the two peers. This structure contains information about the remote peer device and the data exchange status such as pending messages, timeouts, among others.

```
Typedef struct {
    uint8_t  RecvPending;
    uint8_t  SendPending;
    phlnLlcp_Fri_Mac_eLinkStatus_t LinkState;
    phlnLlcp_Fri_Mac_ePeerType_t PeerRemoteDevType;
    phlnLlcp_Fri_Mac_LinkStatus_CB_t LinkStatus_Cb;
    phlnLlcp_Fri_Mac_Send_CB_t MacSend_Cb;
    phlnLlcp_Fri_Mac_Reveive_CB_t MacReceive_Cb;
    phNfc_sData_t *psReceiveBuffer;
    phNfc_sData_t *psSendBuffer;
    phHal_sRemoteDevInformation_t *psRemoteDevInfo; information;
    void  *MacReceive_Context; context;
    void  *LowerDevice;
    void  *MacSend_Context;
    void  *LinkStatus_Context;
    void  *Osal;
    phNfc_sData_t sConfigParam;
    phlnLlcp_Fri_CplRt_t MacCompletionInfo;
    phlnLlcp_Fri_Mac_Interface_t LlcpMacInterface;
} phlnLlcp_Fri_Mac_t;
```

RecvPending: Pending flag for the reception.

SendPending: Pending flag for the transmission.

LinkState: Information of the link status. NFC Reader Library defined values can be found in Table 18.

Table 18. Link state values for MAC layer

Link status	Identifier in the NFC Reader Library
Default link status	`phlnLlcp_Fri_Mac_eLinkDefault`
Link activated	`phlnLlcp_Fri_Mac_eLinkActivated`
Link deactivated	`phlnLlcp_Fri_Mac_eLinkDeactivated`

PeerRemoteDevType: Information about the remote peer´s device role: Initiator or Target.

LinkStatus_Cb: Callback function for link status.

MacSend_Cb: Callback function for MAC sending.

MacReceive_Cb: Callback function for MAC receiving.

***psReceiveBuffer**: Reception buffer.

***psSendBuffer**: Transmission buffer.

***psRemoteDevInfo**: The MAC layer has to be filled with information of the NFC device detected in the RF field. Since there is no function providing this particular operation, it must be done manually by the developer using the `RemDevType` parameter in `phNfc_sRemoteDevInformation_t` component, which contains information of the peer detected according to Table 19.

Table 19. Device type for MAC layer

In NFC Reader Library the MAC layer functionality is authorized for the following peer devices

Device type	Identifier in the NFC Reader Library
ISO18092 P2P initiator	`phHal_eNfcIP1_Initiator`
ISO18092 P2P target	`phHal_eNfcIP1_Target`

***LinkStatus_Context**: Context input for link status callback function.

***MacSend_Context**: Context input for MAC sending callback function.

***MacReceive_Context**: Context input for MAC receiving callback function.

***Osal**: `Pointer to the OSAL component parameter.`

sConfigParam: Buffer for the configurations parameter.

***LowerDevice**: Holds the completion routine information of the MAC Mapping Layer.

MacCompletionInfo: MAC completion routine for the lower layer

LlcpMacInterface: Generic interface structure with the lower layer. It defines five function interfaces: check, activate, deactivate, send and receive.

MAC Mapping API

The MAC Mapping functionalities are managed by the NFC Reader Library core, therefore there is no API exposed to the customer to handle this component.

MAC Mapping Structure

The MAC Mapping functionalities are managed by the NFC Reader Library core, therefore there are no callback functions exposed to the customer by this component.

7.1.4 LLC Link Component

The LLC link component is responsible for the management of LLCP links between peers. As a part of this management, the LLC link component implements the following functionalities:

1. Link Check: `phlnLlcp_ChkLlcp()`
2. Link Activation: `phlnLlcp_Activate()`
3. Link Deactivation: `phlnLlcp_Deactivate()`
4. Packet transmission: `phlnLlcp_Send()`
5. Packet reception: `phlnLlcp_Recv()`

In order to be able to complete all these steps in the correct way, the LLC link should be correctly initialized, which is ensured by a previous call to `phlnLlcp_Reset()` function (see section7.1.2.2).

The Fig 34 shows the proposed flow for a correct execution of the LLCP Link component with both functions and their associated callbacks.

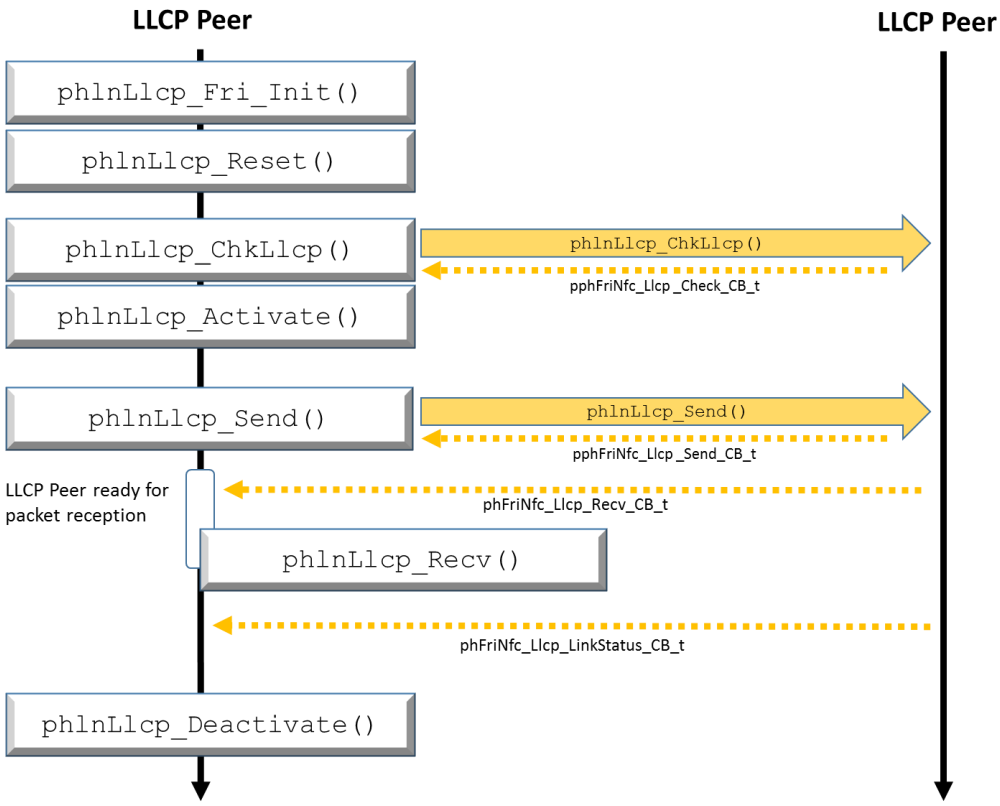


Fig 34. LLCP Link Component operation flow

As it is explained in detail in the LLCP Link API section (see section 7.1.4.2), some of the functions that are depicted in this flow should not be directly called by developers since they are internally called by upper layer components such as the LLCP Transport component.

7.1.4.1 LLC Link Structure

The LLC link component defines a structure containing specific configuration parameters for a LLC link connection. Since LLC links do not have to be equal in both directions, phlnLlcp_Fri_t structure (see section 7.1.2.1) stores a different entrance for each link direction.

```
typedef struct phFriNfc_Llcp_sLinkParameters {
      uint16_t miu;
      uint16_t wks;
      uint8_t lto;
      uint8_t option;
```

```
} phlnLlcp_Fri_sLinkParameters_t;
```

Table 20. Parameters

In the right column there are default values given by the Initialization function (see section 7.1.2.2). Apart from these parameters, there are few others for internal management.

Parameter	Description	Default init value
miu	Maximum Information Unit	128 bytes
wks	Well-Known Services	1
lto	Link Timeout (in steps of 10 ms)	10 (means 100 msec)
option	Options	0

The link characteristics shall be negotiated on the ISO/IEC 18092 communication setup. The LLC link characteristics are exchanged and negotiated as part of the Attribute Request command (see section 4.4.2.3) and/or using the Generic Information bytes (see LLCP NFC Forum specification [20]).

7.1.4.2 LLC Link API

This section defines the functions provided by the NFC Reader Library in order to complete the LLC Link specific functionalities.

Check

This function checks the Attribute Response coming from the remote peer device and decides whether the MAC layer can be enabled for that device or not. It also enables the internal connection between LLC layer and MAC layer.

The General Bytes received in the Attribute Response shall indicate the LLCP capabilities by containing the LLC Magic Number (0x46, 0x66, 0x6D) defined in the specification. The rest of data transmitted as part of the General Bytes is afterwards associated to the remote link configuration parameters rather than sending Parameter Exchange (PAX PDU) commands in the LLC Link layer.

```
phStatus_t phlnLlcp_ChkLlcp(
     void * pDataParams,                                [In]
     phlnLlcp_Check_CB_t pfCheck_CB,                    [In]
     void * pContext );                                 [In]
```

***pDataParams**: Pointer to the `phlnLlcp_Fri_DataParams_t` parameter component.

pfCheck_CB: Pointer to the callback function that will be called when the link checking procedure is completed.

***pContext**: Pointer to the input data to be processed by the callback function.

The values returned by the function can be:

`PH_ERR_SUCCESS`: Successful Operation.

`NFCSTATUS_INVALID_STATE`: LLCP link is not in the `PHFRINFC_LLCP_STATE_RESET_INIT` state.

`NFCSTATUS_INVALID_DEVICE`: Device type unable to perform NFC P2P.

`NFCSTATUS_FAILED`: Attribute Response from remote peer device does not match with LLC Magic Number.

`Other`: Value returned by the underlying component.

Activate LLC Link

This function provides a basic LLC initial configuration and activates both the LLC link and the MAC Mapping components. The LLC link state is changed from checked – PHFRINFC_LLCP_STATE_CHECKED – to activated – PHFRINFC_LLCP_STATE_ACTIVATION –.

In order to activate the link, Maximum Information Unit (MIU), Well-Known Service list (WKS), Link Timeout (LTO), and Option (OPT) values [20] of the remote peer device must be known. These values should have been already received as part of the Discovery Loop component, which implements ISO/IEC 18092 defined Attribute Request procedure. However, if there is not LLCP Magic Number within the Attribute Response and the local device plays the Initiator role, then the activation procedure PAX PDU with local link parameters is transmitted to the remote peer device playing the Target role.

Once the successful activation of the LLC component is completed, a notification regarding the link status change towards the service layer is done via the user defined callback function pfLink_CB(). In case of unsuccessful link activation, the deactivation callback function is automatically executed.

```
phStatus_t phlnLlcp_Activate(
     void * pDataParams );                                    [In]
```

***pDataParams**: Pointer to the phlnLlcp_Fri_DataParams_t component.

The values returned by the function can be:

PH_ERR_SUCCESS: Successful Operation.

NFCSTATUS_INVALID_STATE: LLCP not in state PHFRINFC_LLCP_STATE_CHECKED

Other: Value returned by the underlying component.

Deactivate LLC Link

This function deactivates the MAC interface and disconnects the LLCP link. The LLCP link connections are canceled by sending DISC PDU. If the sending operation is pending, then the disconnect procedure is terminated.

After completing a successful LLC link deactivation, the service layer notifies the link status change through a user defined callback function (pfLink_CB.)

```
phStatus_t phlnLlcp_Deactivate(
     void * pDataParams );                                    [In]
```

***pDataParams**: Pointer to the phlnLlcp_Fri_DataParams_t component.

The values returned by the function can be:

PH_ERR_SUCCESS: Successful Operation.

NFCSTATUS_INVALID_STATE: LLCP link in a state other than PHFRINFC_LLCP_STATE_OPERATION_RECV or PHFRINFC_LLCP_STATE_OPERATION_SEND.

NFCSTATUS_PENDING: the DISC PDU is not sent because the link is waiting for a status notification related to a previous operation.

Other: Value returned by the underlying component.

Send PDU Packet via LLCP Link

This function is used to send a LLC PDU via the LLCP component. The packets must be passed as a header - sequence field - information field sequence.

This function can only be called in a connection-oriented socket in the connected state.

Note: This function is called through other functions sending PDU packets via the LLCP link. In order to send data packets, the developer should make use of the `phlnLlcp_Transport_Send()` function, which sends data within the Information PDU frame and assembles the header and sequence arguments automatically from a given LLCP socket component parameters.

```
phStatus_t phlnLlcp_Send(
     void * pDataParams,                                [In]
     phlnLlcp_sPacketHeader_t * pHeader,                [In]
     phlnLlcp_sPacketSequence_t * pSequence,            [In]
     phNfc_sData_t * pInfo,                             [In]
     phlnLlcp_Send_CB_t pfSend_CB,                      [In]
     void * pContext );                                 [In]
```

***pDataParams**: Pointer to the `phlnLlcp_Fri_DataParams_t` parameter component.

***pHeader**: Pointer to the PDU packet header composed by command type, the Source SAP and the Destination SAP values.

***pSequence**: Pointer to the PDU packet sequence field.

***pInfo**: Pointer to the PDU packet information field.

pfSend_CB: Pointer to the callback function to be called when the transmission of a PDU packet via LLCP link is successfully executed.

***pContext**: Pointer to the input data to be processed by the callback function.

The values returned by the function can be:

`PH_ERR_SUCCESS`: Successful Operation.

`NFCSTATUS_REJECTED`: Previous send operation has not been completed yet.

`NFCSTATUS_PENDING`: Previous receive operation has not been completed yet.

`NFCSTATUS_INVALID_STATE`: LLCP in a state other than `PHFRINFC_LLCP_STATE_OPERATION_RECV` or `PHFRINFC_LLCP_STATE_OPERATION_SEND`.

`Other`: Value returned by the underlying component.

Receive PDU Packet on the LLC Link

This function sets a callback function in the internal LLC link structure for any transport reception on the LLCP component.

Note**:** The link reception callback is mostly used for library internal purposes – receiving the transport packets and further parsed –.

```
phStatus_t phlnLlcp_Recv(
     void * pDataParams,                                [In]
     phlnLlcp_Recv_CB_t pfRecv_CB,                      [In]
     void * pContext );                                 [In]
```

***pDataParams**: Pointer to the `phlnLlcp_Fri_DataParams_t` parameter component.

pfRecv_CB: Callback function to be executed when any transport data on the LLCP link is received. The NFC Reader Library internally assigns this callback function to the transport reception callback function that is normally used by the developers.

***pContext**: Pointer to the input data to be processed by the callback function.

The values returned by the function can be:

NFCSTATUS_SUCCESS: Successful Operation.

NFCSTATUS_REJECTED: Previously assigned callback (most probably the internal library build-in transport receive callback) has not been executed yet.

7.1.4.3 LLC Link Callback functions

The LLC Link component defines a set of callback functions to inform about incoming requests or changes in the links managed by the LLCP component.

Link Check CB

This callback function is executed when an incoming connection request (CONNECT PDU) is received from a remote peer device.

Set by function: phlnLlcp_ChkLlcp()

Function prototype:

```
typedef void (*phFriNfc_Llcp_Check_CB_t) (
     void *pContext,
     NFCSTATUS status );
```

***pContext**: Pointer to the input data to be processed by the callback function.

status: Status code for the callback function.

Link Status CB

This callback function is executed after the link status has changed to activated or deactivated status.

Set by function: phlnLlcp_Reset()

Function prototype:

```
typedef void (*phFriNfc_Llcp_LinkStatus_CB_t) (
     void *pContext,
     phFriNfc_Llcp_eLinkStatus_t eLinkStatus);
```

***pContext**: Pointer to the input data to be processed by the callback function.

eLinkStatus: New value for the link status: activated or deactivated.

Link Send CB

This callback function is called when a generic LLCP packet has been sent via the LLCP link

Set by function: phlnLlcp_Send()

Function prototype:

```
typedef void (*phFriNfc_Llcp_Send_CB_t) ( void *pContext,
     NFCSTATUS stats );
```

***pContext**: Pointer to the input data to be processed by the callback function.

status: Indicates the correct or incorrect completion of the transmission procedure.

Link Receive CB

This callback function is called when a generic LLCP packet has been received via the LLCP link.

Note: This callback is busy by internal NFC Reader Library function.

Set by function: phlnLlcp_Receive()

Function prototype:

```
typedef void (*phFriNfc_Llcp_Recv_CB_t) (
     void *pContext,
     phNfc_sData_t *psData,
     NFCSTATUS status );
```

***pContext**: Pointer to the input data to be processed by the callback function.

***psData**: Pointer to the received data.

status: Indicates the correct or incorrect completion of the reception procedure.

7.1.5 LLC Transport Component

The LLC Transport component establishes the logical connections between the protocol layer services. For the NFC Reader Library, the protocol running on top of the LLCP component is SNEP.

The LLCP specification defines two types of transport services for the transmission of upper layer data: connection-oriented and connectionless. The connection-oriented transport provides a sequenced and guaranteed delivery of service data units according to a previously established connection whereas the connectionless transport provides an unacknowledged data transmission where no previous connection establishment is required.

In line with two transport components identified, we differentiate two types of links:

- The Logical Data Link for Connectionless transport: A combination of source and destination service access points (SSAP and DSAP) addresses used for unnumbered information transfer.
- The Logical Data Link Connection for Connection-oriented transport: A unique combination of SSAP and DSAP addresses used for numbered information transfer.

The Fig 35 shows an example flow of both functions and callbacks for the correct P2P communication between two NFC Devices according to a connection-oriented transport scheme.

This flow should only be taken as guidance since the functions to be called will vary depending on whether the peer creates or listens and connects to a socket.

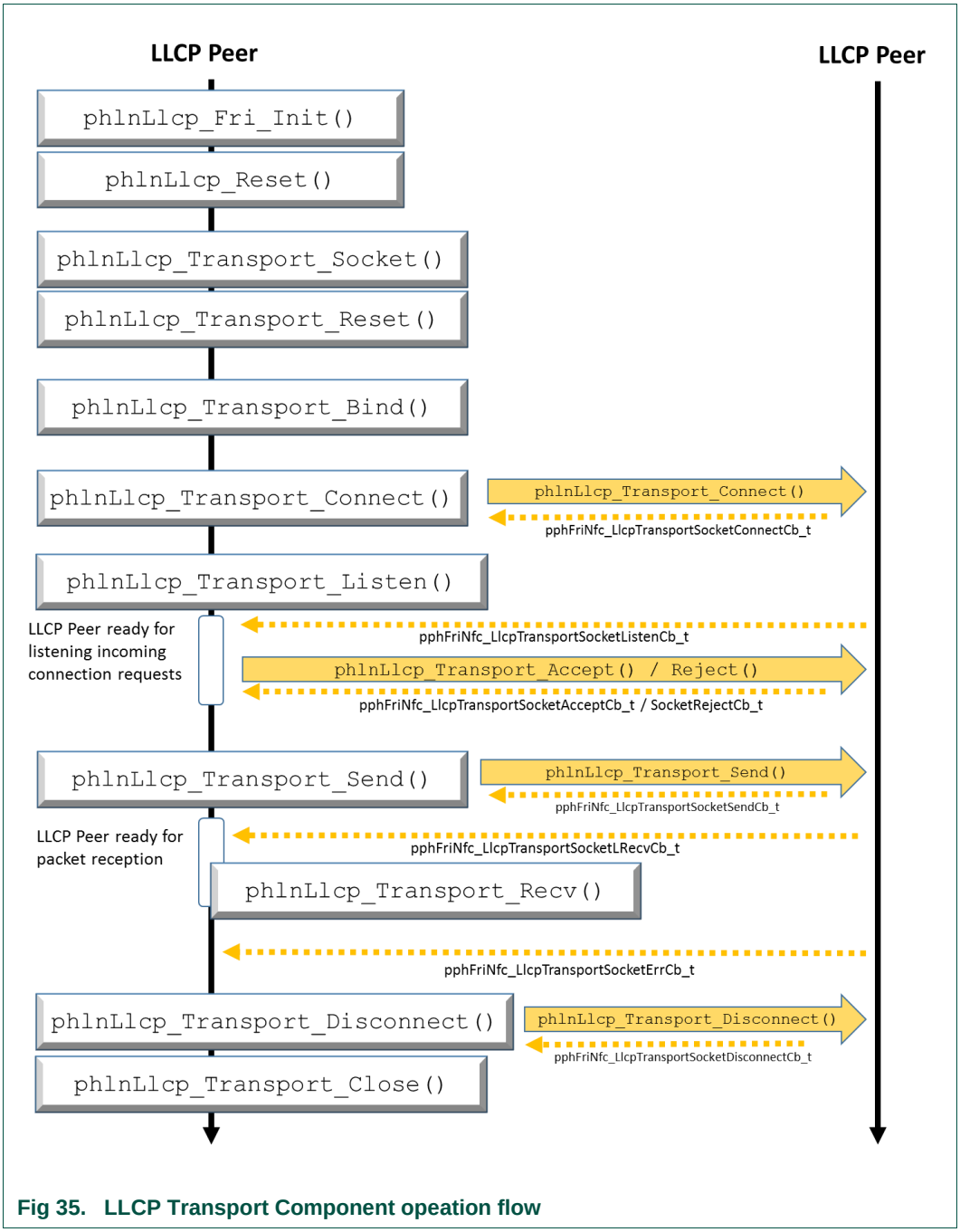


Fig 35. LLCP Transport Component opeation flow

As it is explained in detail in the LLCP Transport API section (see section 7.1.5.2), some of the functions that are described here should not be directly called by the developer since they are called internally by upper layer components such as the SNEP protocol.

Service Access Points - SAP

A SAP is an identifying label of an upper service endpoint.

A LLCP link is uniquely determined by its Source Service Access Point (SSAP), and its Destination Service Access Point (DSAP). These two values are part of LLC PDU

packets and are used by the remote LLC component to identify the destination application layer service.

The LLCP specification defines a set of 15 SAP for well-known services; for instance the SNEP protocol is identified by SAP 04. For those services that are not part of the specification, the service access points can be discovered using the Service Discovery Protocol (SDP).

The SAP assignment rules implemented by the NFC Reader Library are fully compliant with LLCP 1.1 specification by NFC Forum.

Table 21. DSAP/SSAP values

DSAP/SSAP	NFC Forum description
0	Link management
1	Designated well known service access point for the Service Discovery Protocol
2-15	Well-Known Service Access Points
16-31	Shall be assigned by the local LLC component to services registered by itself. These registrations shall be made available by the local SDP instance for discovery and use by a remote LLC.
32-63	Shall be assigned by the local LLC as the result of an upper layer service request and shall not be available for discovery using the SDP

7.1.5.1 LLC Transport structure

The LLC Transport component handles a set of connections associated to upper protocol services. Each LLC link is uniquely identified by its SSAP and DSAP.

The LLC transport structure is the entry point for the `phlnLlcp_Fri_Transport_Socket_t` structure array, which holds information about transport connection between peers associated to upper layer services.

The `phlnLlcp_Fri_Transport` structure maintains the list of all discovered and announced SAPs and their current status. In order to facilitate the management of SAPs and their associated service names, the `phlnLlcp_Fri_Transport` defines the `pCachedServiceNames` parameter which connects both values.

```
struct phlnLlcp_Fri_Transport {
    uint8_t bSendPending;
    uint8_t bRecvPending;
    uint8_t bDmPending;
    uint8_t bFrmrPending;
    uint8_t socketIndex;
    uint8_t LinkStatusError;
    uint8_t nDiscoveryListSize;
    uint8_t nDiscoveryReqOffset;
    uint8_t nDiscoveryResOffset;
    uint8_t nDiscoveryResListSize;
    phlnLlcp_Fri_sPacketSequence_t sSequence;
    phlnLlcp_Fri_sPacketHeader_t sLlcpHeader;
    phlnLlcp_Fri_sPacketHeader_t sDmHeader;
    uint8_t DmInfoBuffer[3];
    uint8_t FrmrInfoBuffer[4];
    uint8_t DiscoveryResTidList[PHLNLLCP_FRI_SNL_RESPONSE_MAX];
    uint8_t nDiscoveryResSapList[PHLNLLCP_FRI_SNL_RESPONSE_MAX];
    uint8_t pDiscoveryBuffer[PHLNLLCP_FRI_MIU_DEFAULT];
```

```
    uint8_t *pnDiscoverySapList;
    uint8_t *pServiceNames[PHLNLLCP_FRI_NB_SOCKET_MAX];
    phNfc_sData_t *psDiscoveryServiceNameList;
    phlnLlcp_Fri_t *pLlcp;
    void *pLinkSendContext;
    void *pDiscoverContext;
    phlnLlcp_Fri_Send_CB_t pfLinkSendCb;
    phNfc_sData_t sDmPayload;
    pphlnLlcp_Fri_Cr_t pfDiscover_Cb;
    phlnLlcp_Fri_Transport_Socket_t pSocketTable[PHLNLLCP_FRI_NB_SOCKET_MAX];
    phlnLlcp_Fri_CachedServiceName_t
                  pCachedServiceNames[PHLNLLCP_FRI_SDP_ADVERTISED_NB];
};
```

bSendPending: Pending flag for PDU transmission.

bRecvPending: Pending flag for PDU reception.

bDmPending: Pending flag for Disconnect Mode PDU.

bFrmrPending: Pending flag for Frame Reject PDU.

socketIndex: Index of the socket from the socket table.

LinkStatusError: Link status error flag.

nDiscoveryListSize: Size of the discovered SAP values list.

nDiscoveryReqOffset: Offset for the request discovery list.

nDiscoveryResOffset: Offset for the response discovery list.

nDiscoveryResListSize: Size of the response discovery list.

sSequence: Packet sequence number for transmission and reception.

sLlcpHeader: Header of a LLCP component PDU according to the specification.

sDmHeader: Header field of the pending Disconnect Mode PDU.

sDmPayload: Payload of the pending Disconnect Mode PDU.

DmInfoBuffer: Information field of the pending Disconnect Mode PDU.

FrmrInfoBuffer: Information field of the pending Frame Reject PDU.

DiscoveryResTidList: List of Transaction Identifiers used to identify a remote service name.

nDiscoveryResSapList: List of response SAP values.

***pDiscoveryBuffer:** Discovery buffer.

***pnDiscoverySapList:** List of discovered SAP values.

***pServiceNames:** List of service names associated to discovered SAP values.

***psDiscoveryServiceNameList:** Service discovery name.

***pLlcp:** Pointer to the `phlnLlcp_Fri_t` component parameter.

***pLinkSendContext:** Context input for the transport sending callback function.

***pDiscoverContext:** Context input for the Discovery procedure callback function.

***pfLinkSendCb:** Callback function for the link transmission.

pfDiscover_Cb: Callback function for the discovery procedure.

pSocketTable: List of established sockets with the remote peer device according to the `phlnLlcp_Fri_Transport_Socket` structure.

phlnLlcp_Fri_CachedServiceName_t: List of discovered services as a Service name/SAP value table.

LLC Transport socket structure

The LLC transport socket structure stores information regarding a connection setup between two upper layer services. Some of the setup parameters, such as the MIU and the LTO value, are negotiated during the link setup and some others during the socket connection setup.

As the `phlnLlcp_Fri_Transport_Socket` has a heavy structure with many arguments, only show the most relevant variables are shown. The transmission and reception buffer related variables and pending flags for LLCP PDU packets have been intentionally omitted.

```
struct phlnLlcp_Fri_Transport_Socket {
    uint8_t socket_sSap;
    uint8_t socket_dSap;
    uint8_t remoteRW;
    uint8_t localRW;
    uint8_t nTid;
    phlnLlcp_Fri_Transport_sSocketOptions_t sSocketOption;
    uint32_t indexRwRead;
    uint32_t indexRwWrite;
    void *pOperationContext;
    phlnLlcp_Fri_TransportSocket_eSocketState_t eSocket_State;
    phlnLlcp_Fri_Transport_eSocketType_t eSocket_Type;
    pphlnLlcp_Fri_TransportSocketOperationCb_t pSocketOperationCb;
    phNfc_sData_t sServiceName;
};
```

socket_sSap: Source SAP identifying the local service.

socket_dSap: Destination SAP identifying the remote service.

remoteRW: Remote Receive Window (RM) value used for the sliding window configuration in connection-oriented transports.

localRW: Local Receive Window (RM) value used for the sliding window configuration in connection-oriented transports.

nTid: Transaction identified for the transport link.

sSocketOption: This structure stores the LLCP socket Receive Window and Maximum Information Unit attributes.

indexRwRead: Receive Window index for reading access.

indexRwWrite: Receive Window index for writing access.

pOperationContext: Pointer to the content – input argument for LLCP Transport socket component defined callback functions.

pfSocketOperation_Cb: Pointer to the LLCP Transport socket component defined callback functions.

eSocket_State: Identifies the status of the LLC Transport socket. Valid values defined by the NFC Reader Library can be found in Table 22.

Table 22. Socket state valid values

NFC Reader Library identifier	Socket state
TransportSocket_eSocketDefault	Default state
TransportSocket_eSocketCreated	Socket created
TransportSocket_eSocketBound	Socket bound
TransportSocket_eSocketRegistered	Socket registered
TransportSocket_eSocketConnected	Socket connected
TransportSocket_eSocketConnecting	Socket connecting
TransportSocket_eSocketAccepted	Socket accepted
TransportSocket_eSocketDisconnected	Socket disconnected
TransportSocket_eSocketDisconnecting	Socket disconnecting
TransportSocket_eSocketRejected,	Socket rejected

eSocket_Type: Type of LLC Transport connection. Valid values defined by the NFC Reader Library can be found in Table 23.

Table 23. Transport connection valid values

NFC Reader Library identifier	Transport communication
Transport_eDefaultType	Default communication type
Transport_eConnectionOriented	Connection-oriented communication
Transport_eConnectionLess	Connection-less communication

sServiceName: Service Name identifying the upper layer service.

7.1.5.2 LLC Transport API

Create LLCP Socket

This function creates a socket for a given LLCP link. The socket can be either *connection-oriented* or *connectionless*.

If the socket is connection-oriented, the caller function must provide a working buffer to the socket in order to handle the incoming data. This buffer must be large enough to fit the reception window (RW * MIU).

Note: The options and working buffer are not required in case of listening sockets which cannot be directly used for communication.

```
phStatus_t phlnLlcp_Transport_Socket(
     void * pDataParams,                                [In]
     phlnLlcp_Transport_eSocketType_t eType,            [In]
     phlnLlcp_Transport_sSocketOptions_t * pOptions,    [In]
     phNfc_sData_t * pWorkingBuffer,                    [In]
     phlnLlcp_Transport_Socket_t ** pLlcpSocket,        [Out]
     phlnLlcp_TransportSocketErrCb_t pErr_Cb,           [In]
     void * pContext );                                 [In]
```

***pDataParams**: Pointer to the `phlnLlcp_Fri_DataParams_t` parameter component.

eType: Type of the socket to be created: *connection-oriented* or *connectionless.*

***pOptions**: Configuration options for connection-oriented sockets.

***pWorkingBuffer**: Working buffer to be used by connection-oriented sockets.

****pLlcpSocket**: Pointer to the socket to be filled with a socket found on the socket table.

pErr_Cb: Application callback function that shall be called whenever an error on the socket occurs.

***pContext**: Pointer to the input data to be processed by the error callback function.

The values returned by the function can be:

`PH_ERR_SUCCESS`: Successful Operation.

`Other`: Value returned by the underlying component.

Reset LLCP Socket

This function sets the transport structure and all the LLCP transport sockets to their default states. The LLCP component structure must be previously reset for a correct operation.

Once the reset operation has been performed, the reception of incoming LLCP packets on the LLCP link is enabled.

```
phStatus_t phlnLlcp_Transport_Reset(
     void * pDataParams );                                    [In]
```

***pDataParams**: Pointer to the `phlnLlcp_Fri_DataParams_t` parameter component.

The values returned by the function can be:

`NFCSTATUS_SUCCESS`: Successful operation.

`Other`: Value returned by the underlying component.

Bind a Socket to a Local Source SAP

This function binds a LLCP transport socket to a user defined SAP and service name. The binding is only performed if the socket has been previously created.

Depending on whether an existing service name is given or not, the SAP assigned will belong to either SDP advertised or unadvertised set of available SAPs (see Table 21).

```
phStatus_t phlnLlcp_Transport_Bind(
     void * pDataParams,                                      [In]
     phlnLlcp_Transport_Socket_t * pLlcpSocket,               [In]
     uint8_t nSap,                                            {In]
     phNfc_sData_t *psServiceName );                          [In]
```

***pDataParams**: Pointer to the `phlnLlcp_Fri_DataParams_t` parameter component.

***pLlcpSocket**: Pointer to the LLCP transport socket.

nSap: Local source SAP number to bind the given socket to. If this parameter is `NULL`, a free SSAP is assigned dynamically according to Table 21.

***psServiceName**: Pointer to the service name. If no service name (`NULL`) is specified, `nSAP` is considered as unadvertised according to Table 21.

The values returned by the function can be:

`NFCSTATUS_SUCCESS`: Successful operation.

`NFCSTATUS_INVALID_STATE`: Attempt to bind a socket that has not been previously created or that has been already bound.

NFCSTATUS_ALREADY_REGISTERED: Passed nSAP already bound to another socket.
NFCSTATUS_INVALID_PARAMETER: SAP out of valid range or service name already in use.
NFCSTATUS_INSUFFICIENT_RESOURCES: There are no free SAPs available.
NFCSTATUS_NOT_ENOUGH_MEMORY: Insufficient memory space to store the given service name.

Connect

This function connects a socket with a SAP in the remote peer device. The connection is performed only for *connection-oriented* sockets that are disconnected. If the socket is not bound to a local SAP, it is implicitly bound to a free unadvertised SAP (see Table 21).

According to the LLCP specification, if MIU, RW and Service Name values are different from the default values, then these values are also sent to their remote peer.

```
phStatus_t phlnLlcp_Transport_Connect(
     void * pDataParams,                                    [In]
     phlnLlcp_Transport_Socket_t * pLlcpSocket,             [In]
     uint8_t nSap,                                          [In]
     phlnLlcp_TransportSocketConnectCb_t pConnect_RspCb,    [In]
     void * pContext );                                     [In]
```

***pDataParams**: Pointer to the phlnLlcp_Fri_DataParams_t parameter component.

***pLlcpSocket**: Pointer to the LLCP transport socket.

nSap: The target SAP to connect to. It shall be in range from 2 to 63.

pConnect_RspCb: Callback function to be called when the connection operation is completed (CC PDU packet is received from the remote peer device).

***pContext**: Pointer to the input data to be processed by the callback function.

The values returned by the function can be:

PH_ERR_SUCCESS: Successful operation.

NFCSTATUS_INVALID_PARAMETER: Socket is not a connection-oriented socket or nSap value is out of valid range (2-63).

NFCSTATUS_INVALID_STATE: The socket has neither been bound nor created, or the socket has been already assigned to another service name.

NFCSTATUS_PENDING: Connection operation in progress; pConnect_RspCb() to be called upon completion.

Other: Value returned by the underlying component.

Connect by URI

This function creates a connection between a given socket and a remote service designated by a URI. If the socket has not been bound to a local SAP, it is implicitly bound to a free unadvertised SAP (see Table 21).

```
phStatus_t phlnLlcp_Transport_ConnectByUri(
     void * pDataParams,                                    [In]
     phlnLlcp_Transport_Socket_t * pLlcpSocket,             [In]
     phNfc_sData_t * psUri                                  [In]
     phlnLlcp_TransportSocketConnectCb_t pConnect_RspCb,    [In]
```

```
    void * pContext );                                  [In]
```

***pDataParams**: Pointer to the `phlnLlcp_Fri_DataParams_t` parameter component.

***pLlcpSocket**: Pointer to the LLCP transport socket.

***psUri**: The URI corresponding to the DSAP in the remote peer device. The length of the URI parameter is limited to 255 characters.

pConnect_RspCb: Callback function to be called when the connection operation is completed (CC PDU packet is received from the remote peer device).

***pContext**: Pointer to the input data to be processed by the callback function.

The values returned by the function can be:

`PH_ERR_SUCCESS`: Successful operation.

`NFCSTATUS_INVALID_PARAMETER`: The socket is not a connection-oriented socket, it has been already connected, it is pending to connect or the URI address is longer than 255 characters.

`NFCSTATUS_PENDING`: Connection operation in progress; `pConnect_RspCb()` to be called upon completion.

`Other`: Value returned by the underlying component.

Listen to Connection Requests

This function sets up a socket into listen mode for any incoming connection request coming from a remote peer device. This step is mandatory for the LLCP connection setup.

Listening is only allowed for connection-oriented sockets which are currently not connected to other sockets. The listening remote peer device parses the incoming connection request and processes it according to the DSAP and service name value. If an invalid TLV data is received, and immediate response with FRMR PDU is sent.

Upon the reception of a connection request, the user defined pListen_Cb() callback function is called in order to accept or reject the incoming connection request for that particular LLCP socket.

Note: This function should be called once the socket has been bound to a particular SAP. Without local SAP bound, the socket is not LLCP addressable.

```
phStatus_t phlnLlcp_Transport_Listen(
    void * pDataParams,                                 [In]
    phlnLlcp_Transport_Socket_t * pLlcpSocket,          [In]
    phlnLlcp_TransportSocketListenCb_t pListen_Cb,      [In]
    void * pContext );                                  [In]
```

***pDataParams**: Pointer to the `phlnLlcp_Fri_DataParams_t` parameter component.

***pLlcpSocket**: Pointer to the LLCP transport socket.

pListen_Cb: Callback function to be called when the socket receives a connection request.

***pContext**: Pointer to the input data to be processed by the callback function.

The values returned by the function can be:

`PH_ERR_SUCCESS`: Successful Operation.

NFCSTATUS_INVALID_PARAMETER: The socket is not a connection-oriented socket or it is already listening for incoming requests.

NFCSTATUS_INVALID_STATE: The socket is not at bound state.

Accept an Incoming Connection Request

This function accepts an incoming connection request for a socket provided within the listening callback. It switches the socket state to *connected*.

```
phStatus_t phlnLlcp_Transport_Accept(
     void * pDataParams,                                    [In]
     phlnLlcp_Transport_Socket_t * pLlcpSocket,             [In]
     phlnLlcp_Transport_sSocketOptions_t * pOptions,        [In]
     phNfc_sData_t * psWorkingBuffer,                       [In]
     phlnLlcp_TransportSocketErrCb_t pErr_Cb,               [In]
     phlnLlcp_TransportSocketAcceptCb_t pAccept_RspCb,      [In]
     void * pContext );                                     [In]
```

***pDataParams**: Pointer to the phlnLlcp_Fri_DataParams_t parameter component.

***pLlcpSocket**: Pointer to the LLCP transport socket.

***pOptions**: Options for the socket configuration.

***psWorkingBuffer**: A working buffer to be used by the library.

pErr_Cb: Callback function that shall be called whenever an error occurs on the socket.

pAccept_RspCb: Callback function to be called when a connection request from a remote peer device is received (CC PDU sent).

***pContext**: Pointer to the input data to be processed by the Accept() callback function.

The values returned by the function can be:

PH_ERR_SUCCESS: Successful Operation.

NFCSTATUS_INVALID_PARAMETER: Socket is not a connection-oriented socket.

NFCSTATUS_INVALID_STATE: Socket is not at bound state.

Other: Value returned by the underlying component.

Reject a Connection Request

This function rejects an incoming connection request for a socket provided within the listening callback. The socket is implicitly closed when the function is called.

```
phStatus_t phlnLlcp_Transport_Reject(
     void * pDataParams,                                    [In]
     phlnLlcp_Transport_Socket_t * pLlcpSocket,             [In]
     phlnLlcp_TransportSocketRejectCb_t pReject_RspCb,      [In]
     void * pContext );                                     [In]
```

pDataParams: Pointer to the phlnLlcp_Fri_DataParams_t parameter component.

***pLlcpSocket**: Pointer to the LLCP transport socket.

pReject_RspCb: Callback function to be called when rejection operation is completed.

***pContext**: Pointer to the input data to be processed by the callback function.

The values returned by the function can be:

`PH_ERR_SUCCESS`: Successful Operation.

`NFCSTATUS_INVALID_PARAMETER`: Socket is not at connection-oriented socket.

`NFCSTATUS_INVALID_STATE`: Socket is not a bound state.

`Other`: Value returned by the underlying component.

Disconnect Socket

This function disconnects a connection-oriented socket by sending a DISC PDU through the LLC link. If the socket contains any data pending to be sent or received, then this data is resolved before socket is disconnected.

Local and destination SAP shall both be cleared from the socket, but the socket itself shall not be closed. First, the socket state is changed to disconnecting. When the socket disconnection is successfully completed (DM PDU received and handled internally), the upper layer is notified by the socket disconnection callback and the socket is left in disconnected state.

Note: As soon as the socket is disconnected, both socket connected and socket disconnected callbacks are set to NULL.

```
phStatus_t phlnLlcp_Transport_Disconnect(
     void * pDataParams,                                    [In]
     phlnLlcp_Transport_Socket_t * pLlcpSocket,             [In]
     phlnLlcp_SocketDisconnectCb_t pDisconnect_RspCb,       [In]
     void * pContext );                                     [In]
```

pDataParams: Pointer to the `phlnLlcp_Fri_DataParams_t` parameter component.

***pLlcpSocket**: Pointer to the LLCP transport socket.

pDisconnect_RspCb: Callback function to be called by `phlnLlcp_Transport_Close()` function when the disconnection operation is completed.

***pContext**: Pointer to the input data to be processed by the callback function.

The values returned by the function can be:

`PH_ERR_SUCCESS`: Successful Operation.

`NFCSTATUS_INVALID_PARAMETER`: Socket is not a connection-oriented socket.

`NFCSTATUS_INVALID_STATE`: Socket is not at connected state.

`NFCSTATUS_PENDING`: LLC link has data pending to be sent. Try later.

`Other`: Value returned by the underlying component.

Send Data Packet – Connection Oriented

This function sends data using a *connection-oriented* transport socket, which shall be already in the *connected* state.

Data is transmitted using I PDU packets defined by the LLCP specification. SSAP and DSAP values that are part of the I PDU packet header are obtained from the socket parameter.

```
phStatus_t phlnLlcp_Transport_Send(
```

```
    void * pDataParams,                                    [In]
    phlnLlcp_Transport_Socket_t * pLlcpSocket,             [In]
    phNfc_sData_t * pBuffer,                               [In]
    phlnLlcp_TransportSocketSendCb_t pSend_RspCb,          [In]
    void * pContext );                                     [In]
```

pDataParams: Pointer to the `phlnLlcp_Fri_DataParams_t` parameter component.

***pLlcpSocket**: Pointer to the LLCP transport socket.

***pBuffer**: Buffer containing the data to be sent.

pSend_RspCb: Callback function to be called when the sending operation is completed.

***pContext**: Pointer to the input data to be processed by the callback function.

The values returned by the function can be:

`PH_ERR_SUCCESS`: Successful Operation.

`NFCSTATUS_INVALID_PARAMETER`: Socket is not a connection-oriented socket or the data in psBuffer is longer than the agreed MIU for sending.

`NFCSTATUS_INVALID_STATE`: The socket is not in a valid state to perform the requested operation.

`NFCSTATUS_REJECTED`: LLC link has data pending to be sent. Try later.

`NFCSTATUS_FAILED`: Operation failed.

`Other`: Value returned by the underlying component.

Receive Data from a Socket – Connection Oriented

This function reads data from the LLCP transport socket. This function can only be called on a *connection-oriented* socket.

The function reads the available data in the socket. The maximum number of bytes to be read is limited by the size of the reception buffer. If there is no data available when the function is called, the function waits for incoming data, and the response will be sent by the callback function.

Once the data has been successfully received, the peer sends a RR PDU packet to acknowledge the reception of the data.

Note: Calling this function from the API does not force the remote peer device to send data.

```
phStatus_t phlnLlcp_Transport_Recv(
    void * pDataParams,                                    [In]
    phlnLlcp_Transport_Socket_t * pLlcpSocket,             [In]
    phNfc_sData_t * pBuffer,                               [Out]
    phlnLlcp_TransportSocketRecvCb_t pRecv_RspCb,          [In]
    void * pContext );                                     [In]
```

pDataParams: Pointer to the `phlnLlcp_Fri_DataParams_t` parameter component.

***pLlcpSocket**: Pointer to the LLCP transport socket.

***pBuffer**: Buffer prepared for the reception of the data.

pRecv_RspCb: Callback function to be called when the received data is copied from the socket buffer to the output pBuffer buffer. If the socket is pending for reception, this callback will provide the data when the operation is completed.

***pContext**: Pointer to the input data to be processed by the callback function.

The values returned by the function can be:

`NFCSTATUS_SUCCESS`: Successful Operation.

`NFCSTATUS_INVALID_PARAMETER`: Socket is not a connection-oriented socket

`NFCSTATUS_REJECTED`: Socket is already pending for reception.

`NFCSTATUS_FAILED`: Operation failed or socket is not connected.

Send Data Packet – Connectionless

This function sends data using a *connectionless* transport socket, which shall be already in the *connected* state.

Data is transmitted using UI PDU packets defined by the LLCP specification. The DSAP value is provided by the application.

```
phStatus_t phlnLlcp_Transport_SendTo(
     void * pDataParams,                                    [In]
     phlnLlcp_Transport_Socket_t * pLlcpSocket,             [In]
     uint8_t nSap,                                          [In]
     phNfc_sData_t * pBuffer,                               [In]
     phlnLlcp_TransportSocketSendCb_t pSend_RspCb,          [In]
     void * pContext );                                     [In]
```

pDataParams: Pointer to the `phlnLlcp_Fri_DataParams_t` parameter component.

***pLlcpSocket**: Pointer to the LLCP transport socket.

nSap: Destination SAP of the service the data is sent to.

***pBuffer**: Buffer containing the data to be sent.

pSend_RspCb: Callback function to be called when the sending operation is completed.

***pContext**: Pointer to the input data to be processed by the callback function.

The values returned by the function can be:

`NFCSTATUS_SUCCESS`: Successful Operation.

`NFCSTATUS_INVALID_PARAMETER`: Socket is not a connectionless socket, the data in pBuffer is longer than the agreed MIU for sending or the DSAP is out of the valid range (2-63).

`NFCSTATUS_INVALID_STATE`: Socket is not a connectionless socket.

`NFCSTATUS_REJECTED`: Socket cannot send data because is waiting for data to arrive from the remote peer device. Try later.

`NFCSTATUS_FAILED`: Operation failed.

Close One Socket

This function closes a given LLCP transport connection-oriented or connectionless socket previously created by phFriNfc_LlcpTransport_Socket() function.

If the socket is connected, first it is disconnected and then it is closed. If the socket has not been connected yet, it is closed by aborting it and setting it to NULL.

```
phStatus_t phlnLlcp_Transport_Close(
     void * pDataParams                                    [In]
     phlnLlcp_Transport_Socket_t * pLlcpSocket );          [In]
```

***pDataParams**: Pointer to the `phlnLlcp_Fri_DataParams_t` parameter component.

***pLlcpSocket:** Pointer to the transport socket.

The values returned by the function can be:

`NFCSTATUS_SUCCESS`: Successful Operation.

`Other`: Value returned by the underlying component.

Close All the Sockets

This function closes all created sockets independently of their current states. In addition, the information from the `pCachedServiceName` in `phlnLlcp_Fri_Transport_t` is completely cleared.

```
phStatus_t phlnLlcp_Transport_CloseAll(
     void * pDataParams );                                 [In]
```

***pDataParams**: Pointer to the `phlnLlcp_Fri_DataParams_t` parameter component.

The values returned by the function can be:

`NFCSTATUS_SUCCESS`: Successful Operation.

`Other`: Value returned by the underlying component.

7.1.5.3 Transport Layer Callback functions

The LLC component defines a set of callback functions to inform about incoming requests, completion of transmitted requests and modifications in the sockets managed by the transport layer.

LLCP Error CB

This callback function is executed when an error on the LLCP link occurs.

Set by function: `phlnLlcp_Transport_Socket()` or `phlnLlcp_Transport_Accept()`

Function prototype:

```
typedef void (*pphFriNfc_LlcpTransportSocketErrCb_t) (
     void* pContext,
     uint8_t nErrCode);
```

***pContext**: Pointer to the input data to be processed in the callback function.

nErrCode: Indicates which error has occurred.

`PHFRINFC_LLCP_ERR_NOT_BUSY_CONDITION`: RR acknowledgement received from the remote peer device after a negative acknowledgement (RNR).

`PHFRINFC_LLCP_ERR_BUSY_CONDITION`: Negative acknowledgement (RNR PDU) received from the remote peer device.

PHFRINFC_LLCP_ERR_FRAME_REJECTED: The remote peer device received an invalid packet and subsequently sent a FRMR frame.

PHFRINFC_LLCP_ERR_DISCONNECTED: Disconnection request (DISC PDU) received from the remote peer device.

LLCP Listen CB

This callback function is triggered when an incoming client connection request (CONNECT PDU) is received.

Set by function: phlnLlcp_Transport_Listen()

Function prototype:

```
void (*pphFriNfc_LlcpTransportSocketListenCb_t)  (
     void* pContext,
     phFriNfc_LlcpTransport_Socket_t *IncomingSocket);
```

***pContext**: Pointer to the user data input to be processed in the callback function.

***IncomingSocket**; Pointer to the socket that is bound to the SAP or service name that the remote peer device is requesting to connect with.

LLCP Connect CB

This callback function is triggered when the procedure of connecting to a service in the remote peer device is completed.

Set by function: phlnLlcp_Transport_Connect()

Function prototype:

```
typedef void (*pphFriNfc_LlcpTransportSocketConnectCb_t) (
     void* pContext,
     uint8_t nErrCode,
     NFCSTATUS status);
```

***pContext**: Pointer to the user data input to be processed by the callback function.

nError: Error code defined by the NFC forum LLCP Disconnect mode.

status: Indicates the status of the connection procedure.

NFCSTATUS_SUCCESS: Connection successful. The data transmission may be performed.

NFCSTATUS_ABORTED: Connection rejected or socket closed by phlnLlcp_Transport_Close().

NFCSTATUS_FAILED: Connection not confirmed on the remote side, and therefore not created.

LLCP Disconnect CB

This callback function is triggered when the disconnecting procedure with a remote peer device is completed.

Set by function: phlnLlcp_Transport_Disconnect()

Function prototype:

```
typedef void (*pphFriNfc_LlcpTransportSocketDisconnectCb_t) (
     void* pContext,
```

```
     NFCSTATUS status);
```

***pContext**: Pointer to the data input to be processed by the callback function.

status: Indicates the status of the disconnection procedure.

NFCSTATUS_SUCCESS: Disconnection has been confirmed by the remote side.

LLCP Accept CB

This callback function is called when the local server confirms (sent CC PDU) the connection request (CONNECT PDU) from the remote client.

Set by function: phlnLlcp_Transport_Accept()

Function prototype:

```
void (*pphFriNfc_LlcpTransportSocketAcceptCb_t)  (
     void* pContext,
     NFCSTATUS status);
```

***pContext**: Pointer to the input data to be processed by the callback function.

status: Indicates the status of the acceptance procedure from the connection request.

NFCSTATUS_SUCCESS: Disconnection has been confirmed by the remote peer device.

LLCP Reject CB

This callback function is triggered when the rejection of an incoming Connection Request from a remote peer device is completed.

Set by function: phlnLlcp_Transport_Reject()

Function prototype:

```
typedef void (*pphFriNfc_LlcpTransportSocketRejectCb_t) (
     void* pContext,
     NFCSTATUS status);
```

***pContext**: Pointer to the input data to be processed in the callback function.

status: Indicates the status of the rejection procedure of the disconnection request.

NFCSTATUS_SUCCESS: The DM PDU has been sent successfully.

LLCP Send CB

This callback function is triggered when the transmission of a packet (I PDU) to the remote peer device is completed.

Set by function: **phlnLlcp_Transport_Send**() or phlnLlcp_Transport_SendTo()

Function prototype:

```
typedef void (*pphFriNfc_LlcpTransportSocketSendCb_t) (
     void* pContext,
     NFCSTATUS status);
```

***pContext**: Pointer to the input data to be processed in the callback function.

status: Indicates the status of the transmission procedure from the connection request.

NFCSTATUS_SUCCESS: The data (I PDU frame) has been successfully transmitted.

LLCP Receive CB

This callback function is triggered when the local peer device receives a data packet (I PDU) from the remote peer device in connection-oriented mode.

Set by function: phlnLlcp_Transport_Send(

Function prototype:

```
typedef void (*pphFriNfc_LlcpTransportSocketRecvCb_t) (
     void* pContext,
     NFCSTATUS status);
```

***pContext**: Pointer to the input data to be processed in the callback function.

status: Indicates the status of the reception procedure.

NFCSTATUS_SUCCESS: The incoming packet has been received without an error in accordance with the LLCP specification.

LLCP Receive CB

This callback function is called when the local peer receives a data packet (UI PDU) from the remote peer device in connectionless mode.

Set by function:

Function prototype:

```
typedef void (*pphFriNfc_LlcpTransportSocketRecvFromCb_t) (
     void* pContext,
     uint8_t ssap,
     NFCSTATUS status);
```

***pContext**: Pointer to the input data to be processed in the callback function.

ssap: Source SAP for the incoming data.

status: Indicates the status of the reception procedure.

NFCSTATUS_SUCCESS: The incoming packet has been received without an error in accordance with the LLCP specification.

7.2 SNEP

7.2.1 Technical Introduction

The Simple NDEF Exchange Protocol (SNEP) [21] is a request/response application level protocol for the exchange of application data units in the form of NDEF messages between two NFC Forum compliant devices.

A SNEP Client sends a request to a SNEP Server to either retrieve data from the server with the GET method or push data to the server using the PUT method. The SNEP Server performs the action indicated by the request method using the information provided. Then, it responds with a NDEF message that contains the response code and application data, if required.

The requests are always sent by the SNEP Client and responses are sent by the SNEP Server.

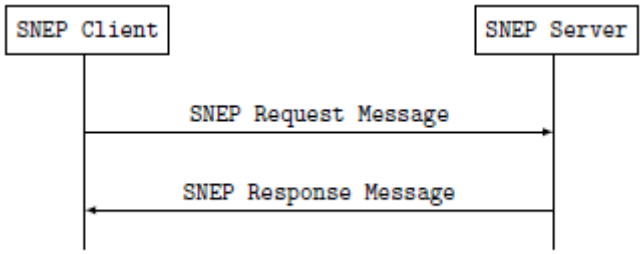


Fig 36. SNEP Communication

Exchanging SNEP messages requires a reliable transport protocol. In the NFC Forum architecture, SNEP is a protocol layer on top of the Logical Link Control Protocol (LLCP). SNEP messages should be transmitted over LLCP data link connections using the LLCP connection-oriented transport service.

NDEF messages can easily be larger than the Maximum Information Unit (MIU) supported by the LLCP data link connection that a SNEP Client establishes with a SNEP Server. The SNEP layer handles fragmentation and reassembly. In the NFC Reader Library, the fragmentation and reassembly of messages is transparent to developers.

The NFC Reader Library supports the Simple NDEF Exchange Protocol (SNEP) 1.0 as specified in the NFC Forum. Both the SNEP Client and the SNEP Server are implemented in the NFC Reader Library.

Remark: The SNEP specification defines a *Default SNEP Server* with well-known LLCP service access point (SAP) address number *4* and service name *urn:nfc:sn:snep*. Certified NFC Forum devices must have the *Default SNEP Server* implemented. The *Default SNEP Server* only implements the PUT request and the NDEF message could be rejected if it is larger than 1024 bytes, though smartphones generally support more.

Remark: The NFC Reader Library supports hosting just one SNEP object, which could be a SNEP Client or SNEP Server instance. The default SNEP Server instance supports one server-client connection due to memory limitations. The limit is set in the `#define PHNPSNEP_FRI_MAX_SNEP_SERVER_CONN` variable in *NxpRdLib_PublicRelease/intfs/phnpSnep.h*. Incrementing the number of possible concurrent connections will consume more RAM memory.

7.2.2 SNEP Client Application

The SNEP Client application is a component which sends a PUT or GET request to a SNEP Server peer in order to either send information or to retrieve data from the remote peer device.

7.2.2.1 SNEP Client Data Structures

The SNEP Client data structures are used to store and organize SNEP Client application operation data. There are three SNEP data structures implemented: the SNEP Configuration structure, the SNEP Client session structure and the SNEP Client PUT/GET request context structure.

SNEP Configuration Structure

This structure contains information about the SNEP Server that the SNEP Client has to connect to. This instance is used as the input argument in phnpSnep_Client_Init() function.

```
typedef struct {
      phnpSnep_Fri_Server_type_t SnepServerType;
      phNfc_sData_t *SnepServerName;
      phlnLlcp_Fri_sSocketOptions_t sOptions:
}phnpSnep_Fri_Config_t, *pphnpSnep_Fri_Config_t
```

SnepServerType: Defines the SNEP Server Type. It could be:

- phnpSnep_Fri_Server_Default: The default SNEP Server name is "*urn:nfc:sn:snep*". In compliance with the SNEP NFC Forum specification [21], the responses to a GET request are not implemented.
- phnpSnep_Fri_Server_NonDefault: Server name taken from *SnepServerName* structure (see below).

***SnepServerName**: SNEP Server name string. This string is used only if SnepServerType == phnpSnep_Fri_Server_NonDefault.

sOptions: LLCP socket options of the local peer. The members RW and MIU determine size of the SNEP and LLCP working buffer.

SNEP Client Session Structure

The SNEP Client instance is called and implemented as a *Client Session* in the NFC Reader Library. It stores the necessary SNEP Client parameters running on an MCU or computer. Only the sWorkingBuffer has to be initialized directly by the developer. The remaining fields are entirely managed by the NFC Reader Library. Once the SNEP module is running, the data structure should not be modified.

```
typedef struct{
      ph_NfcHandle SnepClientHandle;
      ph_NfcHandle hRemoteDevHandle;
      uint32_t iMiu;
      uint32_t iRemoteMiu;
      uint8_t SnepClientVersion;
      phNfc_sData_t sWorkingBuffer;
      phnpSnep_Fri_Client_status_t Client_state;
      pphnpSnep_Fri_ConnectCB_t pConnectionCb;
      void *pClientContext;
      pphnpSnep_Fri_ReqCb_t pReqCb;
      void *pReqCbContext;
      phnpSnep_Fri_putGetDataContext_t putGetDataContext;
      uint32_t acceptableLength;
      phnpSnep_Fri_Config_t *pSnepClientInitDataParams;
      uint8_t bChunking;
      void *pSnepDataParamsContext;
}phnpSnep_Fri_ClientSession_t, *pphnpSnep_Fri_ClientSession_t;
```

hSnepClientHandle: SNEP Client Data link connection handler. The value is assigned by the phnpSnep_Client_Init() function. The handler is used to access the client-server connection by other SNEP Client API functions.

hRemoteDevHandle: Remote device handler for the peer device.

iMiu: Local MIU for LLCP connection. Set by `phnpSnep_Client_Init()` function. The MIU determines the maximum length of the SNEP packet.

iRemoteMiu: Taken from the remote peer device during LLCP initialization. In case no value is received from the remote peer device, the default value is taken (128 bytes).

SnepClientVersion: SNEP protocol version supported is 1.0. This value is hardcoded in *phnpSnep_Fri.h*, and it must not be changed.

```
#define PHNPSNEP_FRI_VERSION_MAJOR    1
#define PHNPSNEP_FRI_VERSION_MINOR    0
```

sWorkingBuffer; Working buffer for the LLCP socket. Its length is calculated within the NFC Reader Library using two socket options (MIU and RW). The buffer needs to be initialized by the developer before the SNEP module starts running.

Client_state: SNEP Client status, for internal library purpose.

pConnectionCb: Callback function triggered when the SNEP Client is initialized and connected to SNEP Server.

***pClientContext**: Upper layer context to be passed in the connect callback function.

pReqCb: Callback function triggered after completion of a PUT or a GET request with either success or failure.

***pReqCbContext**: Pointer to a context to be passed to the request callback function.

putGetDataContext: Pointer to a SNEP packet data and related parameters instance.

acceptableLength: Acceptable length of the GET request.

***pSnepClientInitDataParams**: Defines the SNEP Server type.

bChunking: Chunking buffer flag, not implemented.

***pSnepDataParamsContext**: Pointer to the SNEP data parameter component.

SNEP Client PUT/GET Request Context Structure

This structure contains parameters directly related to the SNEP Client data exchange. The `pSnepPacket, preqResponse` and `pChunkingBuffer` members must be initialized by the developer. All the remaining structure members are completely managed by the NFC Reader Library. Once the SNEP module is running, the values of the structure should not be changed.

```
typedef struct{
      uint32_t iDataSent;
      uint32_t iDataReceived;
      phNfc_sData_t *pSnepPacket;
      uint8_t bWaitForContinue;
      uint8_t bContinueReceived;
      phNfc_sData_t *pReqResponse;
      phNfc_sData_t *pChunkingBuffer;
}phnpSnep_Fri_putGetDataContext_t, *pphnpSnep_Fri_putGetDataContext_t;
```

iDataSent: The number of PUT request data sent so far. Header bytes do not count.

iDataReceived: The number of data bytes received so far.

***pSnepPacket**: SNEP message with a PUT or a GET request. The buffer needs to be initialized before the SNEP module starts running.

bWaitForContinue: Flag indicating whether to wait for a CONTINUE response from the server. This is an internal flag to ensure correct SNEP Client performance.

bContinueReceived: Flag indicating that the server has received a CONTINUE response from the server. The client continues sending the remaining SNEP message (`pSnepPacket`).

***pReqResponse**: Response data received from the SNEP Server (except header). May be null in case of a PUT request. It is passed to the `pphnpSnep_Fri_ReqCb_t` application layer notification callback. The buffer needs to be initialized before the SNEP module starts running.

***pChunkingBuffer**: This buffer temporarily holds SNEP fragments that have just been received from or are transmitted to the server (in other words, the data passed to the LLCP link). The buffer needs to be initialized by the developers. The size of the buffer needs to be larger than two server LLCP MIU vs. client LLCP MIU. The buffer is shared for both sent and received data, one at a time. The buffer needs to be initialized before the SNEP module starts running.

7.2.2.2 SNEP Client API

The general workflow to build a SNEP Client application using the NFC Reader Library requires initializing the SNEP module to store the SNEP Client context. Then, the SNEP Client session has to be initialized to create a LLCP connection with the SNEP Server peer. After the successful connection setup, the SNEP Client can then perform PUT or GET operations to either retrieve or push data to the SNEP Server. Finally, after the data exchange is completed, the SNEP Client session has to be removed to close and erase the LLCP connection with the SNEP Server.

The workflow to build a SNEP Client application is depicted in Fig 37.

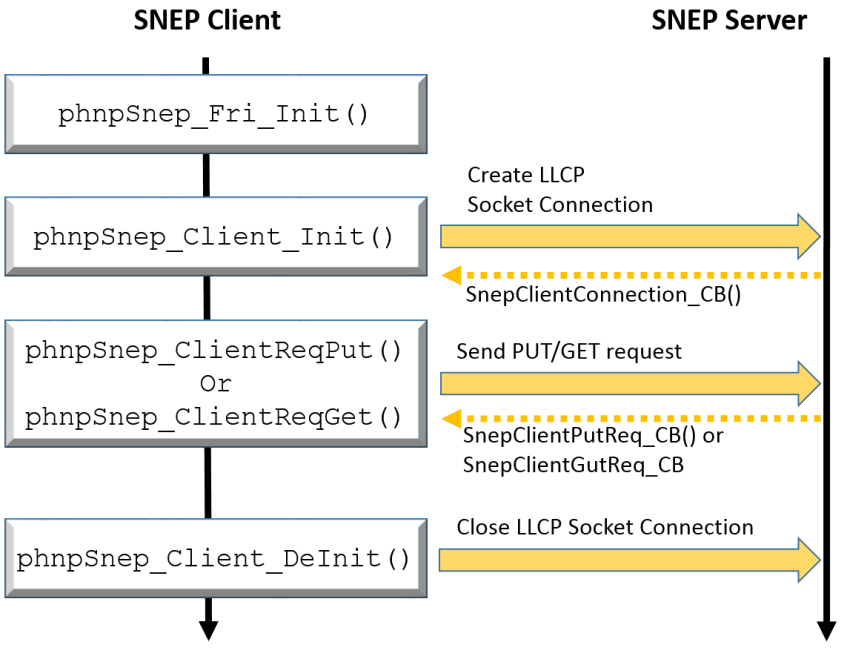


Fig 37. SNEP Client application workflow

The creation of a LLCP socket connection with the SNEP Server and sending PUT/GET requests requires interaction with the SNEP Server. Timeouts and expiration times are implemented between peers. For instance, after a LLCP socket connection request, the SNEP Server should accept or reject the connection and respond accordingly in a specified time frame. Notification events are implemented in the form of callback functions. More information regarding the SNEP Client callback functions is available in Section 7.2.2.3. The SNEP Client API functions are detailed below:

SNEP Module Initialization

The SNEP module initialization creates a SNEP object. A SNEP object instance is needed if either a SNEP Client or a SNEP Server is implemented. The SNEP object structure components (phnpSnep_Fri_DataParams) are:

```
typedef phnpSnep_Fri_DataParams_t{
   void *plnLlcpDataParams,
   phnpSnep_Fri_ClientContext_t gpClientContext,
   phnpSnep_Fri_ServerContext_t gpServerContext,
   void *pOsal };
```

***plnLlcpDataParams:** Pointer to the phlnLlcp_Fri_DataParams_t LLCP Data Parameter component.

gpClientContext: SNEP Client context. Maintains all active SNEP Client entries.

gpServerContext: SNEP Server context. Maintains all active SNEP Server entries.

***pOsal:** Pointer to the OSAL component parameter. Required for timers and dynamic memory allocation. The valid OSAL pointer is taken from the underlying LLCP layer.

The SNEP object is initialized with the phnpSnep_Fri_Init function:

```
phStatus_t phnpSnep_Fri_Init (
      phnpSnep_Fri_DataParams_t *pDataParams,                [In]
      uint16_t wSizeOfDataParams,                            [In]
      void *plnLlcpDataparams );                             [In]
```

***pDataParams**: Pointer to the SNEP object component phnpSnep_Fri_DataParams_t

wSizeOfDataParams: The component size: sizeof(phnpSnep_Fri_DataParams_t)

***plnLlcpDataparams**: Pointer to the underlying LLCP component.

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

Other: Value returned by the underlying component.

SNEP Client Initialization

The phnpSnep_Client_Init() function creates and configures a SNEP Client over LLCP transport protocol.

```
phStatus_t phnpSnep_Client_Init(
      void *pDataParams,                                     [In]
      phnpSnep_Fri_Config_t *pConfigInfo,                    [In]
      ph_NfcHandle hRemDevHandle,                            [In]
      pphnpSnep_Fri_ConnectCB_t pConnClientCb,               [In]
      phnpSnep_Fri_ClientSession_t *pClientSession,          [In]
```

```
      void *pContext );                                    [In]
```

***pDataParams**: Pointer to the SNEP parameter component phnpSnep_Fri_DataParams_t

***pConfigInfo**: Contains the SNEP Server information which the client has to connect to. Stores information like server name, MIU and RW (receive window) for the LLCP layer.

hRemDevHandle: Remote peer device handler.

pConnClientCb: Pointer to a callback function that is triggered when the SNEP Client is initialized and connected to the SNEP Server.

***pClientSession**: SNEP Client instance.

***pContext**: Pointer to the input data to be processed by the callback function.

The values returned by the function can be:

PH_ERR_SUCCESS Operation successful.

Other: Value returned by the underlying component.

SNEP Client PUT Request

This function encapsulates the application data into SNEP packet(s) and sends them to the SNEP Server peer. The application layer is notified about the completion of this request via a callback function.

```
phStatus_t phnpSnep_ClientReqPut(
      void *pDataParams,                                   [In]
      ph_NfcHandle ConnHandle,                             [In]
      phNfc_sData_t *pPutData,                             [In]
      pphnpSnep_Fri_ReqCb_t fCbPut,                        [In]
      void *cbContext );                                   [In]
```

***pDataParams**: Pointer to the SNEP parameter component phnpSnep_Fri_DataParams_t

ConnHandle: Connection handler to a SNEP Client session. It is identified uniquely.

***pPutData**: Pointer to the data to be transmitted to a SNEP Server.

fCbPut: Callback function triggered after the success or failure of a PUT request.

***cbContext**: Pointer to a context to be passed to callback function .

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

Other: Value returned by the underlying component.

SNEP Client GET Request

The phnpSnep_ClientReqGet function generates and sends a GET request to a SNEP Server peer. The application is notified about the incoming SNEP Server response via its callback function.

```
phStatus_t phnpSnep_ClientReqGet(
      void *pDataParams,                                   [In]
      ph_NfcHandle ConnHandle,                             [In]
      phNfc_sData_t *pGetData,                             [Out]
      uint32_t acceptable_length,                          [In]
      pphnpSnep_Fri_ReqCb_t fCbGet,                        [In]
      void *cbContext);                                    [In]
```

***pDataParams**: Pointer to the SNEP parameter component phnpSnep_Fri_DataParams_t

ConnHandle: Connection handler to a SNEP Client session. It is identified uniquely.

***pGetData**: Pointer to the data to be sent to the SNEP Server as part of a GET request.

fCbGet: Callback function triggered when a response from the SNEP Server is received.

acceptable_length: Maximum data length (SNEP packet information field) that the SNEP Client is able to receive as a response from a server.

cbContext: Pointer to a context to be passed to the callback function when called.

The values returned by the function can be:

PH_ERR_INSUFFICIENT_RESOURCES: An internal buffer is not available.

PH_ERR_SUCCESS: Operation successful.

Other: Value returned by the underlying component.

SNEP Client de-Initialization

The phnpSnep_Client_DeInit function removes a SNEP Client session and closes the LLCP socket used by the SNEP Client session.

Remark: The memory used by the SNEP session is not released. Only the link pointing to the SNEP Client instance is set to NULL.

```
phStatus_t phnpSnep_Client_DeInit(
      void *pDataParams,                                [In]
      ph_NfcHandle ConnHandle )                         [In]
```

***pDataParams**: Pointer to the SNEP parameter component phnpSnep_Fri_DataParams_t

ConnHandle: Connection handler to SNEP Client session. It is identified uniquely.

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

Other: Value returned by the underlying component.

7.2.2.3 SNEP Client Callback functions

The callback functions are triggered in the SNEP Client side as notification events. For instance, a callback function is called when the SNEP Server accepts the connection request coming from a SNEP Client. The callback functions prototypes are defined within the SNEP Client source code, but the specific functionality has to be implemented by the developers. Two callback functions are defined in the SNEP Client side: the SNEP Client connect callback function and the SNEP Client request callback function.

The details about these two callback functions are provided below.

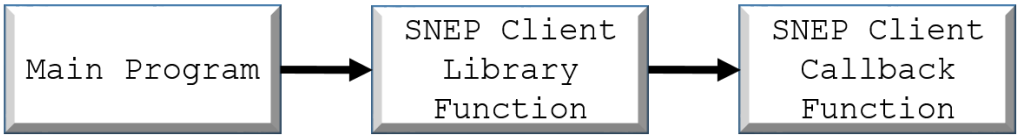


Fig 38. Callback function triggered by a function of the NFC Reader Library

SNEP Client Connect Callback function

The `pphnpSnep_Fri_ConnectCB_t` callback function is triggered as a response of a SNEP Server to a SNEP Client connection request. The caller is implemented as the LLCP connect callback. The developer is in charge of implementing the functionality of the callback function.

Set by function: `phnpSnep_Client_Init()`

Function prototype:

```
typedef void(*pphnpSnep_Fri_ConnectCB_t) (
      void *pContext,                                    [In]
      uint32_t ConnHandle,                               [Out]
      phStatus_t Status );                               [Out]
```

***pContext**: Pointer to the input data to be processed by the callback function.

ConnHandle: Connection handler that uniquely identifies the SNEP Client session.

Status: Informs about the operation success or failure.

`PHNPSNEP_FRI_CONNECTION_SUCCESS`: LLCP socket successfully established for a SNEP Client server connection.

`PHNPSNEP_FRI_CONNECTION_FAILED`: SNEP Client server connection failed.

SNEP Client Request Callback function

The `pphnpSnep_Fri_ReqCb_t` callback function is triggered after the completion of a PUT or a GET request with either success or failure. The status of completion is received via the `Status` function parameter. The developer is in charge of implementing the functionality of the callback function.

Set by function: `phnpSnep_ClientReqPut()`, `phnpSnep_ClientReqGet()`

Function prototype:

```
typedef void(*pphnpSnep_Fri_ReqCb_t) (
      ph_NfcHandle ConnHandle,                           [In]
      void *pContext,                                    [In]
      phStatus_t Status,                                 [Out]
      phNfc_sData_t *pReqResponse );                     [Out]
```

ConnHandle: Connection handler which uniquely identifies a SNEP Client (session).

***pContext**: Pointer to the input data to be processed by the callback function.

Status: Status of the response callback.

`PHNPSNEP_FRI_STATUS_REQUEST_REJECT_FAILED`: Client intended to reject but the REJECT request sending has failed.

`PHNPSNEP_FRI_STATUS_REQUEST_CONTINUE_FAILED`: Client intended to continue but the CONTINUE request sending has failed.

`PHNPSNEP_FRI_STATUS_REQUEST_REJECT`

***pReqResponse**: Received response from a SNEP Server. May be NULL for PUT request.

7.2.3 SNEP Server Application

The SNEP Server application is a component which listens for incoming SNEP Client connection requests to receive or push application data. After the completion of the required action, the SNEP Server responds with a SNEP message containing the response code, which indicates the result of the operation. In case of a GET request, it also responds with application data.

7.2.3.1 SNEP Server Data Structures

The SNEP Server data structures are used to store and organize data related to the operation of a SNEP Server application. There are four data structures: the SNEP Configuration structure, the SNEP Server session structure, the SNEP Server connection structure and the SNEP Server response context structure.

SNEP Configuration Structure

`This structure contains information about the SNEP Server that the SNEP Client has to connect to. This instance is used as the input argument in phnpSnep_Client_Init() function.`

```
typedef struct {
      phnpSnep_Fri_Server_type_t SnepServerType;
      phNfc_sData_t *SnepServerName;
      phlnLlcp_Fri_sSocketOptions_t sOptions:
}phnpSnep_Fri_Config_t, *pphnpSnep_Fri_Config_t
```

SnepServerType: Defines the SNEP Server Type. It could be:

- `phnpSnep_Fri_Server_Default`: The default SNEP Server name is "*urn:nfc:sn:snep*". In compliance with the SNEP NFC Forum specification [21], the responses to a GET request are not implemented.
- `phnpSnep_Fri_Server_NonDefault`: Server name taken from *SnepServerName* structure (see below)

***SnepServerName**: SNEP Server name string. This string is used only if `SnepServerType == phnpSnep_Fri_Server_NonDefault`.

sOptions: LLCP socket options of the local peer. The members RW and MIU determine the size of the SNEP and LLCP working buffer.

SNEP Server Session Structure

The SNEP Server instance is called and implemented as a *Server Session*. It stores the parameters for a SNEP Server management. The `sWorkingBuffer` and `pServerConnection[]` must be initialized directly by the developers. All the remaining

members are completely managed by the SNEP Server implementation. Once the SNEP module is running, the variables shall not be changed.

```
typedef struct{
      ph_NfcHandle hSnepServerHandle;
      uint8_t SnepServerSap;
      uint8_t SnepServerVersion;
      uint8_t SnepServerType;
      phNfc_sData_t sWorkingBuffer;
      phnpSnep_Fri_Server_status_t Server_state;
      phnpSnep_Fri_ServerConnection_t *pServerConnection[];
      uint8_t CurrentConnCnt;
      pphnpSnep_Fri_ConnectCB_t ConnectionCb;
      void *pListenContext;
}pphnpSnep_Fri_ServerSession_t;
```

hSnepServerHandle: SNEP Client Data link connection handler. The value is assigned by the `phnpSnep_Server_Init()` function. The handler is used to access a particular server-client connection by other SNEP Client API functions.

SnepServerSap: SAP on the LLCP that the SNEP Server is bound to. The 0x04 is hardcoded for a SNEP default server and 0x15 for a non-default SNEP Server. Set by the `phnpSnep_Server_Init()` function.

SnepServerVersion: SNEP protocol version supported by the Server. Use SNEP 1.0 version.

SnepServerType: SNEP Server type initialized. This value is taken from `hnpSnep_Fri_Config_t` component.

sWorkingBuffer: Working buffer for the LLCP socket. The buffer needs to be initialized before the SNEP module starts running. The length of the buffer is calculated and assigned by the NFC Reader Library (taking MIU and RW socket options given to `phnpSnep_Server_Init()`) This buffer is shown below:

```
uint8_t workingBuffer[256];
phnpSnep_Fri_ServerSession_t ServerSession;
ServerSession.sWorkingBuffer.buffer = workingBuffer;
```

Server_state: SNEP Server status. For internal management.

***pServerConnection**[]: Table of SNEP Server connections. Each connection is specified by the `phnpSnep_Fri_ServerConnection_t` component, which needs to be assigned by the developer before starting the SNEP module.

```
phnpSnep_Fri_ServerConnection_t pServerConnection;
phnpSnep_Fri_ServerSession_t ServerSession;
ServerSession.pServerConnection[count] = &pServerConnection
```

CurrentConnCnt: Current number of clients connected to the server. The member is incremented when a client connection request is accepted on the server side by the `phnpSnep_Server_Accept()` function. So far, only one server-client connection is supported.

pConnectionCb: Callback function triggered when a SNEP Client request is received or the connection has been accepted by the SNEP Server.

***pListenContext**: Application layer context passed to the above callback.

SNEP Server Connection Structure _

This structure keeps pointers to buffers and variables related to a particular SNEP Server-Client connection. The `sConnWorkingBuffer` and `pSnepWorkingBuffer` must be initialized directly by the developers. All the other members are entirely managed by the NFC Reader Library. The structure values shall not be modified after the SNEP module is running.

```
typedef struct {
      ph_NfcHandle hSnepServerConnHandle;
      ph_NfcHandle hRemoteDevHandle;
      uint8_t SnepServerVersion;
      uint32_t iInboxSize;
      uint32_t iDataTobeReceived;
      phNfc_sData_t *pDataInbox;
      pphnpSnep_Fri_Put_ntf_t pPutNtfCb;
      void *pContextForPutCb;
      pphnpSnep_Fri_Get_ntf_t pGetNtfCb;
      phNfc_sData_t sConnWorkingBuffer;
      phNfc_sData_t *pSnepWorkingBuffer;
      void *pContextForGetCb;
      phnpSnep_Fri_sendResponseDataContext_t responseDataContext;
      uint32_t iMiu;
      uint32_t iRemoteMiu;
      phnpSnep_Fri_Server_status_t ServerConnectionState;
      void *pConnectionContext;
      void *pSnepDataParamsContext;
}pphnpSnep_Fri_ServerConnection_t
```

hSnepServerConnHandle: SNEP Server-client connection handler (related to an LLCP socket).

hRemoteDevHandle: Remote device handler for the peer device.

SnepServerVersion: SNEP protocol version supported by the SNEP Server.

iInboxSize: Buffer size to pick up the data received from a PUT request. Set by `phnpSnep_Server_Accept()` function. For the default SNEP Server it shall be 1024 bytes at least.

iDataTobeReceived: Size of the NDEF message to be read. For internal use.

***pDataInbox**: SNEP connection inbox. Buffer to pick up data received from a PUT request. Set by `phnpSnep_Server_Accept()`. It is linked to the `pSnepWorkingBuffer`.

pPutNtfCb: Callback function triggered when an incoming PUT request from a SNEP Client.

***pContextForPutCb**: Context passed to the above PUT request callback.

pGetNtfCb: Callback function triggered after an incoming GET request from a SNEP Client.

sConnWorkingBuffer: Working buffer for the LLCP connection The buffer needs to be initialized by the developer. It shall be initialized before the SNEP module starts running. The length of the buffer is calculated and assigned by the implementation (from the MIU and the RW socket options given to `phnpSnep_Server_Init()`)

```
uint8_t  sConnWorkingBuffer[];
phnpSnep_Fri_ServerConnection_t pServerConnection;
   pServerConnection->sConnWorkingBuffer.length = sizeof(sConnWorkingBuffer);
   pServerConnection->sConnWorkingBuffer.buffer = sConnWorkingBuffer;
```

***pSnepWorkingBuffer**: The working buffer for the SNEP connection. It is used to store the NDEF message fragments during the data exchange. It must be initiated directly by the developers before the SNEP module starts `phnpSnep_Fri_ServerConnection_t pServerConnection;`

```
phNfc_sData_t pSnepWorkingBuffer;
pServerConnection->pSnepWorkingBuffer = &pSnepWorkingBuffer;
```

***pContextForGetCb**: Context passed to the above GET request callback.

`phnpSnep_Fri_sendResponseDataContext_t` **responseDataContext**: Context of the data transfer transaction.

iMiu: Local MIU (size of information frame of the LLCP). The parameter determines the size of the SNEP fragment. Set by `phnpSnep_Server_Accept()` function.

iRemoteMiu: Remote MIU for the LLCP connection. Determines the length of the SNEP fragment transmitted from the SNEP Server to the SNEP Client. Set by the `phnpSnep_Server_Accept()` function. Taken from the LLCP connection procedure or set by the NFC Reader Library to 128 bytes.

ServerConnectionState: Connection status.

***pConnectionContext**: Context passed to:

- Connection callback when server sends connection response.
- PUT and GET request callbacks whenever called.

***pSnepDataParamsContext**: Pointer to the `phnpSnep_Fri_DataParams_t` SNEP data parameter component.

SNEP Server Response Context Structure

This structure stores parameters directly related to the SNEP Server data exchange. The `pSnepPacket` and `pChunkingBuffer` shall be initialized by the developers. All the remaining members are completely managed by the implementation. The structure values shall not be modified after the SNEP module is running.

```
typedef struct{
      uint32_t iAcceptableLength;
      uint8_t bIsExcessData;
      uint32_t iDataSent;
      phNfc_sData_t *pSnepPacket;
      uint8_t bWaitForContinue;
      uint8_t bContinueReceived;
      phNfc_sData_t *pChunkingBuffer;
      pphnpSnep_Fri_Protocol_SendRspCb_t fSendCompleteCb;
      void *cbContext;
}pphnpSnep_Fri_sendResponseDataContext_t;
```

iAcceptableLength: Acceptable length. This value is taken from the Acceptable length field of the client GET request. The server uses the parameter to decide if it is required to indicate EXCESS DATA response to the client.

bIsExcessData: Flag to indicate excess data.

iDataSent: Number of data sent so far (related to a SNEP message GET request).

***pSnepPacket**: Prepared SNEP packet to be sent as the SNEP Server response.

bWaitForContinue: Flag indicating whether to wait for CONTINUE request from client. Internal flag to ensure the correct SNEP Server performance.

bContinueReceived: Flag indicating that the server has received a CONTINUE request from the client. The server continues sending the remaining part of the SNEP message (`pSnepPacket`).

***pChunkingBuffer**: This buffer temporarily stores SNEP fragments that have just been. The buffer is shared for both sent and received data.

fSendCompleteCb: Callback function triggered when a response to a SNEP Client is sent.

***cbContext**: Pointer to the application layer context passed to above PUT request callback.

7.2.3.2 SNEP Server API

The general workflow to build a SNEP Server application using the NFC Reader Library requires initializing the SNEP module to store the SNEP Server context. After that, the SNEP Server session has to be initialized to set the SNEP Server in listening state for incoming SNEP Client requests, in order to accept and process them accordingly. Finally, after the data exchange is completed, the SNEP Server session has to close the communication channel. The workflow to build a SNEP Client application is depicted in Fig 39.

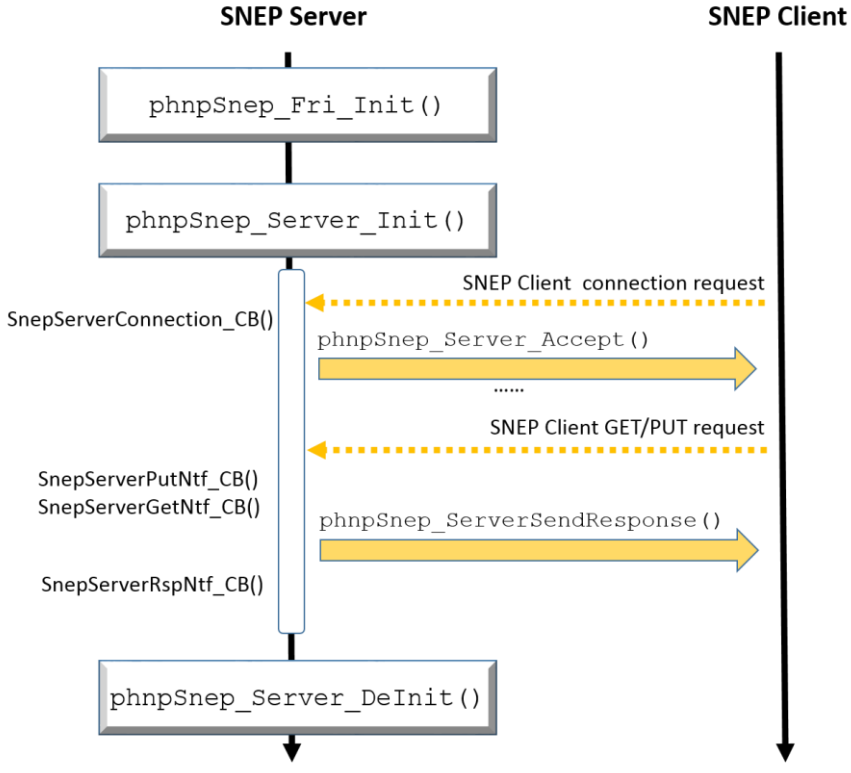


Fig 39. SNEP Server application workflow

A SNEP Server in the listening phase is capable of accepting SNEP Client requests. The connection establishment and the data exchange generate notification events which trigger callback functions. More information of the SNEP Server callback functions is available in Section 7.2.3.3. The SNEP Server API functions are detailed here following:

SNEP Module Initialization

The SNEP module initialization creates a SNEP object. A SNEP object instance is needed if either a SNEP Client or server is implemented. The SNEP object structure components (phnpSnep_Fri_DataParams) are:

```
typedef phnpSnep_Fri_DataParams_t{
   void  *plnLlcpDataParams,
   phnpSnep_Fri_ClientContext_t gpClientContext,
   phnpSnep_Fri_ServerContext_t gpServerContext,
   void  *pOsal};
```

***plnLlcpDataParams:** Pointer to the phlnLlcp_Fri_DataParams_t LLCP Data Parameter component.

gpClientContext: SNEP Client context. It saves all active SNEP Client entries.

gpServerContext: SNEP Server context. It saves all active SNEP Server entries.

***pOsal:** Pointer to the OSAL component parameter. Needed for the timers and for the dynamic memory allocation. The valid OSAL pointer is taken from the underlying LLCP layer.

The SNEP object is initialized with the phnpSnep_Fri_Init function:

```
phStatus_t phnpSnep_Fri_Init (
      phnpSnep_Fri_DataParams_t *pDataParams,                [In]
      uint16_t wSizeOfDataParams,                            [In]
      void *plnLlcpDataparams );                             [In]
```

***pDataParams**: Pointer to the SNEP object component phnpSnep_Fri_DataParams_t

wSizeOfDataParams: The component size: sizeof(phnpSnep_Fri_DataParams_t)

***plnLlcpDataparams**: Pointer to the underlying LLCP component.

The values returned by the function can be:

PH_ERR_SUCCESS: Operation successful.

Other: Value returned by the underlying component.

SNEP Server Initialization

The phStatus_t phnpSnep_Server_Init function creates and configures a SNEP Server over LLCP. The SNEP Server initialization binds a service access point (SAP) with a service name. The default SAP of SNEP Server is 0x04 and the default SNEP Server service name is "*urn:nfc:sn:snep*". It can also bind it to a non-default service name with the hardcoded SAP 0x15 and custom service name. Additionally, it sets the SNEP Server into listening mode to connection requests from SNEP Clients.

```
phStatus_t phnpSnep_Server_Init(
      void *pDataParams,                                     [In]
      phnpSnep_Fri_Config_t *pConfigInfo,                    [In]
      pphnpSnep_Fri_ConnectCB_t pConnCb,                     [In]
```

```
      ph_NfcHandle *pServerHandle,                          [Out]
      phnpSnep_Fri_ServerSession_t *pServerSession,         [In]
      void *pContext );                                     [In]
```

***pDataParams**: Pointer to the SNEP parameter component phnpSnep_Fri_DataParams_t.

***pConfigInfo**: Contains the SNEP Server name, type, LLCP socket options.

***pConnCb**: Callback function which is called when the SNEP Server receives a connection request from a SNEP Client. The callback is also called when the connection is accepted.

***pServerHandle**: Assigned handler to the SNEP Server (session). The handler should be used as reference of the server session in the SNEP API.

***pServerSession**: Pointer to a SNEP Server (session).

***pContext**: Pointer to the input data to be processed by the callback function.

The values returned by the function can be:

PH_ERR_INVALID_PARAMETER: Default server and name also given or non-default server and no server name given.

NFCSTATUS_ALREADY_REGISTERED: When pConfigInfo->SnepServerType == phnpSnep_Fri_Server_Default and SAP==0x04 already occupied.

PH_ERR_INSUFFICIENT_RESOURCES: Some of the given references are invalid.

PH_ERR_SUCCESS: Operation successful.

Other: Value returned by the underlying component.

SNEP Server Accept Connection

This function accepts an incoming connection request from a SNEP Client. Once a connection request from a SNEP Client is received, this phnpSnep_Server_Accept() function should be called. The phnpSnep_Server_Accept should be called inside the pphnpSnep_Fri_ConnectCB_t () connection callback, which has been previously set by the phnpSnep_Server_Init() function.

```
phStatus_t phnpSnep_Server_Accept(
      phnpSnep_Fri_DataParams_t *pDataParams,               [In]
      phNfc_sData_t *pDataInbox,                            [In]
      phlnLlcp_Fri_Transport_sSocketOptions_t *pSockOps,    [In]
      ph_NfcHandle hRemoteDevHandle,                        [In]
      ph_NfcHandle ServerHandle,                            [In]
      ph_NfcHandle ConnHandle,                              [In]
      pphnpSnep_Fri_Put_ntf_t pPutNtfCb,                    [In]
      pphnpSnep_Fri_Get_ntf_t pGetNtfCb,                    [In]
      void *pContext );                                     [In]
```

***pDataParams**: Pointer to the phnpSnep_Fri_DataParams_t SNEP parameter component.

***pDataInbox**: Pointer to the SNEP inbox buffer. The pDataInbox buffer size must be at least of 1024 bytes (Default SNEP Server), otherwise it will return an error.

***pSockOps**: MIU and RW options for a LLCP socket used for a communication with a particular SNEP Client. The size of MIU and RW determine the size of the working buffer.

hRemoteDevHandle: Remote peer device handler.

ServerHandle: Server Session handler (obtained by the phnpSnep_Server_Init() function.

ConnHandle: Handler to incoming connection with a SNEP Client. It is obtained via a connect callback `pphnpSnep_Fri_ConnectCB_t`.The connection handler is unique and represents a connection with particular SNEP Client.

pPutNtfCb: Put Notification callback for incoming data. The callback shall be called when the PUT request from a SNEP Client is received.

pGetNtfCb: Get Notification callback for incoming data request. The callback shall be called when the GET request from a SNEP is client received.

***pContext**: Application layer data to be passed to the above PUT and GET request callbacks functions.

The values returned by the function can be:

`PH_ERR_INVALID_PARAMETER`: Running default server with `pDataInbox->length < 1024`

`PH_ERR_INSUFFICIENT_RESOURCES`: Some of the SNEP Server related components or internal buffer not allocated.

`PH_ERR_SUCCESS`: Operation successful.

`Other`: Value returned by the underlying component.

SNEP Server Response

This function sends a response to a SNEP Client. This function should be called inside the `pphnpSnep_Fri_Put_ntf_t` or `pphnpSnep_Fri_Get_ntf_t` callback functions depending if the SNEP Client performed a GET or PUT operation.

```
phStatus_t phnpSnep_ServerSendResponse (
      void *pDataParams,                                    [In]
      ph_NfcHandle ConnHandle,                              [In]
      phNfc_sData_t *pResponseData,                         [In]
      phStatus_t responseStatus,                            [In]
      pphnpSnep_Fri_Protocol_SendRspCb_t fSendCompleteCb,   [In]
      void *cbContext );                                    [In]
```

***pDataParams**: `Pointer to the phnpSnep_Fri_DataParams_t SNEP parameter component.`

ConnHandle: Handler to a connection with the SNEP Client. The value of the handler should be obtained via a PUT or GET request callback. The connection handler is unique and represents a connection with a particular SNEP Client.

***pResponseData**: Pointer to the `phNfc_sData_t` structure storing the response data to be sent to the SNEP Client.

responseStatus: Response status code.

fSendCompleteCb: The callback function to be called when the response has been sent.

***cbContext**: Application layer content to be passed to the callback function.

The values returned by the function can be:

`PH_ERR_INSUFFICIENT_RESOURCES`: Some of the SNEP Server related components or internal buffer not allocated

`PH_ERR_SUCCESS`: Operation successful.

`Other`: Value returned by the underlying component.

SNEP Server de-Initialization

This function removes the SNEP Server handler and closes all adjacent connections. The memory space occupied by the instance of the server session is not released as only an internal reference is cancelled.

```
phStatus_t phnpSnep_Server_DeInit (
      void *pDataParams,                                    [In]
      ph_NfcHandle ServerHandle );                          [In]
```

***pDataParams**: `Pointer to the SNEP parameter component phnpSnep_Fri_DataParams_t.`

ServerHandle: Handler to a server session to be removed.

The values returned by the function can be:

`PH_ERR_NOT_INITIALISED`: Invalid handle to a server session.

`PH_ERR_SUCCESS`: Operation successful.

`Other`: Value returned by the underlying component.

7.2.3.3 SNEP Server Callback functions

The callback functions are triggered in the SNEP Server side as notification events. For instance, a callback function is called when the SNEP Server has just received a connection request from a SNEP Client or after the completion of a SNEP Client PUT request. The callback functions prototypes are defined but their implementation is in hands of the developer.

There are four callback functions defined: the SNEP Server connect callback function, the SNEP Server PUT request callback function, the SNEP Server GET request callback function and the SNEP Server Send response callback function. The details of these four callback functions are provided below.

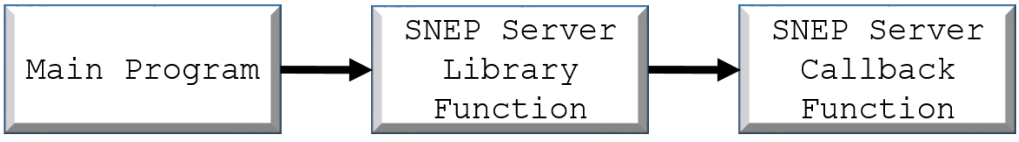


Fig 40. Callback function triggered by a function of the NFC Reader Library

SNEP Server Connect Callback function

This callback function is called when:

- The SNEP Server has just received a connection request from a client.
- The SNEP Server has just accepted a connection request. The server-client connection has just been established.

A particular event is recognized according to the 3rd input argument `status`.

Set by function: `phnpSnep_Server_Init()`

Function prototype:

```
typedef void(*pphnpSnep_Fri_ConnectCB_t) (
     void  *pContext,                                       [In]
```

```
    uint32_t   ConnHandle,                                  [Out]
    phStatus_t Status );                                    [Out]
```

***pContext**: Pointer to the input data to be processed by the callback function. Depending on the caller, one of two possible contexts may be returned:

- Listen context: The last argument passed to the `phnpSnep_Server_Init()` function. This context is returned when the third parameter is `Status == PHNPSNEP_FRI_INCOMING_CONNECTION`.
- Connection context: The last argument passed to the `phnpSnep_Server_Accept()` function. This context is returned when the third parameter is `Status == PHNPSNEP_FRI_CONNECTION_SUCCESS` or `PHNPSNEP_FRI_CONNECTION_FAILED`.

ConnHandle: Connection handler between the server and a particular client. It is assigned as unique to distinguish from other server client connections on the SNEP layer.

Status: Status referring about the connection state:

`PHNPSNEP_FRI_INCOMING_CONNECTION`: A connection request from a SNEP Client has been received.

`PHNPSNEP_FRI_CONNECTION_SUCCESS`: The SNEP Server has just accepted a connection request form a client -sent CC frame on the LLCP layer.

`PHNPSNEP_FRI_CONNECTION_FAILED`: A connection request from a SNEP Client has failed.

SNEP Server PUT Request Callback function

This callback function is called when a PUT request from a SNEP Client is received.

Set by function: `phnpSnep_Server_Accept()`

Function prototype:

```
typedef void(*pphnpSnep_Fri_Put_ntf_t) (
    void *pContext,                                         [In]
    phStatus_t Status,                                      [Out]
    phNfc_sData_t *pDataInbox,                              [Out]
    ph_NfcHandle ConnHandle );                              [Out]
```

***pContext**: Pointer to the input data to be processed by the callback function. The content is shared with the content of the GET request callback.

Status: Status of the response callback. Only `PH_ERR_SUCCESS` should be returned.

***pDataInbox**: Pointer to an incoming data buffer (NDEF Message).

ConnHandle: Connection handler between a SNEP Server and a particular client. It is assigned as unique to distinguish from other server client connections on the SNEP layer.

SNEP Server GET Request Callback function

This callback function is called when a GET request from a SNEP Client is received.

Set by function: `phnpSnep_Server_Accept()`

Function prototype:

```
typedef void(*pphnpSnep_Fri_Get_ntf_t) (
```

```
    void* pContext,                                    [In]
    phStatus_t Status,                                 [Out]
    phNfc_sData_t *pGetMsgId,                          [Out]
    ph_NfcHandle ConnHandle );                         [Out]
```

***pContext**: Pointer to the input data to be processed by the callback function.

Status: Status of the response callback. Only `PH_ERR_SUCCESS` should be returned.

***pGetMsgId**: Pointer to a buffer storing a NDEF message from the SNEP packet. The buffer is cleared by the NFC Reader Library after the callback function finishes.

ConnHandle: Connection handler between a SNEP Server and a particular client. It is assigned as unique to distinguish from other server client connections on the SNEP layer.

SNEP Server Send Response Callback _

This callback function is called when a SNEP Server response to a SNEP Client has been sent.

Set by function: `phnpSnep_ServerSendResponse()`

Function prototype:

```
typedef void(*pphnpSnep_Fri_Protocol_SendRspCb_t) (
      void *pContext,                                  [In]
      phStatus_t Status,                               [Out]
      ph_NfcHandle ConnHandle);                        [Out]
```

***pContext**: Pointer to the input data to be processed by the callback function.

`Status`: Status of the response callback.

`PHNPSNEP_FRI_STATUS_REQUEST_REJECT`: Request rejected. The client is unable to receive the remaining fragments of a SNEP response message. The client is not expecting or willing to handle further fragments of this message, and the reception of more fragments might force the client to close the data link connection.

`PHNPSNEP_FRI_STATUS_RESPONSE_UNSUPPORTED_VERSION`: The SNEP protocol version implemented in the server is different from the SNEP protocol version implemented in the client side.

`PHNPSNEP_FRI_STATUS_INVALID_PROTOCOL_DATA`: The server is unable to understand the request from the client. The server sends BAD REQUEST response.

`PHNPSNEP_FRI_STATUS_RESPONSE_EXCESS_DATA`: The server has found a resource matching the request, but returning the result would exceed the maximum acceptable length that the client has specified within the request message.

`PH_ERR_SUCCESS`: Operation successful.

`Other`: Return the value of the underlying `phlnLlcp_Transport_Recv()` or `phlnLlcp_Transport_Send()`.

ConnHandle: Connection handler between a SNEP Server and a particular client. It is assigned as unique to distinguish from other server client connections on the SNEP layer.

8. NFC Reader Library API: Common Layer

In this section, the Key Store, the Log module and the OSAL components are explained in depth.

8.1 Key Store

A proper key management is critical to ensure security in cryptosystems. The secure storage of cryptographic keys is a must in order to develop reliable solutions and protect data from hackers. The Key Store component is a key handling software module for both communication encryption and authentication operations. The Key Store supports the following symmetric cryptography key types:

- PH_KEYSTORE_KEY_TYPE_AES128 - Key size:128bits
- PH_KEYSTORE_KEY_TYPE_AES192 - Key size: 192bits
- PH_KEYSTORE_KEY_TYPE_AES256 - Key size: 256bits
- PH_KEYSTORE_KEY_TYPE_DES - Key size: 56bits
- PH_KEYSTORE_KEY_TYPE_2K3DES - Key size: 128bits
- PH_KEYSTORE_KEY_TYPE_MIFARE - Key size: 96bits
- PH_KEYSTORE_KEY_TYPE_3K3DES - Key size: 192bits

The NFC Reader Library provides two Key Store implementations: the CLRC663 Hardware Key Store and the Software Key Store.

- CLRC663 Hardware Key Store: It provides a hardware dependent API to store MIFARE Crypto1 secret keys in the reader IC EEPROM. This functionality is available in the CLRC663 reader ICs, where a special EEPROM memory area is dedicated for MIFARE Crypto1 key storage purposes.
- Software Key Store: It provides an API to store the supported types of secret keys into the MCU flash memory.

The API for both the hardware and software Key Store is described in the following sections.

Note: MIFARE key type is a 12-byte array that includes both Key A and Key B.

Note: These Key Store implementations do not guarantee high security key storage since they have not been constructed as tamper-resistant solutions. For high security key storage capabilities, please refer to the NXP Export Controlled Reader Library where the SAM Key Store is implemented.

8.1.1 CLRC663 Hardware Key Store

The CLRC663 Hardware Key Store provides a hardware dependent API to store MIFARE Crypto1 secret keys in the reader IC EEPROM.

The CLRC663 Hardware Key Store solution is available in the CLRC663 contactless reader IC. This IC contains an 8kB EEPROM memory divided in pages of 64 bytes and organized in sections, being section 3 the one specifically reserved for the storage of MIFARE Classic Keys (Crypto1 keys). The maximum number of MIFARE Classic keys that can be stored in the CLRC663 IC is 128. The following figure shows the mentioned EEPROM structure:

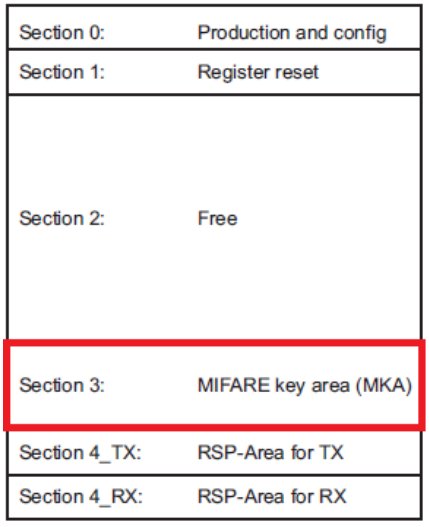


Fig 41. CLRC663 EEPROM memory structure

In order to handle the CLRC663 Hardware Key Store, the NFC Reader Library defines the `phKeyStore_Rc663_DataParams_t` structure:

```
typedef struct{
   void *pHalDataParams;
} phKeyStore_Rc663_DataParams_t;
```

phHalDataParams: Pointer to the parameter structure of the underlying hardware component.

8.1.1.1 CLRC663 Hardware Key Store Initialization

The CLRC663 Hardware Key Store can be initialized using this function:

```
phStatus_t phKeyStore_Rc663_Init(
      phKeyStore_Rc663_DataParams_t * pDataParams,          [In]
      uint16_t wSizeOfDataParams,                           [In]
      void * pHalDataParams );                              [In]
```

***pDataParams:** Pointer to the `phKeyStore_Rc663_DataParams_t` parameter component.

wSizeOfDataParams: Size of the `phKeyStore_Rc663_DataParams_t` data parameter structure.

***pHalDataParams:** Pointer to the underlying HAL data parameter component.

The returned values from the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_ERR_INVALID_DATA_PARAMS`: `wSizeOfDataParams` does not match with the defined size of the `phKeyStore_Rc663_DataParams_t` structure.

8.1.1.2 Format Key Entry

This function formats a key entry to a MIFARE key type. The function sets a pair of MIFARE keys A and B to zero.

```
phStatus_t phKeyStore_FormatKeyEntry(
      void * pDataParams,                                [In]
      uint16_t wKeyNo,                                   [In]
      uint16_t wNewKeyType );                            [In]
```

***pDataParams:** Pointer to the Key Store parameter component.

wKeyNo: Position of the key to be formatted.

wNewKeyType: Key type to be formatted into the Key Store entry. For the CLRC663 Hardware Key Store, only `PH_KEYSTORE_KEY_TYPE_MIFARE` is supported.

The returned values from the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_ERR_INVALID_PARAMETER`: Argument `wKeyNo` out of valid range.

`Other`: Value returned by the underlying component.

8.1.1.3 Set Key Value

This function stores a new key value in a given key entry (array position) of the CLRC663 Hardware Key Store.

```
phStatus_t phKeyStore_SetKey(
      void * pDataParams,                                [In]
      uint16_t wKeyNo,                                   [In]
      uint16_t wKeyVersion,                              [In]
      uint16_t wKeyType,                                 [In]
      uint8_t * pNewKey,                                 [In]
      uint16_t wNewKeyVersion );                         [In]
```

***pDataParams:** Pointer to the Key Store parameter component.

wKeyNo: Position of the key to be updated.

wKeyVersion: This parameter has no effect on the CLRC663 Hardware Key Store.

wKeyType: Type of the key to be stored. For the CLRC663 Harware Key Store, only `PH_KEYSTORE_KEY_TYPE_MIFARE` is supported.

***pNewKey:** Pointer to the key value to be stored.

wNewKeyVersion: This parameter has no effect on the CLRC663 Hardware Key Store.

The returned values from the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_ERR_INVALID_PARAMETER`:

- `wKeyType` of the new key does not match with the key type of the destination key.
- `wKeyNo` out of valid range.

`Other`: Value returned by the underlying component.

8.1.1.4 Set Key Value at position

This function stores a new key value in a given key entry (array position) of the CLRC663 Hardware Key Store. It performs the same action as the `phKeyStore_SetKey()` function described above (this is only true within the CLRC663 Hardware Key Store).

```
phStatus_t phKeyStore_SetKeyAtPos(
```

```
    void * pDataParams,                                    [In]
    uint16_t wKeyNo,                                       [In]
    uint16_t wPos,                                         [In]
    uint16_t wKeyType,                                     [In]
    uint8_t * pNewKey,                                     [In]
    uint16_t wNewKeyVersion );                             [In]
```

***pDataParams:** Pointer to the key store parameter component.

wKeyNo: Position of the key to be updated.

wPos: This parameter has no effect on the CLRC663 Hardware Key Store.

wKeyType:. Only `PH_KEYSTORE_KEY_TYPE_MIFARE` is supported for the CLRC663 Hardware Key Store.

***pNewKey:** Pointer to the new key value.

wNewKeyVersion: This parameter has no effect on the CLRC663 Hardware Key Store.

The returned values from the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_ERR_INVALID_PARAMETER`:

- `wKeyType` of the new key does not match with the key type of the destination key.
- `wKeyNo` out of valid range.

`Other`: Value returned by the underlying component.

8.1.2 Software Key Store

The Software Key Store provides the means to store any kind of cryptographic keys into the MCU flash memory. From now on, each of the keys kept in memory will be referred to as a key entry. Each of these key entries may have different versions, taking the key a different value for each version (or we could say that each key entry has different key-version pairs).

Each of the key entries may also have a Key Usage Counter (KUC) assigned. A Key Usage Counter allows to keep count of and limit the number of times a key entry is used with authentication purposes.

A Software Key Store component contains basically three arrays: the `pKeyEntries` array, the `pKeyVersionPairs` array and the `pKUCentries` array. Each key entry is represented by an element at the `pKeyEntries` array. Each of these elements contains information about the type of the key and a pointer to a KUC. The key-version pairs are all stored in the `pKeyVersionPairs` array. Every key entry in the component has the same number of key-version pairs assigned. This number is stored in the `wNoOfVersions` attribute from the component. As these key-version pairs are ordered in the array in a known way, there is no need for pointers (these can be better understood by having a look at the example below). Finally, the `pKUCEntries` array stores the KUC entries. These entries contain the current value of the counter and its maximum value.

The Software Key Store data structure is the following:

```
typedef struct{
   phKeyStore_Sw_KeyEntry_t *pKeyEntries;
   phKeyStore_Sw_KeyVersionPair_t *pKeyVersionPairs;
   uint16_t wNoOfKeyEntries;
   uint16_t wNoOfVersions;
```

```
   phKeyStore_Sw_KUCEntry_t *pKUCEntries;
   uint16_t wNoOfKUCEntries;
} phKeyStore_Sw_DataParams_t;
```

As explained before, the `phKeyStore_Sw_DataParams_t` structure is composed of three arrays (`*pKeyEntries, *pKeyVersionPairs` and `*pKUCEntries`) and three variables with the information about their sizes (`wNoOfKeyEntries, wNoOfVersion` and `wNoOfKUCEntries`). The Fig 42 depicts the `phKeyStore_Sw_DataParams_t` structure.

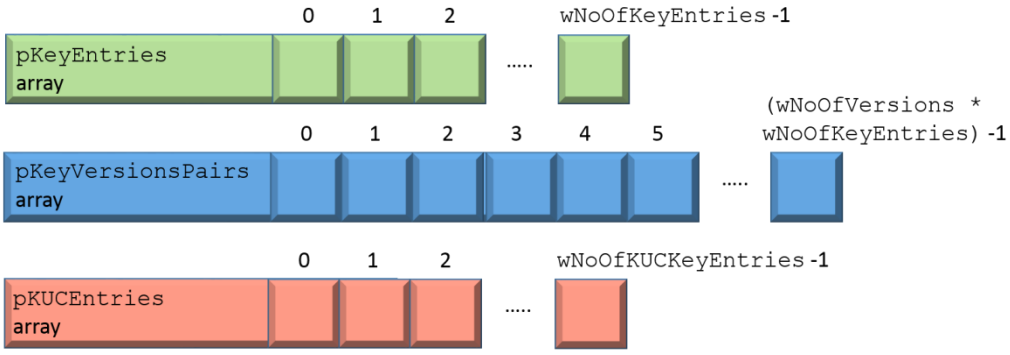


Fig 42. `phKeyStore_Sw_DataParams_t` **structure**

The elements of these arrays are structure variables, which are described in the following lines.

Key entry structure - `phKeyStore_Sw_KeyEntry_t`

The Key entry structure indicates the type of key stored and a Key Usage Counter for this specific key entry.

```
typedef struct{
   uint16_t wKeyType;
   uint16_t wRefNoKUC;
} phKeyStore_Sw_KeyEntry_t;
```

wKeyType: Type of the key stored in `pKey`.

wRefNoKUC: Position of the KUC entry in the `pKUCEntries` array assigned to this key entry.

Key-version pair structure - `phKeyStore_Sw_KeyVersionPair_t`

The key-version pair structure associates a certain key value with a key version number. In authentication and encryption calculations only the key values are used.

```
typedef struct{
   uint8_t pKey [PH_KEYSTORE_SW_MAX_KEY_SIZE];
   uint16_t wVersion;
} phKeyStore_Sw_KeyVersionPair_t;
```

pKey: Variable storing the secret key value. The maximum key size is 32 bytes.

wVersion: Version of this key value.

KUC entry - `phKeyStore_Sw_KUCEntry_t`

The Key Usage Counter (KUC) is used to count and limit the number of authentications a key entry can be used. It is automatically incremented each time the corresponding key entry is used for authentication.

```
typedef struct{
   uint32_t dwLimit;
   uint32_t dwCurVal;
} phKeyStore_Sw_KCUEntry_t;
```

dwLimit: Limit of the Key Usage Counter.

dCurVal: Current value of the Key Usage Counter.

One KUC entry instance may be referenced (linked) by more key entries, but this is a bit hazardous and difficult for management. Thus, it is recommended to keep each key entry with its own KUC entry assigned.

Software Key Store: Structure example

The `phKeyStore_Sw_DataParams_t` data structure with the specific values of `wNoOfKeyEntries=3, wNoOfVersion=2` and `wNoOfKUCEntries=4` is represented in Fig 43.

For this particular case, the `pKeyEntries` array has a size of 3 units, the `pKeyVersionPairs` array has a size of 6 units (two key versions per key entry), and finally, the `pKUCEntries` array has a size of 4 units. Each element of `pKeyEntries` is linked to its corresponding (two, in this case) elements of `pKeyVersionPairs`, where its key values are stored. For instance, the positions 4 and 5 of the `pKeyVersionPairs` array are assigned to the third element of the `pKeyEntries` array. Additionally, each element of `pKeyEntries` may be linked to one specific element of `pKUCEntries`, but it is not mandatory for a key entry to have a KUC assigned.

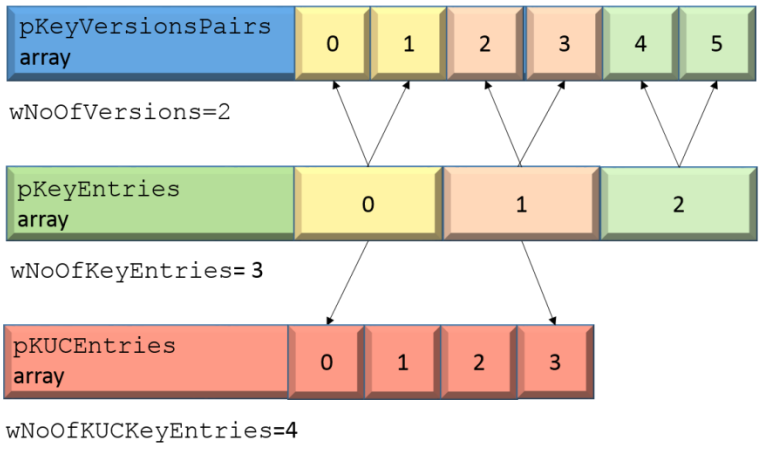


Fig 43. Software Key Store structure example

8.1.2.1 Software Key Store Initialization

The software Key Store can be initialized using this function:

```
phStatus_t phKeyStore_Sw_Init(
```

```
      phKeyStore_Sw_DataParams_t * pDataParams,              [In]
      uint16_t wSizeOfDataParams,                            [In]
      phKeyStore_Sw_KeyEntry_t * pKeyEntries,                [In]
      uint16_t wNoOfKeyEntries,                              [In]
      phKeyStore_Sw_KeyVersionPair_t * pKeyVersionPairs,     [In]
      uint16_t wNoOfVersionPairs,                            [In]
      phKeyStore_Sw_KUCEntry_t * pKUCEntries,                [In]
      uint16_t wNoOfKUCEntries );                            [In]
```

***pDataParams:** Pointer to the `phKeyStore_Sw_DataParams_t` parameter component.

wSizeOfDataParams: Size of the `phKeyStore_Sw_DataParams_t` data parameter structure.

***pKeyEntries:** Pointer to the array containing the key entries.

wNoOfKeyEntries: Number of key entries in the `pKeyEntries` array.

***pKeyVersionPairs:** Pointer to the array containing the key-version pairs.

wNoOfVersionPairs: Number of key-version pairs per key value.

***pKUCEntries**: Pointer to the Key Usage Counter array.

wNoOfKUCEntries: Number of Key Usage Counter entries.

The returned values from the function can be:

`PH_ERR_SUCCESS:` Operation successful.

`PH_ERR_INVALID_DATA_PARAMS: wSizeOfDataParams` does not match with the defined size of the `phKeyStore_Sw_DataParams_t` structure.

8.1.2.2 Format Key Component

This function formats a key entry to a given key type (MIFARE, AES128, AES192, 3DES, etc.). All its key values and their version numbers are set to null.

```
phStatus_t phKeyStore_FormatKeyEntry(
      void * pDataParams,                                    [In]
      uint16_t wKeyNo,                                       [In]
      uint16_t wNewKeyType );                                [In]
```

***pDataParams:** Pointer to the Key Store parameter component.

wKeyNo: Position of the key to be formatted.

wNewKeyType: Key type to be formatted into this Key Store entry.

The returned values from the function can be:

`PH_ERR_SUCCESS:` Operation successful.

`PH_ERR_INVALID_PARAMETER: wKeyNo` argument out of valid range.

`Other:` Value returned by the underlying component.

8.1.2.3 Set Key Value

This function stores a new key value and its corresponding key version number from a certain key entry. After formatting a key entry into a particular key type with the `phKeyStore_FormatKeyEntry()` function, all the key version numbers are set to `PH_KEYSTORE_DEFAULT_ID` (zero). This function has no impact on the KUCs (the KUCs are

neither incremented or decremented nor erased, and they remain linked to the same key entry).

```
phStatus_t phKeyStore_SetKey(
      void * pDataParams,                                [In]
      uint16_t wKeyNo,                                   [In]
      uint16_t wKeyVersion,                              [In]
      uint16_t wKeyType,                                 [In]
      uint8_t * pNewKey,                                 [In]
      uint16_t wNewKeyVersion );                         [In]
```

***pDataParams:** Pointer to the Key Store parameter component.

wKeyNo: Position of the key to be updated.

wKeyVersion: Version of the key to be updated.

wKeyType: Type of the key to be stored.

***pNewKey:** Pointer to the key value to be stored.

wNewKeyVersion: New key version number. It replaces the key version number stored previously in `wVersion`.

The returned values from the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_ERR_INVALID_PARAMETER`:

- `wKeyType` of the new key does not agree with the key type of the destination key.
- `wKeyNo` out of valid range.
- `wKeyVersion` for the given `wKeyNumber` not found.

`Other`: Value returned by the underlying component.

8.1.2.4 Set Key Value at Position

This function changes the key value at a given position. Unlike the `phKeyStore_SetKey()` function, in this case the key is not selected by the version (although it has a version number).

```
phStatus_t phKeyStore_SetKeyAtPos(
      void * pDataParams,                                [In]
      uint16_t wKeyNo,                                   [In]
      uint16_t wPos,                                     [In]
      uint16_t wKeyType,                                 [In]
      uint8_t * pNewKey,                                 [In]
      uint16_t wNewKeyVersion );                         [In]
```

***pDataParams:** Pointer to the Key Store parameter component.

wKeyNo: Position of the key entry to be updated.

wPos: Position of the key-version pair to be updated (within the sub-array of key-version pairs assigned to the selected key entry).

wKeyType: Type of the new key to be stored. It must match with the key type of the key entry. This parameter determines the length of the key (how many bytes to copy from the pNewKey array).

***pNewKey:** Pointer to the new key value.

wNewKeyVersion: Version number of the new key to be updated.

The returned values from the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_ERR_INVALID_PARAMETER`:

- `wKeyType` of the new key does not match with the key type of the destination key.
- `wKeyNo` out of valid range.
- `wPos` is greater than the number of versions (`DataParams->wNoOfVersions`).

`Other`: Value returned by the underlying component.

8.1.2.5 Set Full Key Entry

This function updates the key values and the version numbers of a certain key entry. The replacement of the key value-version pairs begins at the first element of `pKeyVersionPairs` assigned to the given key entry and continues until the given number of keys are written. A KUC can be assigned too. The rest of key values and version numbers in the Key Store component remain untouched.

```
phStatus_t phKeyStore_SetFullKeyEntry(
      void * pDataParams,                                  [In]
      uint16_t wNoOfKeys,                                  [In]
      uint16_t wKeyNo,                                     [In]
      uint16_t wNewRefNoKUC,                               [In]
      uint16_t wNewKeyType,                                [In]
      uint8_t * pNewKeys,                                  [In]
      uint16_t * pNewKeyVersionList );                              [In]
```

***pDataParams:** Pointer to the Key Store parameter component.

wNoOfKeys: Number of key values in `pNewKeys`.

wKeyNo: Position of the key entry to which the values to be updated belong.

wNewRefNoKUC: Position of the Key Usage Counter to be linked to the key entry.

wNewKeyType: Type of the key to be stored.

***pNewKeys:** Pointer to the array of new key values to be stored.

***pNewKeyVersionList:** Pointer to the array of new version numbers.

The returned values from the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_ERR_INVALID_PARAMETER`:

- Invalid key type identifier `wNewKeyType`.
- `wKeyNo` out of valid range.
- `wNoOfKeys` greater than the number of key versions `pDtaParams->wNoOfVersions`.
- `NewwRefNoKUC` greater than `pDataParams->wNoOfKUCEntries-1`.

`Other`: Value returned by the underlying component.

8.1.2.6 Set KUC

This function assigns a KUC entry to a given key entry.

```
phStatus_t phKeyStore_SetKUC(
      void * pDataParams,                                    [In]
      uint16_t wKeyNo,                                       [In]
      uint16_t wRefNoKUC );                                  [In]
```

***pDataParams:** Pointer to the Key Store parameter component.

wKeyNo: Position of the key to which a KUC reference will be assigned.

wRefNoKUC: Position of the KUC to assign to the given key entry.

The returned values from the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_ERR_INVALID_PARAMETER`: `wKeyNo` or `wReffNoKUC` out of valid range.

`Other`: Value returned by the underlying component.

8.1.2.7 Get Key Entry

This function reads the attributes of a key stored in the Key Store. The function does not return the key value, thus the KUC is not incremented.

```
phStatus_t phKeyStore_GetKeyEntry(
      void * pDataParams,                                [In]
      uint16_t wKeyNo,                                   [In]
      uint16_t wKeyVersionBufSize,                       [In]
      uint16_t * wKeyVersion,                            [Out]
      uint16_t * wKeyVersionLength,                      [Out]
      uint16_t * pKeyType );                                      [Out]
```

***pDataParams:** Pointer to the Key Store parameter component.

wKeyNo: Position of the key entry.

wKeyVersionBufSize: Size of the `wKeyVersion` buffer where the version information will be written. It needs to be at least `sizeof(uint16_t)*pDataParams->wNoOfVersions`. If the buffer is not large enough, this function quits without doing anything.

***wKeyVersion:** Array for the version numbers. All the version numbers of the key entry are subsequently written here.

***wKeyVersionLength:** Number of key values of the key entry. The limit is determined by the `wNoOfVersions` Key Store parameter and it is common for all the key entries.

***pKeyType:** Type of the keys stored in the Key Store entry.

The returned values from the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_ERR_BUFFER_OVERFLOW`: The buffer prepared for loading the version numbers is too small.

`PH_ERR_INVALID_PARAMETER`: `wKeyNo` argument is greater than the number of key entries `pDataParams->wNoOfKeyEntries`.

`Other`: Value returned by the underlying component.

8.1.2.8 Get Key Value

`This function returns a key value stored in the software Key Store.`

`Note: The corresponding KUC is incremented by one.`

```
phStatus_t phKeyStore_GetKey(
```

```
      void * pDataParams,                                   [In]
      uint16_t wKeyNo,                                      [In]
      uint16_t wKeyVersion,                                 [In]
      uint8_t bKeyBufSize,                                  [In]
      uint8_t * pKey,                                       [Out]
      uint16_t * pKeyType );                                [Out]
```

***pDataParams:** Pointer to the Key Store parameter component.

wKeyNo: Position of the key to be retrieved.

wKeyVersion: Version of the key to be retrieved.

bKeyBufSize: Size of the buffer where the read key will be written. Use the `phKeyStore_GetKeySize()` function to check the required buffer size.

***pKey:** Pointer to the array where the target key is returned.

***pKeyType:** Type of the requested key.

The returned values from the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_ERR_KEY`: The limit of read accesses for the requested key has been reached.

`PH_ERR_INVALID_PARAMETER`

- `wKeyVersion` for the given `wKeyNo` not found.
- `wKeyNo` greater than `pDataParams->wNoOfKeyEntries - 1`.

`PH_ERR_BUFFER_OVERFLOW`: Too small `bKeyBufSize` for the type of the requested key.

`Other`: Value returned by the underlying component.

8.1.2.9 Change KUC

This function changes the value of a KUC entry.

```
phStatus_t phKeyStore_ChangeKUC(
      void * pDataParams,                                   [In]
      uint16_t wRefNoKUC,                                   [In]
      uint32_t dwLimit );                                   [In]
```

***pDataParams:** Pointer to the Key Store parameter component.

wRefNoKUC: The KUC entry number (position) to be changed.

dwLimit: The Key Usage Counter limit to be assigned.

The returned values from the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_ERR_INVALID_PARAMETER`:

- `wRefNoKUC` greater than `pDataParams->wNoOfKUCEntries-1`.

`Other`: Value returned by the underlying component.

8.1.2.10 Get KUC

This function returns the value of a KUC entry.

```
phStatus_t phKeyStore_GetKUC(
      void * pDataParams,                                   [In]
      uint16_t wRefNoKUC,                                   [In]
```

```
    uint32_t * pdwLimit,                                    [Out]
    uint32_t * pdwCurVal );                                 [Out]
```

***pDataParams:** Pointer to the Key Store parameter component.

wRefNoKUC: KUC number (position) to be retrieved.

***pdwLimit:** KUC limit attribute in the KUC entry instance.

***pdwCurVal:** Current value of the KUC counter in the KUC entry instance.

The returned values from the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_ERR_INVALID_PARAMETER`:

- `wRefNoKUC` greater than `pDataParams->wNoOfKUCEntries-1`.

`Other`: Value returned by the underlying component.

8.2 Log Module

The NFC Reader Library Log module provides a tracking mechanism that records information about library modules during the execution of a project, which is especially useful during the code debugging phase.

This module belongs to the Common layer of the NFC Reader Library and it is independent of both the hardware and the platform selected for the project deployment.

Logging functionality requires I/O operations in order to print values on the console or create log files. Thus, the semihosting feature must be enabled in the project.

The module is activated by enabling the following macro in the *NxpRdbLib_PublicRelease/types/ph_NxpBuild.h* file:

```
#define NXPBUILD__PH_LOG
```

8.2.1 Log Parameter Structure

The NFC Reader Library defines a structure to handle logging operations.

```
typedef struct{
   pphLog_Callback_t  pLogCallback,
   phLog_RegisterEntry_t *  pRegisterEntries,
   uint16_t wNumRegisterEntries,
   uint16_t wMaxRegisterEntries
} phLog_DataParams_t;
```

pLogCallback: Function to be called when the logging is executed. This function shall be programmed by the developer.

pRegisterEntries: Array of `phLog_RegisterEntry_t` component entries.

wNumRegisterEntries: Number of valid entries in the `pRegisterEntries` array.

wMaxRegisterEntries: Maximum number of entries the `pRegisterEntries` array can hold.

8.2.1.1 Register Entries Structure

The Log module associates entries to each component registered for logging purposes.

```
typedef struct{
   void * pDataParams,
   phLog_LogEntry_t *  pLogEntries,
   uint16_t wNumLogEEntries,
   uint16_t wMaxLogEntries
} phLog_RegisterEntry_t;
```

pDataParams: Pointer to the component registered for the logging functionality.

pLogEntries: Array of `phLog_LogEntry_t` entries associated to a particular component.

wNumLogEEntries: Number of valid entries in the `pLogEntries` array.

wMaxLogEntries: Maximum number of entries the `pLogEntries` array can hold.

8.2.1.2 Log Entries Structure

Each component registered for logging has its own array of log entries structure associated. Each entry in this array contains information associated to a particular value, for instance a variable.

```
typedef struct {
   uint8_t bLogType,
   uint8_t const * pString,
   void const *  pData,
   uint16_t wDataLen,
   uint8_t bDataType
} phLog_LogEntry_t;
```

bLogType: Type of entry of the stored data:

- `PH_LOG_LOGTYPE_INFO (0x00): Log Type Info`
- `PH_LOG_LOGTYPE_ERROR (0x01): Log Type Error`
- `PH_LOG_LOGTYPE_WARN (0x02): Log Type Warning`
- `PH_LOG_LOGTYPE_DEBUG (0x03): Log Type Debug`

pString: String that describes the stored value.

pData: Pointer to the stored data. Type defined by the `bDataType` value.

wDataLen: Length of the stored data.

bDataType: Type of data stored at the entry:

- `PH_LOG_DATATYPE_BUFFER (0x00): Data Type Buffer`
- `PH_LOG_DATATYPE_VALUE (0x01): Data Type Value`

8.2.1.3 Logging Component Structures Example

The three structures that have been exposed should be properly associated and configured in order to setup the logging component correctly.

The figure below shows a logging example that is focused on the upper layers of a MIFARE Classic example. Three components have been registered for logging purposes: MIFARE Classic, ISO14443-3A and MIFARE components. Each component registered is limited to record two log entries where variables or buffers can be stored.

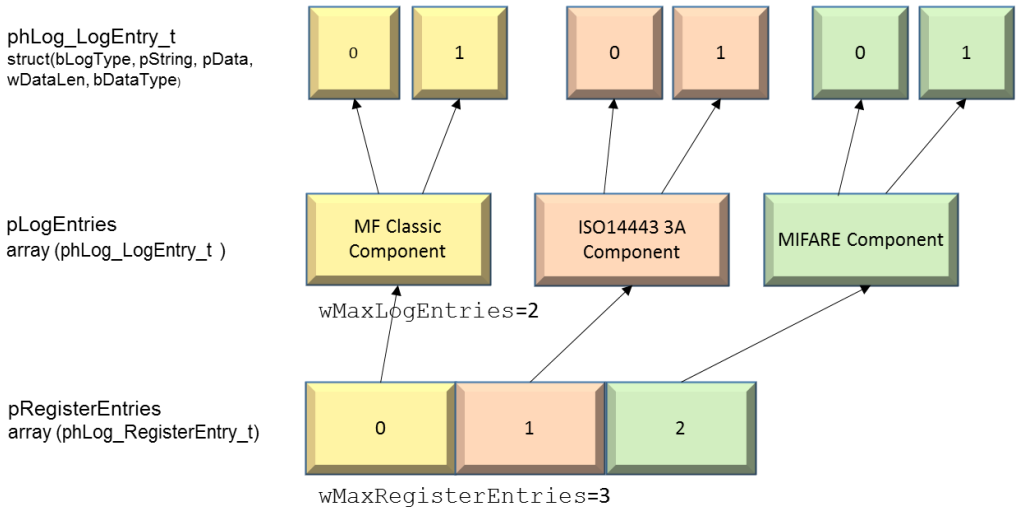


Fig 44. Logging structures example

8.2.1 Module Initialization and Registration

The Log Module must be initialized before using it for the data storage. During its initialization the callback function must be stated. This callback function will be executed when the logging is triggered. The single instance that is created stores the array that points to the entries associated to each component. Therefore, it shall only be called once.

```
phStatus_t phLog_Init(
      pphLog_Callback_t pLogCallback,                    [In]
      phLog_RegisterEntry_t * pRegisterEntries,          [In]
      uint16_t wMaxRegisterEntries);                     [In]
```

pLogCallback: Callback function that will be executed when the logging of some data is required.

*** pRegisterEntries:** An array of log register entries.

wMaxRegisterEntries: Maximum number of entries that the `pRegisterEntries` array can hold.

The returned values from the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_ERR_INVALID_PARAMETER`: Invalid combination of input parameters.

The Log Module creates an entry associated to each particular component of the NFC Reader Library (i.e. BAL Component, MIFARE Classic Component, etc.) by using the function below. Each component shall be registered in order to store data in its associated `pLogEntries` array. Thus, there shall be as many calls to this function as components to be registered.

```
phStatus_t phLog_Register(
      void *  pDataParams,                               [In]
      phLog_LogEntry_t *  pLogEntries,                   [In]
      uint16_t  wMaxLogEntries);                         [In]
```

pDataParams: Component of the NFC Reader Library to which the log entry is associated.

***pLogEntries:** An array of single log entries.

wMaxLogEntries: Maximum number of entries that the pLogEntries array can hold.

The returned values from the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_ERR_USE_CONDITION`: Logging initialization was not properly completed.

`PH_ERR_INVALID_PARAMETER`: Invalid combination of input parameters.

`PH_ERR_BUFFER_OVERFLOW`: The maximum size of the array of Log Register entries has been reached.

8.2.2 Information Storage

The log entries can be stored after the correct initialization of the logging module and the registration of the logging components. The logging information is recorded as long as the execution of the project continues.

The NFC Reader Library provides a set of functions to record variables depending on their type. The definition of these functions is similar as only the input data type changes. For this reason, only the function used to store `uint8_t` values is shown.

```
void phLog_AddParam_Uint8(
      void * pDataParams,                                   [In]
      uint8_t bLogType,                                     [In]
      char const * pName,                                   [In]
      uint8_t * pParam);                                    [In]
```

pDataParams: Component of the NXP Reader Library to which the log entry is associated.

bLogType: Type of entry. Valid values defined in `phLog.h`:

- `PH_LOG_LOGTYPE_INFO (0x00): Log Type : Inf`
- `PH_LOG_LOGTYPE_ERROR (0x01): Log Type : Error`
- `PH_LOG_LOGTYPE_WARN (0x02): Log Type : Warn`
- `PH_LOG_LOGTYPE_DEBUG (0x03): Log Type : Debug`

pName: Name of the entry to be recorded.

pParam: Value of the entry to be recorded.

In addition to these set of functions defined in the library API, the NFC Reader Library defines a set of helper functions to facilitate the recording of values. These helper functions can be found in the *NxpRdLib_PublicRelease/types/ph_Status.h* file.

8.2.3 Information Handling

The callback function that processes the recorded data shall be launched by the developer. This task is performed calling the `phLog_Execute()` function.

```
void phLog_Execute(
      void *  pDataParams,                                  [In]
      uint8_t  bOption)                                     [In]
```

pDataParams: Component of the NXP Reader Library to which the log entry is associated.

bOption: Option byte indicating the moment at which the callback function is called. Valid values are defined in `phLog.h`:

- `PH_LOG_OPTION_CATEGORY_ENTER (0x01)`: Logging takes place at the beginning of the function.
- `PH_LOG_OPTION_CATEGORY_GEN (0x02)`: Logging takes place in the middle of the function.
- `PH_LOG_OPTION_CATEGORY_LEAVE (0x03)`: Logging takes place before leaving the function.

The Log Module does not create any file or print any information by itself. It is developer's responsibility to implement the operations to be performed at the callback function when the logging module is executed. These operations could be, for instance, to print data on the console or to append data to a log file. A function that shows some recorded data on the console could look this way:

```
1    #ifdef NXPBUILD__PH_LOG
2    void my_Log_Function(void * pDataParams, uint8_t bOption, phLog_LogEntry_t *
     pEntries, uint16_t wEntryCount) {
3        printf("--- Function %s called at %x ---\n", pEntries[0].pString, bOption);
4        if(wEntryCount <= 0)
5          return;
6        uint16_t i;
7        for (i = 1; i < wEntryCount; i++ ) {
8            uint16_t wIndex;
9            uint8_t * pBuffer = (uint8_t *)(pEntries[i].pData);
10
11           printf("%s: ", pEntries[i].pString);
12           for (wIndex = 0; wIndex < pEntries[i].wDataLen; ++wIndex) {
13              printf("%02X\n ", pBuffer[wIndex]);
14           }
15       }
16       printf("\n\n");
17   }
18   #endif /* NXPBUILD__PH_LOG */
```

8.3 OSAL

The Operating System Abstraction Layer provides an API that isolates the embedded software from the underlying MCU. This way, developers can test their projects in different environments in a fast and convenient way.

In the NFC Reader Library, the OSAL module provides basic OS services like dynamic memory allocation and the management of hardware timers.

8.3.1 OSAL Structure

The OSAL component of the NFC Reader Library stores the information needed for the management of OSAL functionalities. Since the main objective of the OSAL component is to provide abstraction from the target MCU, a set of structures addressing specific

MCUs are defined by the NFC Reader Library. Depending on the hardware being used for the project deployment, the developer should declare the appropriate type of variable.

Table 24. OSAL component structures for valid target MCUs

OSAL Parameter structure	Target MCU
phOsal_Lpc12xx_DataParams_t	NXP LPC12xx MCU family
phOsal_Lpc17xx_DataParams_t	NXP LPC17xx MCU family

In order to provide abstraction, fields defined in each structure are the same, but the size of the arrays or the content of the variables may change.

```
typedef struct {
     Timer_Struct_t gTimers[LPC17XX_MAX_TIMERS];
} phOsal_Lpc17xx_DataParams_t;
```

***gTimers[]:** Structure containing information to be used for timers management. For further information see the Timer Management section (see section 8.3.3)

8.3.2 Memory management API

The memory management API provides a simple interface that allows developers to allocate and release memory in the MCU heap (the dynamically allocated memory area section). The allocated memory is reserved until it is released or the program terminates.

8.3.2.1 Allocate Memory

This function allocates free memory from the heap segment. If the requested amount of free memory was successfully allocated, a pointer to the granted memory is returned.

In fact, the built-in C function `malloc()` is called.

```
phStatus_t phOsal_GetMemory(
     void * pDataParams,                    [In]
     uint32_t dwLength,                     [In]
     void ** pMem );                        [Out]
```

***pDataParams**: Pointer to the MCU defined `phOsal` parameter component.

dwLength: Required memory size.

****pMem**: Pointer to the allocated memory.

The values returned by the function can be:

`PH_ERR_SUCCESS` - Operation successful.

`PH_ERR_RESOURCE_ERROR` - Requested memory space allocation failed.

8.3.2.2 Free Memory

This function releases memory previously allocated using `phOsal_GetMemory()` function.

In fact, the built-in C function `free()` is called.

```
phStatus_t phOsal_FreeMemory(
     void * pDataParams,                    [In]
     void * ptr );                          [In]
```

***pDataParams**: Pointer to the MCU defined `ph0sal` parameter component.

***ptr**: Pointer to the previously allocated memory to be released.

The value returned by the function is:

`PH_ERR_SUCCESS`: Operation successful.

8.3.3 Timer management API

The NFC Reader Library defines a set of functions that allow developers to make use of timers in their projects. These functions provide abstraction from the MCU on which the program is running, making the development and migration of the software much more convenient.

The OSAL component provides abstraction to developers, but it is obviously restricted by the underlying hardware. The number of timers available to the programmer is limited by the MCU on which the program is running. For instance, there are two 32 bit hardware timers in the LPC1227 MCU whereas the LPC1769 offers four 32 bit hardware timers.

The OSAL module provides utilization of hardware timers in two ways: software time delay and general timer usage.

Timers are an essential part of the NFC Reader Library, as they allow the correct functioning of many of the components. For instance, the Discovery Loop performs time delay after setting the reader chip for a particular NFC protocol or for guard interval between detection of Type B tags and Type F tags; the LLCP component relies on hardware timers for the correct management of the LLCP SYMM timer that ensures the Asynchronous Balanced Communication; and so on.

Note: Some of the components, especially on the PAL and HAL layers, rely on internal hardware timers of the attached reader IC for measuring timeouts defined by a specific NFC protocol.

Timer Management structure

The OSAL structure defines the `gTimers` array of structures to be used for the management of the hardware timers. The size of this array will vary depending on the target MCU on which the project is running.

```
typedef struct {
     uint32_t dwTimerId;
     uint8_t bTimerFree;
     ppCallback_t pApplicationCallback;
     void  *pContext;
} Timer_Struct_t;
```

dwTimerId: Unique identifier of the hardware timer .

bTimerFree: Indicates whether the timer is currently free or it is being used.

pApplicationCallback: Callback function to be called for this timer at completion.

***pContext**: Pointer to the input data to be processed by the callback function.

Operation flow

The following figure depicts the flow of functions to be called in order to ensure the correct functioning and management of hardware timers.

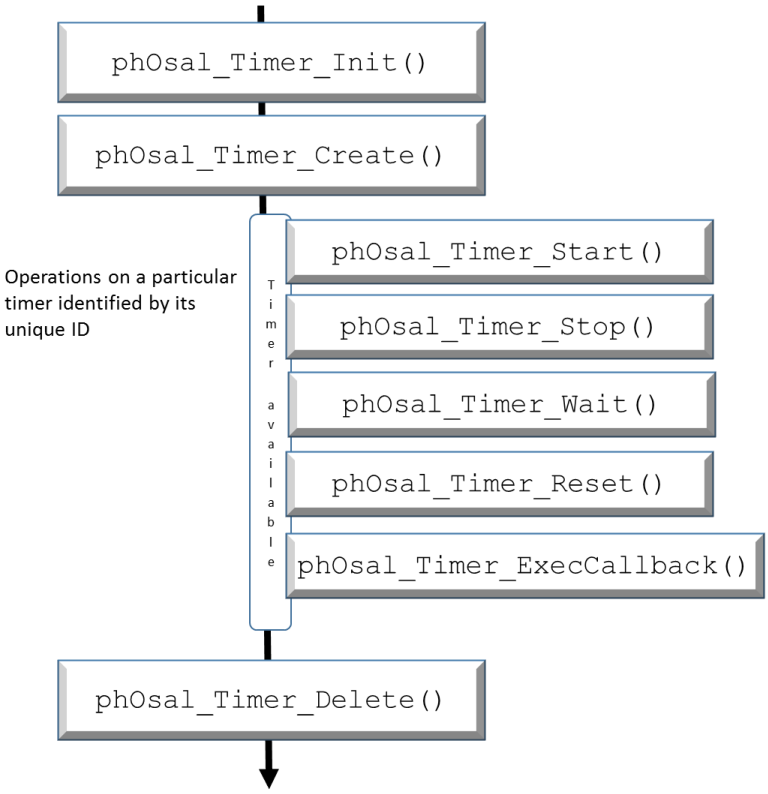


Fig 45. OSAL Component operation flow

8.3.3.1 Timer Init

The first step to be completed in order to work with timers in the NFC Reader Library is to call the `phOsal_Timer_Init()` that initializes the `phOsal_LpcXXxx_DataParams_t` structure and internal software structures aimed for the storage of timers related relevant information.

Once this function has been executed, timer interrupts are enabled and all the timers can be run.

```
phStatus_t phOsal_Timer_Init(
     phOsal_LpcXXxx_DataParams_t * pDataParams );            [In]
```

***pDataParams**: Pointer to the MCU defined `phOsal` parameter component.

The value returned by the function is:

`PH_ERR_SUCCESS`: Operation successful.

8.3.3.2 Timer Create

This function assigns an unused hardware timer of the target MCU to a particular task. The number of hardware timers that can be assigned this way is limited by the MCU

being used. Once the timer has been successfully created and its timer ID has been assigned, it can be used calling `phOsal_Timer_Start()` and `phOsal_Timer_Stop()` functions. In order to release an allocated timer, function `phOsal_Timer_Delete()` should be called, which sets the timer as free.

If all the available timers are currently in use, then no timer is assigned.

Note: Timer 0 is used by the NFC Reader Library as the LLCP LTO timer, and its creation is internally performed.

```
phStatus_t phOsal_Timer_Create(
     void * pDataParams,                                [In]
     uint32_t *timerId );                               [Out]
```

***pDataParams**: Pointer to the MCU defined `phOsal` parameter component.

***timerId**: ID of the assigned timer. If no free timer has been found, then this parameter is returned with value `PH_OSALNFC_INVALID_TIMER_ID` equal to 0xFFFF.

The values returned by the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_OSAL_ERR_NO_FREE_TIMER`: Both the timers are currently in use.

8.3.3.3 Timer Start

This function makes a particular timer start. When the timer expires after a user defined amount of time, the given application callback function is executed.

In order to stop the counting of a particular timer, function `phOsal_Timer_Stop()` should be called.

The timer should have previously been created by `phOsal_Timer_Create()`.

```
phStatus_t phOsal_Timer_Start(
     void * pDataParams,                                [In]
     uint32_t dwTimerId,                                [In]
     uint32_t dwRegTimeCnt,                             [In]
     ppCallBck_t pApplication_callback,                 [In]
     void  * pContext ); ,                              [In]
```

***pDataParams**: Pointer to the MCU defined `phOsal` parameter component.

***timerId**: Valid timer ID as returned by `phOsal_Timer_Create()`.

dwRegTimeCnt: Amount of time in MS after which the timer expires.

pApplication_callback: Pointer to the callback function that will be called once the timer expires. The user defined function must satisfy the following function prototype:

```
void (*ppCallBck_t)(uint32_t TimerId, void *pContext);
```

***pContext**: Pointer to the input data to be processed by the callback function.

The values returned by the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_OSAL_ERR_INVALID_TIMER`: Passed timer ID does not exist or it has not been created before.

8.3.3.4 Timer Stop

This function makes a particular timer stop. It does not release the timer, it only disables the timer by stopping the counting.

The timer should have previously been created by `phOsal_Timer_Create()`.

```
phStatus_t phOsal_Timer_Stop(
     void * pDataParams,                                [In]
     uint32_t dwTimerId );                              [In]
```

***pDataParams**: Pointer to the MCU defined `phOsal` parameter component.

***timerId**: ID of the timer to be stopped.

The values returned by the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_OSAL_ERR_INVALID_TIMER`: Passed timer ID does not exist or it has not been created before.

8.3.3.5 Timer Delete

This function stops a particular timer and releases it. The current content of the timer is erased.

```
phStatus_t phOsal_Timer_Delete(
     void * pDataParams,                                [In]
     uint32_t dwTimerId );                              [In]
```

***pDataParams**: Pointer to the MCU defined `phOsal` parameter component.

***timerId**: ID of the timer to be released.

The values returned by the function can be:

`PH_ERR_SUCCESS`: Operation successful.

`PH_OSAL_ERR_INVALID_TIMER`: Passed timer ID does not exist or it has not been created before.

8.3.3.6 Timer Wait

This function freezes a thread for a given amount of time determined by both the value and the time unit. While the thread is frozen nothing else is executed within the thread. After the completion of the user defined time, the thread continues its execution.

Note: The NFC Reader Library always uses hardware timer 1 for thread wait delay performed by this function, even if it was currently being used for any other purpose.

```
phStatus_t phOsal_Timer_Wait(
     void * pDataParams,                                [In]
     uint8_t bTimerDelayUnit,                           [In]
     uint16_t wDelay );                                 [In]
```

***pDataParams**: Pointer to the MCU defined `phOsal` parameter component.

bTimerDelayUnit: `It defines the` time unit.

- `PH_OSAL_TIMER_UNIT_MS` for milliseconds.
- `PH_OSAL_TIMER_UNIT_US` for microseconds.

wDelay: Amount of time in user defined time unit after which the timer expires.

The value returned by the function is:

`PH_ERR_SUCCESS`: Operation successful.

8.3.3.7 Timer Reset

This function resets the timer to its previously defined expiration value, which was defined using the `phOsal_Timer_Start()` function. Once the expiration value has been reset, the counting continues.

This is especially useful for those timers that are continuously reset. For instance, the LLCP defined LTO timer is reset every time a new packet is received.

The timer should have previously been created by `phOsal_Timer_Create()`.

```
phStatus_t phOsal_Timer_Reset (
     void * pDataParams,                                [In]
     uint32_t dwTimerId );                              [In]
```

***pDataParams**: Pointer to the MCU defined `phOsal` parameter component.

***timerId**: ID of the timer to be reset.

`PH_ERR_SUCCESS`: Operation successful.

8.3.3.8 Timer Execution Callback

When the developer generates a timer interruption, its corresponding timer ISR (Interrupt Service Routine) is assigned. After the completion of the timer, the systems calls its IRQ (Interruption Request) function, which is defined by the system – TIMER0_IRQHandler is assigned to hardware timer 0, and so on –. `phOsal_Timer_ExecCallback()` function shall be called within the IRQHandler function in order to execute the callback function that was defined during the `phOsal_Timer_Start()` function call.

Note: The need to call this function manually depends on the underlying hardware and its capacity to register the interrupt callback function on a specific interruption directly from the API, which would be done in the `phOsal_Timer_Start()` function.

```
phStatus_t phOsal_Timer_ExecCallback (
     void * pDataParams,                                [In]
     uint32_t dwTimerId );                              [In]
```

***pDataParams**: Pointer to the MCU defined `phOsal` parameter component.

***timerId**: ID of the timer of which the callback function is going to be executed.

The value returned by the function is:

`PH_ERR_SUCCESS`: Operation successful.

9. References

[1] NXP Generic Reader Library, http://www.nxp.com/documents/software/200312.zip

[2] NXP Export Controlled Library. (Available in DocStore [30]).

[3] NXP NFC Reader Library (To be published)

[4] **Data Sheet** MF1S503X MIFARE Classic 1K - Mainstream contactless smart card IC for fast and easy solution development, available on http://www.nxp.com/documents/data_sheet/MF1S503x.pdf

[5] **Data Sheet** - MIFARE Ultralight ; MF0ICU1, MIFARE Ultralight contactless single-ticket IC, BU-ID Doc. No. 0286**[1], available on http://www.nxp.com/documents/data_sheet/MF0ICU1.pdf

[6] **Data Sheet** – MIFARE Ultralight EV1- contactless ticket IC, available on http://www.nxp.com/documents/data_sheet/MF0ULX1.pdf

[7] **Data Sheet** – MIFARE MF0ICU2 – MIFARE Ultralight C , available on http://www.nxp.com/documents/short_data_sheet/MF0ICU2_SDS.pdf

[8] **Data Sheet** - MIFARE DESFire; MF3ICDx21_41_81, MIFARE DESFire EV1 contactless multi-application IC, BU-ID Doc. No. 1340**, available on http://www.nxp.com/documents/short_data_sheet/MF3ICDX21_41_81_SDS.pdf

[9] **Data Sheet** - JIS Standard JIS X 6319 Specification of implementation for integrated circuit(s) cards - Part 4: High Speed proximity cards

[10] **Data Sheet** – Innovision Topaz, http://downloads.acs.com.hk/drivers/en/TDS_TOPAZ.pdf

[11] **Data sheet** - MFRC523; Contactless reader IC, BU-ID Doc. No. 1152**, available on http://www.nxp.com/documents/data_sheet/MFRC523.pdf

[12] **Data sheet** - CLRC663; Contactless reader IC, BU-ID Doc. No. 1711**, available on http://www.nxp.com/documents/data_sheet/CLRC663.pdf

[13] **Data sheet** - MFRC522; Contactless reader IC, BU-ID Doc. No. 1121**, available on http://www.nxp.com/documents/data_sheet/MFRC522.pdf

[14] **Data sheet** – PN512; Transmission module, BU-ID Doc. No. 1113**, available on http://www.nxp.com/documents/data_sheet/PN512.pdf

[15] **Data sheet** – MFRC631; Contactless reader IC, BU-ID Doc. No. 2274**, available on http://www.nxp.com/documents/data_sheet/MFRC631.pdf

[16] **Data sheet** – MFRC630; Contactless reader IC, BU-ID Doc. No. 2275**, available on http://www.nxp.com/documents/data_sheet/MFRC630.pdf

[17] **Data sheet** – SLRC610; Contactless reader IC, BU-ID Doc. No. 2276**, available on http://www.nxp.com/documents/data_sheet/SLRC610.pdf

[18] **ISO/IEC Standard** - ISO/IEC 14443 Identification cards - Contactless integrated circuit cards - Proximity cards

[19] **ISO/IEC Standard** - ISO/IEC 18092 Information technology - Telecommunications and information exchange between systems - Near Field Communication- Interface and Protocol (NFCIP-1)

1. [1] ** … BU ID document version number

[20] **Technical Specification** Logical Link Control Protocol, NFCForum-TS-LLCP_1.1, available on www.nxp.com/redirect/nfc-forum.org/specs/spec_license

[21] **Technical Specification** – Simple NDEF Exchange Protocol, NFCForum-TS-SNEP_1.0, available on www.nxp.com/redirect/nfc-forum.org/specs/spec_license

[22] **Technical Specification** – Type 1 Tag Operation, NFCForum-TS-Type-1-Tag_1.1, available on www.nxp.com/redirect/nfc-forum.org/specs/spec_license

[23] **Technical Specification** – Type 2 Tag Operation, NFCForum-TS-Type-2-Tag_1.1, available on www.nxp.com/redirect/nfc-forum.org/specs/spec_license

[24] **Technical Specification** – Type 3 Tag Operation, NFCForum-TS-Type-3-Tag_1.1, available on www.nxp.com/redirect/nfc-forum.org/specs/spec_license

[25] **Technical Specification** – Type 4 Tag Operation, NFCForum-TS-Type-4-Tag_2.0, available on www.nxp.com/redirect/nfc-forum.org/specs/spec_license

[26] **Technical Specification** – NFC Data Exchange Format, NFCForum-TS-NDEF_1.0, available on www.nxp.com/redirect/nfc-forum.org/specs/spec_license

[27] **Application note** - AN11211 Quick Start Up Guide RC663 Blueboard, available on http://www.nxp.com/documents/application_note/AN11211.pdf

[28] **Application note** – AN11308 Quick Start Up Guide PNEV512B, available on http://www.nxp.com/documents/application_note/AN11308.pdf

[29] LPCZone, http://www.nxp.com/techzones/microcontrollers-techzone/news.html

[30] NXP DocStore, https://www.docstore.nxp.com/flex/DocStoreApp.html#/l

[31] LPCXPresso IDE, http://www.lpcware.com/lpcxpresso/code-red

[32] LPCXpresso target boards, http://www.nxp.com/techzones/microcontrollers-techzone/tools-ecosystem/lpcxpresso.html

[33] AN11211 CLEV663B Blueboard Quick Start Guide, http://www.nxp.com/documents/application_note/AN11211.pdf

[34] AN11308 PNEV512B Blueboard Quick Start Guide, http://www.nxp.com/documents/application_note/AN11308.pdf

[35] NXP Contactless reader IC Demoboards ordering, http://www.nxp.com/products/identification_and_security/#demoboards

[36] AN11342 How to Scale Down the NXP Reader Library, http://www.nxp.com/documents/application_note/AN11342.pdf

[37] UM10721 NXP NFC Reader Library User Manual http://www.nxp.com/documents/user_manual/UM10721.pdf

10. Legal information

10.1 Definitions

Draft — The document is a draft version only. The content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included herein and shall have no liability for the consequences of use of such information.

10.2 Disclaimers

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.

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 accepts 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.

Evaluation products — This product is provided on an "as is" and "with all faults" basis for evaluation purposes only. NXP Semiconductors, its affiliates and their suppliers expressly disclaim all warranties, whether express, implied or statutory, including but not limited to the implied warranties of non-infringement, merchantability and fitness for a particular purpose. The entire risk as to the quality, or arising out of the use or performance, of this product remains with customer.

In no event shall NXP Semiconductors, its affiliates or their suppliers be liable to customer for any special, indirect, consequential, punitive or incidental damages (including without limitation damages for loss of business, business interruption, loss of use, loss of data or information, and the like) arising out the use of or inability to use the product, whether or not based on tort (including negligence), strict liability, breach of contract, breach of warranty or any other theory, even if advised of the possibility of such damages.

Notwithstanding any damages that customer might incur for any reason whatsoever (including without limitation, all damages referenced above and all direct or general damages), the entire liability of NXP Semiconductors, its affiliates and their suppliers and customer's exclusive remedy for all of the foregoing shall be limited to actual damages incurred by customer based on reasonable reliance up to the greater of the amount actually paid by customer for the product or five dollars (US$5.00). The foregoing limitations, exclusions and disclaimers shall apply to the maximum extent permitted by applicable law, even if any remedy fails of its essential purpose.

10.3 Licenses

Purchase of NXP <xxx> components
<License statement text>

10.4 Patents

Notice is herewith given that the subject device uses one or more of the following patents and that each of these patents may have corresponding patents in other jurisdictions.

<Patent ID> — owned by <Company name>

10.5 Trademarks

Notice: All referenced brands, product names, service names and trademarks are property of their respective owners.

<Name> — is a trademark of NXP B.V.

11. Index

No index entries found.

12. List of figures

13. List of tables

14. Contents

For more information, please visit: http://www.nxp.com
For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 07 April 2014
Document identifier: 294210

For more information, please visit: http://www.nxp.com
For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 07 April 2014
Document identifier: 294210

For more information, please visit: http://www.nxp.com
For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 07 April 2014
Document identifier: 294210

For more information, please visit: http://www.nxp.com
For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 07 April 2014
Document identifier: 294210