UM10663

NXP Reader Library User Manual based on CLRC663 and PN512 Blueboard Reader projects

Rev. 1.2 — 24 July 2013
257412

User Manual
COMPANY PUBLIC

Document information

Info	Content
Keywords	NXP Reader Library, MFRC523, MFRC522, MFRC500, MFRC530, MFRC531, MFRC630, MFRC631, SLRC610
Abstract	This document describes the implementation and usage of the NXP Reader Library Public with special stress on MIFARE Classic command and CLRC663 native command set implementation.

Revision history

Rev	Date	Description
1.2	20130724	Added “PN512” in the descriptive title
1.1	20130502	Updated the URL of the NXP Reader library in the first reference.
1.0	20130301	Initial version

Contact information

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

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

1. Introduction

1.1 NXP libraries comparison

NXP Reader Library

This document describes the **NXP Reader Library (Public),** which is the software written in C language enabling the customers to create their own software stack for their contactless reader. The NXP Reader Library is available at [1].

There are another two NXP Reader Libraries giving to the developer advanced possibilities. Particular libraries are restricted for usage on LPC microcontroller models. For example the NXP Reader Library that is intended to run on NXP LPC1227 board.

NXP Reader Library Export Controlled: comprehensive software API enabling the customers to create their own software stack for their contactless reader. The library includes software representing cards, which may be export controlled or common criteria certified. Therefore the whole software is export controlled and subject to NDA with NXP. Library enables usage of SAM module which enables encrypted communication between the host and reader chip (PCD). There must be hardware support from SAM unit, of course.

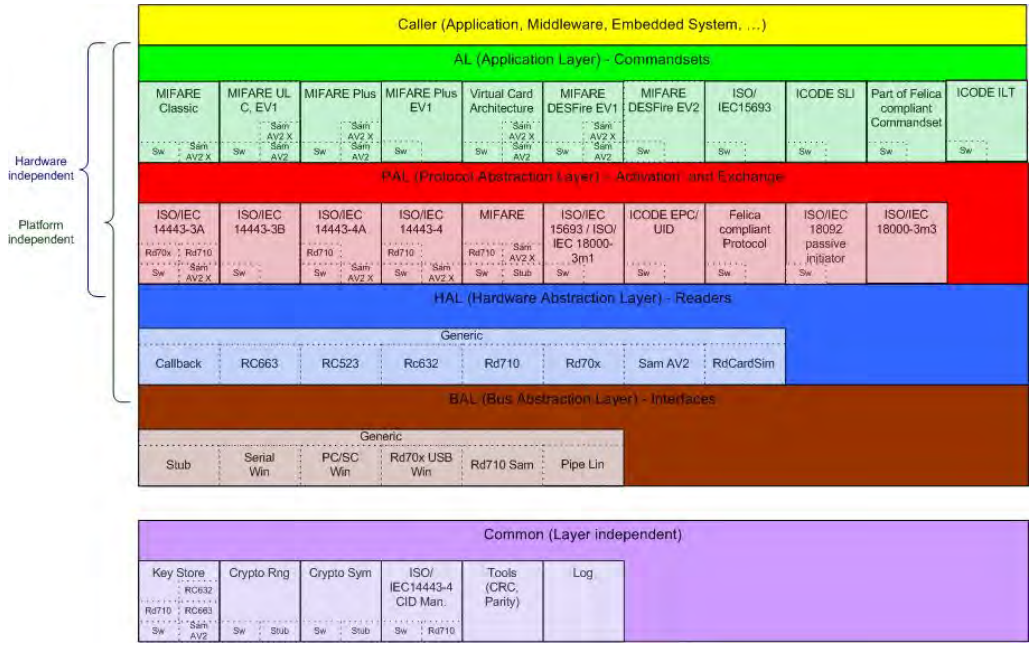

Fig 1. Structure of the NXP Reader Library Export Controlled

NXP Reader Library NFC P2P: is intended to run on NXP LPC1227 board which is either connected to PNEV512 v1.4 blue board or CLRC663 Blue board v2.1. It is the software support for Near Field Communication (NFC) including Data Exchange Format.

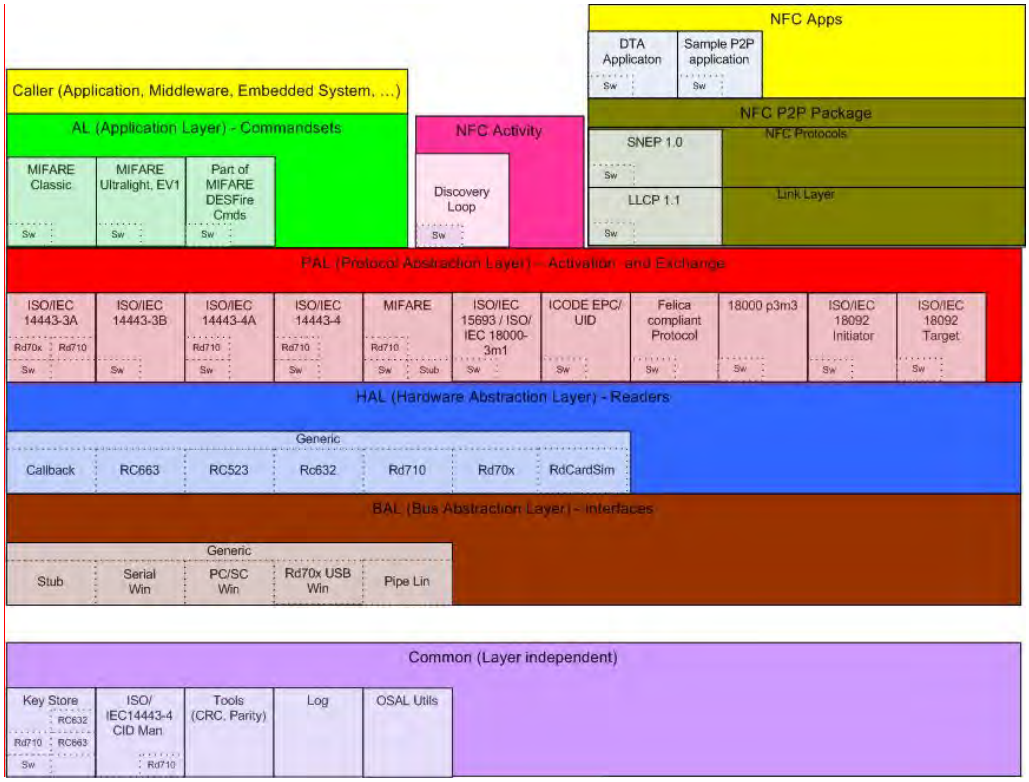

Fig 2. Structure of the NXP Reader Library NFC P2P

1.1.1 Layer Structure of the NXP Reader Library

The MCU implementing the Library is able to utilize several types of reader chips to access any MIFARE card. To satisfy such versatile feature, architecture of the Library was designed with multi layered structure – see Fig 3. Each layer is independent from the other layers.

Vertical structure of the NXP Reader Library is classified into following layers:

- Application layer (AL)
- Protocol abstraction layer (PAL)
- Hardware abstraction layer (HAL)
- Bus abstraction layer (BAL)

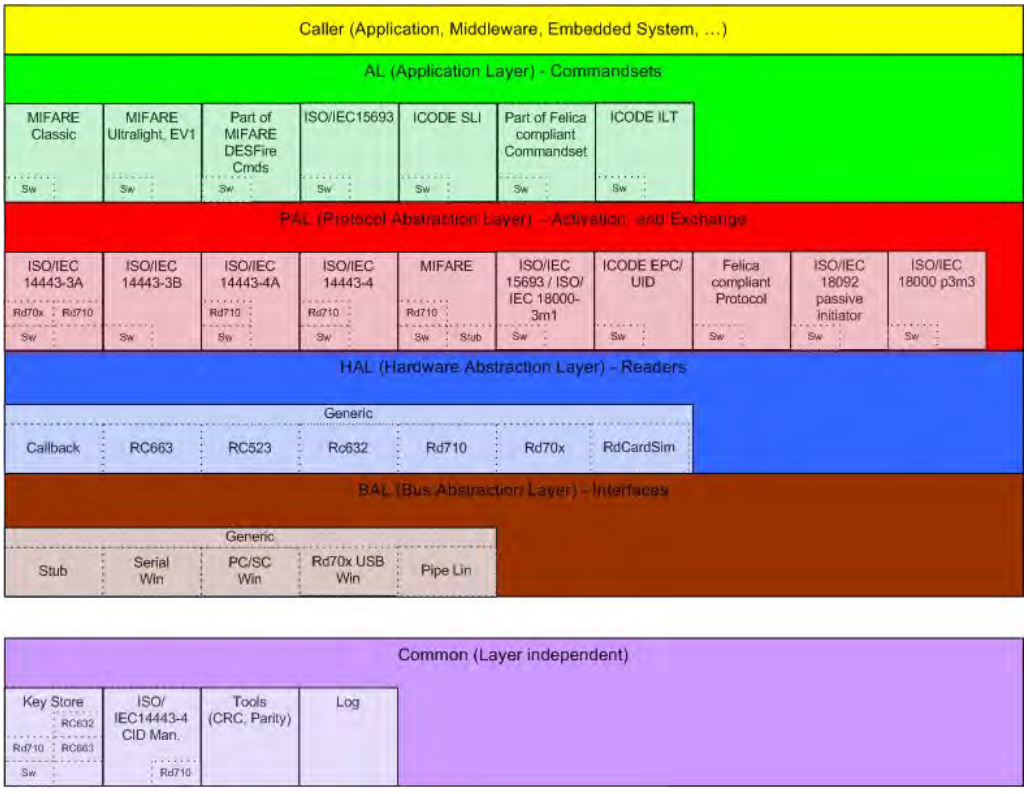

Fig 3. Structure of the NXP Reader Library

1.1.1.1 Application Layer

Although this document describes the NXP Reader Library from Fig 3, in this section there is slightly introduced most of the cards (components) from (green) AL in Fig 1. The Application Layer of the NXP Reader Library implements commands for only MIFARE Classic and MIFARE Ultralight among contactless cards listed below. The Application Layer enables developer to access particular card by using its command set – it means reading, writing, modifying and other operations with data on the card.

MIFARE Classic[2]: the leading industry standard for contactless and dual interface smart card schemes, with an immense worldwide installed base. The platform offers a full range of compatible contactless smartcard and reader ICs, as well as dual-interfaces ICs. The MIFARE Classic family covers contactless smart cards used in applications like public transport, access management, loyalty cards and many more. MIFARE Classic is fully compliant with ISO/IEC 14443 Type A, available with 1k and 4k memory and 7 Byte as well as 4 Byte identifiers. The Application Layer of the MIFARE Classic is closer described in Section 2.4.

MIFARE Ultralight EV1[3]:It is intended for use with single trip tickets in public transportation networks, loyalty cards or day passes for events as a replacement for conventional ticketing solutions such as paper tickets, magnetic stripe tickets or coins. The mechanical and electrical specifications of MIFARE Ultralight are tailored to meet the requirements of paper ticket manufacturers. It can be easily integrated into existing

contactless system without need for serious changes of the system. MIFARE Ultralight is fully compliant with ISO/IEC 14443 Type A.

The Application Layer of the MIFARE Ultralight is closer described in Section 2.5.

MIFARE Plus[4]: Migrate classic contactless smart card systems to the next security level. After the security upgrade, MIFARE Plus uses AES-128 (Advanced Encryption Standard) for authentication, data integrity and encryption. MIFARE Plus is based on open global standards for both air interface and cryptographic methods at the highest security level.

MIFARE DESFire[5]: fully compliant with ISO/IEC14443A(part 1 - 4) and uses optional ISO/IEC7816-4 commands. The selectable cryptographic methods include 2KTDES, 3KTDES and AES128. The highly secure microcontroller based IC is certified with Common Criteria EAL4+. MIFARE DESFire is multi-application smart card used in public transport schemes, access management or closed-loop e-payment applications. It fulfills the requirements for fast and highly secure data transmission, flexible memory organization and interoperability with existing infrastructure.

ISO/IEC15693[6]: contactless vicinity card defined by ISO/IEC Standard.

ICODE SLI[7]: the first member of a product family of smart label ICs based on ISO/IEC15693. This IC is dedicated for intelligent label applications like supply chain management as well as baggage and parcel identification in airline business and mail service.

Felica[11][12]: contactless smart card developed by the Sony company with usage spread in Japan.

ICODE ILT[8]: dedicated chip for passive, intelligent tags and labels supporting the ISO18000-3 mode 3 RFID standard. It is especially suited for applications where reliable identification and high anti-collision rates are required. The ICODE ILT supports ISO/IEC18000-3mode3 RFID standard.

1.1.1.2 Protocol Abstraction Layer

The Protocol Abstraction Layer implements the **activation and exchange** operations regarding the protocol of the contactless communication. Each protocol has its own folder in the library structure *NxpRdbLib_PublicRelease/comps/phpal<protocolName>*. The NXP Reader Library supports following ISO standard protocols:

ISO/IEC14443-3A[9]: air interface communication at 13.56MHz for the cards of type A

ISO/IEC14443-3B[9]: air interface communication at 13.56MHz for the cards of type B

ISO/IEC14443-4[9]: 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.

ISO/IEC14443-4A[9]: previous protocol for the cards of type A

MIFARE(R): needs to be included for any MIFARE card. Contains support for MIFARE authentication and data exchange reader chip and PC or PICC according to protocols ISO/IEC14443-3A and ISO/IEC14443-4.

ISO/IEC15693[6]/**ISO18000-3m1**[10]: smart cards based on ISO/IEC15693 are used like SKIPASS.

ISO/IEC18000-3m3[10]: The ISO 18000-3 mode 3/EPC Class-1 HF standard allows the commercialized provision of mass adoption of HF RFID technology for passive smart tags and labels.

applications are supply chain management and logistics for worldwide use.

ISO/IEC18092 Mode Passive Initiator[12]: the DEP protocol as well as the passive communication mode

Felicia: compliant protocol[11], parts of it are also part of ISO 18092[12]

This document is focused on description of MIFARE Classic card which is compliant to the ISO/IEC14443-3A protocol. The protocol layer ensures software functionality of specified procedures: RequestA, Activate, Anti-collision, Select, HaltA, WakeUpA. The PAL layer is software solution, how to ensure on the PC or MCU side all the procedures, states and the states changes regarding to ISO/IEC14443-3A[9].

1.1.1.3 Hardware abstraction layer

The Hardware Abstraction Layer implements the hardware specific elements of the reader chip. The reader chip could be considered as an additional module of MCU or PC – connection provider between the MCU or PC and the card. From this point of view the HAL layer is software how to utilize this module. There are many different card readers supported by the NXP Reader Library. They differ in peripheral modules, memory organization, command set etc or support various package of ISO protocols. According to Fig 3 the NXP Reader Library supports two Pegoda readers and three reader chips and their derivatives.

RD70x and RD710 Pegoda readers: contactless reader designed for an easy reader adaptation to a PC to use these devices for test and application purposes and reference design for new reader development based on the CLRC632 MFRC500 reader ICs respectively.

Three major card reader chips and their derivatives:

MFRC523: MFRC522, PN512 (reader functionality)

CLRC632: MFRC500, MFRC530, MFRC531

CLRC663: MFRC631, MFRC630, SLRC610

There are several aspects that can be each reader chip considered from:

- support for card types
- support for types of ISO protocols from previous section 1.1.1.2
- support for secure access module (SAM)
- support for host communication interface

Check datasheet of particular reader chip to check if it satisfies your requirements.

Although there are some differences between these reader chips within each group also, PCDs of one group share the same HAL layer source code – each PCD group owns a separate folder in *NxpRdbLib_PublicRelease/comps/phhalHw<readerChip>*.

Functions of this layer are responsible for execution of native commands of particular reader chip and all the management to be utilized effectively concurrently with stress on preventing software from infinite loops. This means mainly:

- reading/writing from/into reader's registers.

- RF field management, receiver and transmitter configuration
- Timers configuration
- Resolving interrupt sources from reader chip
- FIFO management

Data passed from upper layers are packed together with command code into special order, thus reader is able to decode them as command code, address and data. Section 2.1 is focused on description of functions performing CLRC663 native commands and section 2.2 describes some HAL layer functions independent of reader chip.

The NXP Reader Library also supports Pegoda reader

1.1.1.4 Bus Abstraction Layer

The Bus Abstraction Layer ensures correct communication interface between the master device and the reader chip. The master device can be PC with Windows or Linux platform installed or MCU and it sends to the reader chip commands and other command related data like addresses and data bytes. The card reader can send some register values or received data like the response to requests from the mater device.

The NXP Reader Library supports following communication interfaces:

SerialWin: serial connection for Windows platform

Rd70x USB Win: drivers for Windows platform to enable connection to Pegoda reader

PcscWin: driver for PC/SC interface running on Windows platform

Stub: Originally it was intended like component without functionality to ease implementation of additional busses. Currently it supports SPI, I2C and RS232 interfaces enabling connection to the Blueboard[27] or Xpresso board.

The reader chip can possibly send replies – mostly when MCU requests value of particular register.

1.1.1.5 Common Layer

The NXP Reader Library also includes utilities independent of any card and hardware (card reader) – meaning they can be implemented regardless of hardware components. All of them are encapsulated into the Common Layer

phTools: this part of common layer is able to perform binary operations related to CRC and bit parity both for various lengths. CRC and parity check are used very rarely in communication between PC or MCU and the card reader.

Note: These functions must not be implemented into the reader and any card intercommunication. All the CRC checksum is done by both the sides automatically.

phKeyStore: is key handling software – storage, changing key format etc.. But the NXP Reader Library supports only key storage utility. Only the NXP Reader Library Export Controlled version supports full key storage functionalities and for those card readers from *phKeyStore* part in Fig 3.

phLog: software module enabling log files creation.

1.1.1.6 Data structures of the layers

Each layer uses its own data structure to store important data, parameters for hardware and software configuration. Members of structures determine branching of program flow

during software running, which means triggering, enabling or disabling of particular software utilities and hardware modules.

At the beginning of program, it is necessary to load data parameter structure of layer with default values by calling an initialization function of that layer. At the moment, when particular data parameter structure is being initialized, underlaying structure must have already been allocated - pointer to the structure must be known. So it is recommended to begin initialization of particular data parameter structures from bottom to top layers. To successfully implement functions described in this document, all the layer components from Table 1 need to be incorporated and their structures initialized:

Table 1. Library layer components

Component	Component structure	Initialization function	Description	Layer
BAL	`phbalReg_Stub_DataParams_t`	`phbalReg_Stub_Init()`	Bus interface	BAL
HAL CLRC663	`phhalHw_Rc663_DataParams_t`	`phhalHw_Rc663_Init()`	Reader chip CLRC663 component	HAL
ISO14443-3A	`phpalI14443p3a_Sw_DataParams_t`	`phpalI14443p3a_Sw_Init()`	Protocol ISO14443-3A component	PAL
MIFARE	`phpalMifare_Sw_DataParams_t`	`phpalMifare_Sw_Init()`	Authentication function and ISO14443-3A implements this layer component	PAL
MIFARE Classic	`phalMfc_Sw_DataParams_t`	`phalMfc_Sw_Init()`	MIFARE Classic card component	AL
MIFARE Ultralight	`phalMful_Sw_DataParams_t`	`phalMful_Sw_Init()`	MIFARE Ultralight card component	AL

Initialization functions for HAL CLRC663, PAL ISO14443-3A, MIFARE Classic, MIFARE Ultralight are described in sections 2.2.2 2.3.2, 2.4.1, 2.5.1 respectively. BAL and MIFARE layer components initializations are mentioned in sample code in section 3 in line 150 and 153 respectively.

Check each init function of data parameter structures whether its default values agrees with your intended cause, otherwise, init manually. Pay special attention to `bBalConnectionType` member in `phhalHw_Rc663_DataParams_t`, to set custom bus communication between reader chip and MCU. However, you will need to initiate `bBalConnectionType` “manually” (see sample code in section 3 line 164 or line 169), due to HAL init function sets RS232 interface on default. The partial description of CLRC663 HAL layer structure and its initialization function are in sections 2.2.1 and 2.2.2 respectively. The HAL layer has its own couple of `phhalHw_SetConfig()` and `phhalHw_GetConfig()` functions (see sections 2.2.3 and 2.2.4) to modify parameters of the structure and change configuration of the reader.

2. Exemplary explanation of the library functions

A project defining a CLRC663 Blueboard communication with a MIFARE classic is herewith explained.

2.1 HAL: CLRC663 commands

2.1.1 General

The reader chip is not able to execute compiled C code, but executes commands of command set. For that reason the PCD reader is controlled by host device able to execute the library code (PC or MCU) and sends particular commands to the PCD via communication bus. In this chapter are described command related functions. Together with command itself, described functions mostly call many other support routines to ensure flawless command run. So the described functions are not PCD's native commands literally, but larger functions ensuring particular command execution. This attitude intends to focus developer on most important issues without redundant knowledge about underlaying functions.

2.1.2 Idle command

This command indicates that the PCD is in idle mode. This command is used to terminate actually executed command. It also indicates. Unlike other commands this command does not have its own function. Only way, how to perform the Idle command is

```
phhalHw_WriteRegister(pDataParams, PHHAL_HW_RC663_REG_COMMAND,
PHHAL_HW_RC663_CMD_IDLE));
```

2.1.3 LPCD Low Power Card Detection - phhalHw_Rc663_Cmd_Lpcd()

This function performs the low-power card detection and an automatic trimming of the LPO. The values of the sampled I and Q channel are stored in the register map. The value is compared with the min/max values in the register. If it exceeds the limits, an LPCD IRQ will be raised. After the command the standby is activated if selected.

```
phStatus_t phhalHw_Rc663_Cmd_Lpcd(
                                phhalHw_Rc663_DataParams_t *pDataParams,    [In]
                                uint8_t bMode,                              [In]
                                uint8_t bI,                                 [In]
                                uint8_t bQ,                                 [In]
                                uint16_t wPowerDownTimeMs,                  [In]
                                uint16_t wDetectionTimeUs );                [In]
```

***pDataParams:** pointer to the HAL layer data parameter structure.

bMode: there are three different modes that LPCD can run. Each is matched with one of three following defines:

`PHHAL_HW_RC663_CMD_LPCD_MODE_DEFAULT:` Try LPCD until timeout is reached. T4timer runs on 64 kHz frequency. Function ignores last two input arguments, but uses `PHHAL_HW_RC663_LPCD_T3_RELOAD_MIN` and `PHHAL_HW_RC663_LPCD_T4_RELOAD_MIN` from *phhalHw_Rc663_Config.h* file to set power-down and detection time amounts.

`PHHAL_HW_RC663_CMD_LPCD_MODE_POWERDOWN:` Powers down the PCD for a certain amount of time and performs LPC after wakeup. If no card is found the PCD is powered down again and the procedure is restarted. If a card is found the function returns and the PCD remains powered up.

PHHAL_HW_RC663_CMD_LPCD_MODE_POWERDOWN_GUARDED: Same as previous, but uses the timeout set with either PHHAL_HW_CONFIG_TIMING_US or PHHAL_HW_CONFIG_TIMING_MS as abort criteria. Guard-timer in this case is only running during the power-up phases, so the timeout has to be adjusted properly.

PHHAL_HW_RC663_CMD_LPCD_MODE_OPTION_TRIMM_LPO: Or this bit to the desired mode to perform LPO trimming together with the LPCD command.

PHHAL_HW_RC663_CMD_LPCD_MODE_OPTION_IGNORE_IQ Or this bit to the desired mode to prevent the function to set I and Q channel values.

bI: I-Channel value in case of no card on antenna.

bQ: is Q-Channel value in case of no card on antenna.

wPowerDownTimeMs: power-down time in milliseconds if power-down mode is used.

wDetectionTimeUs: detection time in microseconds if power-down mode is used.

returnValues:

PH_ERR_SUCCESS - card present.

PH_ERR_IO_TIMEOUT - no card found.

PH_ERR_SUCCESS - operation successful.

Other: depending on implementation and underlaying component.

2.1.4 LoadKey - phStatus_t phhalHw_Rc663_Cmd_LoadKey()

This function loads a MIFARE Key (6 bytes) into the Key buffer of the reader, that necessary for further authentication.

```
phStatus_t phhalHw_Rc663_Cmd_LoadKey(
                                phhalHw_Rc663_DataParams_t * DataParams,    [In]
                                uint8_t * pKey );                           [In]
```

***pDataParams:** pointer to the HAL layer data parameter structure.

***pKey:** pointer to the 6 byte MIFARE key array. Value of this key is defined by software, so any value of key satisfying 6 byte size may be used for attempt to Authenticate. This is different from LoadKeyE2 command - phhalHw_Rc663_Cmd_LoadKeyE2() function, which loads key from EEPROM without any possibility to find out actual value of the key.

returnValues:

PH_ERR_SUCCESS - operation successful.

Other: depending on implementation and underlaying component.

2.1.5 LoadKeyE2 - phhalHw_Rc663_Cmd_LoadKeyE2()

This function loads the MIFARE Key (6 bytes) from a position in the PCD EEPROM Key Storage Area into the Key buffer. Subsequently, with such loaded key the authentication can be executed.

Note: Before you decide to use this function, see more complex phalMfc_Authenticate() function in section 2.4.2 and consider which one is more suitable for your intended purpose.

```
phStatus_t phhalHw_Rc663_Cmd_LoadKeyE2(
                                phhalHw_Rc663_DataParams_t * DataParams,    [In]
                                uint8_t bKeyNo );                           [In]
```

***pDataParams:** pointer to the HAL layer data parameter structure.

bKeyNo: key number in EEPROM in range 0x00 - 0xFF.

returnValues:

`PH_ERR_SUCCESS` - operation successful.

Other: depending on implementation and underlaying component.

2.1.6 MFAuthent - `phhalHw_Rc663_MfcAuthenticate_Int()`

This function performs MFAuthent command (entire three pass Authentication procedure to the given EEPROM block of the card) with key currently loaded in the Key buffer. If Authentication passes through all three phases successfully, then the EEPROM block is fully accessible for data manipulation MIFARE Classic commands: reading, writing, increment, decrement, restore and data transfer. For MIFARE Classic, if once access to the block is obtained, then access to entire sector (4 blocks) retains until next attempt to authenticate to any EEPROM block of card.[2]

Note: This function does not provide any loading into the Key buffer. The key intended to be used for the authentication needs to be loaded by `phhalHw_Rc663_Cmd_LoadKey()` function (section 2.1.4). In case of using the key from the key storage area of PCD's EEPROM see an alternative to this function `phalMfc_Authenticate()` in section 2.4.2.

```
phStatus_t phhalHw_Rc663_MfcAuthenticate_Int(
                                phhalHw_Rc663_DataParams_t * pDataParams,   [In]
                                uint8_t bBlockNo,                           [In]
                                uint8_t bKeyType,                           [In]
                                uint8_t * pUid );                           [In]
```

***pDataParams:** pointer to the HAL layer data parameter structure.

Set `pDataParams->pKeyStoreDataParams = NULL` to run EEPROM key store branch.

bBlockNo: number of block in EEPROM of the card to authenticate to. Authentication to block enables access to entire section which the block is part of. In case of the MIFARE Classic it must be in range from 0 to 63.

bKeyType: can be either `PHAL_MFC_KEYA` or `PHAL_MFC_KEYB`. In case of MIFARE Classic each block can be authenticated by using anyone of these two keys specified for that block.[2]

***pUid:** pointer to UID of the card to be authenticated. Each card has its own unique identification number.

Note: In this function there is no pointer to array containing 6 byte key used for authentication. Thus before calling this function, the key intended to be used for authentication must be loaded into the Key buffer by the function `phhalHw_Rc663_Cmd_LoadKey()` – see section 2.1.4. or by the `phKeyStore_SetKey()`– see section 2.1.14. In case of key stored in MKA see `phalMfc_Authenticate()` in section 2.4.2 as an alternative to this function.

returnValues:

`PH_ERR_AUTH_ERROR`: Authentication procedure fails, if the key currently loaded in the Key buffer of reader does not match with key for given block in card tried to be accessed.

`PH_ERR_INVALID_PARAMETER` – `wKeyVersion` other than NULL.

`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.

PH_ERR_SUCCESS - operation successful.

Other: depending on implementation and underlaying component.

2.1.7 Receive - phhalHw_Rc663_Cmd_Receive()

This function handles receiver, then waits for eventually coming data. There are many data related errors that can occur during reception: framing, CRC, buffer overflow, protocol, integrity, collision error. Weather successful frame reception or any among many possible errors occur, they are used as return values, so developer can further manage particular situation himself. Receiving also terminates as soon as number of received data overflows FIFO WaterLevel. Waterlevel is set on default size FIFO-1 bytes, so abort from this interrupt source should not occur. If necessary, use WriteRegister(pDataParams, PHHAL_HW_RC663_REG_WATERLEVEL, customWaterlevelValue) to set the Waterlevel on custom value.

```
phStatus_t phhalHw_Rc663_Cmd_Receive(
                                phhalHw_Rc663_DataParams_t * DataParams,    [In]
                                uint16_t wOption,                           [In]
                                uint8_t ** ppRxBuffer,                      [Out]
                                uint16_t * pRxLength );                     [Out]
```

***pDataParams:** pointer the HAL layer data parameter structure.

pDataParams->bRfResetAfterTo == PH_ON: provokes RF field reset, after unsuccessful reception due to timeout. Although this is just software parameter (no register related), it is recommended to set this option by phhalHw_SetConfig(pDataparams, PHHAL_HW_CONFIG_RFRESET_ON_TIMEOUT, PH_ON) function (see section 2.2.3).

wOption: PHHAL_HW_RC663_OPTION_RXTX_TIMER_START should be passed here to prevent software from freezing while waiting for received data for a long time. This option employs both T0 and T1 timers.

****ppRxBuffer:** the pointer to byte where function writes received data.

***pRxLength:** is pointer to the number of received data bytes.

returnValues:

PH_ERR_IO_TIMEOUT – no coming data detected and while timeout from timer 1 terminated.

PH_ERR_BUFFER_OVERFLOW - Data is written into the FIFO when it is already full.

PH_ERR_READ_WRITE_ERROR - Data was written into the FIFO, during a transmission of a possible CRC, during "RxWait", "Wait for data" or "Receiving" state, or during an authentication command.

PH_ERR_FRAMING_ERROR - A valid SOF was received, but afterwards less than 4 data bits were received.

PH_ERR_COLLISION_ERROR – A collision has occurred. Software routine retrieves valid bits of last byte.

PH_ERR_PROTOCOL_ERROR – A protocol error has occurred. When a protocol error occurs the last received data byte is not written into the FIFO.

PH_ERR_INTEGRITY_ERROR - A data integrity error has been detected. Possible cause can be a wrong parity or a wrong CRC.

PH_ERR_SUCCESS - operation successful.

even if waterlevel reached no error indicated also

Other: depending on implementation and underlaying component.

For further information of particular error see CLRC663 datasheet [2].

2.1.8 Transmit - phhalHw_Rc663_Cmd_Transmit()

This function just transmits data from the transmit buffer without starting receiving afterwards.

```
phStatus_t phhalHw_Rc663_Cmd_Transmit(
                                phhalHw_Rc663_DataParams_t * pDataParams,   [In]
                                uint16_t wOption,                           [In]
                                uint8_t *pTxBuffer,                         [In]
                                uint16_t wTxLength );                       [In]
```

***pDataParams:** pointer to the HAL layer data parameter structure.

pDataParams->wMaxPrecachedBytes: defines maximal amount of data that into FIFO buffer. Use the phhalHw_SetConfig(pDataParams, PHHAAL_HW_MAX_PRECACHED_BYTES, customPrecacheBufferSize) function to set custom number of precached bytes.

wOption: option parameter must be only one of following values, else function returns error status:

PH_EXCHANGE_BUFFERED_BIT buffers Tx-Data into internal buffer instead of transmitting it.

PH_EXCHANGE_LEAVE_BUFFER_BIT does not clear the internal buffer before operation. If this bit is set and data is transmitted, the contents of the internal buffer are sent first.

PHHAL_HW_RC663_OPTION_RXTX_TIMER_START starts timer after transmission before reception, reset timer counter value

***pTxBuffer:** data to transmit.

wTxLength: length of data to transmit.

returnValues:

PH_ERR_INVALID_PARAMETER – invalid value of the wOption parameter.

PH_ERR_INTERNAL_ERROR – 'No data' error. Data should be sent, but no data is in FIFO.

PH_ERR_BUFFER_OVERFLOW – Data written into the FIFO when it is already full.

PH_ERR_READ_WRITE_ERROR – Data was written into the FIFO, during a transmission of a possible CRC, during "RxWait", "Wait for data" or "Receiving" state, or during an authentication command.

PH_ERR_SUCCESS - operation successful.

Other: depending on implementation and underlaying component.

2.1.9 WriteE2 - phhalHw_Rc663_Cmd_WriteE2

Write one byte of data to the given EEPROM address. There is no software waiting loop to protect program flow from delay, while writing data into EEPROM.

```
phStatus_t phhalHw_Rc663_Cmd_WriteE2(
                                phhalHw_Rc663_DataParams_t * pDataParams,   [In]
                                uint16_t wAddress,                          [In]
                                uint8_t bData );                            [In]
```

***pDataParams:** pointer to the HAL layer data parameter structure.

wAddress: address in EEPROM, where data byte shall be written to. Firmware does not verify validity of address.

bData: data byte to be written.

returnValues:

`PH_ERR_READ_WRITE_ERROR` – error occurred during writing data into EEPROM – invalid `wAddress` parameter

`PH_ERR_SUCCESS` - operation successful.

Other: depending on implementation and underlaying component.

2.1.10 WriteE2 Page - `phhalHw_Rc663_Cmd_WriteE2Page`

Write up to 64 bytes of data to a given EEPROM page. There is no software waiting loop to stop program flow, while writing data into EEPROM. In comparison with previous function, this function handles medium amount of data.

```
phStatus_t phhalHw_Rc663_Cmd_WriteE2Page(
                                phhalHw_Rc663_DataParams_t * pDataParams,   [In]
                                uint16_t wAddress,                          [In]
                                uint8_t *pData,                             [In]
                                uint8_t bDataLen );                         [In]
```

***pDataParams:** pointer to the HAL layer data parameter structure.

wAddress: 2 byte address. Better said page number in range from 0 to 128.

***pData:** pointer to data byte array to be stored in EEPROM.

bDataLen: length of data to be written, up to 64 bytes.

returnValues:

`PH_ERR_INVALID_PARAMETER` - `wAddress` greater than 64

- `bDataLen` out of range 0 – 128

`PH_ERR_READ_WRITE_ERROR` – error occurred during writing data into EEPROM

`PH_ERR_SUCCESS` - operation successful.

Other: depending on implementation and underlaying component.

2.1.11 ReadE2 - `phhalHw_Rc663_Cmd_ReadE2()`

This function reads up to 256 bytes from the EEPROM of the reader. Since all requested data are firstly loaded into FIFO buffer, High Alert Interrupt is enabled. This causes abort as soon as loaded data in FIFO reaches FIFO top Waterlevel defined in Waterlevel (03h) register. Outside of this function, you can increase FIFO size from default 256 to 512 bytes by setting FIFO. Use `phhalHw_SetConfig(pDataParam, PHHAL_HW_RC663_CONFIG_FIFOSIZE, PHHAL_HW_RC663_VALUE_FIFOSIZE_512)` function (see section 2.2.3) to increase FIFO size (remember, this clears FIFO also). The Waterlevel is set one less than actual FIFO size by default.

```
phStatus_t phhalHw_Rc663_Cmd_ReadE2(
                                phhalHw_Rc663_DataParams_t * pDataParams,   [In]
                                uint16_t wAddress,                          [In]
                                uint16_t wNumBytes,                         [In]
                                uint8_t * pData );                          [Out]
```

***pDataParams:** pointer to the HAL layer data parameter structure.

wAddress: 2 byte address; Range is 0x0000 - 0x1FFF.

wNumBytes: number of data bytes to read. If `wAddress+wNumBytes > EEPROM_SIZE`, then function aborts before any read operation, although some addresses might be valid.

***pData:** pointer to requested data. This value must not be NULL, else no data will be returned.

returnValues:

`PH_ERR_INVALID_PARAMETER` - `wNumBytes` greater than 256 or greater than actual size of FIFO invalid address

- `bDataLen` out of range 0 – 128

`PH_ERR_READ_WRITE_ERROR` - error occurred during writing data into EEPROM

`PH_ERR_SUCCESS` - operation successful.

Other: depending on implementation and underlaying component.

2.1.12 LoadReg command - `phhalHw_Rc663_Cmd_LoadReg()`

Reads a defined number of bytes from the EEPROM and copies the value into the register set, beginning at the given register address. Function includes a software protection from attempt to access invalid address. Verification is done before even the first read command execution. All the register and EEPROM addresses intended to be read/written must be in valid range, otherwise no load register operation is performed, although some of EEPROM or register addresses might be valid.

```
phStatus_t phhalHw_Rc663_Cmd_LoadReg(
                                phhalHw_Rc663_DataParams_t * pDataParams,   [In]
                                uint16_t wEEAddress,                        [In]
                                uint8_t bRegAddress,                        [In]
                                uint8_t bNumBytes );                        [In]
```

***pDataParams:** pointer to the HAL layer data parameter structure.

wEEAddress: 2 byte address within the section 2 of EEPROM: from 192 to 6143.

bRegAddress: register address in range of 0 – 128.

bNumBytes: number of bytes to copy.

returnValues:

`PH_ERR_INVALID_PARAMETER` – attempt to access EEPROM address out of section 2 or register address out of 0 – 128.

`PH_ERR_READ_WRITE_ERROR` – error occurred during writing data into EEPROM

`PH_ERR_SUCCESS` - operation successful.

Other: depending on implementation and underlaying component.

2.1.13 LoadProtocol - phhalHw_Rc663_ApplyProtocolSettings()

This function configures receiver and transmitter of reader to communicate according given ISO protocol. The function needs to be called before the Activation. Reader chip keeps configuration of the latest protocol set, unless changed by this function or direct command LoadProtocol or LoadRegister. All values necessary to configure RX and TX registers correctly are read from special EEPROM section and loaded to RX/TX configuration registers important to protocol selection.

```
phStatus_t phhalHw_Rc663_ApplyProtocolSettings(
                                phhalHw_Rc663_DataParams_t * pDataParams,   [In]
                                uint8_t bCardType );                        [In]
```

***pDataParams:** pointer to the HAL layer data parameter structure.

Except the load protocol settings themselves, it is necessary receiver module to be set additionally. For this purpose following parameter is dedicated:

`pDataParams->bLoadRegMode` is binary switch and It must be one of two following values

`PH_OFF`: function performs loading RX/TX registers with default software defined values from `gpkhhalHw_Rc663_<protocol_name>` array in *phhalHw_Rc663_Int.c* file. This is much more comfortable and highly recommended option than the next one. This is default value after the HAL layer data parameter structure initiated.

`PH_ON`: function calls native Load Register command to load RX/TX registers with values from dedicated EEPROM addresses. This option assumes that the configuration values have been stored yet in the EEPROM of the reader. This implies other parameters to be set:

`pDataParams->pLoadRegConfig[0]`: MSB byte of the EEPROM address

`pDataParams->pLoadRegConfig[1]`: LSB byte of the EEPROM address

`pDataParams->pLoadRegConfig[2]`: register address. 0x34 is the first address for the configuration of the receiver of CLRC663 reader.

`pDataParams->pLoadRegConfig[3]`: number of bytes to be copied. Recommended value for configuration of CLRC663 reader is 6, which is length of configuration registers of receiver in register space.

All EEPROM and register addresses must be valid.

bCardType: protocol related to card intended to be handled. Constants representing all the cards are defined in *phhalHw.h* file.

`PHHAL_HW_CARDTYPE_CURRENT`: reapply settings for current Communication Mode

`PHHAL_HW_CARDTYPE_ISO14443A`: ISO/IEC 14443A Mode

`PHHAL_HW_CARDTYPE_ISO14443B`: ISO/IEC 14443B Mode

`PHHAL_HW_CARDTYPE_FELICA`: Felica (JIS X 6319) Mode

`PHHAL_HW_CARDTYPE_ISO15693`: ISO/IEC 15693 Mode

`PHHAL_HW_CARDTYPE_ICODEEPCUID`: NXP I-Code EPC/UID Mode

`PHHAL_HW_CARDTYPE_I18000P3M3`: ISO/IEC 18000-3 Mode3

`PHHAL_HW_CARDTYPE_I18092MPI`: ISO/IEC ISO18092 (NFC) Passive Initiator Mode

`PHHAL_HW_CARDTYPE_I18092MPT`: ISO/IEC ISO18092 (NFC) Passive Target Mode

PH_ERR_SUCCESS - operation successful.

Note: This function configures same communication protocol for both receiver and transmitter as well. If developer needs for any purpose to run them on different protocols concurrently, in this case it is necessary to do it via direct calling of lower layer function phhalHw_Rc663_Cmd_LoadProtocol() - software equivalent of native Load Protocol command. But be careful about using this function, because it does not provide necessary timer configurations.

returnValues:

PH_ERR_INVALID_PARAMETER – attempt to access EEPROM address out of section 2 or register address out of 0 – 128.

PH_ERR_READ_WRITE_ERROR – error occurred during writing data into EEPROM

PH_ERR_SUCCESS - operation successful.

Other: depending on implementation and underlaying component.

2.1.14 StoreKeyE2 - phKeyStore_SetKey()

This function writes a given key to Firstly, a given key is passed to the FIFO and subsequently written to specified position in EEPROM. Unlike CLRC663 native SetKeyE2 command, this function stores just 2 keys into reader's key storage area. If an incomplete key (less than 6 bytes) is passed to this function, that key is not stored into the EEPROM.

```
phStatus_t phKeyStore_SetKey(
                                void * pDataParams,                 [In]
                                uint16_t wKeyNo,                    [In]
                                uint16_t wKeyVersion,               [In]
                                uint16_t wKeyType,                  [In]
                                uint8_t pNewKey,                    [Out]
                                uint16_t wNewKeyVersion );          [Out]
```

***pDataParams:** pointer to the HAL layer data parameter structure.

wKeyNo: indicates the first key in MIFARE key area (up to 128) that will be written. In other words number of section that key is matched to.

wKeyVersion: is key version of the key to be loaded. Parameter has no effect.

wKeyType: is key type of the key to be loaded. Only PH_KEYSTORE_TYPE_MIFARE is supported.

***pNewKey:** is pointer to the new key to be stored in EEPROM.

wNewKeyVersion: is new key version of the key to be updated. Parameter has no effect.

returnValues:

PH_ERR_READ_WRITE_ERROR – unsuccessful writing the key into EEPROM. If writing not finished after time defined by PH_KEYSTORE_RC663_EEP_WR_TO_MS in phKeyStore_Rc663_Int.h then function stops command executing and returns this error.

- error from write into EEPROM

PH_ERR_UNSUPPORTED_PARAMETER – attempt to store an unsupported type of key

PH_ERR_INVALID_PARAMETER – wKeyNo is greater than 128

PH_ERR_SUCCESS - operation successful.

Other: depending on implementation and underlaying component.

2.1.15 SoftReset - phhalHw_Rc663_Cmd_SoftReset()

This function performs the SoftReset command. Triggered by this command all the default values for the register setting will be read from the EEPROM and copied into the register set. It is up to the caller to wait until the IC is powered-up and ready again. In addition to that, the caller should call phhalHw_ApplyProtocolSettings() again to re-configure the reader chip.

```
phStatus_t phhalHw_Rc663_Cmd_SoftReset(
                                    phhalHw_Rc663_DataParams_t * pDataParams );   [In]
```

***pDataParams:** pointer to the HAL layer data parameter structure.

returnValues:

PH_ERR_SUCCESS - operation successful.

Other: depending on implementation and underlaying component.

2.2 HAL: Additional description of HAL layer

In this section there is firstly partially described HAL layer data parameter structure and for CLRC663 reader, then HAL layer functions common for all the reader chips supported by the library. Those functions can be implemented independently of hardware. Field controlling and reading/writing registers functions are general functionalities necessary for various the PCDs.

2.2.1 CLRC663 HAL layer data parameter structure

Entire structure is defined in the file *phhalHw.h* in folder *NxpRdLib_PublicRelease\intfs.* In this section there are mentioned only some parameters from the CLRC663 HAL layer data parameter structure which are implemented in the functions described in this document.

void ***pBalDataParams:** pointer to the undelaying BAL data parameter structure.

uint8_t **bLoadRegMode:** binary switch for enabling/disabling (PH_ON/PH_OFF) Load Register mode in ApplyProtocolSettings() function (see section 2.1.13). If Load Register mode intended to be used, pLoadRegConfig (see line bellow) must have already been defined.

uint8_t ***pLoadRegConfig:** pointer to configuration buffer for Load Register mode.

uint8_t **bCardType:** type of card for which the HAL component is configured for. This member is set by ApplyProtocolSettings() function (see section 2.1.13).

uint16_t wCfgShadow[PHHAL_HW_RC663_SHADOW_COUNT]; configuration shadow. Stores configuration of receiver and transmitter for current card type.

uint16_t **wFieldOffTime:** field off time in milliseconds for FieldReset() function described in section 2.2.9. How long FieldReset() function holds the field turned off while doing nothing else.

`uint16_t` **wFieldRecoveryTime:** field recovery time in milliseconds. How long `FieldReset()` function (see section 2.2.9) holds the field turned on while doing nothing else.

uint8_t **bBalConnectionType:** type of the underlying BAL connection interface. This parameter is necessary from software point of view, thus it has no impact on hardware pin configuration communication interface in the MCU.

`PHHAL_HW_BAL_CONNECTION_RS232:` reader connected to the MCU via RS232.

`PHHAL_HW_BAL_CONNECTION_SPI:` reader connected to the MCU via SPI.

`PHHAL_HW_BAL_CONNECTION_I2C:` reader connected to the MCU via I2C.

`uint8_t` **bRfResetAfterTo:** binary switch for field reset after timeout. If any PAL ISO/IEC14443-3A function (described in section) receives no response from PICC after timeout expiration and this parameter is set `PH_ON` then PCD RF field is reset by `FieldReset()` function.

2.2.2 Initialization HAL - `phhalHw_Rc663_Init()`

This function loads entire HAL layer data parameter structure with values. The HAL layer data parameter structure contains much more parameters than number of input parameters of this function. Consequently, by this function developer can influence only those HAL layer parameters matched to input parameters whilst the rest of the HAL parameters are initialized on default values. Check definition of initialization function for particular reader chip in *NxpRdLib_PublicRelease\comps\phhalHw\src\<FolderOfReaderChip>\phhalHw<ReaderChip>* whether default initialization values satisfy your intended purpose. Use `phhalHw_SetConfig()` function (see section 2.2.3) to modify any parameter of a HAL layer component on any legal value.

```
phStatus_t phhalHw_Rc663_Init(
                                phhalHw_Rc663_DataParams_t * pDataParams,   [In]
                                uint16_t wSizeOfDataParams,                 [In]
                                void * pBalDataParams,                      [In]
                                uint8_t * pLoadRegConfig,                   [In]
                                uint8_t * pTxBuffer,                        [In]
                                uint16_t wTxBufSize,                        [In]
                                uint8_t * pRxBuffer,                        [In]
                                uint16_t wRxBufSize );                      [In]
```

***pDataParams:** pointer to the HAL layer parameter structure.

wSizeOfDataParams: specifies the size of the data parameter structure.

***pBalDataParams:** pointer to the underlaying BAL layer data parameter structure.

***pLoadRegConfig:** pointer to configuration buffer for Load Register mode in `ApplyprotocolSettings()` (see section 2.1.13).

***pTxBuffer:** pointer to global transmit buffer used by data exchange in PAL ISO/IEC14443-3A layer.

wTxBufSize: size of the global transmit buffer.

***pRxBuffer:** pointer to global receive buffer used by data exchange in PAL ISO/IEC14443-3A layer.

wRxBufSize: size of the global receive buffer. Specify the buffer +1 byte, because one byte is reserved for SPI communication.

2.2.3 SetConfig - phhalHw_SetConfig()

This function performs changing the value of the particular parameter (member) of the HAL layer data parameter structure, but it does not modify the structure itself (adding or erasing new parameters - members) because one structure is fix defined for particular reader chip. For example, this function can be used to set timeouts for `FieldReset()` function. This function is implemented by many functions of HAL and PAL layer as well, for example to set timeouts and bits according to set desired communication protocol. If a parameter is closely matched with particular register, in addition `phhalHw_SetConfig()` function executes writing the value to correct PCD register (for example `bFifoSize` or `wMaxPrecachcedBytes` are related to FifoSize register). Therefore rather use this function to modify desired parameter of HAL layer component.

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

***pDataParams:** pointer to the HAL layer data parameter structure.

wConfig: configuration identifier specifies parameter (member) to be changed. All the configuration identifiers (`PHHAL_HW_CONFIG_<CONFIGURATION_IDENTIFIER>`) are listed in *phhalHw.h* file in folder *NxpRdLib_PublicRelease\intfs*.

wValue: configuration value. New value of the parameter.

returnValue:

2.2.4 GetConfig - phhalHw_GetConfig()

This function is complementary to the previous `phhalHw_SetConfig()` function (see previous section 2.2.3). It reads current configuration value of given parameter from HAL data parameter structure component.

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

***pDataParams:** pointer to the HAL layer data parameter structure.

wConfig: configuration identifier specifies parameter (member) to be read. All the configuration identifiers (`PHHAL_HW_CONFIG_<CONFIGURATION_IDENTIFIER>`) are listed in *phhalHw.h* file in folder *NxpRdLib_PublicRelease\intfs*.

***pValue:** pointer to variable, where read value is copied into.

2.2.5 ReadRegister - phhalHw_ReadRegister()

This function reads a value from any PCD register.

```
phStatus_t phhalHw_ReadRegister(
                                void * pDataParams,                    [In]
                                uint8_t bAddress,                      [In]
```

```
                                    uint8_t * pValue );                    [Out]
```

***pDataParams:** pointer to the HAL layer data parameter structure

bAddress: address of the register to be the value read from. Addresses of all the registers are defined in folder *NxpRdLib_PublicRelease\intfs* in *phhalHw_<readerChip>_Cmd.h* file according to the reader chip datasheet.

***pValue:** new value of the register to be written.

returnValues:

`PH_ERR_SUCCESS` operation successful.

`PH_ERR_INTERFACE_ERROR` hardware problem.

2.2.6 WriteRegister - `phhalHw_WriteRegister()`

This function writes a given value to any PCD register. The MCU controls PCD via this function including sending commands to PCD command register.

```
phStatus_t phhalHw_WriteRegister(
                                    void * pDataParams,                    [In]
                                    uint8_t bAddress,                      [In]
                                    uint8_t bValue );                      [In]
```

***pDataParams:** pointer to the HAL layer data parameter structure

bAddress: address of the register to be the value read from. Addresses of all the registers are defined in folder *NxpRdLib_PublicRelease\intfs* in *phhalHw_<readerChip>_Cmd.h* file according to the reader chip datasheet.

bValue: new value of the register to be written.

returnValues:

`PH_ERR_SUCCESS` operation successful.

`PH_ERR_INTERFACE_ERROR` hardware problem.

2.2.7 FieldOn - `phhalHw_FieldOn()`

This function turns on RF field by enabling transmitter pins of the reader chip.

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

***pDataParams:** pointer to the HAL layer data parameter structure.

returnValues:

`PH_ERR_SUCCESS` operation successful.

`PH_ERR_INTERFACE_ERROR` communication error.

2.2.8 FieldOff - `phhalHw_FieldOff()`

This function turns off RF field by disabling transmitter pins of the reader chip.

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

***pDataParams:** pointer to the HAL data layer parameter structure.

returnValues:

PH_ERR_SUCCESS operation successful.

PH_ERR_INTERFACE_ERROR communication error.

2.2.9 FieldReset - phhalHw_FieldReset()

This function performs reset of the RF field. Firstly the RF field is turned off by FieldOff() (see section 2.2.8) function and subsequently the RF field is enabled by FieldOn() (see section 2.2.7) function. Both those states are hold for a certain amount of time while doing nothing else.

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

***pDataParams:** pointer to the HAL layer data parameter structure. HAL structure parameters pDataParams->wFieldOffTime and pDataParams->wFieldRecoveryTime determine how long (in milliseconds) FieldOff() and FieldOn() functions hold their field states. Default value for both the timeouts is 5 milliseconds. Whilst field off and field recovery times can be changed by phhalHw_SetConfig() function individually, units are unchangeable (milliseconds are fixed).

returnValues:

PH_ERR_SUCCESS operation successful.

PH_ERR_INTERFACE_ERROR communication error.

2.2.10 Wait - phhalHw_Wait()

This function utilizes reader chip timers T0 and T1 to hold program flow for a given amount of time. Program leaves this function when timeout elapses.

```
phStatus_t phhalHw_Wait(
                void * pDataParams,                  [In]
                uint8_t bUnit,                       [In]
                uint16_t wTimeout );                 [In]
```

***pDataParams:** pointer to the HAL layer data parameter structure.

bUnit: unit of given timeout value. Either PHHAL_HW_TIME_MICROSECONDS or PHHAL_HW_TIME_MILLISECONDS.

wTimeout: timeout value.

returnValue:

PH_ERR_SUCCESS operation successful.

PH_ERR_INVALID_PARAMETER bUnit is invalid.

PH_ERR_INTERFACE_ERROR communication error.

2.3 PAL: ISO/IEC14443-3A

2.3.1 ISO14443A part 3

The ISO/IEC 14443 part 3 defines the start of communication and how to select the PICC and resolve situations, when more cards are present in the field of reader. Sometimes this is called “Card activation Sequence”. ISO/IEC 14443-3 protocol is split into A and B version. MIFARE Classic uses A type.

A particular feature of these functions is complexity. They represent software implementation of procedures defined in ISO/IEC14443-3 exploiting functions of underlaying HAL layer to execute partial steps of procedures. All the following mentioned functions are involved in protocol abstraction layer with `phpalI14443p3a_Sw_DataParams_t` storing parameters of layer.

It is recommended to use activation function `phpalI14443p3a_ActivateCard()` described in section 2.3.4 which implements all other functions inside and executes all the procedures according ISO/IEC14443p3. This is just a recommendation due to comfort. If any reason to call particular function, it is possible call it separately, of course.

2.3.2 Init ISO14443-3A - `phpalI14443p3a_Sw_Init()`

This function initiates PAL ISO14443-3A 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 structure.

wSizeOfDataParams: specifies the size of the data parameter structure. Recommended to pass here `sizeof(phalMfc_Sw_DataParams_t)`.

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

returnValues:

`PH_ERR_SUCCESS` - operation successful.

`PH_ERR_INVALID_DATA_PARAMS` – `wSizeOfDataParams` does not agree with defined size of PAL ISO14443p3A structure.

Except previous function this layer implements also one additional important component MIFARE Although PAL MIFARE component

2.3.3 Request A - `phStatus_t phpalI14443p3a_RequestA()`

This function transmits request type A (REQA) then it expects immediate answer to that request (ATQA). Data rates are automatically configured on 106kHz data rate for both receiver and transmitter. During this operation CRC module of reader chip is turned off for both reception and transmission signal. After request is transmitted, routine waits for any answer until timeout (timers T1, T0) expires.

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

***pDataParams:** pointer to the PAL layer data parameter structure.

***pAtqa:** pointer to ATQA. If RequestA is successful, then 2 byte ATQA is written to this variable.

returnValues:

`PH_ERR_PROTOCOL_ERROR` - invalid response received.

`PH_ERR_SUCCESS` - operation successful.

Other: depending on implementation and underlaying component.

Note:. If no answer to request is received nor any changes of field near subcarrier detected by receiver during dedicated time, then function is aborted with timeout error. There are two time constants defined in *phpalI14443p3a_Sw_Int.h* file to determine for ATQA timeout: `PHPAL_I14443P3A_EXT_TIME_US, PHPAL_I14443P3A_SELECTION_TIME_US`.

Resulting waiting time is in microseconds and equals to sum of both the values.

2.3.4 Activate Card - `phpalI14443p3a_ActivateCard()`

This function gets card into Active state, whether it was is Halt state or has not been activated yet. After successful activation, the card may be turned into Halt state due to intention to be handled near future use. Activate procedure is quite complex, therefore this function calls nearly all the other functions from this chapter: RequestA or Wake up, Anticollision, Selection. Inside this function they are integrated into powerful software tool executing entire Activation procedure covering all the states and situations specified in ISO/IEC14443p3.

This function performs Anti-collision and Selection operation respectively. Successful Activation results in acquiring actual and complete UID of given card. Even if there are more PICCs present in the range of PCD's field, the activation function ensures just one UID captured. If the card has been pushed to Halt state before, this function provides Wake up command.

```
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 structure. `UidLength` and `Uidab` are changed.

***pUidIn:** pointer to UID of card to get activated. If this parameter equals NULL and there is at least one card in range of PCD's field, then one card will be activated – UID of a card will captured.

bLenUidIn: number of relevant bytes of `pUidIn` array. Just 0, 4, 7, 10 are valid values:

0 – means UID is unknown, thus function begins with RequestA. At the end of function complete UID of card should be captured.

4, 7, 10 - Wake up command is performed.

***pUidOut:** pointer to complete UID of activated card.

***pLenUidOut:** length of `pUidOut`. Only values 4, 7, 10 are possible.

***pSak:** pointer to one byte SAK. byte code specifying type of card. MIFARE card type is determined by bits according to table of SAK values in [26].

***pMoreCardsAvailable:** indicates whether one or more cards are in range of PCD field at the same time. But still only one card' UID is captured in one function run.

`PH_ON`: more cards available. Collision occurred.

`PH_OFF`: just one card in PCD field.

returnValues:

`PH_ERR_INVALID_PARAMETER`: `bLenUidIn` does not equal any of 0, 3, 7 or 10.

`PH_ERR_SUCCESS` - operation successful.

Other: depending on implementation and underlaying component.

2.3.5 Anticollision - `phpalI14443p3a_Anticollision()`

This function is responsible for handling just one card during Activation procedure in case more cards are present in the PCD field. Anti-collision routine performs 1 or 3 loops depending on card currently activating, until complete UID acquired. The Anti-collision procedure is mandatory for ISO/IEC 14443A compliant products and all the NXP MIFARE products support the Anti-collision to ISO/IEC14443A.

```
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 structure. Function handles some UID related variables.

bCascadeLevel: number of sub-loop which Anti-collision procedure is currently in. For this parameter there are three legal values in total:

```
#define PHPAL_I14443P3A_CASCADE_LEVEL_1     0x93
#define PHPAL_I14443P3A_CASCADE_LEVEL_2     0x95
#define PHPAL_I14443P3A_CASCADE_LEVEL_3     0x97
```

PCD transmits cascade level byte part of anti-collision command to PICC. If value differs from these three values, command is invalid.

Cards owning 4 byte long UID, run only. After third cascade complete, UID must be known for any UID length case.

***pUidIn:** is pointer to UID of card.

bNvbUidIn: is number of valid bits UID of the card currently processed by Anti-collision procedure. Actually, this variable carries two information encoded within. MSB four bytes keeps information about number of valid bytes and lower foursome of bytes represents number of valid bits for currently processed UID.

***pUidOut:** is pointer to array, where updated UID of card will be load after function finishes successfully. The array is increased in amount of bytes depending on a current Anticollision phase of the Anti-collision represented by `bCascadeLevel`. During operation of

this function the first UID byte can equal PHPAL_I14443P3A_CASCADE_TAG, which indicates, that UID not complete yet, therefore next anti-collision loop required.

***pNvbUidOut:** is length of array of UID. It specifies how many bytes of UID are currently relevant.

returnValues:

PH_ERR_INVALID_PARAMETER – invalid cascade level or invalid bNvbUidIn.

PH_ERR_PROTOCOL_ERROR - invalid response received.

PH_ERR_SUCCESS - operation successful.

Other: depending on implementation and underlaying component.

2.3.6 Selection - phpalI14443p3a_Select()

This function provides Selection - a partial operation of Activation procedure. This function and phpalI14443p3a_Anticollision() are both together implemented within the phpalI14443p3a_ActivateCard() function. Cooperation of this couple of functions results in obtaining of compete UID in a way satisfying ISO/IEC14443p3.

After this Selection executes successfully, card specified by UID responds by SAK byte code which refers to the card type (MIFARE Classis 1k, 4K, MIFARE DESFire etc.). If the PCD reader requested to be able to handle more card types, particular MIFARE card type can be recognized by SAK one byte code. Further task for developer is to design branching of software to run correct routines compliant to type (SAK byte) of the card currently selected.

Note: Use this function just for purpose of finding out SAK byte, which determines type of card.

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

***pDataParams:** pointer to PAL data parameter structure. Selection function sets pDataParam->UidComplete flag to 1, if acquiring UID was completed successfully.

bCascadeLevel: refers to degree which anti-collision procedure is currently in. For this parameter there are three legal values in total:

```
#define PHPAL_I14443P3A_CASCADE_LEVEL_1     0x93
#define PHPAL_I14443P3A_CASCADE_LEVEL_2     0x95
#define PHPAL_I14443P3A_CASCADE_LEVEL_3     0x97
```

Cards owning 4 byte long UID, run only one cascade level. After the third cascade level is completed, UID must be obtained for any card - all UID lengths' cases. For further information see ISO/IEC14443p3.

***pUidIn:** UID of card to be selected. This value should not be NULL, due to this function process only relevant even if incomplete input UID.

***pSak:** byte code specifying type of card. MIFARE card type is determined by bits according to table of SAK values in [26]. There is one special case SAK==0x04 which means UID of currently selected card is not complete yet and next cascade of activation loop is necessary to perform. Therefore Selection function is used inside Activation

function, where SAK == 0x04 provokes next cascade level to obtain complete UID by Anti-collision function `phpalI14443p3a_Anticollision()`.

returnValues:

`PH_ERR_PROTOCOL_ERROR:` Disagreement between first byte of UID and SAK. First byte of UID `pUidIn[0] == PHPAL_I14443P3A_CASCADE_TAG` together with `SAK != 0x04` or vice versa.

`PH_ERR_SUCCESS` - operation successful.

Other: depending on implementation and underlaying component.

2.3.7 Exchange - `phpalI14443p3a_Exchange()`

In comparison with PCD CLRC663 native commands performing functions are ISO/IEC14443p3 related functions more complex and exploit different communication principle to achieve correct performing of particular ISO operation. PCD transmits partial command to PICC (if any), afterwards PCD receives eventual reply from PICC. Common issue for all ISO/IEC14443p3A related functions is bidirectional communication (half-duplex base) between PCD and PICC. Exchange function performs transmission and reception respectively, via suitably exploiting native Transceive command and additional register settings ensuring flawless communication.

Note: Although ISO/IEC14443-3A functions directly implement `phhalHw_Exchange()` function, in this section it is described another one (from PAL layer), which implements that function also.

```
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 parameter structure.

wOption: all ISO/IEC14443p3 functions pass default parameter EXCHANGE_DEFAULT

***pTxBuffer:** pointer to data array to be transmitted. Actually, this array contains PICC command defined by byte code and data for PICC.

wTxLength: number of data bytes to be transmitted. If function performing a card command calls this function, then this variable is one greater than length of data to be transmitted, due to length of command itself. Developer does not need to care about this, command related functions handles it automatically.

****ppRxBuffer:** pointer to pointer to received data.

***pRxLength:** pointer to address, where information about received data is passed.

returnValues:

`PH_ERR_SUCCESS` - operation successful.

Other: depending on implementation and underlaying component.

2.3.8 Halt A - `phStatus_t phpalI14443p3a_HaltA()`

After MIFARE card has been activated one option is to get the card into Halt state. The card can be later reactivated by WakeUpA or `phpalI14443p3a_ActivateCard()` function.

```
phStatus_t phpalI14443p3a_HaltA(
```

```
                                    void * pDataParams );                    [In]
```

***pDataParams:** pointer to the PAL layer data parameter structure.

returnValues:

`PH_ERR_SUCCESS`: card has been turned into Halt mode successfully. After Halt command has been transmitted to the card and no subsequent RF signal from card received. Time to wait is by value `PHPAL_I14443P3A_HALT_TIME_US` + `PHPAL_I14443P3A_EXT_TIME_US` in microseconds.

`PH_ERR_PROTOCOL_ERROR`: protocol error occurred. After transmitting Halt command some echo signal has been detected, which is considered that card has not turned Halt mode.

Other: depending on implementation and underlaying component.

2.3.9 Wake up A - `phStatus_t phpalI14443p3a_WakeUpA()`

This function reactivates a card if the card has been pushed to Halt state before. The Request Guard Time (see 6.2.2, ISO/IEC 14443-3:2009(E)) is mandatory and is neither implemented here nor implemented in every HAL layer. Make sure that either the used HAL or the used application does comply to this rule.

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

***pDataParams** pointer to the PAL layer data parameter structure.

***pAtqa** pointer to ATQA. If WakuUpA is successful, then 2 byte ATQA is written to this variable.

returnValues:

`PH_ERR_PROTOCOL_ERROR:` invalid response code received.

`PH_ERR_SUCCESS` - operation successful.

Other: depending on implementation and underlaying component.

2.4 AL: MIFARE Classic commands

This section describes MIFARE Authentication function and data manipulating functions reflecting data manipulating MIFARE commands. You can find further information about the MIFARE Classic commands in [2]. All of them are implemented in the Application Layer of the NXP Reader Library in the folder *NxpRdbLib_PublicRelease/comps/phalMfc.*

2.4.1 Init MIFARE Classic - `phalMfc_Sw_Init()`

This function initiates MIFARE Classic card component.

```
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 AL parameter structure.

wSizeOfDataParams: specifies the size of the data parameter structure. Recommended to pass here `sizeof(phalMfc_Sw_DataParams_t)`.

***pPalMifareDataParams:** pointer to the undelaying PAL parameter structure.

***pKeyStoreDataParams:** pointer to the underlaying KeyStore parameter structure.

returnValues:

`PH_ERR_SUCCESS` - operation successful.

`PH_ERR_INVALID_DATA_PARAMS` – `wSizeOfDataParams` does not agree with defined size of AL MFC component.

2.4.2 MF Authentication - `phalMfc_Authenticate()`

This function loads a key from the key store area of PCD's EEPROM into the Key buffer. Then it performs entire procedure of three pass Authentication to the given EEPROM block of the card. If Authentication passes through all the three phases successfully, then the EEPROM block is fully accessible for data manipulation MIFARE Classic commands: reading, writing, increment, decrement, restore and data transfer. Once access to block is obtained, access to entire section (4 blocks) retains until next attempt to authenticate to any EEPROM block of card. This function integrates two functions: . In case of authentication by "software" defined key - out of PCD key storage area use `phhalHw_Rc663_Cmd_LoadKey()` function (section 2.1.4) to load the key into the Key buffer and subsequently function `phhalHw_Rc663_MfcAuthenticate_Int()` (section 2.1.6) to authenticate.

This function could be divided into two branches:

Software key store branch: unfortunately contains functions with de facto no executable source code.

EEPROM key store: utilizes `phhalHw_Rc663_Cmd_LoadKeyE2` to load a key from EEPROM into crypto unit then executes MIFARE three pass Authentication.

```
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.

`pDataParams->pKeyStoreDataParams == NULL` to run EEPROM key store branch. Otherwise bodyless functions from the Software Key store branch run and authentication has no effect.

bBlockNo: number of block in EEPROM of card to authenticate to. Authentication to block enables access to entire section which the block is part of.

bKeyType: can be either `PHAL_MFC_KEYA` or `PHAL_MFC_KEYB`. Each block can be authenticated by using anyone of these two keys specified for that block.[2]

wKeyNumber: key number from the PCD's EEPROM to be used in authentication. RC663 has key storage area of volume of 1024 bytes. This variable has no special predefined match with block number of the MIFARE card. Developer must to manage storing of the keys responsibly (by reliable system) to avoid later mismatch in authentication.

wKeyVersion: key version to be used in authentication. This is a useless variable, but must be NULL.

***pUid:** pointer to UID of the card to be authenticated. Each card has its own unique identification number.

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

returnValues:

`PH_ERR_AUTH_ERROR`: Authentication procedure fails, if the key currently loaded in the Key buffer of reader does not match with key for given block in card tried to be accessed.

`PH_ERR_INVALID_PARAMETER` – `wKeyVersion` other than NULL.

`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.

`PH_ERR_SUCCESS`: operation successful.

Other: depending on implementation and underlaying component.

2.4.3 ReadValue - `phalMfc_ReadValue()`

This function performs MIFARE Classic Read command first then in addition verifies all the read bytes from the 16 byte receive buffer regarding to data format in Fig 4. Verification is done by `phalMfc_Int_CheckValueBlockFormat()` function.

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

Fig 4. One value block in EEPROM of the MIFARE Classic

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

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

bBlockNo: block number to be read from.

***pValue:** pointer to 4 byte array containing read value from the MIFARE Classic card value block.

***pAddrData:** pointer to one byte containing address read from the MIFARE Classic card value block.

returnValues:

`PH_ERR_PROTOCOL_ERROR` – error when other than 16 bytes read from MIFARE Classic card value block.

- data or address bytes within the 16 byte receive buffer do not satisfy MIFARE Block data rules according to Fig 4.

`PH_ERR_SUCCESS` - operation successful.

Other: depending on implementation and underlaying component.

2.4.4 WriteValue - `phalMfc_WriteValue()`

Firstly, this function creates from 4 byte input value and input address the 16 byte formatted structure according to Fig 4. Then it performs MIFARE Classic Write command of that 16 byte value. Formatting is done automatically by MCU software - `phalMfc_Int_CreateValueBlockFormat()`.

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

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

bBlockNo: block number to be written into.

***pValue:** 4 byte array to be written to the MIFARE Classic card value block. This function converts and includes it to the 16 byte format ready to be written.

bAddrData: one byte array of address data to be written to the MIFARE Classic card value block. This function converts and includes it to the 16 byte format ready to be written.

returnValues:

`PH_ERR_SUCCESS` - operation successful.

Other: depending on implementation and underlaying component.

2.4.5 Increment - `phalMfc_Increment()`

It adds the operand to the value of the addressed block, and stores the result in a volatile

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

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

bBlockNo: block number to be incremented.

***pValue:** pValue[4] containing value (LSB first) to be incremented on the MIFARE(R) card.

returnValues:

`PH_ERR_SUCCESS` - operation successful.

Other: depending on implementation and underlaying component.

2.4.6 Decrement - `phalMfc_Decrement()`

It subtracts the operand from the value of the addressed block, and stores the result in a volatile memory.

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

```
                                    uint8_t bBlockNo,                   [In]
                                    uint8_t * pValue );                 [In]
```

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

bBlockNo: block number to be decremented.

***pValue:** pValue[4] containing value (LSB first) to be decremented on the MIFARE(R) card

returnValues:

`PH_ERR_SUCCESS` - operation successful.

Other: depending on implementation and underlaying component.

2.4.7 Restore - `phalMfc_Restore()`

It copies the value of the addressed block into a volatile memory.

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

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

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

returnValues:

`PH_ERR_SUCCESS` - operation successful.

Other: depending on implementation and underlaying component.

2.4.8 Transfer - `phalMfc_Transfer()`

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

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

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

bBlockNo: block number the transfer buffer shall be transferred to.

returnValues:

`PH_ERR_SUCCESS` - operation successful.

Other: depending on implementation and underlaying component.

2.4.9 IncrementTransfer - `phalMfc_IncrementTransfer()`

This function executes both Increment and Transfer command respectively. Value from source block is copied into volatile memory where it is incremented by given value then stored into destination in EEPROM memory. All the mentioned is executed within the MIFARE Classic card.

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

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

bSrcBlockNo: source block number in MIFARE Classic card EEPROM. Value from this block to be incremented.

bDstBlockNo: destination block number. Incremented value from source block is stored to this block.

***pValue:** 4 byte value that is value from source block incremented by.

returnValues:

`PH_ERR_SUCCESS` - operation successful.

Other: depending on implementation and underlaying component.

2.4.10 DecrementTransfer - `phalMfc_DecrementTransfer()`

This function executes both Decrement and Transfer command respectively. Value from source block is copied into volatile memory where it is decremented by given value then stored into destination in EEPROM memory. All the mentioned is executed within the MIFARE Classic card.

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

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

bSrcBlockNo: block number to be decremented.

bDstBlockNo: destination block number. Decremented value from source block is stored to this block.

***pValue:** 4 byte value that is value from source block decremented by.

returnValues:

`PH_ERR_SUCCESS` - operation successful.

Other: depending on implementation and underlaying component.

2.4.11 RestoreTransfer - `phalMfc_RestoreTransfer()`

This function executes both Decrement and Transfer command respectively. Value from source block is copied into volatile memory where it is incremented by given value then stored into destination in EEPROM memory. All the mentioned is executed within the MIFARE Classic card.

```
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 decremented.

bDstBlockNo: block number to be transferred to.

returnValues:

`PH_ERR_SUCCESS` - operation successful.

2.4.12 PersonalizeUID - phalMfc_PersonalizeUid()

This function configures UID to given personalization option, which has direct impact on behavior during Anti-collision, Selection and. The default configuration at delivery enables the ISO/IEC 14443-3 compliant anti-collision and selection. The execution of this command requires an authentication to sector 0. Once this command has been issued and accepted by the PICC, the configuration is automatically locked.

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

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

bUidType: UID type.

PHAL_MFC_UID_TYPE_UIDF0 - anti-collision and selection with the double size UID according to ISO/IEC14443-3

#PHAL_MFC_UID_TYPE_UIDF1 - anti-collision 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 - anti-collision and selection with a single size random ID according to ISO/IEC

14443-3

PHAL_MFC_UID_TYPE_UIDF3 - anti-collision and selection with a single size NUID according to ISO/IEC 14443-3 where the NUID is calculated out of the 7-byte UID

returnValues:

PH_ERR_SUCCESS - operation successful.

Other: depending on implementation and underlaying component.

2.5 AL: MIFARE Ultralight commands

You can find further information about MIFARE Ultralight commands in [3]. All of them are implemented in the Application Layer of the NXP Reader Library in the folder *NxpRdbLib_PublicRelease/comps/phalMful.*

2.5.1 Init MIFARE Ultralight - phalMfu_Sw_Init()

This function initiates MIFARE Ultralight card 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 MFUL AL layer parameter structure.

wSizeOfDataParams: specifies the size data parameter structure. Recommended to pass here sizeof(phalMful_Sw_DataParams_t).

***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.

`PH_ERR_SUCCESS` - operation successful.

`PH_ERR_INVALID_DATA_PARAMS` – `wSizeOfDataParams` does not agree with defined size of MFUL component.

2.5.2 Read - `phalMful_Read`

This function performs MIFARE Ultralight Read command.

```
phStatus_t phalMful_Read(
                                void * pDataParams,                 [In]
                                uint8_t bAddress,                   [In]
                                uint8_t * pData );                  [Out]
```

***pDataParams:** pointer to the MIFARE Ultralight AL layer data parameter structure.

bAddress: addres on the card to read from. In range from 00h to FFh. If out of range of valid address, MFUL returns NAK.

***pData:** pointer to 16 byte data array containing data read from the Ultralight card.

returnValues:

`PH_ERR_PROTOCOL_ERROR` – number of received data differs from 16 bytes.

`PH_ERR_SUCCESS` - operation successful.

Other Depending on implementation and underlaying component.

2.5.3 Write - `phalMful_Write()`

This function performs MIFARE Ultralight Write command – writes 4 bytes to given memory page.

```
phStatus_t phalMful_Write(
                                void * pDataParams,                 [In]
                                uint8_t bAddress,                   [In]
                                uint8_t * pData );                  [In]
```

***pDataParams:** pointer the MIFARE Ultralight AL layer data parameter structure.

bAddress: address on the card to write to. It can be the lock bytes in page 02h, the one time only programmable bytes in page 03h or the data bytes in pages 04h to 0Fh.

***pData:** pointer to 4 byte data array containing data read from the Ultralight card.

returnValues:

`PH_ERR_SUCCESS` - operation successful.

Other Depending on implementation and underlaying component.

2.5.4 CompatibilityWrite - `phalMful_CompatibilityWrite()`

This function performs MIFARE Ultralight Compatibility-Write command.

```
phStatus_t phalMful_CompatibilityWrite(
                                void * pDataParams,                 [In]
                                uint8_t bAddress,                   [In]
                                uint8_t * pData );                  [Out]
```

***pDataParams:** pointer to the MIFARE Ultralight AL layer data parameter structure.

bAddress: Address on Picc to read from.

***pData:** pointer to 16 byte data array containing data read from the Ultralight card.

returnValues:

PH_ERR_SUCCESS - operation successful.

Other Depending on implementation and underlaying component.

3. Sample code

In this chapter we introduce a sample code performing basic handling with MIFARE Classic card. There are only fragments of the complete original source code presented with focus on most important parts. Main purpose of example code is to clarify functions and their aim in context of application, what are particular functions responsible for, prerequisites for some functions necessary to do or call.

```
104  int main (void)
105  {
110        phStatus_t status;
111        uint8_t bHalBufferReader[0x40];
112        uint8_t bBufferReader[0x60];
113        uint8_t bSak[1];
114        uint8_t bUid[10];
115        uint8_t bMoreCardsAvailable;
116        uint8_t bLength;
117        void *pHal;
```

Before any operations with card and hardware configuration either, declare data parameter structures for all the layers. When particular data parameter structure will be initiated later, underlaying structure must already exist. Particular layers and their mutual relationships and hierarchy are described in section 1.1.1.

```
122        phbalReg_Stub_DataParams_t balReader;
123        phhalHw_Rc663_DataParams_t halReader;
124        phpalI14443p3a_Sw_DataParams_t I14443p3a;
125        phpalI14443p4_Sw_DataParams_t I14443p4;
126        phpalMifare_Sw_DataParams_t palMifare;
127
128        phKeyStore_Rc663_DataParams_t Rc663keyStore;
129        phalMfc_Sw_DataParams_t alMfc;
```

Set interface between the MCU and the reader chip for correct communication. This includes bus type selection and MCU's PIN configuration for that purpose.

```
142        Set_Interface_Link();
```

Perform reset of reader chip. One of the MCU GPIO pin is connected to the CLRC663 reader chip. MCU sets reset signal on this pin.

```
145        Reset_RC663_device();
```

Initialize data parameter structure for the BAL layer.

```
150        phbalReg_Stub_Init(&balReader, sizeof(phbalReg_Stub_DataParams_t));
```

Initialize data parameter structure for the HAL layer. The `phhalHw_Rc663_DataParams_t` structure is related to the card reader CLRC663 and stores some software configuration parameters which MCU controls the reader through. This function initiates PCD card reader. The third passed parameter is address of existing BAL data parameter structure. Initiate bus connection type separately (line 164 or 169), because `phhalHw_Rc663_Init()` function (see section 2.2.2) sets RS232 interface by default, although real hardware bus selection has been done in line 142.

```
153       status = phhalHw_Rc663_Init(
154            &halReader,
155            sizeof(phhalHw_Rc663_DataParams_t),
156            &balReader,
157            0,
158            bHalBufferReader,
159            sizeof(bHalBufferReader),
160            bHalBufferReader,
161            sizeof(bHalBufferReader));
162  #ifdef I2C_USED
163       /* Set the parameter to use the I2C interface */
164       halReader.bBalConnectionType = PHHAL_HW_BAL_CONNECTION_I2C;
165  #endif
166
167  #ifdef SPI_USED
168       /* Set the parameter to use the SPI interface */
169       halReader.bBalConnectionType = PHHAL_HW_BAL_CONNECTION_SPI;
170  #endif
```

Pointer to the HAL layer data parameter structure will be needed by following initiations of upper layer data parameter structures.

```
173  pHal = &halReader;
```

Initialize the 14443-3A PAL (Protocol Abstraction Layer) component (see section 2.3.2).

```
181  PH_CHECK_SUCCESS_FCT(status, phpalI14443p3a_Sw_Init(&I14443p3a,
182                       sizeof(phpalI14443p3a_Sw_DataParams_t), pHal));
```

This initialization is redundant, since MIFARE Classic is incompliant with the ISO/IEC14443-4.

```
185  PH_CHECK_SUCCESS_FCT(status, phpalI14443p4_Sw_Init(&I14443p4,
186                       sizeof(phpalI14443p4_Sw_DataParams_t), pHal, &I14443p4));
```

Initialize the MIFARE PAL component, due to it is implemented by `phalMfc_Authenticate()` function and PAL ISO14443-3A.

```
189  PH_CHECK_SUCCESS_FCT(status, phpalMifare_Sw_Init(&palMifare,
190                       sizeof(phpalMifare_Sw_DataParams_t), pHal, &I14443p4));
```

Initialize the KeyStore component. We will need it to store key into EEPROM key storage area.

```
193   PH_CHECK_SUCCESS_FCT(status, phKeyStore_Rc663_Init(&Rc663keyStore,
194                         sizeof(phKeyStore_Rc663_DataParams_t), pHal));
```

Initialize the MIFARE Classic AL component (see section 2.4.1) - set NULL because the keys are loaded in EEPROM by the function `phKeyStore_SetKey()` (line 260).

```
199         PH_CHECK_SUCCESS_FCT(status, phalMfc_Sw_Init(&alMfc,
200                     sizeof(phalMfc_Sw_DataParams_t), &palMifare, NULL));
```

The SoftReset (see section 2.1.15) only resets the CLRC663 to EEPROM configuration.

```
207         PH_CHECK_SUCCESS_FCT(status, phhalHw_Rc663_Cmd_SoftReset(pHal));
```

Just for sure, reset CLRC663 field.

```
218         PH_CHECK_SUCCESS_FCT(status, phhalHw_FieldReset(pHal));
```

Before any communication attempt with MIFARE Classic, set up reader chip to ISO/IEC14443p3A communication protocol, which is MIFARE Classic compliant protocol (see section 2.1.13). Even we don't know if in the PCD's field any card at all.

```
222   PH_CHECK_SUCCESS_FCT(status, phhalHw_ApplyProtocolSettings(pHal,
223                     PHHAL_HW_CARDTYPE_ISO14443A));
```

From this time on, reader chip is prepared to communicate with ISO/IEC14443p3A compliant cards.

It might seem like program ignores the RequestA operation (if there is any card in the range of field) and skips directly to Activation phase, that would be violating of ISO14443p3A. But `phpalI14443p3a_ActivateCard()` includes RequestA also. Section 2.3.4 provides quite precious description of this function.

```
227         status = phpalI14443p3a_ActivateCard(&I14443p3a, NULL, 0x00, bUid,
                &bLength, bSak, &bMoreCardsAvailable);
```

If the is a card in the field and the activation is successful, since last four passed parameters are pointers. Anti-collision procedure is also included here, so even if there are more cards in the PCD's field, only one card is selected – one UID captured.

If there is a card in the field of PCD reader, previous Activate function runs successfully and obtains SAK byte from the card also. That is byte code saying about the type of the card.

```
232   if (PH_ERR_SUCCESS == status)
233         {
237         if (0x20 == (*bSak & 0x20))
238         {
239               debug_printf_msg("ISO-4 compliant card detected");
240         }
```

From this point nearly to the end is branch for MIFARE Classic card detected.

```
242         else if (0x08 == (*bSak & 0x08))
243         {
244             debug_printf_msg("Mifare Classic card detected");
```

Memory of MIFARE Classic is divided into 16 sectors and each sector contains 4 blocks of 16 byte length. Firstly we need to get access to sector by authentication with key. The key must agree with key stored in the card for particular sector.

Before authentication we must store the key into key storage area in the EEPROM of the reader chip. The key is defined array Key[6] – we use a new card here so the key is 6 bytes of 0xFF.

```
260             PH_CHECK_SUCCESS_FCT(status, phKeyStore_SetKey(&Rc663keyStore, 0, 0,
261                                     PH_KEYSTORE_KEY_TYPE_MIFARE, &Key[0], 0));
```

Now the 6 byte key is in key buffer of the crypto unit and retains until next key is buffered by `phKeyStore_SetKey()`.

Performing authentication function loads from the key storage EEPROM area according blocknumber passed as parameter to authentication function. See section 2.4.2 to understand parameters of this function.

```
265             status = phalMfc_Authenticate(&alMfc, 0, PHHAL_HW_MFC_KEYA, 0, 0,
                                                 bUid, bLength);
```

Successful authentication to block 0 gives access to all the other blocks of sector 0.

Although we already have captured UID since Authentication, here is an alternative way: UID of the card is read from block 0 in sector 0 by Read MIFARE native command. But this assumes sector 0 has already been authenticated.

```
280             PH_CHECK_SUCCESS_FCT(status, phalMfc_Read(&alMfc, 0,
281                                     &bBufferReader[0]));
```

Now let's authenticate through block 6 to entire sector 1 – block 4, 5, 6, 7.

```
313             PH_CHECK_SUCCESS_FCT(status, phalMfc_Authenticate(&alMfc, 6,
314                                     PHHAL_HW_MFC_KEYA, 0, 0, bUid, bLength));
```

Block 4 is accessible, so we can write data into. To be correct, 4 bytes should be written to card EEPROM via `phhalMfc_ WriteValue()`, which ensures correct form of data to be written. That function is described in section 2.4.4.

```
320             PH_CHECK_SUCCESS_FCT(status, phalMfc_Write(&alMfc, 4, bBufferReader));
```

Clear the read buffer to forget written data and later read data can be load buffer and verified.

```
324             memset(bBufferReader, '\0', 0x60);
```

Read just written 16 byte data and copy them into `bReaderBuffer`. Since we have written data into card EEPROM , we read the data without checking the format as well.

```
330             PH_CHECK_SUCCESS_FCT(status, phalMfc_Read(&alMfc, 4, bBufferReader));
```

Last possibility of SAK card type remains. If no MIFARE Classis, nor ISOp14443p4 compliant and activation was successful it must be Ultralight card.

```
343         else
344         {
345             debug_printf_msg("Mifare UltraLight card detected");
346         }
```

Activation failed due to no card in the field detected.

```
349         else
350         {
351             debug_printf_msg("No card detected");
352         }
```

4. Appendix

4.1 Error codes

The NXP Reader Library return defined return values – error codes. This approach should help the developer with error identification if it occurs and thus make easier way how to fix bugs during software development. Each error code is type of 16 uint and includes two pieces of information. Upper byte identifies the layer which an error has occurred on and lower byte is the byte code of particular error. All the error codes are defined in *ph_Status.h* file in the folder *NxpRdLib_PublicRelease/types*.

5. Abbreviations

Table 2. Abbreviations

Acronym	Description
AL	Application Layer
ACK	ACKnowledge
ATQA	Answer To reQuest, type A
BAL	Bus Abstraction Layer
CRC	Cyclic Redundancy Check
EEPROM	Electrically Erasable Programmable Read-Only Memory
GPIO	General Purpose Input Output
HAL	Hardware Abstraction Layer
I2C	Inter-Interchanged Circuit
IC	Integrated Circuit
IRQ	Interrupt ReQuest
LPCD	Low Power Card Detection
MCU	MicroController Unit
MF	MIFARE
MFC	MIFARE Classic
MFUL	MIFARE UltraLight
MKA	MIFARE key area (same as EEMPROM Key Storage Area)
NAK	Negative AcKnowledge
NFC	Near Field Communication
PAL	Protocol Abstraction Layer
PCD	Proximity Coupling Device (Contactless Reader)
PICC	Proximity Integrated Circuit Card (Contactless Card)
REQA	REQuest command, type A
SAK	Select AcKnowledge, type A
SAM	Secure Access Module
SPI	Serial Peripheral Interface
UID	Unique IDentifier

6. References

[1] **Link to the NXP Reader Library**
http://www.nxp.com/products/identification_and_security/reader_ics/contactless_reader_systems/series/CLRC663.html#documentation

[2] **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

[3] **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

[4] **Data Sheet** - MIFARE Plus; MF1PLUSx0y1, Mainstream contactless smart card IC for fast and easy solution development, BU-ID Doc. No. 1635**, available on
http://www.nxp.com/documents/short_data_sheet/MF1PLUSX0Y1_SDS.pdf

[5] **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

[6] **Data Sheet** - ISO/IEC Standard - ISO/IEC15693 Identification cards - Contactless integrated circuit(s) cards - Vicinity cards, available on
http://www.nxp.com/redirect/waazaa.org/download/fcd-15693-3

[7] **Data Sheet** - I CODE SLI; Standard Label IC SL2 ICS20, available on
http://www.nxp.com/documents/data_sheet/SL058030.pdf

[8] **Data Sheet** - ICODE ILT, smart label IC; will be available on NXP Web

[9] **ISO/IEC Standard** - ISO/IEC14443 Identification cards - Contactless integrated circuit cards - Proximity cards

[10] **Data Sheet** - ISO/IEC Standard - ISO 18000-3 Information technology AIDC techniques - RFID for item management - Air interface, Part 3 - Parameters for air interface communications at 13.56 MHz, M1 stands for Mode 1, M3 stands for Mode 3

[11] **Data Sheet** - JIS Standard JIS X 6319 Specification of implementation for integrated circuit(s) cards - Part 4: High Speed proximity cards

[12] **Data Sheet** - ISO/IEC Standard - ISO 18092 Information technology - Telecommunications and information exchange between systems - Near Field Communication- Interface and Protocol (NFCIP-1)

[13] **Data sheet** - MFRC523; Contactless reader IC, BU-ID Doc. No. 1152**, available on http://www.nxp.com/documents/data_sheet/MFRC523.pdf

[14] **Data sheet** - CLRC632; Multiple protocol contactless reader IC (MIFARE/I-CODE1), BU-ID Doc. No. 0739**, available on
http://www.nxp.com/documents/data_sheet/CLRC632.pdf

[15] **Data sheet** - CLRC663; Contactless reader IC, BU-ID Doc. No. 1711**, available on http://www.nxp.com/documents/data_sheet/CLRC663.pdf

[16] **Data sheet** - MFRC522; Contactless reader IC, BU-ID Doc. No. 1121**, available on http://www.nxp.com/documents/data_sheet/MFRC522.pdf

1. [1] ** … BU ID document version number

[17] **Data sheet** – PN512; Transmission module, BU-ID Doc. No. 1112**, available on http://www.nxp.com/documents/data_sheet/PN512.pdf

[18] **Data sheet** – MFRC500; Highly Integrated ISO/IEC 14443 A Reader IC, BU-ID Doc. No. 0480**, available on http://www.nxp.com/documents/data_sheet/MFRC500.pdf

[19] **Data sheet** – MFRC530; ISO/IEC 14443 A Reader IC, BU-ID Doc. No. 0574**, available on http://www.nxp.com/documents/data_sheet/MFRC530.pdf

[20] **Data sheet** – MFRC531; ISO/IEC 14443 reader IC, BU-ID Doc. No. 0566**, available on http://www.nxp.com/documents/data_sheet/MFRC531.pdf

[21] **Data sheet** – MFRC631; Contactless reader IC, BU-ID Doc. No. 2274**, available on http://www.nxp.com/documents/data_sheet/MFRC631.pdf

[22] **Data sheet** – MFRC630; Contactless reader IC, BU-ID Doc. No. 2275**, available on http://www.nxp.com/documents/data_sheet/MFRC630.pdf

[23] **Data sheet** – SLRC610; Contactless reader IC, BU-ID Doc. No. 2276**, available on http://www.nxp.com/documents/data_sheet/SLRC610.pdf

[24] **Data Sheet** - RD701; Pegoda Contactless Smart Card Reader, BU-ID Doc. No. 0992**, available on http://www.nxp.com/documents/data_sheet/PE099231.pdf

[25] **Data Sheet** MFEV710, Pegoda EV710, available on http://www.nxp.com/documents/short_data_sheet/MFEV710_SDS.pdf

[26] **Application note -** AN10833 MIFARE Type Identification Procedure

[27] **Application note** – Quick Start Up GuideRC663 Blueboard, available on http://www.nxp.com/demoboard/CLEV663B.html

7. Legal information

7.1 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.

7.2 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.

7.3 Licenses

Purchase of NXP ICs with NFC technology

Purchase of an NXP Semiconductors IC that complies with one of the Near Field Communication (NFC) standards ISO/IEC 18092 and ISO/IEC 21481 does not convey an implied license under any patent right infringed by implementation of any of those standards.

Purchase of NXP ICs with ISO/IEC 14443 type B functionality

This NXP Semiconductors IC is ISO/IEC 14443 Type B software enabled and is licensed under Innovatron's Contactless Card patents license for ISO/IEC 14443 B.

The license includes the right to use the IC in systems and/or end-user equipment.

RATP/Innovatron Technology

7.4 Trademarks

Notice: All referenced brands, product names, service names and trademarks are property of their respective owners.

MIFARE — is a trademark of NXP B.V.

MIFARE Ultralight — is a trademark of NXP B.V.

8. Contents

1. Introduction 3
1.1 NXP libraries comparison 3
1.1.1 Layer Structure of the NXP Reader Library 4
1.1.1.1 Application Layer 5
1.1.1.2 Protocol Abstraction Layer 6
1.1.1.3 Hardware abstraction layer 7
1.1.1.4 Bus Abstraction Layer 8
1.1.1.5 Common Layer 8
1.1.1.6 Data structures of the layers 8
2. Exemplary explanation of the library functions 10
2.1 HAL: CLRC663 commands 10
2.1.1 General 10
2.1.2 Idle command 10
2.1.3 LPCD Low Power Card Detection - `phhalHw_Rc663_Cmd_Lpcd()` 10
2.1.4 LoadKey - `phStatus_t phhalHw_Rc663_Cmd_LoadKey()` 11
2.1.5 LoadKeyE2 - `phhalHw_Rc663_Cmd_LoadKeyE2()` 11
2.1.6 MFAuthent - `phhalHw_Rc663_MfcAuthenticate_Int()` 12
2.1.7 Receive - `phhalHw_Rc663_Cmd_Receive()` 13
2.1.8 Transmit - `phhalHw_Rc663_Cmd_Transmit()` 14
2.1.9 WriteE2 - `phhalHw_Rc663_Cmd_WriteE2` 14
2.1.10 WriteE2 Page - `phhalHw_Rc663_Cmd_WriteE2Page` 15
2.1.11 ReadE2 - `phhalHw_Rc663_Cmd_ReadE2()` 15
2.1.12 LoadReg command - `phhalHw_Rc663_Cmd_LoadReg()` 16
2.1.13 LoadProtocol - `phhalHw_Rc663_ApplyProtocolSettings()` 17
2.1.14 StoreKeyE2 - `phKeyStore_SetKey()` 18
2.1.15 SoftReset - `phhalHw_Rc663_Cmd_SoftReset()` 19
2.2 HAL: Additional description of HAL layer 19
2.2.1 CLRC663 HAL layer data parameter structure 19
2.2.2 Initialization HAL - `phhalHw_Rc663_Init()` 20
2.2.3 SetConfig - `phhalHw_SetConfig()` 21
2.2.4 GetConfig - `phhalHw_GetConfig()` 21
2.2.5 ReadRegister - `phhalHw_ReadRegister()` 21
2.2.6 WriteRegister - `phhalHw_WriteRegister()` 22
2.2.7 FieldOn - `phhalHw_FieldOn()` 22
2.2.8 FieldOff - `phhalHw_FieldOff()` 22
2.2.9 FieldReset - `phhalHw_FieldReset()` 23
2.2.10 Wait - `phhalHw_Wait()` 23
2.3 PAL: ISO/IEC14443-3A 24
2.3.1 ISO14443A part 3 24
2.3.2 Init ISO14443-3A - `phpalI14443p3a_Sw_Init()` 24
2.3.3 Request A - `phStatus_t phpalI14443p3a_RequestA()` 24
2.3.4 Activate Card - `phpalI14443p3a_ActivateCard()` 25
2.3.5 Anticollision - `phpalI14443p3a_Anticollision()` 26
2.3.6 Selection - `phpalI14443p3a_Select()` 27
2.3.7 Exchange - `phpalI14443p3a_Exchange()` 28
2.3.8 Halt A - `phStatus_t phpalI14443p3a_HaltA()` 28
2.3.9 Wake up A - `phStatus_t phpalI14443p3a_WakeUpA()` 29
2.4 AL: MIFARE Classic commands 29
2.4.1 Init MIFARE Classic - `phalMfc_Sw_Init()` 29
2.4.2 MF Authentication - `phalMfc_Authenticate()` 30
2.4.3 ReadValue - `phalMfc_ReadValue()` 31
2.4.4 WriteValue - `phalMfc_WriteValue()` 32
2.4.5 Increment - `phalMfc_Increment()` 32
2.4.6 Decrement - `phalMfc_Decrement()` 32
2.4.7 Restore - `phalMfc_Restore()` 33
2.4.8 Transfer - `phalMfc_Transfer()` 33
2.4.9 IncrementTransfer - `phalMfc_IncrementTransfer()` 33
2.4.10 DecrementTransfer - `phalMfc_DecrementTransfer()` 34
2.4.11 RestoreTransfer - `phalMfc_RestoreTransfer()` 34
2.4.12 PersonalizeUID - `phalMfc_PersonalizeUid()` 35
2.5 AL: MIFARE Ultralight commands 35
2.5.1 Init MIFARE Ultralight - `phalMfu_Sw_Init()` 35
2.5.2 Read - `phalMful_Read` 36
2.5.3 Write - `phalMful_Write()` 36
2.5.4 CompatibilityWrite - `phalMful_CompatibilityWrite()` 36
3. Sample code 38
4. Appendix 43
4.1 Error codes 43
5. Abbreviations 43
6. References 44
7. Legal information 46
7.1 Definitions 46
7.2 Disclaimers 46
7.3 Licenses 46
7.4 Trademarks 46
8. Contents 47

For more information, please visit: http://www.nxp.com
For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 24 July 2013
257412
Document identifier: UM10663