UM10721

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

Rev. 1.2 — 5 August 2018   **User manual**
270112   **COMPANY PUBLIC**

Revision history

Rev	Date	Description
1.2	20180805	Broken links in section References updated
1.1	20130724	Change of descriptive title
1.0	20130613	First release

Contact information

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

1. NXP Reader Libraries comparison

NXP Reader Libraries are C written software packages aimed to be used in either embedded or desktop applications for MCUs or PCs both enhanced by a contactless reader. Depending on particular library the MCU or PC gets capability to handle various RF cards, NFC tags and NFC Peers.

There are three NXP Reader libraries. Although altogether have some common functionalities specially for handling the most common MIFARE product-based cards, each of them includes special enhancement for advanced contactless communication.

This document is focused on description of the NXP Reader Library NFC Peer to Peer, thus from section 2 deals with explanation of this Library. In the section 4 there are parts of sample code performing SNEP client using the functions of the NXP P2P Library.

1.1.1 NXP Reader Library Public

The basic NXP Reader Library [1][1] is dedicated for card readers. It supports multiple RF protocols compliant to many contactless cards, in addition several common RF cards are supported. Those cards may be directly handled using library APIs. Since there are many bus interfaces and contactless readers supported the Reader Library may be used either in desktop (PC with Pegoda Reader) or embedded (MCU with reader chip board) application. This NXP Reader Library Public does not support SAM unit or the NFC Peer to Peer communication.
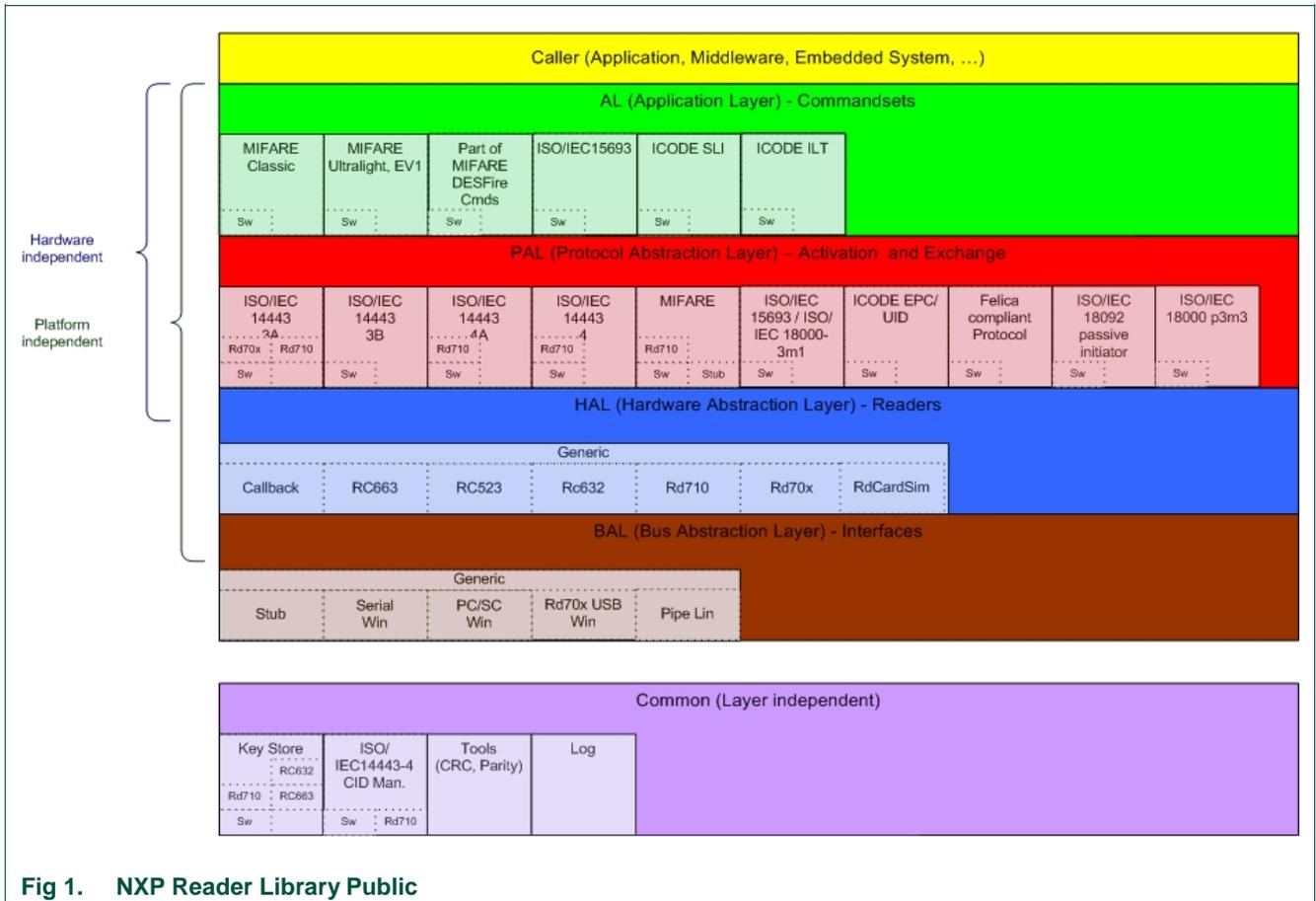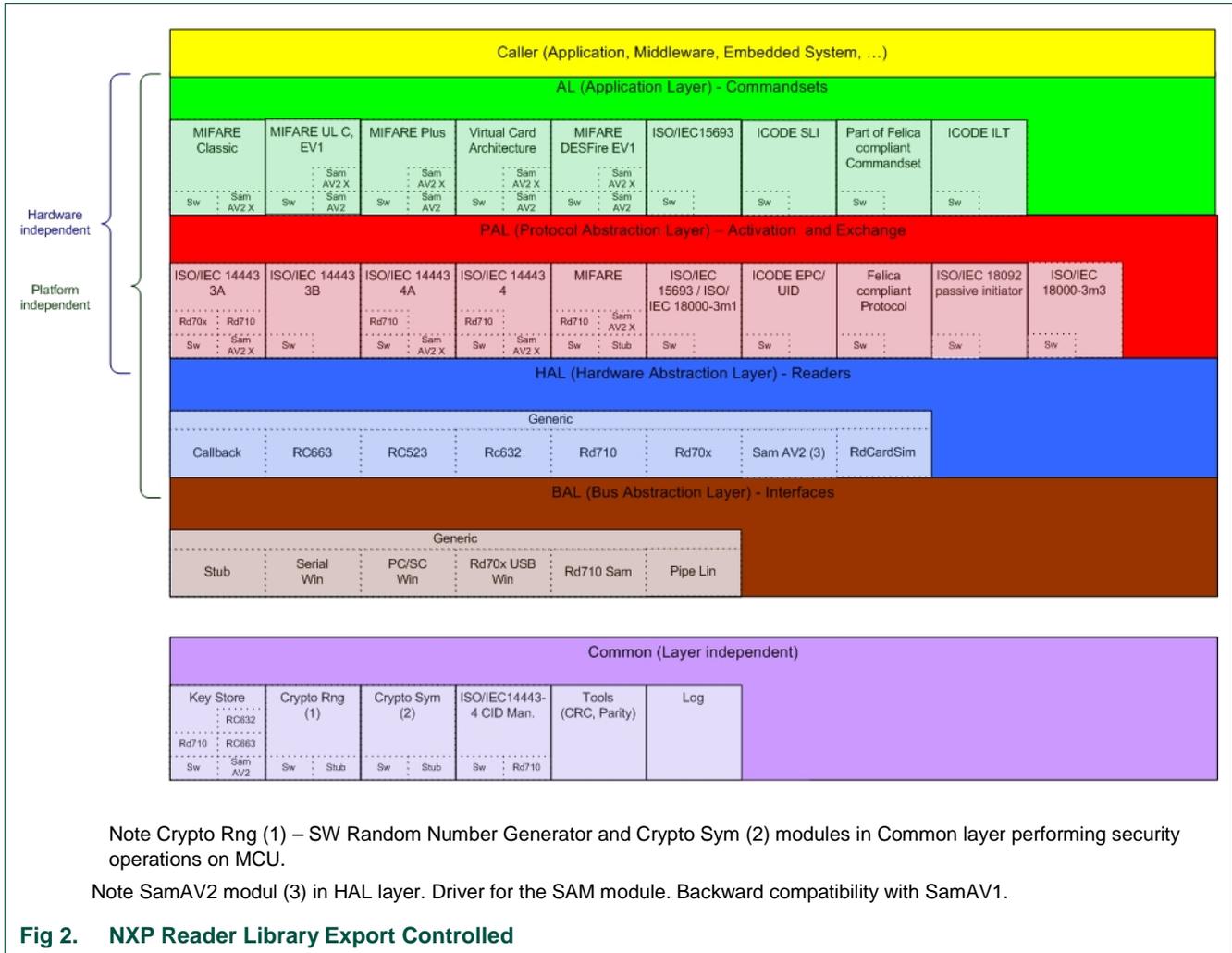

Fig 1. NXP Reader Library Public

UM10721

User manual **Rev. 1.2 — 5 August 2018** **3 of 77**
COMPANY PUBLIC 270112

1.1.2 NXP Reader Library Export controlled

Comprehensive software API enables the customer 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.

Note Crypto Rng (1) – SW Random Number Generator and Crypto Sym (2) modules in Common layer performing security operations on MCU.

Note SamAV2 modul (3) in HAL layer. Driver for the SAM module. Backward compatibility with SamAV1.

Fig 2. NXP Reader Library Export Controlled

UM10721

© NXP B.V. 2018. All rights reserved.

User manual **Rev. 1.2 — 5 August 2018** **4 of 77**
COMPANY PUBLIC **270112**

2. General information about NXP Reader Library P2P

The main purpose and difference from the other NXP Reader Libraries is software extension performing NFC P2P that is compliant with P2P defined by the NFC Forum.

The compatibility with NFC P2P is due to the P2P Library structure, which designed in accordance with P2P part of Protocol Stack defined by NFC forum.

An embedded application implementing the P2P Library is capable to communicate with another NFC P2P device (compliant to NFC Forum P2P concept), including connection establishment and maintenance, data exchange in NDEF, up to the correct connection cancelation. Apart from the P2P functionality it provides tools to create applications for handling some MIFARE product-based cards.

The Library is designed for the particular hardware platform: **NXP LPC1227** board either connected to **PNEV512 Blueboard v1.4** or **CLRC663 Blueboard v2.1**. The Library enables the hardware to be in a role of NFC P2P **Passive Initiator**. See section 2.2.3 for further hardware restrictions.
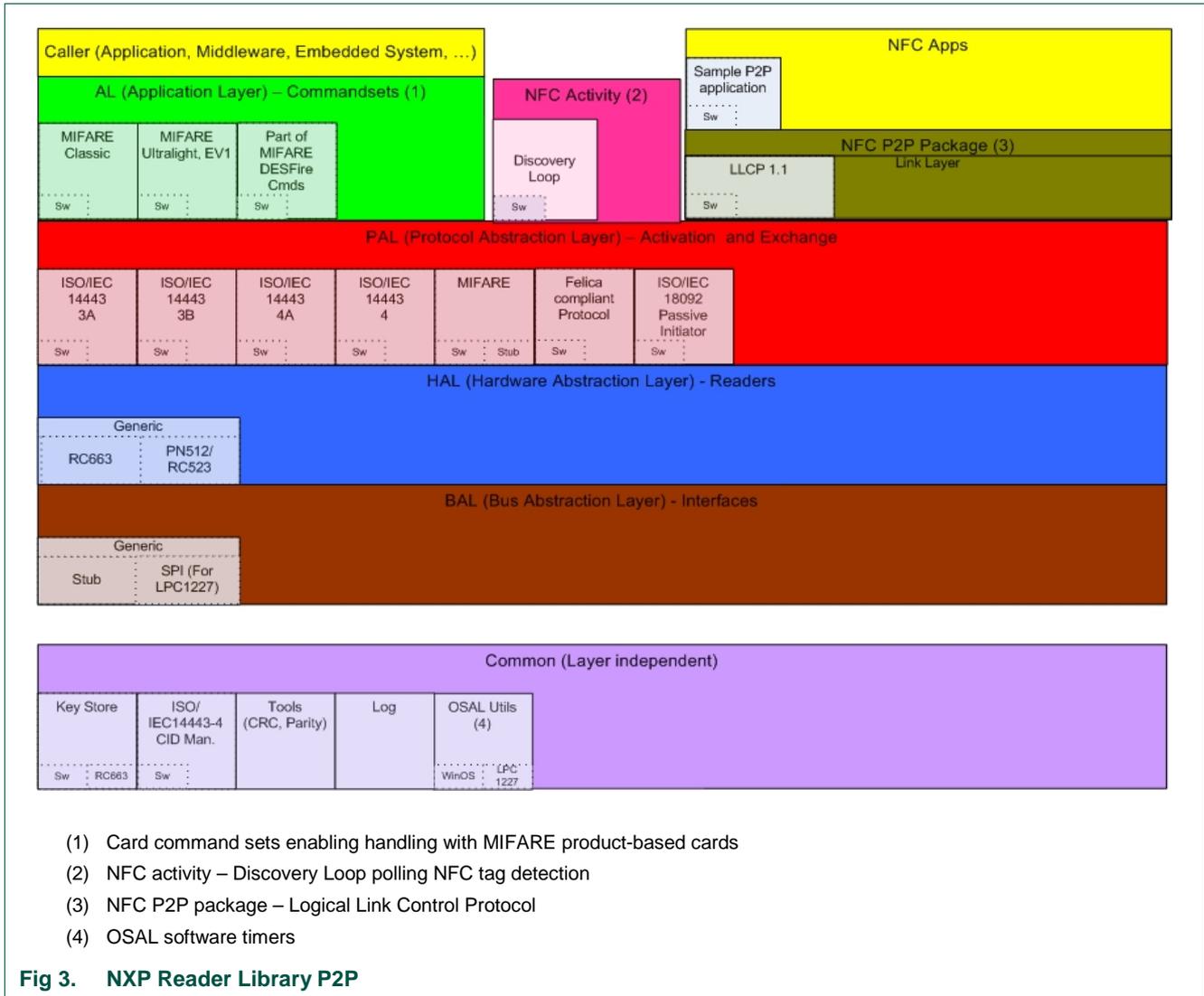
2.1.1 Document structure

First of all, we introduce general structure of the P2P Library dividing it vertically into layers and further dividing particular layer into modules horizontally (see Fig 3). Particular modules imply compatibility of the Library with reader chips, compliance with protocols and ability to communicate with various PICCs.

The document provides overview of implementation Logical Link Control Protocol and ISO18092 protocol which in accordance with NFC Forum are responsible for performance of NFC P2P. The LLCP module description is emphasized in section 3.2 due to it is the uppermost layer and its APIs are designed for usage in developer's embedded application. The ISO18092 module is described more sketchily just to get a complete image of NFC P2P Protocol Stack within the P2P Library.

After those layers described, in the section 4 there is a simple SNEP client application which performs sending NDEF message to a SNEP server (NFC mobile device with android platform). The SNEP client implements functions of the LLCP layer described in the section 3.2 to present usage of the library API functions. The SNEP client simply sends NDEF message to android (SNEP server).

2.2 Layer Structure of the NXP Reader Library

The MCU implementing the Library is able to utilize several types of reader chips to handle any of few types of MIFARE product-based cards and NFC peer to peer communication. To satisfy such versatile usage, architecture of the Library was designed with multi layered structure (see Fig 3). Each layer is independent from the other layers.

(1) Card command sets enabling handling with MIFARE product-based cards

(2) NFC activity – Discovery Loop polling NFC tag detection

(3) NFC P2P package – Logical Link Control Protocol

(4) OSAL software timers

Fig 3.   NXP Reader Library P2P

As seen on Fig 3 vertical structure of the NXP Reader Library is classified into following layers:

- API layer
  - ➤ MIFARE product-based cards command set
  - ➤ NFC Activity
  - ➤ NFC P2P package
    - Logical Link Control Protocol (LLCP)
- Protocol abstraction layer (PAL)
- Hardware abstraction layer (HAL)
- Bus abstraction layer (BAL)
- Common layer
  - ➤ Software timers and timer handlers

2.2.1 API layer

The uppermost API layer is divided into three individual and independent modules (taking place on one layer, vertically equal). Each provides different services which are designed like APIs to be implemented into customer's application (yellow).

2.2.1.1 Card command sets

This module ((1) in Fig 3) provides software implementation of certain commands of MIFARE product-based cards [4][5][6]. Commands enable to access memory and execute certain operations on the particular card specified in datasheet of the card.

2.2.1.2 NFC Activity

This module ((2) in Fig 3) provides probing the RF field in loop with finding out the card type of detected cards additional possibility to register and activate particular cards. This procedure is called Discovery loop and it is deeper described in section 3.1.

2.2.1.3 NFC P2P Package

This module ((3) in Fig 3) provides complete software support for Logical link management. The goal of the Link layer is to create, manage, maintain and correctly disconnect the connection among peers. In result, **multiple logical connections** between two or more peers can exist in the same time, but still using the 13,54MHz RF link. The NFC forum specified Logical Link Control Protocol (LLCP)[3] as standard for the *NFC peer to peer* communication. Thus, LLCP is implemented in NXP Reader Library P2P to process traffic on Logical Link. LLCP ensures sending and reception of separate packets while being not "aware" the upper NDEF message entirety.

Since LLCP module is the most upper (among P2P related), LLCP APIs are expected to be the most used from the Library at all. Those APIs are described in section 3.2. There is a "hidden" part of the LLCP layer called MAC layer. According to LLCP specification [3] from the NFC forum *activation* and *deactivation* procedures are related to the MAC layer.

2.2.2 Protocol Abstraction Layer

The Protocol Abstraction Layer implements the activation and control and data exchange and other operations regarding the protocol of the contactless communication. Each protocol is placed in separate own folder in the library folder structure *NxpRdbLib/comps/phpal<protocolName>*.

Since this document is focused on NFC *peer to peer* communication, which is compliant to ISO18092 protocol, this one is deeper described in section 3.3.

The NXP Reader Library supports following ISO standard protocols:

ISO14443-3A [7]: air interface communication at 13.56MHz for the cards of type A. MIFARE Classic and Ultralight EV1 are based on this protocol.

ISO14443-3B [7]: air interface communication at 13.56MHz for the cards of type B

ISO14443-4 [7]: 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. MIFARE DESFire EV1 card is based on this protocol.

ISO14443-4A [7]: previous protocol for the cards of type A

MIFARE: needs to be included for any MIFARE product-based card. Contains support for MIFARE authentication, proximity check and data exchange between the PCD and PICC according to protocols ISO/IEC14443-3A and ISO/IEC14443-4.

UM10721

© NXP B.V. 2018. All rights reserved.

User manual **Rev. 1.2 — 5 August 2018** **7 of 77**
COMPANY PUBLIC 270112

Felicia: compliant protocol [10], parts of it are also part of ISO 18092[9]

ISO18092 [9]: enables the NFC Data Exchange Protocol. ISO18092 protocol is Request&Response based NFC communication protocol, which is in accordance with the LLCP acknowledgement-based approach. This library version supports **only Passive Initiator mode**.

2.2.3 Hardware abstraction layer

The master devices which have no peripherals to perform any RF communication at all a reader chip needs to be connected to them. The main goal of the reader chip is to convert digital signal to analog which is transmitted and carries information via RF field. Apart from modulator and demodulator the PCD includes many digital registers for temporarily storing sent/received data and registers for PCD configuration, for key storage for cards etc. In addition, the reader chip is able to execute just specified commands. There are several different card readers which may differ one from another in command set, peripheral modules, internal register addresses etc. Software of HAL layer is kind of driver which enables the master device to exploit the utilities of the connected reader chip. HAL layer is responsible for encapsulating and buffering raw data coming from PAL layer to the form to be processed by particular reader chip correctly. It also ensures reading of received data from the reader chip in correct order. There are many different reader chips, but the P2P Library supports only **PN512 [12]** and **RC663 [11]** reader chips. Only PN512 is able to perform NFC P2P Target mode, but this option is not supported by this version of the P2P library.

2.2.4 Bus Abstraction Layer

The Bus Abstraction Layer ensures correct communication interface between the MCU and the reader chip. The MCU sends to the reader chip commands and other command related data like addresses and data bytes. The card reader can possibly to send some register values or received data like the response to requests from the MCU.

The NXP Reader Library supports following communication interfaces:

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 Bluebord [13] or Xpresso board.

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

2.2.5 Common layer

The NXP Reader Library also includes utilities independent of any card and hardware (card reader) – meaning they can be implemented regardless of reader chip. 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 MCU and the card reader.

Note: This has nothing to do with the communication between the reader chip and the PICC or another NFC device. All the CRC checksum is done by both the sides participating on NFC communication automatically.

phKeyStore: is key handling software – storage, changing key format etc. But the NXP Reader Library P2P supports only key storage utility. Only the NXP Reader Library Export Controlled version supports full key storage functionalities.

phLog: software module enabling log files creation.

OSAL Utils: This module provides some basic utilities like dynamic memory allocation and handles functions for LPC1227 timers. This module is described in section 3.4.

3. Explanation of the Library modules for P2P

3.1 Discovery Loop

Discovery loop is tool for detecting, activating various NFC tags and NFC peer devices. Due to OSAL layer does not provide any thread creation, Discovery loop can run only in single thread called from upper application without any possibility of interruption during running, therefore the upper application (caller) is blocked till the Discovery Loop exits. The Discovery loop can include ability to detect more tag types in the future.

3.1.1 Discovery Loop data parameter structure

Discovery loop can be controlled by values of the parameters from `phacDiscLoop_Sw_DataParams_t` parameter component. Furthermore, while running the Discovery loop information about detected and activated devices are hold by that structure.

Each parameter of the `phacDiscLoop_Sw_DataParams_t` structure is described in library file *NxpRdLib_PublicRelease/intfs/phacDiscLoop.h*.

Generally the structure holds information regarding to:

guard time for each tag type. This timeout is applied after PCD is configured for particular protocol. Only after the guard time Detection of that tag type is attempted.

number of **polling loops** for entire **Discovery loop**. Number of times the Discovery loop procedure should look for cards before discovery loop gives up.

number of **polling loops** for each **tag type** separately.

mode to run Discovery **loop**: **listen** and **poll** or vice versa and additional pause mode.

permission for particular **tag type detection** within the Discovery loop. For all of the three tag types the detection can be executed or skipped separately.

pointers to underlaying **Protocol Abstraction Layer** parameter components.

pointers to **NFC tag type components**: These components store information of detected tags.

`bDetectionConfiguration`: Or this variable with any of `PHAC_DISCLOOP_CON_POLL_A`, `PHAC_DISCLOOP_CON_POLL_B`, `PHAC_DISCLOOP_CON_POLL_F` to allow detection of the particular configuration for detection of various tag types.

`uint8_t bState`: Indicates the current state.

`uint16_t wTagsFound`: Sets bit masks indicating tags that were found.

`uint8_t bLoopMode`: Holds the combination of poll, listen and pause mode for discover loop.

`pErrHandlerCallback pErrHandler`: Pointer to the user error handler function.

`uint16_t wPausePeriod`: The delay to be used in pause mode.

3.1.2 Initialization of the parameter structure

This function initiates *phacDiscLoop_Sw_DataParams_t* parameter component (see section 3.1.1) to zeros or default values. All the components that are passed as input arguments to this function should be initialized before the call is made to start the Discovery loop.

```
phStatus_t phacDiscLoop_Sw_Init(

        phacDiscLoop_Sw_DataParams_t * pDataParams,                [In]
```

UM10721

All information provided in this document is subject to legal disclaimers.

© NXP B.V. 2018. All rights reserved.

User manual
COMPANY PUBLIC

Rev. 1.2 — 5 August 2018
270112

10 of 77

```
uint16_t wSizeOfDataParams,                        [In]

void * pHalDataParams,                             [In]

void * pOsal );                                    [In]
```

*pDataParams: pointer to `phacDiscLoop_Sw_DataParams_t` parameter component. Pointers to PAL components are initialized to zero by this function. Only way is to set them "manually" due to `SetConfig()` function does not provide this initialization neither.

wSizeOfDataParams: size of `phacDiscLoop_Sw_DataParams_t` data parameter component. It is highly recommended to use standard C `sifeof()` function.

*pHalDataParams: pointer to the HAL component.

*pOsal: reference to the OS AL data parameters.

returnValues:

`PH_ERR_SUCCESS` - Operation successful.

Other - Depending on implementation and underlaying component.

3.1.3 Discovery Loop routine

NXP P2P Reader Library provides useful routine – Discovery Loop which performs different NFC cards and tag types detection and also their activation. The procedure implements detection of the NFC tags of types A, B, F and P2P, whether there is any tag in the RF field or not. If any, it provides ISO14443 Activation of tag A and tag B type and Anticollision for both the tag types and Attribute Request ISO18092 for tags type F and P2P tags. Discovery loop is clarified by flow chart on Fig 4.

Since reader chips MFRC523 nor CLRC663 do not support listen nor pause mode (passive), active – poll mode is switched on automatically.

If any NFC Tag type detected (in case of type A and B also activated) parameters of that tag are obtained and saved into new tag component. Information about the detected tag is retained within the structured tag component but they are replaced by a new tag object as soon as that is activated.

If listen mode is not supported by the hardware or no RF field is detected, then listen mode is automatically switched off and discovery procedure continues in *polling* mode.

(5) It does not describe all the states within the Discovery loop.

Fig 4.   Discovery loop flow chart.
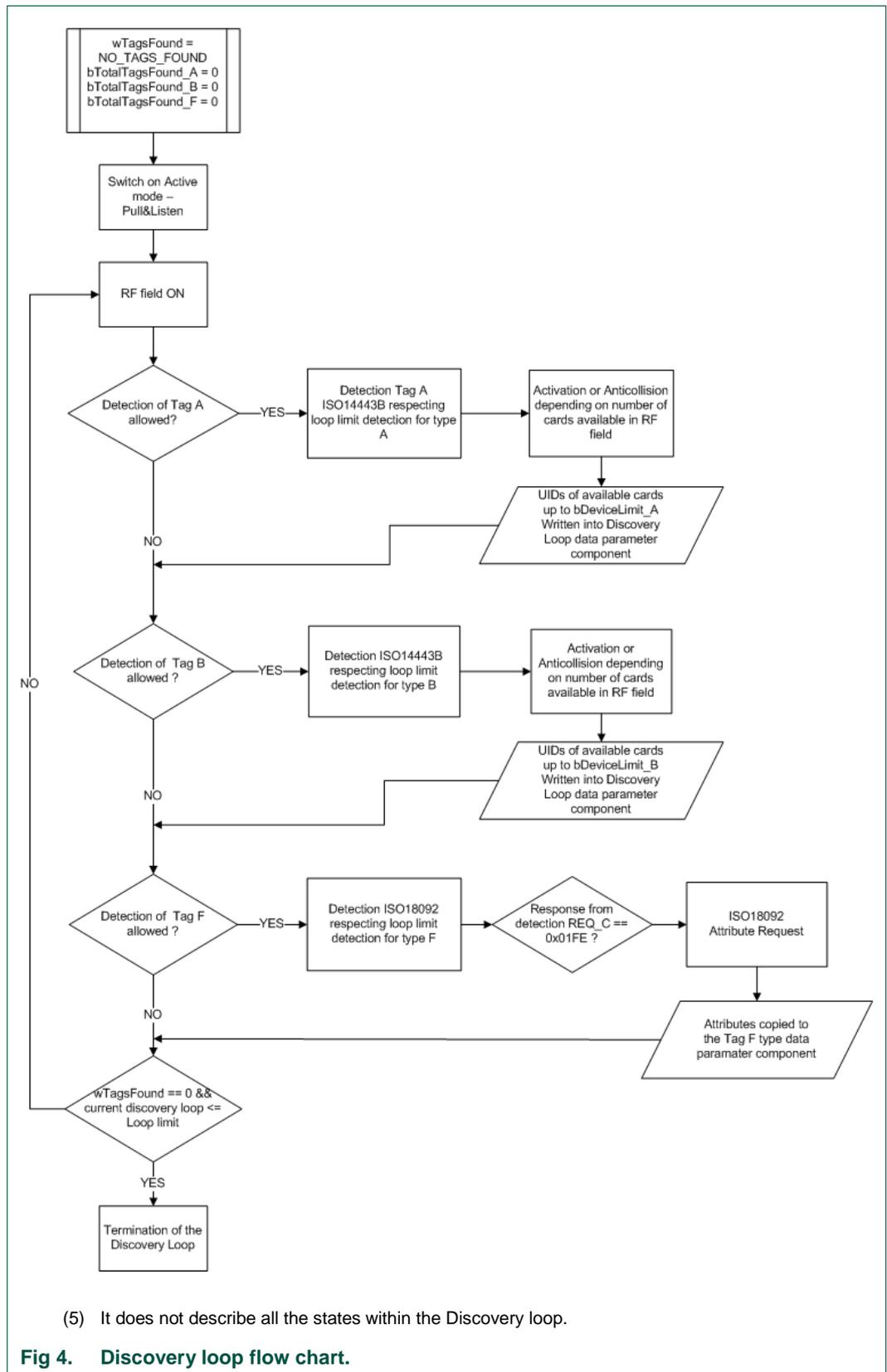
```
phStatus_t phacDiscLoop_Start(

                void * pDataParams );                          [In]
```

*pDataParams: pointer to `phacDiscLoop_Sw_DataParams_t` parameter component.

Execution of the function is influenced by parameters of this component, especially following ones:

->bNumPollLoops: number of times the Discovery loop should loop looking for tags before Discovery loop gives up.

->bDetectionConfiguration: or this parameter with desired combination of `PHAC_DISCLOOP_CON_POLL_A`, `PHAC_DISCLOOP_CON_POLL_B`, `PHAC_DISCLOOP_CON_POLL_F` to allow detection of a particular tag type.

->wTagsFound: sets bit masks indicating tags that were detected. Whenever the function is called, this flag is cleared and particular bits are set according to which tag types are currently detected in the range of the RF field.

->sTypeATargetInfo.bTotalTagsFound_A: this parameter refers to the number of the tags A detected in the field. It is set to zero with each function call – Discovery loop (NOT the partial loop detection within the Discovery loop) increased by one when any tag is detected in the range of RF field.

->sTypeATargetInfo.bLoopLimit_A: this is upper limit for attempts to detect Tag A type inside the `phacDiscLoop_Sw_Int_DetectA()` function (see section 3.1.5).

->sTypeATargetInfo.bActivatedTagNumber_A: number of activated Tags of type A

returnValues:

`PH_ERR_INVALID_PARAMETER`: invalid *listen/poll/pause* mode. If the HAL component does not provide *listen* mode, the Discovery loop is automatically run in *poll* mode with *listen* switched off and detection procedure runs in poll mode, without termination the Discovery loop.

Unknown NFC Tag type, other than A/B/F

`PH_ERR_SUCCESS`: Operation successful.

Other: Depending on implementation and underlaying component.

3.1.4 Activate Card

Activates the given tag type with given index. This function should follow previous successful tag detection, passing the tag type as the input argument. In case of tag type A according to detected SAK the tag is additionally activated as ISO14443p4A or ISO18092 for P2P. This function is implemented within the Discovery loop - `phacDiscLoop_Start()` function (see section 3.1.3).

Note: This function does not provide tags F activation since those are activated with the detection.

Activate card does not provide Anticollision resolution, whilst both Anticollision resolution either Activation are implemented within the `phacDiscLoop_Start()`

```
phStatus_t phacDiscLoop_Sw_ActivateCard(

        phacDiscLoop_Sw_DataParams_t *pDataParams,         [In]

        uint8_t bTagType,                                  [In]

        uint8_t bTagIndex );                               [In]
```

*pDataParams: pointer to `phacDiscLoop_Sw_DataParams_t` parameter component. If the tag type is activated successfully, the tag component is fulfilled with parameters of the activated tag.

bTagType: `PHAC_DISCLOOP_TYPEA_ACTIVATE` - activate of the tag type A.

`PHAC_DISCLOOP_TYPEB_ACTIVATE` - activate f the tag type B.

bTagIndex: The tag which has to be activated. Only I3P3 of tag type A and B can have three references – tag indexes, while all the others have just one component to be stored within the to `phacDiscLoop_Sw_DataParams_t` structure.

returnValues:

`PH_ERR_INVALID_PARAMETER`: invalid value of `bTagType`

`PH_ERR_INVALID_DATA_PARAMS`: `bTagIndex` is greater than number of previously found (detected) tags A.

`PH_ERR_SUCCESS`: Operation successful.

Other: Depending on implementation and underlaying component.

3.1.5 Detect A

This function broadcasts Request A of ISO14443p3A by NFC. If there is present any tag type A in a range of the RF field, then answer to request ARQ_A is received.

```
phStatus_t phacDiscLoop_Sw_Int_DetectA(

                 phacDiscLoop_Sw_DataParams_t * pDataParams );              [In]
```

*pDataParams: pointer to `phacDiscLoop_Sw_DataParams_t` data parameter component.

If a tag type A is detected, then received Answer to Request type A ATQ_A is copied to

`pDataParams->sTypeATargetInfo.aTypeA_I3P3[0].aAtqa`

returnValues:

`PHAC_DISCLOOP_ERR_TYPEA_NO_TAG_FOUND`: Answer to Request A not detected.

`PH_ERR_COLLISION_ERROR`: more tags type A in the RF field detected.

`PH_ERR_SUCCESS`: Operation successful.

Other: Depending on implementation and underlaying component.

3.1.6 Detect B

This function broadcasts Request B of ISO14443p3B by NFC. If there is present any tag type B in a range of the RF field, then answer to request 8 bytes ID and 8 byte PM are received.

```
phStatus_t phacDiscLoop_Sw_Int_DetectB(

                 phacDiscLoop_Sw_DataParams_t * pDataParams );            [In]
```

*pDataParams: pointer to `phacDiscLoop_Sw_DataParams_t` data parameter component.

If a tag type B is detected, then received Answer to Request type A ATQ_B is copied to

`pDataParams->sTypeBTargetInfo.aI3P3B[].aPupi`

returnValues:

`PHAC_DISCLOOP_ERR_TYPEA_NO_TAG_FOUND`: Answer to Request B not detected.

`PH_ERR_INTEGRITY_ERROR`: more tags B in the RF field detected.

`PH_ERR_SUCCESS`: Operation successful.

Other: Depending on implementation and underlaying component.

3.1.7 Detect F

This function broadcasts Felica Request C. If there is present any tag type F in a range of the RF field, then answer to request ATQ_B is received. Detects if there is any Type B tag in the field.

```
phStatus_t phacDiscLoop_Sw_Int_DetectF(

        phacDiscLoop_Sw_DataParams_t * pDataParams );    [In]
```

*pDataParams: pointer to `phacDiscLoop_Sw_DataParams_t` parameter component.

If a tag type F is detected successfully, then received Answer to Request type C is copied to `pDataParams->sTypeFTargetInfo.aTypeF[0].aIDmPMm`

returnValues:

`PHAC_DISCLOOP_ERR_TYPEA_NO_TAG_FOUND`: tag type F not detected.

`PH_ERR_COLLISION_ERROR`: more tags type F in the RF field available.

`PH_ERR_SUCCESS`: Operation successful.

Other: Depending on implementation and underlaying component.

3.2 LLCP modul

In the P2P Library there is logical link implemented in full compliance with LLCP specified by NFC forum.[3] The purpose of the LLCP is to create, manage, maintain and correctly disconnect the connection among peers. In result, **multiple logical connections** between two or more peers can exist in the same time, while still using the 13,54MHz RF link. This has nothing to do with the anticollision nor the collision avoidance mechanism (those are regarding to ISO14443 and ISO18092 protocols). According to LLCP specification it is possible to run up to 62 services (applications, peers) on LLCP link connection. NXP Reader Library is limited to establish the LLCP link connection with other 5 services in maximum. The LLCP ensures the transmission of data (or link) packets separately (they are just numbered from 0 to 15 for internal purposes).

3.2.1 LLCP Library structures

3.2.1.1 LLCP parameter component - `phlnLlcp_Fri_DataParams_t`

The LLCP basic and uppermost parameter component that holds pointers to buffers and other LLCP related structures and underlaying parameter components. Since this parameter component holds pointers to several different components, those data of those parameter components are accessible via `phlnLlcp_Fri_DataParams_t`. Therefore, there is one common input parameter for all the LLCP API functions. Within a particular API there are handled only required data parameter components.

```
typedef struct {

    uint16_t                    wId;

    phlnLlcp_t                  * pLlcp;

    phlnLlcp_sLinkParameters_t  * pLinkParams;
```

```
phlnLlcp_Transport_t           * pLlcpSocketTable;
void                           * pTxBuffer;
uint16_t                         wTxBufferLength;
void                           * pRxBuffer;
uint16_t                         wRxBufferLength;
phHal_sRemoteDevInformation_t  * pRemoteDevInfo;
void                           * pLowerDevice;
} phlnLlcp_Fri_DataParams_t;
```

See section 4.1.5 and 4.1.6 that only content of `*pRemoteDevInfo` and `*pLinkParams` is needed for "direct access". Other data parameter components pointed by members of parameter component do not need to be initialized to any particular values. They must be only declared before initialization of the `phlnLlcp_Fri_DataParams_t` parameter component.

*pLlcp: LLCP data parameter component

*pLinkParams: pointer to `phlnLlcp_sLinkParameters_t` data parameter component storing the link parameters

*pLlcpSocketTable: pointer to `phlnLlcp_Transport_t` parameter component.

*pRemoteDeviceInfo: pointer to `phHal_sRemoteDevInformation_t` parameter component. This structure holds information about device detected in the RF field. The P2P Library assumes there is stored information taken from the remote's Attribute Response. But there is no P2P Library function to fill parameter component of this structure, therefore must be done manually (see section 4.1.5). Once parameter component of this structure passed to `phlnLlcp_Fri_Init()` contains relevant link parameters, further link activation procedure takes link parameters from this parameter component, otherwise PAX exchange performed.

*pLlcp: pointer to the LLCP layer

*pLinkParams: pointer to the Link parameters.

*pLlcpSocketTable: pointer to the Socket table.

*pRemoteDevInfo: pointer to the Remote device information.

*pTxBuffer: pointer to the transmit buffer

wTxBufferLength: length of the transmit buffer

*pRxBuffer: pointer to the receive buffer

wRxBufferLength: length of the receive buffer

*pLowerDevice: pointer to the underlaying `palI18092mPI` PAL data parameter component

3.2.1.2 Link Parameters - `phFriNfc_Llcp_sLinkParameters_t, phlnLlcp_sLinkParameters_t`

This data parameter component holds information about remote.

```
typedef struct phFriNfc_Llcp_sLinkParameters {
    uint16_t   miu;
    uint16_t   wks;
    uint8_t    lto;
```

```
    uint8_t    option;

} phFriNfc_Llcp_sLinkParameters_t, phlnLlcp_sLinkParameters_t;
```

miu: remote Maximum Information Unit MIU.

wks: remote Well-Known Services

lto: remote Link TimeOut (in 1/100s)

option: remote Options

3.2.1.3 Buffer structure `phNfc_sData_t`

This structure holds the Data in the Buffer of the specified size.

```
typedef struct phNfc_sData_t {

    uint8_    *buffer;

    uint32_    length;

} phNfc_sData_t;
```

buffer: pointer to the buffer that stores the data

length: length of the buffer. Number of valid bytes in the buffer.

Pointer to this structure is taken as input argument in many API functions, so the developer needs to define some instances of it.

3.2.2 Initialization of the LLCP layer

Initialize the LLCP FRI component. All the input parameters are pointers to dedicated structures or buffers. The first input parameter is pointer to data parameter component to be initialized. This function assigns given input arguments (pointers) to members of parameter component `phlnLlcp_Fri_DataParams_t` pointed by `pDataParams`. apart the

```
phStatus_t phlnLlcp_Fri_Init(

            phlnLlcp_Fri_DataParams_t * pDataParams,          [In]

            uint16_t wSizeOfDataParams,                       [In]

            phlnLlcp_t * pLlcp,                               [In]

            phlnLlcp_sLinkParameters_t * pLinkParams,         [In]

            phlnLlcp_Transport_t * pLlcpSocketTable,          [In]

            phHal_sRemoteDevInformation_t * pRemoteDevInfo,   [In]

            void * pTxBuffer,                                 [In]

            uint16_t wTxBufferLength,                         [In]

            void * pRxBuffer,                                 [In]

            uint16_t wRxBufferLength,                         [In]

            void * pLowerDevice );                            [In]
```

*pDataParams: pointer to `phlnLlcp_Fri_DataParams_t` parameter component to be initiated.

wSizeOfDataParams: size of the parameter component – `sizeof(wSizeOfDataParams)`.

Other input parameters are members of `phlnLlcp_Fri_DataParams_t` (see section 3.2.1.1).

See section 4.1.5 and 4.1.6 that only content of *`pRemoteDevInfo` and *`pLinkParams` is needed for "direct access". Other data parameter components pointed by members of parameter component do not need to be initialized to any particular values. They must be only declared before initialization of the `phlnLlcp_Fri_DataParams_t` parameter component.

returnValues:

`PH_ERR_SUCCESS` Operation successful.

3.2.3 Pending

Sometimes the current request cannot be performed in time, due to another operation is running. For example, the local intends to send disconnect request, but the link is currently performing receiving from the remote. Thus, the disconnect request is pending means placed into "waiting queue". The disconnect request shall be send as soon as the link will perform sending operations. All the pending cases are managed internally, thus the developer need not to care about it.

3.2.4 Callbacks

In the P2P Library provides notification to the application layer via callbacks when an important event on LLCP layer occurs. The developer may build his own function, declare it and set it like a callback. The developer's defined callback function must fulfill the function prototype for the particular callback type defined in the library. All the socket related callbacks prototypes are in *NxpRdLib_PublicRelease/comps/phlnLlcp/src/Fri/phFriNfc_LlcpTransport.h.* But the developer is totally free in building the body of the callback function. Except the callback function must be declared right, it can be triggered at an event after set by the dedicated function. In the sections 3.2.8 and 3.2.10 there are link and callback prototypes clarified through following pattern:

Called at event: the event on LLCP that triggers the user CB function is called.

Set by: the user defined CB function address need to be passed like the input argument to the certain function. In fact, the function address is assigned to internal library parameter structure. From this time on, the user defined CB can be called when the event occurs.

Function prototype: although the function body may be defined by the user, the function header and return value need to agree with the CB function prototype defined by the P2P library. Function may be a type void.

Input arguments: are explained one by one. In general, there are two input arguments passed to the called CB. The first input argument passed to each CB is `pContext` - pointer to application layer data of any type. The pointer needs to be passed to the dedicated LLCP Link API or LLCP Transport Socket API function from section 3.2.7 or 3.2.9 respectively in the same time as pointer to CB function. The data may be later used during the CB function execution. The second on third argument (if any) are information passed by the caller of the CB – from within the P2P library, giving precious information about the event related circumstances.

There are two types of callbacks in general. Link related callbacks and socket related callbacks. The link callbacks are called when link status changes from activated to deactivated or vice versa. The socket callbacks are triggered on dedicated socket events.

3.2.5 SYMM

The NFC forum defines LTO to avoid infinite waiting for activity from a remote while the remote is off without letting know. Therefore, the P2P Library implements SYMM timer to perform LTO. The SYMM timer is adjusted according to LTO of the remote received during link activation phase. The LTO in P2P Library is implemented in compliance with the NFC Forum LLCP. All the SYMM timer and SYMM exchange is managed internally, thus the developer does not need to care about at all. The SYMM timer is reset with each sent packet (regardless link or transport). If the SYMM timer expires during local link receive state, then link is deactivated (consequently disconnected). If the SYMM timer expires while the local is something to send, then it local sends SYMM PDU.

3.2.6 Medium access control – MAC layer

The MAC layer is necessary for nearly all the LLCP layer link related functionalities based on receiving/sending and LLC link activation/deactivation as well. Functions from the section 3.2.7 providing LLC link activation must be run before the first LLCP PDU sent/received.

The MAC layer parameter component `LlcpMac` needs to be fulfilled with information of the NFC device or tag type detected in the RF field. But there is no function providing passing of detected tag/device type to `RemDevType` in `phNfc_sRemoteDevInformation_t` component, thus must be done manually. MAC layer can be initiated only for those devices and tag types listed in Table 1, because they are capable of NFC P2P communication. Furthermore, just in case P2P initiator or target (two bottom lines) the MAC layer interface can be fully activated, consequently send and receive functionalities are fully provided.

Table 1.   Device and tag type for MAC layer
In NXP Reader Library P2P the MAC layer functionality is legal for following remote devices/tags

Tag/device type	Identifier in the P2P Library
ISO14443 type A tag	`phHal_eISO14443_A_PICC`
ISO14443 type B tag	`phHal_eISO14443_B_PICC`
ISO18092 type F – P2P initiator	`phHal_eNfcIP1_Initiator`
ISO18092 type F – P2P target	`phHal_eNfcIP1_Target`

The LLC link layer activation procedure needs three functions to be called in following order:

1. Reset: `phlnLlcp_Reset()` section 3.2.7.1
2. Check: `phlnLlcp_ChkLlcp()` section 3.2.7.2
3. Activation: `phlnLlcp_Activate()` section 3.2.7.3
4. Send/Receive: LLCP packets: upper layer communication
5. Deactivation: `phlnLlcp_Deactivate()` section 3.2.7.4

Once the Activate successes, the LLC link is fully connected with undelaying PAL layer (represented by ISO18092 Initiator). This needs to be done, to send any Afterward the functions from section 3.2.9 can be called (socket creation, connect, listen etc.).

Internal MAC Send and Receive are used for all the LLCP PDUs (SYMM, CONNECT, etc.) transaction between the LLCP layer and the PAL layer. This should be hidden for a developer, who is expected to use APIs from section 3.2.9.

3.2.7 LLCP Link APIs

3.2.7.1 Reset LLCP link

This function makes initial configuration steps necessary for later LLC link activation procedure. This reset function is prerequisite for correct execution of `phlnLlcp_Activate()` and `phlnLlcp_Deactivate()`.

```
phStatus_t phlnLlcp_Reset(

        void * pDataParams,              [In]

        phlnLlcp_LinkStatus_CB_t pfLink_CB,     [In]

        void * pContext );               [In]
```

*pDataParams: pointer to `phFriNfc_Llcp_t` parameter component.

pfLink_CB: pointer to the user defined callback assigned to `pfLinkCB` in `phFriNfc_Llcp_t` component. It is called from within `phlnLlcp_Activate()` and `phlnLlcp_Deactivate()` function and refers to the change of the link status when activated or deactivated. Developer can pass own `phlnLlcp_LinkStatus_CB_t` function here. If none function to pass, then this parameter must be NULL. The second input argument of the callback is passed by lower layer caller and it refers about latest (updated) link status. See section 3.2.8.2.

*pContext: pointer to the content – input argument for the callback function.

returnValues:

`NFCSTATUS_BUFFER_TOO_SMALL`	-	Receive buffer is not large enough to support 131 bytes (128 + 2 + 1 == MIU + Header + Sequence)
	-	Transmit buffer too small to support maximal LLCP frame size (Header + Sequence + MIU)
`NFCSTATUS_INVALID_PARAMETER`	-	MIU in `psLinkParams` lower than 128 bytes
`PH_ERR_SUCCESS`	-	Operation successful.
Other	-	Depending on implementation and underlaying component.

3.2.7.2 Check

This function checks some Attribute Response from remote device then decides if the MAC layer can be enabled for that device. The device should have been previously detected in the Discovery loop. If the remote device type is NCF P2P Initiator or Target type, then it enables the internal connection between LLCP layer and MAC layer (necessary for Activation, Deactivation, Sending, Reception and Checking itself). In case ISO14443A or ISO14443B no internal MAC connection provided. In addition, General Bytes from the Attribute Response matched to that remote device (received in Discovery loop) are tested whether they equal to *LLC Magic Number* – 3 byte array 0x46, 0x66, 0x6D. If it equals, then the rest of data of the Attribute Response is later assigned as remote link parameters rather than doing PAX exchange.

Note: This function requires the LLC link to be in correct state which is ensured by previous run of `phlnLlcp_Reset()` function.

```
phStatus_t phlnLlcp_ChkLlcp(

            void * pDataParams,             [In]

            phlnLlcp_Check_CB_t pfCheck_CB,  [In]

            void * pContext );               [In]
```

*pDataParams: pointer to `phFriNfc_Llcp_t` parameter component.

pfCheck_CB: pointer to the callback function that will be called when check procedure is complete. The accepted and called just for remote device type P2P Initiator or Target.

*pContext: pointer to the content – input argument for the callback function.

returnValues:

`NFCSTATUS_INVALID_STATE`	-	LLCP link is not in `PHFRINFC_LLCP_STATE_RESET_INIT` state
`NFCSTATUS_INVALID_DEVICE`	-	Device type unable to perform NFC P2P
`NFCSTATUS_FAILED`	-	Attribute Response from remote device does not agree with *LLC Magic Number*
`PH_ERR_SUCCESS`	-	Operation successful
Other	-	Depending on implementation and underlaying component

3.2.7.3 Activate LLC link

This function provides basic LLC initial configuration, activates LLC link and MAC layer. These are all prerequisite necessary for a later flawless communication via LLC link. The LLC link state is changed from *checked* - `PHFRINFC_LLCP_STATE_CHECKED` to *activated* - `PHFRINFC_LLCP_STATE_ACTIVATION`. Before this function run it is recommended to run `phlnLlcp_Reset()` function (see section 3.2.7.1), which preconfigures MAC layer (internal callback) correctly.

There are two ways, how find out link parameters (LTO, MIU, WKS, OPT) of the remote device. Remote link parameters should have been already known from the Discovery loop, since that implements ISO18092 defined Attribute Request. The remote device sends Attribute Response containing LLCP Magic Number (specified by NFC forum) together with link parameters in TLV format. After parsing, first of all the version agreement is performed then the LTO timer is triggered.

However, if there is not LLCP Magic Number within Attribute Response and the local is NFC P2P Initiator type, then in the activation procedure PAX PDU with local link parameters is sent to the remote.

After LLC activated successfully the notification about change of the link status toward service layer is done via user defined callback function `Llcp->pfLink_CB()` which is passed like an input argument to `phlnLlcp_Reset()`.

The function does not create LLCP socket. It should be created by `phlnLlcp_Transport_Socket()`.

Note: If deactivation callback is called, it indicates that activation procedure failed, thus deactivation procedure has been automatically executed to revert steps done by unsuccessful activation.

```
phStatus_t phlnLlcp_Activate(

                void * pDataParams );        [In]
```

*pDataParams: pointer to `phFriNfc_Llc_t` component.

returnValues:

`NFCSTATUS_INVALID_STATE` – LLCP not in state `PHFRINFC_LLCP_STATE_CHECKED`

`PH_ERR_SUCCESS` - Operation successful.

Other - Depending on implementation and underlaying component.

3.2.7.4 Deactivate LLC link

This function deactivates MAC interface and disconnects LLCP link. LLCP link connection is canceled by sending DISC PDU via LLCP link (DSAP == 0x00, SSAP == 0x00) as soon as local is ready for sending operation. If the send operation is pending, then disconnect procedure is terminated. After deactivated successfully the notification about change of the link status toward service layer is done via user defined callback function `Llcp->pfLink_CB()` which is passed like an input argument to `phlnLlcp_Reset()`.

```
phStatus_t phlnLlcp_Deactivate(

                  void * pDataParams );                  [In]
```

*pDataParams: pointer to `phFriNfc_Llc_t` component.

returnValues:

`NFCSTATUS_INVALID_STATE`	-	LLCP link state other than `PHFRINFC_LLCP_STATE_OPERATION_RECV` or `PHFRINFC_LLCP_STATE_OPERATION_SEND`.
`NFCSTATUS_PENDING`	-	DISC PDU not sent due to link send pending.
`PH_ERR_SUCCESS`	-	Operation successful.
Other	-	Depending on implementation and underlaying component.

3.2.7.5 Send PDU packet via LLCP link

This function is used to send a given PDU via LLCP. Arguments need to be passed in form *header - sequence field - information field*. The function can only be called on a *connection-oriented* socket which is already in the *connected state*.

Note: In fact, this function is implemented by some other functions sending any PDU via LLCP link. To send data packed within via LLCP it is recommended rather to use `phlnLlcp_Transport_Send()` (see section 3.2.9.10) which sends data within the *Information* PDU frame and assembles *Header* and *Sequence* automatically from given LLCP socket component parameters.

```
phStatus_t phlnLlcp_Send(

                  void * pDataParams,                        [In]

                  phlnLlcp_sPacketHeader_t * pHeader,        [In]

                  phlnLlcp_sPacketSequence_t * pSequence,    [In]

                  phNfc_sData_t * pInfo,                     [In]

                  phlnLlcp_Send_CB_t pfSend_CB,              [In]

                  void * pContext );                         [In]
```

*pDataParams: pointer to `phFriNfc_Llcp_t` parameter component.

*pHeader: pointer to the PDU packet *header* composed from SSAP, PTYPE and DSAP.

*pSequence: pointer to the PDU packet *sequence field*.

*pInfo: pointer to the PDU *information field*.

pfSend_CB: pointer to the callback of LLCP link structure that shall be called when sending via LLCP link executed successfully.

*pContext: pointer to the content – input argument for the callback function.

returnValues:

`NFCSTATUS_REJECTED`	- previous sending operation has not been finished yet
`NFCSTATUS_PENDING`	- Previous receive operation has not been finished yet.
`NFCSTATUS_INVALID_STATE`	- LLCP state other than `PHFRINFC_LLCP_STATE_OPERATION_RECV` or `PHFRINFC_LLCP_STATE_OPERATION_SEND`.
`PH_ERR_SUCCESS`	- Operation successful.
Other	- Depending on implementation and underlaying component.

3.2.7.6 Receive PDU packet on the LLC link

This function sets a given callback in the internal link structure resulting in the callback shall be called if any transport reception on LLCP link occurs. Setting the callback by the function is rejected until the previously set link reception callback called. This receive occurs even before parsing the header of the packet.

Note: The link reception callback is mostly (dominantly) utilized for the library internal purposes – receiving the transport packets and further parsed. Thus there is a high probability, it is occupied by this internal receive callback function at any time after reset sockets by `phlnLlcp_Transport_Reset()` function.

```
phStatus_t phlnLlcp_Recv(
            void * pDataParams,             [In]
            phlnLlcp_Recv_CB_t pfRecv_CB,   [In]
            void * pContext );              [In]
```

*pDataParams: pointer to `phlnLlcp_Fri_DataParams_t` parameter component.

pfRecv_CB: the receive callback to be called when any transport data on the LLCP link received. Within the `phlnLlcp_Transport_Reset()` function the internal library build-in transport receive callback is assigned to this callback and is reassigned repeatedly internally and the link callback keeps being captured by it unless manually set to another callback.

*pContext: upper layer context to be used as input arguments in callback.

returnValues:

`NFCSTATUS_REJECTED`: previously assigned callback (most probably the internal library build-in transport receives callback) has not been executed yet.

`NFCSTATUS_SUCCESS` - Operation successful.

3.2.8 LLCP link Callbacks

3.2.8.1 Link Check CB

Called at event: Incoming connection request (CONNECT PDU) from a client received.

Set by API: `phlnLlcp_ChkLlcp()`   section 3.2.7.2

Function prototype:

```
typedef void (*phFriNfc_Llcp_Check_CB_t) ( void *pContext,

                                            NFCSTATUS status );
```

*pContext: pointer to user data passed to processed in the CB

status: refers to

3.2.8.2 Link Status CB

Called at event: Link status changed to activated or deactivated

Set by API: `phlnLlcp_Reset()`   section 3.2.7.1

```
typedef void (*phFriNfc_Llcp_LinkStatus_CB_t) ( void *pContext,

                                    phFriNfc_Llcp_eLinkStatus_t eLinkStatus);
```

*pContext: pointer to user data passed to processed in the CB

status: enum type referring to updated LLCP link status. Following enum values can occur:

```
phFriNfc_LlcpMac_eLinkActivated
```

```
phFriNfc_LlcpMac_eLinkDeactivated
```

3.2.8.3 Link Send CB

Called at event: generic LLCP packet sent via LLCP link

Set by API `phlnLlcp_Send()`   section 3.2.7.5

Set by API: `phlnLlcp_Send()`

```
typedef void (*phFriNfc_Llcp_Send_CB_t) ( void *pContext,

                                           NFCSTATUS stats );
```

*pContext: pointer to user data passed to processed in the CB

status:

3.2.8.4 Link Receive CB

Called at event: generic LLCP packet received via LLCP link

Note: This CB is occupied by internal library function.

Set by API `phlnLlcp_Receive()`   section 0

```
typedef void (*phFriNfc_Llcp_Recv_CB_t) ( void *pContext,

                                           phNfc_sData_t *psData,

                                           NFCSTATUS status );
```

*pContext: pointer to user data passed to processed in the CB

*psData: pointer received data

UM10721
© NXP B.V. 2018. All rights reserved.

User manual
COMPANY PUBLIC

Rev. 1.2 — 5 August 2018
270112

24 of 77

status:

3.2.9 LLCP Transport Socket APIs

3.2.9.1 Create LLCP socket

This function creates a socket for a given LLCP link. Sockets can be of two types: *connection-oriented* and *connectionless*. If the socket is connection-oriented, the caller must provide a working buffer to the socket in order to handle incoming data. This buffer must be large enough to fit the receive window (RW * MIU), the remaining space being used as a linear buffer to store incoming data as a stream. Data will be readable later using the `phLibNfc_LlcpTransport_Recv()` function (section 3.2.9.11).

<u>**Note:**</u> The options and working buffer are not required if the socket is used as a listening socket, since it cannot be directly used for communication.

```
phStatus_t phlnLlcp_Transport_Socket(

            void * pDataParams,                        [In]

            phlnLlcp_Transport_eSocketType_t eType,    [In]

            phlnLlcp_Transport_sSocketOptions_t * pOptions,  [In]

            phNfc_sData_t * pWorkingBuffer,            [In]

            phlnLlcp_Transport_Socket_t ** pLlcpSocket,  [Out]

            phlnLlcp_TransportSocketErrCb_t pErr_Cb,   [In]

            void * pContext );                         [In]
```

*pDataParams: pointer to `phFriNfc_Llcp_t` parameter component.

eType: type of the socket to be created: *connection-oriented* or *connectionless.*

*pOptions: options to be used with the socket.

*pWorkingBuffer: working buffer to be used by the library.

**pLlcpSocket: pointer on the socket to be filled with a socket found on the socket table.

pErr_Cb: error callback function from application that shall be called whenever an error on the socket occurs. The cases when the error callback function is called, and error callback type are described in section 3.2.10.1.

*pContext: pointer to application layer data to be used in the callback.

`PH_ERR_SUCCESS`	-	Operation successful.
Other	-	Depending on implementation and underlaying component.

3.2.9.2 Reset LLCP socket

This function transport structure and all the LLCP transport sockets to default states or zero values. LLCP link structure must exist and should be reset before. Once reset function is done, receiving of incoming LLCP packets from LLCP link is enabled.

```
phStatus_t phlnLlcp_Transport_Reset(

            void * pDataParams );                      [In]
```

*pDataParams: pointer to `phlnLlcp_Fri_DataParams_t` parameter component.

returnValues:

`NFCSTATUS_SUCCESS`	-	Operation successful.
Other	-	Depending on implementation and underlaying component.

3.2.9.3 Bind a socket to a local source SAP

This function binds a given LLCP transport socket with a SAP and service name altogether. Only the LLCP socket mandatory - must have been already created and passed like nonzero input argument. Binding is performed only if the socket is created (after `phlnLlcp_Transport_Socket()` or `phlnLlcp_Transport_Disconnect()` function). SAP and Service name are optional parameters (but must be passed as `NULL` at least).

The function provides custom SAP assignment and dynamic SAP assignment as well. Depending on whether an existing service name is given or not, SAP value is assigned among either advertised or unadvertised free SAPs. Thus, the SAP assignment rules are implemented in full compliance with LLCP 1.1[3] by NFC Forum - see Table 2.

Table 2.   DSAP/SSAP values

DSAP/SSAP	NFC Forum description
0	Link management
1	Designate well known service access point for the Service Discovery Protocol
2-15	Well-Known Service Access Points
16-31	Shall be assigned by the local LLC to services registered by the service environment. These registrations shall be made available by the local SDP instance for discovery and use by a remote LLC.
32-63	Shall be assigned by the local LLC as the result of an upper layer service request and shall not be available for discovery using the SDP

```
phStatus_t phlnLlcp_Transport_Bind(

            void * pDataParams,                         [In]

            phlnLlcp_Transport_Socket_t * pLlcpSocket,  [In]

            uint8_t nSap,                               [In]

            phNfc_sData_t *psServiceName );             [In]
```

*pDataParams: pointer to `phFriNfc_Llcp_t` parameter component.

*pLlcpSocket: pointer to the LLCP transport socket.

nSap: local source SAP number to bind the given socket with. If this parameter is `NULL`, a free SSAP is assigned dynamically respecting the intervals from Table 2.

*psServiceName: pointer to service name, or `NULL` if no service name. If no service name (`NULL`), `nSAP` is considered as unadvertised see impact on SAP value in Table 2.

returnValues:

`NFCSTATUS_INVALID_STATE`	-	Attempt to bind a socket not created yet or already connected or bound.
`NFCSTATUS_ALREADY_REGISTERED`	-	Passed `nSAP` already bound to another socket.
`NFCSTATUS_INVALID_PARAMETER`	-	The given SAP out of valid range – see Table 2.
	-	The given service name already in use - bound with another SAP.

`NFCSTATUS_INSUFFICIENT_RESOURCES`	-	No free SAP available.
`NFCSTATUS_NOT_ENOUGH_MEMORY`	-	Insufficient memory space to store the given service name.
`NFCSTATUS_SUCCESS`	-	Operation successful.

3.2.9.4 Connect

This function tries to connect given socket to a given SAP on the remote peer. Connection is performed only for *connection-oriented* sockets that is not currently connected. If the socket is not bound to a local SAP, it is implicitly bound to a free unadvertised SAP (32-63). In accordance with the LLCP defined by NFC forum if MIUX, RW and Service Name are different from default values, then those are also sent to the remote.

```
phStatus_t phlnLlcp_Transport_Connect(

        void * pDataParams,                                [In]

        phlnLlcp_Transport_Socket_t * pLlcpSocket,         [In]

        uint8_t nSap,                                      [In]

        phlnLlcp_TransportSocketConnectCb_t pConnect_RspCb, [In]

        void * pContext );                                 [In]
```

*pDataParams: pointer to `phFriNfc_Llcp_t` parameter component.

*pLlcpSocket: pointer to the LLCP transport socket.

nSap: the destination SAP to connect to. Must be in range from 2 to 63.

pConnect_RspCb: connect callback function to be called when the connection operation is completed– CC frame from the remote received. The connection callback type is more precisely described in section 3.2.10.3.

*pContext: pointer to application layer data to be used in the callback.

returnValkues:

`NFCSTATUS_INVALID_PARAMETER`	-	Non *connection-oriented* socket.
	-	`nSap` value out of valid range (2-63).
`NFCSTATUS_INVALID_STATE`	-	The socket already connected - neither bound nor created.
	-	Socket has already service name assigned.
`NFCSTATUS_PENDING`	-	LLCP link send pending. Connection operation is in progress, `pConnect_RspCb()` will be called upon completion.
`PH_ERR_SUCCESS`	-	Operation successful.
Other	-	Depending on implementation and underlaying component.

3.2.9.5 Connect by URI

This function tries to create connection between given socket and a remote service designated by a given URI. If the socket is not bound to a local SAP, it is implicitly bound to a free SAP.

```
phStatus_t phlnLlcp_Transport_ConnectByUri(

        void * pDataParams,                              [In]

        phlnLlcp_Transport_Socket_t * pLlcpSocket,       [In]

        phNfc_sData_t * psUri                            [In]

        phlnLlcp_TransportSocketConnectCb_t pConnect_RspCb,  [In]

        void * pContext );                               [In]
```

*pDataParams: pointer to `phFriNfc_Llcp_t` parameter component.

*pLlcpSocket: pointer to the transport socket to be connected to the remote designated by the given URI address.

*psUri: the URI corresponding to the destination SAP on remote to connect to. Length of URI is limited to 255 bytes (characters).

pConnect_RspCb: connect callback function to be called when the connection operation is completed- CC frame from the remote received. The connection callback type is more precisely described in section 3.2.10.3.

*pContext: pointer to application layer data to be used in the callback.

returnValkues:

`NFCSTATUS_INVALID_PARAMETER`	-	Non *connection-oriented* socket.
-	-	The given socket already connected or pending connect.
	-	URI address longer than 255 characters
`NFCSTATUS_PENDING`	-	LLCP link send pending. Connection operation is in progress, `pConnect_RspCb()` will be called upon completion.
`PH_ERR_SUCCESS`	-	Operation successful.
Other	-	Depending on implementation and underlaying component.

3.2.9.6 Listen to Connection Requests

This function makes a given socket listen to any incoming connection request from a remote. This is prerequisite for establishing the LLCP connection initialized from a remote. Listening is only allowed for *connection-oriented* sockets which are currently not connected. The listening itself does not load a thread execution, because it is based on callback. In fact, this function just sets a given callback.

The P2P library parses the incoming connection request and further ensures resolving of following situations that may occur:

- Immediate respond with FRMR to a sender of the connection request when invalid TLV data received.

- According to remote's DSAP to find the socket that the connection request is designated for.

- According to service name to find the socket that the connection request is designated for.

The callback function shall be called when an incoming connection request from a remote received and found out the socket requested for conenction. Then an application layer may decide whether to accept `phlnLlcp_Transport_Accept()` (section 3.2.9.7) or to reject `phlnLlcp_Transport_Reject()` (section 3.2.9.8) the incoming connection request for that particular LLCP socket.

Note: This function should be called after socket bound by `phlnLlcp_Transport_Bind()` (section 3.2.9.3). Without local SAP bound, the socket is not LLCP addressable.

```
phStatus_t phlnLlcp_Transport_Listen(

                void * pDataParams,                              [In]

                phlnLlcp_Transport_Socket_t * pLlcpSocket,       [In]

                phlnLlcp_TransportSocketListenCb_t pListen_Cb,   [In]

                void * pContext );                               [In]
```

*pDataParams: pointer to `phFriNfc_Llcp_t` parameter component.

*pLlcpSocket: pointer to the transport socket.

pListen_Cb: listen callback function to be called when the socket receives a connection request. The second input argument refers to the socket requested to be connected (see section 3.2.10.2).

*pContext: pointer to application data to be used in the callback.

returnValues:

`NFCSTATUS_INVALID_PARAMETER`	- Non *connection-oriented* socket.
	- This socket is already listening and it is pending.
`NFCSTATUS_INVALID_STATE`	- Socket in other state than socket bound.
`PH_ERR_SUCCESS`	- Operation successful.

3.2.9.7 Accept an incoming connection request

This function is used to accept an incoming connection request the client to accept an incoming connection request. It must be used with the socket provided within the listen callback. The socket is implicitly switched to the connected state when the function is called.

```
phStatus_t phlnLlcp_Transport_Accept(

                void * pDataParams,                                  [In]

                phlnLlcp_Transport_Socket_t * pLlcpSocket,           [In]

                phlnLlcp_Transport_sSocketOptions_t * pOptions,      [In]

                phNfc_sData_t * psWorkingBuffer,                     [In]

                phlnLlcp_TransportSocketErrCb_t pErr_Cb,             [In]

                phlnLlcp_TransportSocketAcceptCb_t pAccept_RspCb,    [In]

                void * pContext );                                   [In]
```

*pDataParams: pointer to `phFriNfc_Llcp_t` parameter component.

*pLlcpSocket: pointer to the transport socket.

*pOptions: options to be used with the socket.

*psWorkingBuffer: a working buffer to be used by the library.

pErr_Cb: error callback function from application that shall be called whenever an error on the socket occurs. The cases when the error callback function is called and error callback type are described in section 3.2.10.1.

pAccept_RspCb: accept connection callback function that shall be called when the connection request from a remote accepted – CC PDU sent. The accept callback type is more precisely described in section 3.2.10.3.

*pContext: pointer to application data to be used in the callback.

returnValues:

NFCSTATUS_INVALID_PARAMETER	- Non *connection-oriented* socket.
NFCSTATUS_INVALID_STATE	- Socket in other state than socket bound.
PH_ERR_SUCCESS	- Operation successful.
Other	- Depending on implementation and underlaying component.

3.2.9.8 Reject a connection request

This function allows the client to reject an incoming connection request. It must be used with the socket provided within the listen callback. The socket is implicitly closed when the function is called.

```
phStatus_t phlnLlcp_Transport_Reject(

            void * pDataParams,                              [In]

            phlnLlcp_Transport_Socket_t * pLlcpSocket,       [In]

            phlnLlcp_TransportSocketRejectCb_t pReject_RspCb,  [In]

            void * pContext );                               [In]
```

*pDataParams: pointer to phFriNfc_Llcp_t parameter component.

*pLlcpSocket: pointer to the transport socket.

pReject_RspCb: the callback to be called when reject operation is completed.

*pContext: pointer to application data to be used in the callback.

returnValues:

NFCSTATUS_INVALID_PARAMETER	- Non *connection-oriented* socket.
NFCSTATUS_INVALID_STATE	- Socket in other state than socket bound.
PH_ERR_SUCCESS	- Operation successful.
Other	- Depending on implementation and underlaying component.

3.2.9.9 Disconnect socket

This function disconnects currently previously connected *connection-oriented* socket by sending DISC PDU on LLC link. Local and the remote SAP shall both be cleared from the socket but the socket itself shall not not closed. Firstly, the socket state is changed to *disconnecting.* If the socket sends or receive data pending, then this pending are resolved before socket disconnected. When the socket disconnected successfully (DM

PDU received and handled internally), then upper layer is notified by the socket disconnect callback and the socket is left in *disconnected state*.

Note: As soon as the socket disconnected, both socket connected and socket disconnected callbacks are set to `NULL`.

```
phStatus_t phlnLlcp_Transport_Disconnect(
            void * pDataParams,                        [In]
            phlnLlcp_Transport_Socket_t * pLlcpSocket,  [In]
            phlnLlcp_SocketDisconnectCb_t pDisconnect_RspCb,  [In]
            void * pContext );                         [In]
```

*pDataParams: pointer to `phFriNfc_Llcp_t` parameter component.

*pLlcpSocket: pointer to the transport socket to be disconnected.

pDisconnect_RspCb: the callback to be called when the disconnection operation is completed. `phlnLlcp_Transport_Disconnect()` function just puts address of this callback to disconnectCB in socket. Actually, the callback is called by `phlnLlcp_Transport_Close()` function (see section 3.2.9.13).

*pContext: pointer to application data to be used in the callback.

returnValues:

`NFCSTATUS_INVALID_PARAMETER`	- Non *connection-oriented* socket.
`NFCSTATUS_INVALID_STATE`	- Socket in not connected state.
`NFCSTATUS_PENDING`	- LLC link status in send pending state. Try later.
`PH_ERR_SUCCESS`	- Operation successful.
Other	Depending on implementation and underlaying component.

3.2.9.10 Send data packed – connection oriented

This function provides sending data via *connection-oriented* LLCP transport socket, which must be already in the *connected* state. To transmit the data the function performs on LLCP layer sending the I PDU frame with header created according to DSAP and SSAP from the socket.

```
phStatus_t phlnLlcp_Transport_Send(
            void * pDataParams,                        [In]
            phlnLlcp_Transport_Socket_t * pLlcpSocket,  [In]
            phNfc_sData_t * pBuffer,                    [In]
            phlnLlcp_TransportSocketSendCb_t pSend_RspCb,  [In]
            void * pContext );                         [In]
```

*pDataParams: pointer to `phFriNfc_Llcp_t` parameter component.

*pLlcpSocket: pointer to the LLCP transport socket, which the data to be sent via.

*pBuffer: buffer containing the data to send. Data are in the format of the `phNfc_sData_t` structure, where n-1 bytes represents the data itself and one byte is length of the data.

UM10721

User manual
COMPANY PUBLIC

Rev. 1.2 — 5 August 2018
270112

31 of 77

pSend_RspCb: callback to be called when the send operation is completed (see section 3.2.10.7).

*pContext: pointer to application data to be used in the callback.

returnValues:

`NFCSTATUS_INVALID_PARAMETER`	- Socket is another than *connection-oriented* type.
	- Socket is not in *connected* state.
	- Data in `psBuffer` to be sent are longer than remote MIU (of the receiver)
`NFCSTATUS_INVALID_STATE`	- The socket is not in a valid state, or not of a valid type to perform the requested operation.
`NFCSTATUS_REJECTED`	- Socket is in send pending state.
`NFCSTATUS_FAILED`	- Operation failed.
`NFCSTATUS_SUCCESS`	- Operation successful.
Other	- Depending on implementation and underlaying component.

3.2.9.11 Receive data from socket

This function is used to read received data from a given socket. It reads at most the size of the reception buffer, but can also return less bytes if less bytes are available. If no data is available, the function will be pending until more data comes, and the response will be sent by the callback. This function can only be called on a *connection-oriented* socket. When the data received successfully, RR frame is sent subsequently to the sender of the received I PDU frame.

Note: Calling this function from API does make the remote to send data.

```
phStatus_t phlnLlcp_Transport_Recv(

              void * pDataParams,                          [In]

              phlnLlcp_Transport_Socket_t * pLlcpSocket,   [In]

              phNfc_sData_t * pBuffer,                     [out]

              phlnLlcp_TransportSocketRecvCb_t pRecv_RspCb, [In]

              void * pContext );                           [In]
```

*pDataParams: pointer to `phFriNfc_Llcp_t` parameter component.

*pLlcpSocket: pointer to the LLCP transport socket, which the received data to be read from.

pRecv_RspCb: callback to be called as soon as receive operation is completed.

*pBuffer: buffer prepared for received data. Data are in the format of the `phNfc_sData_t` structure, where n-1 bytes represents the data itself and one byte refers to the length of the data.

pRecv_RspCb: callback function to be called when received data copied from socket buffer to output `pBuffer`. If socket is in pending receive, this callback is stored to socket component receive callback `pLlcpSocket->pfSocketRecv_Cb`.

*pContext: pointer to application data to be used in the callback.

returnValues:

NFCSTATUS_SUCCESS	- Operation successful.
NFCSTATUS_INVALID_PARAMETER	- Socket is another than *connection-oriented* type.
NFCSTATUS_REJECTED	- Socket is already in receive pending.
NFCSTATUS_FAILED	- Operation failed.
	- Socket not connected.

3.2.9.12 Send data - connectionless

This function provides sending data via *connectionless* LLCP transport socket, which must be already in the *connected* state. To transmit the data the function performs on LLCP layer sending the UI PDU frame with header created according to DSAP and *SSAP* from the socket.

```
phStatus_t phlnLlcp_Transport_SendTo(

        void * pDataParams,                              [In]

        phlnLlcp_Transport_Socket_t * pLlcpSocket,       [In]

        uint8_t nSap,                                    [In]

        phNfc_sData_t * pBuffer,                         [In]

        phlnLlcp_TransportSocketSendCb_t pSend_RspCb,    [In]

        void * pContext );                               [In]
```

*pDataParams: pointer to `phFriNfc_Llcp_t` parameter component.

*pLlcpSocket: pointer to the LLCP transport socket, which the data to be sent via.

nSap: destination SAP that the data to be sent to.

*pBuffer: buffer containing the data to send. Data are in the format of the `phNfc_sData_t` structure, where n-1 bytes represents the data itself and the one byte is length of the data.

pSend_RspCb: callback to be called when the send operation is completed.

*pContext: pointer to application data to be used in the callback.

returnValues:

NFCSTATUS_SUCCESS	- Operation successful.
NFCSTATUS_INVALID_PARAMETER	- Socket is another than *connectionless* type.
	- Socket is not *bound*.
-	Data in `psBuffer` to be sent are longer than MIU of the link.
	- `nSap` out of range 2 - 63
NFCSTATUS_INVALID_STATE	- Socket not in *connectionless* state.
NFCSTATUS_REJECTED	- Socket is in send pending state.
	- Socket is in error status.
NFCSTATUS_FAILED	- Operation failed.

3.2.9.13 Close one socket

This function closes a given LLCP transport *connection-oriented* or *connectionless* socket previously created by `phFriNfc_LlcpTransport_Socket()` function. If the socket was connected, it is first disconnected, and then closed.

This function manages closing of given previously created socket whether c*onnection oriented* or c*onnectionless.* If a socket has been already connected, then disconnect from LLC link by sending DICS PDU is performed automatically. If *connection-oriented* socked has not been *connected* yet, it is closed by abort (Accept, Connect CBs are executed, then set to NULL).

```
phStatus_t phlnLlcp_Transport_Close(
                    void * pDataParams                       [In]
                    phlnLlcp_Transport_Socket_t * pLlcpSocket );    [In]
```

*pDataParams: pointer to `phlnLlcp_Fri_DataParams_t` parameter component.

*pLlcpSocket: pointer to the transport socket.

returnValues:

`NFCSTATUS_SUCCESS`	- Operation successful.
Other	- Depending on implementation and underlaying component.

3.2.9.14 Close all the sockets

This function closes all the created sockets independently of their current states. In addition, information from the `pCachedServiceName` is entirely cleared.

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

*pDataParams: pointer to `phlnLlcp_Fri_DataParams_t` parameter component.

returnValues:

`NFCSTATUS_SUCCESS`	- Operation successful.
Other	- Depending on implementation and underlaying component.

3.2.10 LLCP Socket Callbacks

3.2.10.1 Error CB

Called at event: Error on LLCP link occured. Origin of the error is passed like the second input argument by the caller from within the library.

Set by: `phlnLlcp_Transport_Socket()` or `phlnLlcp_Transport_Accept()`

Function prototype:

```
typedef void (*pphFriNfc_LlcpTransportSocketErrCb_t) ( void* pContext,
                                                uint8_t nErrCode);
```

*pContext: pointer to user data passed to processed in the CB

nErrCode: value tells, what is the error caused by.

UM10721

© NXP B.V. 2018. All rights reserved.

User manual
COMPANY PUBLIC

Rev. 1.2 — 5 August 2018
270112

34 of 77

`PHFRINFC_LLCP_ERR_NOT_BUSY_CONDITION` – RR acknowledgement received from the remote after negative acknowledgement (RNR) received before.

`PHFRINFC_LLCP_ERR_BUSY_CONDITION` – negative acknowledgement (RNR PDU) from the remote received.

`PHFRINFC_LLCP_ERR_FRAME_REJECTED` – the remote received an invalid packet and subsequently sent FRMR frame.

`PHFRINFC_LLCP_ERR_DISCONNECTED` - disconnection request (DISC PDU) from the remote received.

3.2.10.2 Listen CB

Called at event: Incoming connection request (CONNECT PDU) from a client received.

Set by: `phlnLlcp_Transport_Listen()`

Function prototype:

```
void (*pphFriNfc_LlcpTransportSocketListenCb_t) (void* pContext,
                               phFriNfc_LlcpTransport_Socket_t *IncomingSocket);
```

*pContext: pointer to user data passed to processed in the CB

*IncomingSocket; pointer to socket that bound to the SAP or service name that the remote is requesting to connect with.

3.2.10.3 Connect CB

Called at event: The local has just sent data packet (I PDU) to the remote.

Set by: `phlnLlcp_Transport_Connect()`

Function prototype:

```
typedef void (*pphFriNfc_LlcpTransportSocketConnectCb_t) (void* pContext,
                                           uint8_t nErrCode,
                                           NFCSTATUS status);
```

*pContext: pointer to user data passed to processed in the CB

nError: error code defined by NFC forum LLCP Disconnect mode

status: refers to the status of connection procedure. Following values may be passed in

`NFCSTATUS_SUCCESS`: connection successful. Once the connection established, the data transmission may be performed.

`NFCSTATUS_ABORTED`: socket closed by `phlnLlcp_Transport_Close()` or as along with frame reject.

`NFCSTATUS_FAILED`: the connection not confirmed on the remote side. Connection failed and not created.

3.2.10.4 Disconnect CB

Called at event: The local has just sent data packet (I PDU) to the remote.

Set by: `phlnLlcp_Transport_Disconnect()`

Function prototype:

```
typedef void (*pphFriNfc_LlcpTransportSocketDisconnectCb_t) ( void* pContext,
```

```
                                                              NFCSTATUS status);
```

*pContext: pointer to user data passed to processed in the CB

status: refers to the status of disconnection procedure. Following values may be passed:

`NFCSTATUS_SUCCESS`: disconnection has been confirmed by the remote side.

3.2.10.5 Accept CB

<u>**Called at event**</u>: The local server has just confirmed (sent CC PDU) the connection request (CONNECT PDU) from the remote client.

<u>**Set by**</u>: `phlnLlcp_Transport_Accept()`

<u>**Function prototype**</u>:

```
void (*pphFriNfc_LlcpTransportSocketAcceptCb_t) (void* pContext,
                                                     NFCSTATUS status);
```

*pContext: pointer to user data passed to processed in the CB

status: refers the status of acceptance procedure of connection request. If the socket has been connected successfully, status is 0 or `NFCSTATUS_SUCCESS`.

3.2.10.6 Reject CB

<u>**Called at event**</u>: The local has just received a data packet (I PDU) to the remote.

<u>**Set by**</u>: `phlnLlcp_Transport_Reject()`

<u>**Function prototype**</u>:

```
typedef void (*pphFriNfc_LlcpTransportSocketRejectCb_t) (void* pContext,
                                                             NFCSTATUS status);
```

*pContext: pointer to user data passed to processed in the CB

status: refers the status of rejection procedure of disconnection request. If the DM has been sent successfully, status is 0 or `NFCSTATUS_SUCCESS`.

3.2.10.7 Send CB

<u>**Called at event**</u>: The local has just sent data the packet (I PDU) to the remote.

<u>**Set by**</u>: **`phlnLlcp_Transport_Send`**`()` or `phlnLlcp_Transport_SendTo()`

<u>**Function prototype**</u>:

```
typedef void (*pphFriNfc_LlcpTransportSocketSendCb_t) (void* pContext,
                                                           NFCSTATUS status);
```

*pContext: pointer to user data passed to processed in the CB

status: refers the status of acceptance procedure of connection request. If the data (I PDU frame) has been sent successfully via the socket, status is 0 or `NFCSTATUS_SUCCESS`.

3.2.10.8 Receive CB – connection oriented

<u>**Called at event**</u>: The local has just received a data packet (I PDU) from the remote in connection-oriented mode.

<u>**Set by**</u>: **`phlnLlcp_Transport_Send(`**

<u>**Function prototype**</u>:

UM10721

All information provided in this document is subject to legal disclaimers.

© NXP B.V. 2018. All rights reserved.

User manual
COMPANY PUBLIC

Rev. 1.2 — 5 August 2018
270112

36 of 77

```
typedef void (*pphFriNfc_LlcpTransportSocketRecvCb_t) ( void* pContext,
                                                         NFCSTATUS status);
```

*pContext: pointer to user data passed to processed in the CB

status: refers to the status of reception procedure. If the incoming packet has been received without an error in accordance with LLCP, status is 0 or `NFCSTATUS_SUCCESS`.

3.2.10.9 Receive CB - connectionless

Receive from is notification about information packet has just been received from a remote in connectionless mode.

<u>**Called at event**</u>: The local has just received data packet (UI PDU) from the remote in connectionless mode.

<u>**Set by**</u>:

<u>**Function prototype**</u>:

```
typedef void (*pphFriNfc_LlcpTransportSocketRecvFromCb_t) ( void* pContext,
                                                            uint8_t ssap,
                                                            NFCSTATUS status);
```

*pContext: pointer to user data passed to processed in the CB

ssap: source SAP the data coming from

status: refers to the status of reception procedure. If the incoming packet has been received without an error in accordance with LLCP, status is 0 or `NFCSTATUS_SUCCESS`.

3.3 Protocol layer - ISO18092 protocol commands

All the LLCP packets must be encapsulated regardless of their purpose on the Link layer (data carrying packets and link maintenance packets as well). ISO18092 protocol ensures encapsulating link layer packets into final binary format which is ready to be transmitted via RF. NXP P2P Library version supports just Passive Initiator mode.

Since the LLCP is acknowledgement-based communication, ISO18092 PAL exchange function awaits response from another peer within defined time period. Thus, an MCU implementing the P2P Library sends ISO18092 request commands subsequently waits for a response coming from Target. But the MCU is not able to listen to requests from another Initiator. Wake Up Request and Response are not implemented in the P2P Library.

For a developer implementing the LLCP APIs there is no way how to influence this layer.

The ISO18092 standard protocol parameters and values are mostly taken from NFC Forum-TS-Digital Protocol-1.0.[14]

All the defines representing those parameter values are defined in the library folder *NxpRdLib_PublicRelease/comps/phpalI18092mPI/src/Sw* in file *phpalI18092mPI_Sw_Int*.

3.3.1 ISO18092 PAL parameter component

PAL layer component uses `phpalI18092mPI_Sw_DataParams_t` parameter structure to store important ISO18092 protocol attributes. The structure is defined in *NxpRdLib_PublicRelease/intfs/phpalI18092mPI.h*. Important members of the structure are listed in Table 3. An instance of this structure is called ISO18092 PAL component.

Table 3. Some parameters from ISO18092 Pal component

In the right column there are default values given by Initialization function (see section 3.3.2). Apart from these parameters, there are few others for internal management.

parameter	description	default init value
pHalDataParams	Pointer to the HAL parameter component of the underlaying HAL layer.	input parameter Init (section 3.3.2)
NFC ID3	NFC ID 10 bytes long	input parameter ATR_REQ (section 3.3.4)
DID	Device identifier	NULL
NAD enabled	NAD enabler	PH_OFF
NAD	Node Address	NULL
WT	Waiting timeout for a target	14 ()
FSL	Frame length	0 (means 64 bytes)
PNI	Packet number	NULL
DSi	Divisor Send from Parameter Select Request (Initiator to target)	NULL (106kbit/s)
DRi	Divisor Receive initiator	NULL (106kbit/s)
bMaxRetryCount	Number of attempts to send a Request despite no Response from a Target. The limit is common for all the types of Requests.	2

3.3.2 Protocol initialization

This function is used to initialize `phpalI18092mPI_Sw_DataParams_t` parameter component. Thus, it should be called as the first among the ISO18092 related functions (section 3.3). It calls `phpalI18092mPI_ResetProtocol()` function and registers a pointer to a component of the undelaying HAL layer.

```
phStatus_t phpalI18092mPI_Sw_Init(

          phpalI18092mPI_Sw_DataParams_t * pDataParams,        [In]

          uint16_t wSizeOfDataParams,                          [In]

          void * pHalDataParams );                             [In]
```

*pDataParams: pointer to `phpalI18092mPI_Sw_DataParams_t` parameter component.

wSizeOfDataParams: size of the parameter component. It is highly recommended to use built in C function `sizeof(wSizeOfDataParams)`.

*pHalDataParams: pointer to the component of the underlaying hardware layer. This component is bound to the used PCD.

returnValues:

`PH_ERR_SUCCESS` - Operation successful.

UM10721

All information provided in this document is subject to legal disclaimers.

© NXP B.V. 2018. All rights reserved.

User manual
COMPANY PUBLIC

Rev. 1.2 — 5 August 2018
270112

38 of 77

3.3.3 Reset Protocol

This function reset values of given ISO18092mPI PAL parameter component to zero or default values.

Frame length FSL is set to 64 bytes.

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

*pDataParams: pointer to `phpalI18092mPI_Sw_DataParams_t` parameter component.

returnValues:

`PH_ERR_SUCCESS` - Operation successful.

3.3.4 Attribute Request

This function performs ISO18092 Attribute Request command then listens to Attribute Response from the Target. DID and NAD parameters from Target's Attribute Response are verified whether they agree with Initiator side. Initiator timeout value is set according to TO parameter from Attribute Response. After this function finishes successfully, in the ISO18092 PAL parameter component there are stored parameters of NFC communication.

In the P2P Library this function is implemented in the Discovery Loop to detect any of NFC P2P tags either of types F or types A.

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

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

*pDataParams: pointer to `phpalI18092mPI_Sw_DataParams_t` parameter component.

*pNfcid3i: NFCID3 - randomly generated by application in case of 106kbit/s initial data rate or NFCID2 from a target in case of 212 or 424kbit/s data rate. In the Discovery Loop there is UID from `phacDiscLoop_Sw_Int_DetectA()` or ID+PM from `phacDiscLoop_Sw_Int_DetectF()` used as NFCID2. The P2P Library does not provide NFCID3 random generator.

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

UM10721

All information provided in this document is subject to legal disclaimers.

© NXP B.V. 2018. All rights reserved.

User manual **Rev. 1.2 — 5 August 2018** **39 of 77**
COMPANY PUBLIC 270112

bLri: Length Reduction of the Transport Data on the Initiator side. Put into this parameter one defines from the third column of Table 4. If this LRI differs from one received from Target Attribute Response, then smaller among them is used in the NFC communication. Negotiated Length Reduction value retains as FSL in `phpalI18092mPI_Sw_DataParams_t` parameter component.

Table 4.   Table of Length Reduction values
The first column refers to LRi bits how they are placed in the Protocol Parameter Initiator byte or FSL byte defined by ISO18092 standard.
The second column refers number of bytes that he Initiator shall send in the Transport Data field within DEP.
In the third column there are identifiers from the P2P Library analogous to the previous columns.

LRi bits ISO18092	Max byte count in the DEP Transport Data	P2P Library identifier
00	64 bytes	`PHPAL_I18092MPI_FRAMESIZE_64`
01	128 bytes	`PHPAL_I18092MPI_FRAMESIZE_128`
10	192 bytes	`PHPAL_I18092MPI_FRAMESIZE_192`
11	254 bytes	`PHPAL_I18092MPI_FRAMESIZE_254`

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

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

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

bGiLength: number of General Information bytes sent by the Initiator. The upper limit is 48 bytes according [14] section 14.6.1.1. The value `PHPAL_I18092MPI_MAX_GI_LENGTH` is defined in *NxpRdLib_PublicRelease/intfs* in *phpalI18092mPI.h* file.

*pAtrRes: pointer to Attribute Response from the Target.

*pAtrResLength: Attribute Response length.

returnValues:

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

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

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

`PH_ERR_SUCCESS` - Operation successful.

Other - Depending on implementation and underlaying component.

3.3.5 Parameter Selection

This function provides ISO18092 initiator defined Parameter Selection Request, then listens to the Parameter Selection Response from the Target.

```
phStatus_t phpalI18092mPI_Psl(

            void * pDataParams,              [In]

            uint8_t bDsi,                    [In]

            uint8_t bDri,                    [In]

            uint8_t bFsl );                  [In]
```

*pDataParams: pointer to `phpalI18092mPI_Sw_DataParams_t` parameter component.

bDsi: Divisor Send (data rate Send by Target to Initiator). Only values from Table 5 (column *P2P Library identifier*) are legal.

Table 5.   Table of Divisor Send/Receive

ISO18092	ISO18092 bit duration Divisor D	Bit rate kbit/s	ISO18092 Divisor D	P2P Library identifier
0	1	106	1	`PHPAL_I18092MPI_DATARATE_106`
1	2	212	2	`PHPAL_I18092MPI_DATARATE_212`
2	4	424	4	`PHPAL_I18092MPI_DATARATE_424`

bDri: Divisor Receive (data Received by Initiator from Target). Only values from Table 5 (column *P2P Library identifier*) are legal.

bFsl: Frame Size of the Transport Data. Put into this parameter one P2P Library identifier from the third column of Table 4 representing Length Reduction Initiator.

returnValues:

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

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

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

`PH_ERR_SUCCESS` - Operation successful.

Other - Depending on implementation and underlaying component.

3.3.6 Activate Card

This function just integrates Attribute request and Parameter Selection respectively.

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

*pDataParams: pointer to `phpalI18092mPI_Sw_DataParams_t` parameter component.

*pNfcid3i: NFCID3 - randomly generated in case of 106kbps initial data rate or NFCID2 in case of 212/424kbps data rate.

bDid: Device Identifier. DID is used for multiple data transport protocol activation with more than one target. If zero, then DID shall be. Value must be in range from 0 to 14.

bDsi: Divisor Send (target to initiator). Only values from are allowed:

`PHPAL_I18092MPI_DATARATE_106`

`PHPAL_I18092MPI_DATARATE_212`

`PHPAL_I18092MPI_DATARATE_424`

where last three number refers to data rate in kbit/s.

bDri: Divisor Receive (initiator to target). Only values from are allowed:

`PHPAL_I18092MPI_DATARATE_106`

`PHPAL_I18092MPI_DATARATE_212`

`PHPAL_I18092MPI_DATARATE_424`

where last three number refers to data rate in kbit/s.

bFsl: Frame Length of the Transport Data. Put into this parameter one defines from the third column of Table 4.Frame Length table of values

bNadEnable: enable usage of Node Address. If equal to zero, then NAD is disabled.

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

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

bGiLength: number of General Information bytes sent by the Initiator. Must be less or equal to `PHPAL_I18092MPI_MAX_GI_LENGTH` defined in *NxpRdLib_PublicRelease/intfs* in *phpalI18092mPI.h* file.

*pAtrRes: pointer to Attribute Response from the Target.

*pAtrResLength: Attribute Response length.

`PH PH_ERR_INVALID_PARAMETER` — `bDid,` `bDsi,` `bDri` or `bFsl` value out of .valid range.

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

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

`PH_ERR_SUCCESS` - Operation successful.

Other - Depending on implementation and underlaying component.

3.3.7 Deselect

This function performs ISO18092 defined either Deselect Request or Release Request, then waits for either Deselect Response or Release Response.

```
phStatus_t phpalI18092mPI_Deselect(

                void * pDataParams,              [In]

                uint8_t bDeselectCommand );      [In]
```

*pDataParams: pointer to `phpalI18092mPI_Sw_DataParams_t` parameter component.

bDeselectCommand: Request to send, either `PHPAL_I18092MPI_DESELECT_DSL` for Deselect command or `PHPAL_I18092MPI_DESELECT_RLS` for Release command.

returnValues:

UM10721

All information provided in this document is subject to legal disclaimers.

© NXP B.V. 2018. All rights reserved.

User manual
COMPANY PUBLIC

Rev. 1.2 — 5 August 2018
270112

42 of 77

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

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

`PH_ERR_SUCCESS` - Operation successful.

Other - Depending on implementation and underlaying component.

3.3.8 Presence check

This function performs presence check for current target whether it is still in the RF field range. Firstly, PCD sends ISO18092 defined Supervisory Attention PDU, then listens to Supervisory Attention Response from the Target.

This function performs ISO18092 defined Data Exchange Request, then waits for Data Exchange Response. All the LLCP packets communicated between the initiator and the target are transmitted by this function.

```
phStatus_t phpalI18092mPI_PresCheck(

                void * pDataParams );                    [In]
```

*pDataParams: pointer to `phpalI18092mPI_Sw_DataParams_t` parameter component.

returnValues:

`PH_ERR_SUCCESS` - Operation successful.

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

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

`PHPAL_I18092MPI_ERR_RECOVERY_FAILED` - Recovery failed, target does not respond any more.

`PH_ERR_SUCCESS` - Operation successful.

Other - Depending on implementation and underlaying component.

3.3.9 Exchange

This function performs ISO18092 defined Data Exchange Request, then waits for Data Exchange Response. All the LLCP packets communicated between the initiator and the target are transmitted by this function.

```
phStatus_t phpalI18092mPI_Exchange(

                void * pDataParams,         [In]

                uint16_t wOption,           [In]

                uint8_t * pTxBuffer,        [In]

                uint16_t wTxLength,         [In]

                uint8_t ** ppRxBuffer,      [Out]

                uint16_t * pRxLength );     [Out]
```

*pDataParams: pointer to `phpalI18092mPI_Sw_DataParams_t` parameter component.

wOption: option parameter for internal method how to send DEP frame sequence. OR `wOption` parameter with any of above defines.

`PH_EXCHANGE_BUFFERED_BIT`, `PH_EXCHANGE_LEAVE_BUFFER_BIT` are advanced buffer methods related to HAL buffer.

`PH_EXCHANGE_TXCHAINING`, `PH_EXCHANGE_RXCHAINING`, `PH_EXCHANGE_RXCHAINING_BUFSIZE` mean ISO18092 frame chaining. But this does not need to be set. If the data to be exchanged is more than Frame Size configured by `phpalI18092mPI_Atr()` Length Reduction or `phpalI18092mPI_Psl()`, this function performs the chaining automatically. In result frame of any size is transmitted correctly.

`PH_EXCHANGE_DEFAULT` is sufficient to perform NFC P2P correctly.

For better explanation see *NxpRdLib_PublicRelease/types/ph_Status.h*.

*pTxBuffer: data to transmit from the initiator to target.

wTxLength: length of data to transmit.

**ppRxBuffer: pointer to received data.

*pRxLength: number of received data bytes.

returnValues:

`PH_ERR_INVALID_PARAMETER` – invalid flag bit in `xOption` used.

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

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

`PHPAL_I18092MPI_ERR_RECOVERY_FAILED` - Recovery failed, target does not respond any more.

`PH_ERR_SUCCESS` - Operation successful.

Other - Depending on implementation and underlaying component.

3.3.10 Get serial Number

This function retrieves the serial number NFCID3 from the PAL component. The serial number must have been previously retrieved by `phpalI18092mPI_Atr()` function.

```
phStatus_t phpalI18092mPI_GetSerialNo(
            void * pDataParams,                    [In]
            uint8_t * pNfcId3Out );                [Out]
```

*pDataParams: pointer to `phpalI18092mPI_Sw_DataParams_t` parameter component.

*pNfcId3Out: the latest NFCID3 registered into PAL `pDataParams` component.

returnValues:

`PH_ERR_SUCCESS` - Operation successful.

`PH_ERR_USE_CONDITION` – NFCID3 value is invalid.

3.3.11 Get protocol parameter

Developer may use this function to get parameter value of the ISO18092 PAL component.

```
phStatus_t phpalI18092mPI_GetConfig(
            void * pDataParams,             [In]
            uint16_t wConfig,               [In]
            uint16_t wValue );              [Out]
```

*pDataParams: pointer to `phpalI18092mPI_Sw_DataParams_t` parameter component.

wConfig: identifier of the parameter to be read from `phpalI18092mPI_Sw_DataParams_t` parameter component. Parameters that can be read are listed in Table 6.

Table 6.   P2P Library identifiers of the ISO18092 parameters

Parameter	ISO18092 acronym	P2P Library identifier	Maximal value
Packet number	PNi	`PHPAL_I18092MPI_CONFIG_PACKETNO`	3
Device ID	DID	`PHPAL_I18092MPI_CONFIG_DID`	14
Node Address	NAD	`PHPAL_I18092MPI_CONFIG_NAD`	255
Target Timeout	TO	`PHPAL_I18092MPI_CONFIG_WT`	14
Frame Length	FSL	`PHPAL_I18092MPI_CONFIG_FSL`	3
Retry Count		`PHPAL_I18092MPI_CONFIG_MAXRETRYCOUNT`	5

wValue: value of the read parameter.

returnValues:

`PH_ERR_UNSUPPORTED_PARAMETER` – unknown parameter `wConfig` or cannot be modified.

`PH_ERR_SUCCESS` - Operation successful.

3.3.12 Set protocol parameter

Developer may use this function to set parameter value of the ISO18092 PAL component. In addition, validity check of parameter value is done internally. The updated value shall be afterward used in the communication protocol.

```
phStatus_t phpalI18092mPI_SetConfig(

              void * pDataParams,                    [In]

              uint16_t wConfig,                      [In]

              uint16_t wValue );                     [In]
```

*pDataParams: pointer to `phpalI18092mPI_Sw_DataParams_t` parameter component.

wConfig: identifier of the parameter to be set in `phpalI18092mPI_Sw_DataParams_t` parameter component. Parameters that can be set are listed in Table 6.

wValue: value to be written into parameter. See value limits in Table 6 for each parameter.

returnValues:

`PH_ERR_INVALID_PARAMETER` – illegal value of parameter `wValue` (maximal values in Table 6).

`PH_ERR_UNSUPPORTED_PARAMETER` – unknown parameter `wConfig` or cannot be modified.

`PH_ERR_SUCCESS` - Operation successful.

3.4 OSAL

OSAL module provides basic OS services like dynamic memory allocation and handling hardware timers.

3.4.1 Allocate memory

This function allocates free memory from heap segment. If desired amount of free memory found successfully, pointer to allocated memory is returned. In fact, the build in C function `malloc()` from *stdlib.h* is called.

UM10721

© NXP B.V. 2018. All rights reserved.

User manual
COMPANY PUBLIC

Rev. 1.2 — 5 August 2018
270112

45 of 77

```
phStatus_t phOsal_GetMemory(

                 void * pDataParams,          [In]

                 uint32_t dwLength,           [In]

                 void ** pMem );              [Out]
```

*pDataParams: pointer to `phOsal_Lpc12xx_DataParams_t` parameter component.

dwLength:

**pMem: pointer to memory allocated.

returnValues:

`PH_ERR_SUCCESS` - Operation successful.

`PH_ERR_RESOURCE_ERROR` - Requested memory space allocation failed.

3.4.2 Free memory

This function frees previously allocated memory. In fact, the build in C function `free()` is called.

```
phStatus_t phOsal_FreeMemory(

                 void * pDataParams,      [In]

                 void * ptr );            [In]
```

*pDataParams: pointer to `phOsal_Lpc12xx_DataParams_t` parameter component.

*ptr: pointer to memory to be freed.

returnValues:

`PH_ERR_SUCCESS` - Operation successful.

3.4.3 Timer services

There are two 32 bit hardware timers in the LPC1227 microcontroller. The OSAL module provides utilization of HW timers in two ways: SW time delay and general timer usage

General Timer usade

The P2P Library uses 32 bit hardware timers of LCP1227 to cause forced time delay in thread and one timer interrupt events.

The Discovery Loop performs time delay after setting the reader chip for particular NFC protocol or for guard interval between detection of B and F tag type.

Before the LLCP link activation (section 3.2.7.3) there must be at least one hardware timer free for the purpose of LLCP SYMM timer. The timer is released by link deactivation – `phlnLlcp_Deactivate` (section 3.2.7.4).

Note: On the PAL and HAL layer while the initiator waits for the response from the Target, instead of MCU timers there are internal hardware timers of attached reader chip used for measuring the timeouts defined by particular NFC protocol.

3.4.4 Timer Init

This function is necessary prerequisite for any timer usage. Firstly, it makes software configurations. It initiates timer parameter component `phOsal_Lpc12xx_DataParams_t` and internal software timer structures aimed for storage of important information of particular

UM10721

© NXP B.V. 2018. All rights reserved.

User manual **Rev. 1.2 — 5 August 2018** **46 of 77**
COMPANY PUBLIC 270112

hardware timers. Once this function executed all the timers related can be run. After all the software issues initiated, timer interrupts are enabled.

```
phStatus_t phOsal_Lpc12xx_Timer_Init(

                phOsal_Lpc12xx_DataParams_t * pDataParams );          [In]
```

returnValues:

`PH_ERR_SUCCESS` - Operation successful.

3.4.5 Timer Create

This function assigns an unused hardware timer of MCU LPC1227. Number of hardware timers that can be assigned this way is limited to 2. If both the timers are currently in use, then no timer is assigned. Once the timer created, it can be used for counting by `phOsal_Timer_Start()` and `phOsal_Timer_Stop()`. Counterpart to this function is `phOsal_Timer_Delete` (section 3.4.8), which releases the timer as free.

Internally, an array is maintained which stores timers along with information as to whether the timer is available or not. This function searches a free timer that is available and returns the timer ID.

<u>Note</u>: Timer 0 is used by the NXP P2P library as LLCP LTO timer.

```
phStatus_t phOsal_Timer_Create(

                void * pDataParams,                            [In]

                uint32_t *timerId );                           [Out]
```

*pDataParams: pointer to `phOsal_Lpc12xx_DataParams_t` parameter component.

*timerId: ID of assigned timer. If no free timer has been found, then this parameter is returned with value `PH_OSALNFC_INVALID_TIMER_ID` equal to 0xFFFF.

returnValues:

`PH_OSAL_ERR_NO_FREE_TIMER`   - Both the timers are currently in use.

`PH_ERR_SUCCESS`   - Operation successful.

3.4.6 Timer Start

Timer starts counting. When the timer expires after given time amount, then given application callback function is executed. The counterpart to this function is `phOsal_Timer_Stop()` function (section 3.4.7). The timer must be created by `phOsal_Timer_Create()` (section 3.4.5) before started or stopped timer may be restarted.

```
phStatus_t phOsal_Timer_Start(

                void * pDataParams,                            [In]

                uint32_t dwTimerId,                            [In]

                uint32_t dwRegTimeCnt,                         [In]

                ppCallBck_t pApplication_callback, ,           [In]

                void  * pContext ); ,                          [In]
```

*pDataParams: pointer to `phOsal_Lpc12xx_DataParams_t` parameter component.

*timerId: valid timer ID as returned by `phOsal_Timer_Create()`.

dwRegTimeCnt: required amount of time in MS, after which the timer expires.

UM10721

User manual **Rev. 1.2 — 5 August 2018** **47 of 77**
COMPANY PUBLIC 270112

pApplication_callback: pointer to the callback function that will be called once timer expires.

The user defined function must satisfy the prototype of callback function:

```
void (*ppCallBck_t)(uint32_t TimerId, void *pContext);
```

*pContext: argument with which the call back function will be called.

returnValues:

`PH_OSAL_ERR_INVALID_TIMER` – requesting non-existing timer or the timer has not been created before.

`PH_ERR_SUCCESS` - Operation successful.

3.4.7 Timer Stop

Stop the given timer. This function does not free the timer. It only disables the timer. Use `phOsal_Timer_Delete()` to free the timer. The timer may be restarted by `phOsal_Timer_Start()`.

```
phStatus_t phOsal_Timer_Stop(
                void * pDataParams,                        [In]
                uint32_t dwTimerId );                      [In]
```

*pDataParams: pointer to `phOsal_Lpc12xx_DataParams_t` parameter component.

*timerId: ID of the timer to be stopped.

returnValues:

`PH_OSAL_ERR_INVALID_TIMER` – Attempt to stop non-existing timer.

`PH_ERR_SUCCESS` - Operation successful.

3.4.8 Timer Delete

This function returns the timer with given ID to the free timer pool. Content of the timer component is completely forgotten (erased). In addition, if the timer is running, this function stops it also therefore it does not need to be stopped by `phOsal_Timer_Stop()`. This function is counterpart to `phOsal_Timer_Create()` (section 3.4.5).

```
phStatus_t phOsal_Timer_Delete(
                void * pDataParams,                        [In]
                uint32_t dwTimerId );                      [In]
```

*pDataParams: pointer to `phOsal_Lpc12xx_DataParams_t` parameter component.

*timerId: ID of the timer to be released to free timer pool.

returnValues:

`PH_OSAL_ERR_INVALID_TIMER` – attempt to delete non-existing timer

`PH_ERR_SUCCESS` - Operation successful.

3.4.9 Timer Wait

This function freezes the thread for a given amount of time determined by both the value and time unit. While the thread is being frozen nothing else is being executed within the thread. This runs function differs from `phOsal_Timer_Start()`. The timer triggered by

UM10721

User manual
COMPANY PUBLIC

Rev. 1.2 — 5 August 2018
270112

48 of 77

`phOsal_Timer_Start()` function runs concurrently with the thread without affecting execution of the thread (unless timer expiration).

Note: The NXP P2P Library uses always uses hardware timer 1 for thread wait delay performed by this function. The hardware timer is not even checked whether free or currently used for any other purpose.

```
phStatus_t phOsal_Timer_Wait(
                void * pDataParams,                        [In]
                uint8_t bTimerDelayUnit,                   [In]
                uint16_t wDelay );                         [In]
```

*pDataParams: pointer to `phOsal_Lpc12xx_DataParams_t` parameter component.

bTimerDelayUnit: time units. Identifier `PH_OSAL_TIMER_UNIT_MS` for milliseconds and `PH_OSAL_TIMER_UNIT_US` for microseconds.

wDelay: amount of time. This parameter together with `bTimerDelayUnit` determines the length of delay.

returnValues:

`PH_ERR_SUCCESS` - Operation successful.

4. Sample code

The section 4.1 shows how the P2P Library APIs described in sections 3.2.7 and 3.2.9 can be implemented into functions of an application layer. There fragments of C code taken from *SnepClient.c.*

Functions from section 4.1 are implemented into the SNEP client application. The SNEP client performs only Put request [15]. It is fully described in section 4.1. The SENP server may be performed by any NFC mobile device with Android platform (4.0 and later). In result, after application runs a picture should be received and displayed on mobile devices' screen.

In the entire section 4 we will use several terms representing only two devices. One device is board with LPC1227 with reader chip extension (Blueboard 2.1). Name we use to point particular device depends on layer point of view s presented in Table 7.

Table 7.   Two peers and their names as called in this section

point of view		peer's name
hardware/device	mobile device (with Android)	LPC1227 (MCU)
SNEP	client	server
LLCP layer	local	remote
ISO18092	initiator	Target

4.1 Implementation of the LLCP API

4.1.1 Global variables

Firstly, there will be introduced global variables that shall be used by the SNEP client and underlaying LLCP layer. All the global variables from this section are declared in *src/SnepClient.c.*

4.1.2 Data structures used by the P2P library

Components (instances) of the data structures used by the P2P Library need to be declared.

```
phNfc_sData_t sData;

phFriNfc_LlcpTransport_t      LlcpTransport; /**< LLCP transport layer component */

phFriNfc_Llcp_sLinkParameters_t  LinkParam;      /**< LLCP link parameter */

phFriNfc_Llcp_t               Llcp;            /**< LLCP pointer */

phHal_sRemoteDevInformation_t  RemoteInfo;      /**< Remote Info component */

phLibNfc_Llcp_sSocketOptions_t  sOptions = {128, 1}; /**< LLCP options */
```

Working buffer shall be assigned to a socket.

```
phNfc_sData_t                 sWorkingBuffer = {bLLCP_WorkingBuffer,
sizeof(bLLCP_WorkingBuffer)};
```

LLCP basic (uppermost) parameter component see section .

```
phlnLlcp_Fri_DataParams_t     lnLlcpDataparams;
```

Component of the socket. Shall be used for socket client.

```
phFriNfc_LlcpTransport_Socket_t *pSocket_Client;
```

ISO18092 parameter component (see section).

```
phpalI18092mPI_Sw_DataParams_t  palI18092mPI;     /**< PAL MPI component */
```

OSAL component necessary for usage of the SYMM timer.

```
phOsal_Lpc12xx_DataParams_t    osal;            /**< OSAL component holder */

phacDiscLoop_Sw_DataParams_t   discLoop;        /**< Discovery loop component */
```

4.1.3 Buffers

SNEP buffers are not mentioned in this section but they are used in the SNEP client application.

```
uint8_t bSnepRx[8];                              /**< SNEP RX buffer */

uint8_t bSnepTx[128];                            /**< SNEP TX buffer */

uint8_t bRxBuffer[256];                          /**< LLCP TX buffer */

uint8_t bTxBuffer[256];                          /**< LLCP RX buffer */
```

Working buffer is shall be assigned to socket communicating with the SNEP server via. It is dedicated for temporary storage of incoming data. Pointer to the working buffer structure needs to be passed as the fourth input parameter into `ph_lnLlcp_Transport_Socket()` (see section 3.2.9.1).

```
uint8_t bLLCP_WorkingBuffer[800];                /**< LLCP working buffer */

phNfc_sData_t  sWorkingBuffer = {bLLCP_WorkingBuffer, sizeof(bLLCP_WorkingBuffer)};
```

4.1.4 Application layer LLCP flags

Application layer LLCP flags are used indicate a state that the LLCP layer is currently in.

This flag shall be modified at the beginning of the link initialization.

```
uint8_t Llcp_running;                          /**< Flag indicating whether LLCP
                                                   is running OR not */
```

This flag is shall be modified by application connect CB function.

```
uint8_t SocketConnected;                       /**< Flag indicating socket
                                                   connection */
```

This flag shall be modified immediately after LLC link activation

```
uint8_t Link_Activated = 0;                    /**< Flag indicating link
                                                   activation */
```

4.1.5 LLC link pre-step actions

After initial hardware configurations the Discovery loop needs to detect whether there is any NFC P2P device in the RF field. To perform detection it transmits Attribute Request, then listens for any Attribute Response. In target's Attribute Response there is information about link parameters mentioned in [3] in section 6.2.3.1. During link activation procedure the NXP P2P Library looks for this data in special structure RemoteInfo. If there are some parameters, the Library activates the LLC link with those parameters rather than doing PAX exchange.

```
RemoteInfo.SessionOpened = 1;

                  RemoteInfo.RemDevType = phNfc_eNfcIP1_Target;

RemoteInfo.RemoteDevInfo.NfcIP_Info.ATRInfo_Length =
(discLoop.sTypeFTargetInfo.sTypeF_P2P.bAtrResLength - 17);

                  memcpy(RemoteInfo.RemoteDevInfo.NfcIP_Info.ATRInfo,
&discLoop.sTypeFTargetInfo.sTypeF_P2P.pAtrRes[17],
(discLoop.sTypeFTargetInfo.sTypeF_P2P.bAtrResLength - 17));
```

4.1.6 Initializing the LLC link

Following sample code presents how to activate LLC link. After activation no transport packet can be sent, because LLCP transport socket has not been connected to a DSAP.

```
phStatus_t NFC_LLCPInitialize(void) {

    uint32_t          DummyContext;

    phStatus_t status = PH_ERR_SUCCESS;


    Llcp_running = TRUE;
```

MIU is limit for maximal length of one LLCP frame. Consequently, this determines length of SNEP fragments.

```
    LinkParam.miu = 128;

    LinkParam.lto = 100;

    LinkParam.wks = 0x0001;

    LinkParam.option = 0x02;
```

This function initiates parameter component. It stores important pointers to underlaying components.

```
    status = phlnLlcp_Fri_Init(
```

```
        &lnLlcpDataparams,

        sizeof(lnLlcpDataparams),

        &Llcp,

        &LinkParam,

        &LlcpTransport,

        &RemoteInfo,

        bTxBuffer,

        sizeof(bTxBuffer),

        bRxBuffer,

        sizeof(bRxBuffer),

        &palI18092mPI);
```

Complete link activation consists of sequence of three functions `phlnLlcp_Reset()`, `phlnLlcp_ChkLlcp()`, `phlnLlcp_Activate()`.

```
    status = phlnLlcp_Reset(

            &lnLlcpDataparams,

            &LinkCB,

            &DummyContext);


CHECK_SUCCESS(status);
```

This function resets the Transport layer – base of all the socket related functions.

```
status = phlnLlcp_Transport_Reset(&lnLlcpDataparams);

CHECK_SUCCESS(status);
```

The OSAL component `osal` is necessary for usage of LLCP hardware timers which are used by delay function and SYMM LLCP timer. The component must be assigned before `phlnLlcp_Activate()` function called.

```
Llcp.osal = &osal;


status = phlnLlcp_ChkLlcp(

    &lnLlcpDataparams,

    &CheckCb,

    (void*) &DummyContext);

CHECK_SUCCESS(status);


status = phlnLlcp_Activate(&lnLlcpDataparams);

CHECK_SUCCESS(status);


if (status == NFCSTATUS_SUCCESS) {
```

```
    Link_Activated = 1;

    }


    return status;

}
```

4.1.7 Establishing the LLCP connection

All the data packets between the MCU and the mobile device are exchanged via LLCP transport sockets. After the LLC link activation, such LLCP socket needs to be created by `phlnLlcp_Transport_Socket()` function for further LLCP communication.

Once the socket created it sends connection request to the mobile device. Into `psUri` there is passed the string `"urn:nfc:sn:snep"` which means SNEP server service. The string is sent together with the connection request saying that the local requests the remote for providing SNEP server service.

```
static phStatus_t NFC_LLCPCreateClient(phFriNfc_LlcpTransport_Socket_t **ppSocket,
                                       phNfc_sData_t *psUri) {

    phStatus_t status = NFCSTATUS_SUCCESS;

    uint32_t        DummyContext;


    status = phlnLlcp_Transport_Socket(

        &lnLlcpDataparams,

        phFriNfc_LlcpTransport_eConnectionOriented,

        &sOptions,

        &sWorkingBuffer,

        ppSocket,

        &ErrCb,

        (void*) &DummyContext);

    CHECK_SUCCESS(status);
```

This assignment totally redundant, because later `phlnLlcp_Transport_ConnectByUri()` function implements service discovery protocol within.

```
    (*ppSocket)->socket_dSap = 1;

    SocketConnected = FALSE;


    status = phlnLlcp_Transport_ConnectByUri(

        &lnLlcpDataparams,

        *ppSocket,

        psUri,

        &ConnectCb,

        (void*) &DummyContext);
```

UM10721
All information provided in this document is subject to legal disclaimers.
© NXP B.V. 2018. All rights reserved.

User manual
COMPANY PUBLIC
Rev. 1.2 — 5 August 2018
270112
53 of 77

```
CHECK_SUCCESS(status);
```

Program flow may be halt by bodyless loop, until `SocketConnected` flag modified within the connect CB function. `Llcp_running` allows passing through since link activation.

```
while(!SocketConnected && Llcp_running);


    return status;
}
```

4.1.8 Preparation of NDEF message

The sample application builds SNEP message and its header as well. It also encapsulates NDEF message and header into a SNEP message. All the user needs to do is to choose particular NDEF message before compilation as described in the next section 4.1.9. The chosen NDEF message shall be transmitted to Android SNEP server.

4.1.9 Choosing a file as NDEF message

By default, the software sends the picture of NXP loge in *JPEG* format. In the source code of SNEP client project there are few files prepared tables in separate header files.

Table 8. Table of files that are part of sample application written as C headers
Information from the last two columns is necessary for choosing the right line from `n_mess[]` array – see below.

Content	Header name	NDEF message identifier	File type identifier
PNG image	*c_tablepng.h*	NDEF_TYPE_IMAGE	NDEF_IMAGE_PNG
QR code of NXP logo	*c_tableQR.h*	NDEF_TYPE_IMAGE	NDEF_IMAGE_PNG
Image of NXP logo	*c_tablenxp.h*	NDEF_TYPE_IMAGE	NDEF_IMAGE_JPEG
Long text message	*c_tabletxt.h*	'T'	LANG_EN

Because of lack of flash memory space in LPC1227 the developer is allowed to choose just one file here, otherwise the compiler returns error "*redefinition of 'c_table'*" due to multiple declarations of the `c_table[]`.

By doing some easy modifications in *ndef_message.c* file (explained below) the user can choose among several predefined NDEF messages like text messages, URI links and JPEG or PNG images. For example, to choose picture of PNG uncomment the line `#include <c_tablepng.h>` and comment all the other lines referring to `c_table[]`.

```
#include <c_tablepng.h>

//#include <c_tableQR.h>

//#include <c_tablenxp.h>

//#include <c_tabletxt.h>
```

Then choose the correct file type and format in *ndef_message.c*. Get this information from Table 8 from the same row as the PNG image file. Uncomment the line with `NDEF_TYPE_IMAGE` and `NDEF_IMAGE_PGN` and let other lines commented. Thanks to this option correct NDEF head is built so an Android application can recognize type of the incoming NDEF message.

```
NDEF_messages n_mess[]={
```

UM10721

© NXP B.V. 2018. All rights reserved.

User manual
COMPANY PUBLIC

Rev. 1.2 — 5 August 2018
270112

54 of 77

```
    /* type, parameter, string */
//    {NDEF_TYPE_IMAGE,NDEF_IMAGE_JPEG,c_table, sizeof(c_table)},
     {NDEF_TYPE_IMAGE,NDEF_IMAGE_PNG,c_table, sizeof(c_table)},
//    {'T', LANG_NO, text1, sizeof(text1)},
//    {'T', LANG_EN, c_table, sizeof(c_table)},
//    {'T', LANG_EN, text6, sizeof(text6)},
//    {'T', LANG_GR, text3, sizeof(text3)},
//    {'T', LANG_FR, text4, sizeof(text4)},
//    {'U', NDEF_URI_WWW, uri_11, sizeof(uri_11)},
//    {'U', NDEF_URI_WWW, uri_12, sizeof(uri_12)},
//    {'U', NDEF_URI_WWW, uri_13, sizeof(uri_13)},
//    {'U', NDEF_URI_HTTP, uri_13, sizeof(uri_13)},
//    {'U', NDEF_URI_TEL, uri_52, sizeof(uri_52)},
};
```

Each time you intend to send another NDEF message than previously, the source code must be recompiled.

4.1.10 Sending a fragment of SNEP message

SNEP client uses LLCP Send API `phlnLlcp_Transport_Send()` function which performs sending a SENP fragment as I PDU frame to the SNEP server.

Routine for sending an I PDU frame via LLCP transport socket:

```
phStatus_t NFC_LLCPSend( phFriNfc_LlcpTransport_Socket_t *pSocket,
                      uint8_t *buf, uint32_t len,
                  pphFriNfc_LlcpTransportSocketSendCb_t pSndCb)
{
    static phNfc_sData_t sData;
    phStatus_t status;


    /* set the call-back function */
    if(pSndCb == NULL) pSndCb = &SendCb;


    sData.buffer = buf;
    sData.length = len;
```

Using the LLCP API for sending via LLCP transport socket (see section 3.2.9.10):

```
status = phlnLlcp_Transport_Send( &lnLlcpDataparams, pSocket, &sData,
                                  pSndCb, pSocket);
CHECK_SUCCESS(status);
```

UM10721

© NXP B.V. 2018. All rights reserved.

User manual **Rev. 1.2 — 5 August 2018** **55 of 77**
COMPANY PUBLIC 270112

```
     return status;

}
```

4.1.11 Receiving of SNEP response

The SNEP client needs to receive SNEP messages from the SNEP server. Particularly after sending the first SNEP fragment receiving of Continue SNEP message from the SNEP server is expected. After sending the last fragment the SNEP Success is expected. Any SNEP message/fragment in server – client communication it is I PDU from the LLCP layer's point of view.

Implementation of routine for receiving I PDU frame via LLCP transport socket may look as one underneath.

```
phStatus_t NFC_LLCPRecv(

     phFriNfc_LlcpTransport_Socket_t *pSocket,

     phNfc_sData_t *psData,

     uint8_t *buf,

     uint32_t len,

     pphFriNfc_LlcpTransportSocketSendCb_t pRecvCb )

{

   phStatus_t status;


   /* set the call-back function */

   if(pRecvCb == NULL) pRecvCb = &RecvCb;


   psData->buffer = buf;

   psData->length = len;


   status = phlnLlcp_Transport_Recv(

         &lnLlcpDataparams,

         pSocket,

         psData,

         pRecvCb,

         (void*)psData);

   CHECK_SUCCESS(status);


   return status;

}
```

In `psData->buffer` there is a SNEP response from the SNEP server. It can be compared with codes of defined SNEP responses.

UM10721
All information provided in this document is subject to legal disclaimers.
© NXP B.V. 2018. All rights reserved.

User manual
COMPANY PUBLIC
Rev. 1.2 — 5 August 2018
270112
56 of 77

Receiving of the LLCP I frame is a little different from as one would expect according LLCP defined by the NFC forum. There is a bug in November 2012 Release NXP Reader Library P2P which causes the receiving need to be subsequently followed by the "Dummy send". The bug is fixed in July 2013 release. Receiving works in the normal way, just calling `phlnLlcp_Transport_Recv()` shall be sufficient for receiving I PDU.

Entire receive routine looks like following:

Setting the receive busy status flags before reception itself.

```
status = NFC_LLCPRecv(pSocket_Client, &sData, bSnepRx, sizeof(bSnepRx),
                      NULL);

status = NFC_LLCPSend(pSocket_Client, NULL, NULL, &SendNULLCb);
```

4.1.12 Closing the connection

After data transmission complete the LLCP socket connection with SNEP server service on the remote is no more needed for this purpose. The LLCP socket should be disconnected in the correct way and then closed – socket component cleared. This function shall be also when the any other than SNEP Continue or Success response from the SENP server received.

```
phStatus_t NFC_LlcpClose(void) {

    phStatus_t status = PH_ERR_SUCCESS;

    uint32_t        DummyContext;


    status = phlnLlcp_Transport_Disconnect(

            &lnLlcpDataparams,

            pSocket_Client,

            DisconnectCb,

            (void*) &DummyContext);


    status = phlnLlcp_Transport_Close(

            &lnLlcpDataparams,

            pSocket_Client);

    CHECK_SUCCESS(status);


    return status;

}
```

4.2 SNEP client

4.2.1 How it works

Briefly, SNEP client sends initial fragment of length 128 bytes. Then it is waiting for a SNEP response from the server. Because in SNEP header it is declared longer SNEP message than one fragment, the server should response with Continue SNEP message. The SNEP client (MCU) can go on with sending rest of the SNEP message without

getting any acknowledgements form the mobile device. As soon as entire SNEP message transmitted, the SNEP client shall receive SNEP Success from the mobile device. Or first sending shall be started only after socket connected.

The SNEP client is represented by the uppermost function `SNEPClientDemo()`. The entire SNEP client source code is in *SnepClient.c*. SNEP client is implemented like state machine with several states defined in *SnepClient.h* header file. For each state of the SNEP client there is the specific function-handler managing the particular state. The handlers are described in sections 4.2.6 - 4.2.13.

4.2.2 SNEP client parameter structure

The SNEP client uses one parameter structure `SnepClientData_t` defined in *SnepClient.h* that holds state of the SNEP client and four flags. The flags should avoid the overlapping of sending and receiving of SNEP fragments and responses.

`SnepClientData_t` consists of 6 members

->`Hstate`: states of the SNEP client valid values are defined by the `HandleState_t` *enum* in *SnepClient.h*. The states are also included in the flowcharts in Fig 5 - Fig 11. Each flowchart begins in one particular (input) state and end at one only or one among multiple possible output states.

->`SafetyCnt`: Safety Counter which should avoid infinite loops when an unintended bhehaviour on the LLCP layer occurs

->`bSendStart`: Send start flag ensures the same SNEP is not sent repeatedly as the SNEP client loops in a state. See implementation of this flag in `HandleSnepFirst()` Fig 6 and `HandleSendNext()` Fig 9 functions.

Legal values:

`SEND_START` indicates that sending of the SNEP fragment or request has just been started

`SEND_NOT_START` indicates that sending of a SNEP fragment or request has not been started yet. This value is set inside the application callbacks `SendCb()` and `SendNULLCb()`.

->`bSendFinish`: Send finish flag ensures that sending is not started until previous sending finished. See implementation of this flag in `HandleSnepFirst()` Fig 6 and `HandleSendNext()` Fig 9 functions.

Legal values:

`SEND_FINISHED` indicates that sending of a SNEP fragment has just been completed. This value is set inside the application callbacks `SendCb()` and `SendNULLCb()`.

`SEND_NOT_FINISHED` indicates that sending of a SNEP fragment is currently in progress.

Note: This flag is implemented in the application layer (SNEP client) and it is not directly influenced by the LLCP layer. Alternatively, this could be implemented by return value `NFCSTATUS_REJECTED` of `phlnLlcp_Transport_Send()` (see section 3.2.9.10)

->`bRecvStart`: Receive start flag ensures the receive procedure (see section 4.1.11) is not when one SNEP response is already expected to be received. See

implementation of this flag in `HandleSendFirts()`,Fig 6 `HandleSendNext()` Fig 9 and `HandleRecv()` Fig 7 functions.

<u>Legal values</u>:

`RECV_START` indicates that the SNEP client is ready to receive a SNEP response.

`RECV_NOT_START` indicates that the application layer has not been set yet to listen to SNEP responses. This value is set inside the application callbacks `RecvCb()`.

<u>Note</u>: This flag is implemented in the application layer (SNEP client) and it is not directly influenced by the LLCP layer. Alternatively, this could be implemented by return value `NFCSTATUS_REJECTED` of `phlnLlcp_Transport_Recv()` (see section 3.2.9.11)

`->bRecvFinish`: Receive finish flag indicated that the SNEP client has already received SNEP response from the SNEP server. See implementation in `HandleRecv()` Fig 7 .

<u>Legal values</u>:

`RECV_FINISHED` indicates that the SNEP client has already received the response from the server. This value is set inside the application callbacks `RecvCb()`.

`RECV_NOT_FINISHED` indicates that the SNEP client has not received the response from the server yet.

4.2.3 Application callbacks

As mentioned in the entire document, the NXP P2P Library notifies upperlaying application via callback functions. We define simple callback functions modifying binary flag within. The flag shall indicate a state the application is currently in (see flags of the implemented SNEP client in section 4.2.2). In send callback 4.2.3.1 and receive callback 4.2.3.2 and there are modified flags from dedicated SNEP client parameter structure 4.2.2. In connect callback and link status callback there are modified flags (section 4.1.4) defined as global variables.

4.2.3.1 Application callback for send

The socket send callback function shall be called after the Library completes the sending operation.

```
static void SendCb ( void* pContext, phStatus_t status ) {

    sSnepClientData.bSendStart = SEND_NOT_START;

    sSnepClientData.bSendFinish = SEND_FINISHED;

}
```

Once `bSendStart` flag cleared, performing next send operation on the application layer (see Fig 6 or Fig 9). Once `bSendFinished` flag is set, on the application layer (SNEP client) there is one of two conditions fulfilled to perform listening to incoming SNEP responses (see Fig 6 or Fig 9).

The socket send callback function needs to be passed as parameter to `ph_lnLlcp_Transport_Send()` (see section 3.2.9.10).

UM10721

User manual **Rev. 1.2 — 5 August 2018** **59 of 77**
COMPANY PUBLIC 270112

When send operation is completed by the NXP P2P Library the Send socket callback is set to NULL thus need to be set with a next Transport Send API like input parameter (see section 3.2.10.7)

4.2.3.2 Application callback for receive

The receive callback function shall be called as soon as I PDU frame receive (all the SNEP messages/fragments are I PDUs). Modifying the receive flags enables performing next receive operation on the application layer (see Fig 6 or Fig 9).

```
static void RecvCb ( void* pContext, phStatus_t status ) {

    sSnepClientData.bRecvStart = RECV_NOT_START;

    sSnepClientData.bRecvFinish = RECV_FINISHED;

}
```

Once `bRecvStart` flag cleared, on the application layer (SNEP client) there is one of two conditions fulfilled to perform listening to incoming SNEP responses (see Fig 6 or Fig 9).

Once `bRecvFinished` flag is set, SNEP response can be tested (see Fig 7 and Fig 8).

The socket send callback function needs to be passed as parameter to `ph_lnLlcp_Transport_Recv()` (see section 3.2.10.8).

When send operation is completed by the NXP P2P Library the socket receive socket callback is set to NULL thus need to be set with a next Transport Send API like input parameter (see section 3.2.10.7).

4.2.4 SNEP client pre-step initialization

First of all, at the beginning of the `SNEPClientDemo()` there is `NFC_LLCPInitialize(void)` called which performs initialization of the LLCP layer and the LLC link activation (see section 4.1.6). Afterward the first function of SNEP client state machine - `HandleSnepStart()` (see section 4.2.6) can be run.

4.2.5 Generic state handler

```
phStatus_t SnepClientHandle(SnepClientData_t *pClientData);
```

The generic handler function that integrates all the particular handlers according to a current SNEP state it runs a particular handler.

4.2.6 Start the SNEP client

```
phStatus_t HandleSnepStart(SnepClientData_t *pClientData);
```

This function connects the local to the remote on LLCP layer and concurrently the local requests from the remote to provide SNEP server service specified by the string `"urn:nfc:sh:snep"`. If the request accepted, the connection on LLCP layer is established and on the application layer the local becomes the SNEP client connected to the SNEP server. From this point the client is ready to send the PUT request – the first fragment of SNEP message (see section 4.2.7).

UM10721

© NXP B.V. 2018. All rights reserved.

User manual
COMPANY PUBLIC

Rev. 1.2 — 5 August 2018
270112

60 of 77

Fig 5. Flowchart of the `HandleSnepStart()`
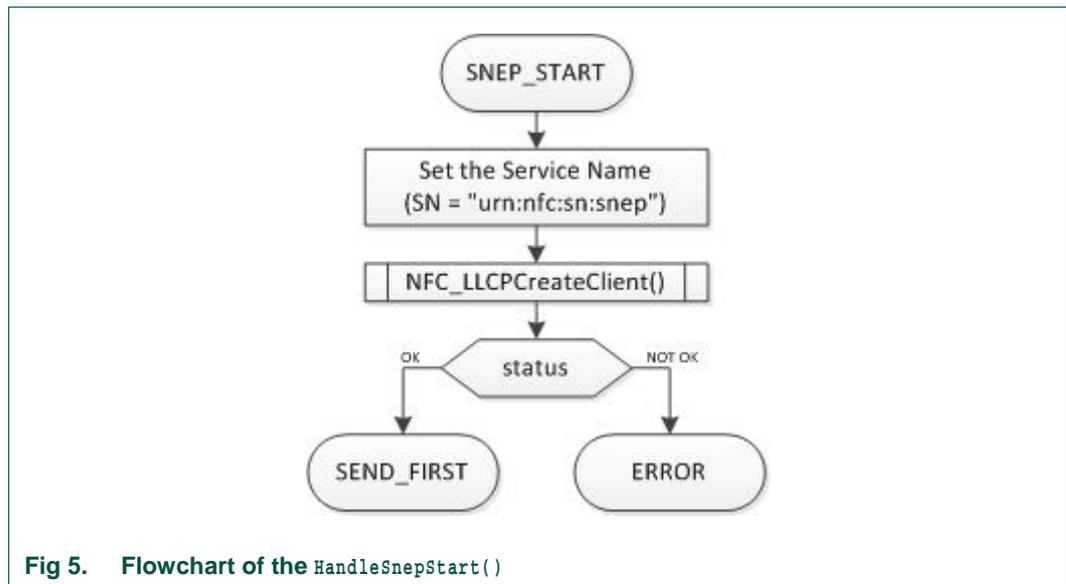
See implementation of the LLCP API functions in section 4.1.7. `HandleSnepStart()` converts the service name string `"urn:nfc:sh:snep"` to the correct data type – see input arguments of `NFC_LLCP_CreateClient()` in section 4.1.7.

4.2.7 Sending the first SNEP fragment

`phStatus_t HandleSendFirst(SnepClientData_t *pClientData);`

The first SNEP fragment is little different from the rest of the SNEP message. There is encapsulated SNEP header and at the beginning of the SNEP payload there is NDEF header of the NDEF message placed. Both the headers must be assembled. It is obvious that, content of both the payloads depends on data intended to be transmitted – see Choosing a file as NDEF message in section 4.1.9. Both the headers (length attribute) depend on size of the chosen file, additionally the NDEF header carries the information about type of the file. Therefore, the first SNEP fragment is considered as special state of the SNEP client state machine.
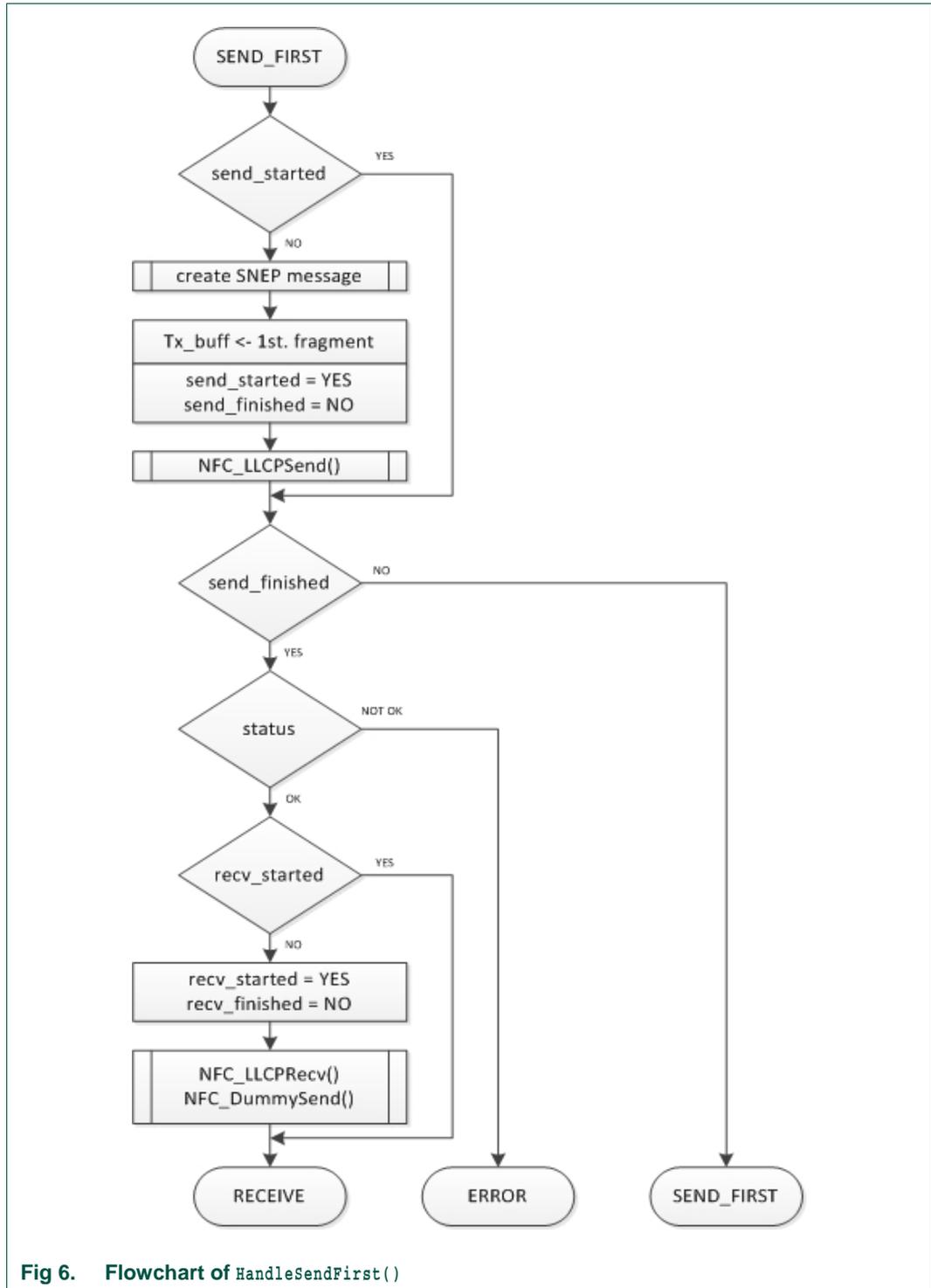
Fig 6.   Flowchart of `HandleSendFirst()`

The `HandleSendFirst()` is designed be implemented in loop, so it is called repeatedly. In the first run it passes only through the upper part of the diagram only which prepares the SNEP message and sends the first fragment containing the PUT request. Since then the upperpart is" locked" – shall not be executed when the handler called. Then it can be called several times without executing the second part (after *send_finished* decision). As soon as the sending operation on the LLCP layer completed, `bSendFinish` flag shall be set

UM10721

All information provided in this document is subject to legal disclaimers.

© NXP B.V. 2018. All rights reserved.

User manual
COMPANY PUBLIC

Rev. 1.2 — 5 August 2018
270112

62 of 77

by send application callback `SendCb()` (it is not in the picture) and *send_finished* decision shall succeed. Assuming no error occurred during the LLCP send API at *recv_started* the client shall run NO option in the first run immediately after *send_finished* YES. The client shall get ready the reception on the LLCP ("DummySend" is clarified in section 4.1.11). Since then the client's state is set to receive – SNEP response from the server is expected (see section 4.2.8).

4.2.8 Handling of the SENP response

`phStatus_t HandleRecv(SnepClientData_t *pClientData);`

As soon any SNEP Response from the SNEP server received, `bRecvFinish` flag is set to `RECV_FINISHED` by the application receive callback `RecvCb()` is this handler parses the response. Each software flow branch is matched with the particular SNEP response. If everything on the client's and server's side goes correctly, then the client goes only twice through the cases of this handler

- First time after sending Put request and subsequently receiving **Continue** response

- Second time after sending the last fragment of the SNEP message and subsequently receiving the **Success** response

(1)   The second part of the flowchart is in Fig 8

Fig 7.   Flowchart of the `HandleRecv()` part 1

UM10721

All information provided in this document is subject to legal disclaimers.

© NXP B.V. 2018. All rights reserved.

User manual
COMPANY PUBLIC

Rev. 1.2 — 5 August 2018
270112

64 of 77

(1) The rest of the flowchart from (1)

Fig 8. Flowchart of the `HandleRecv()` part 2
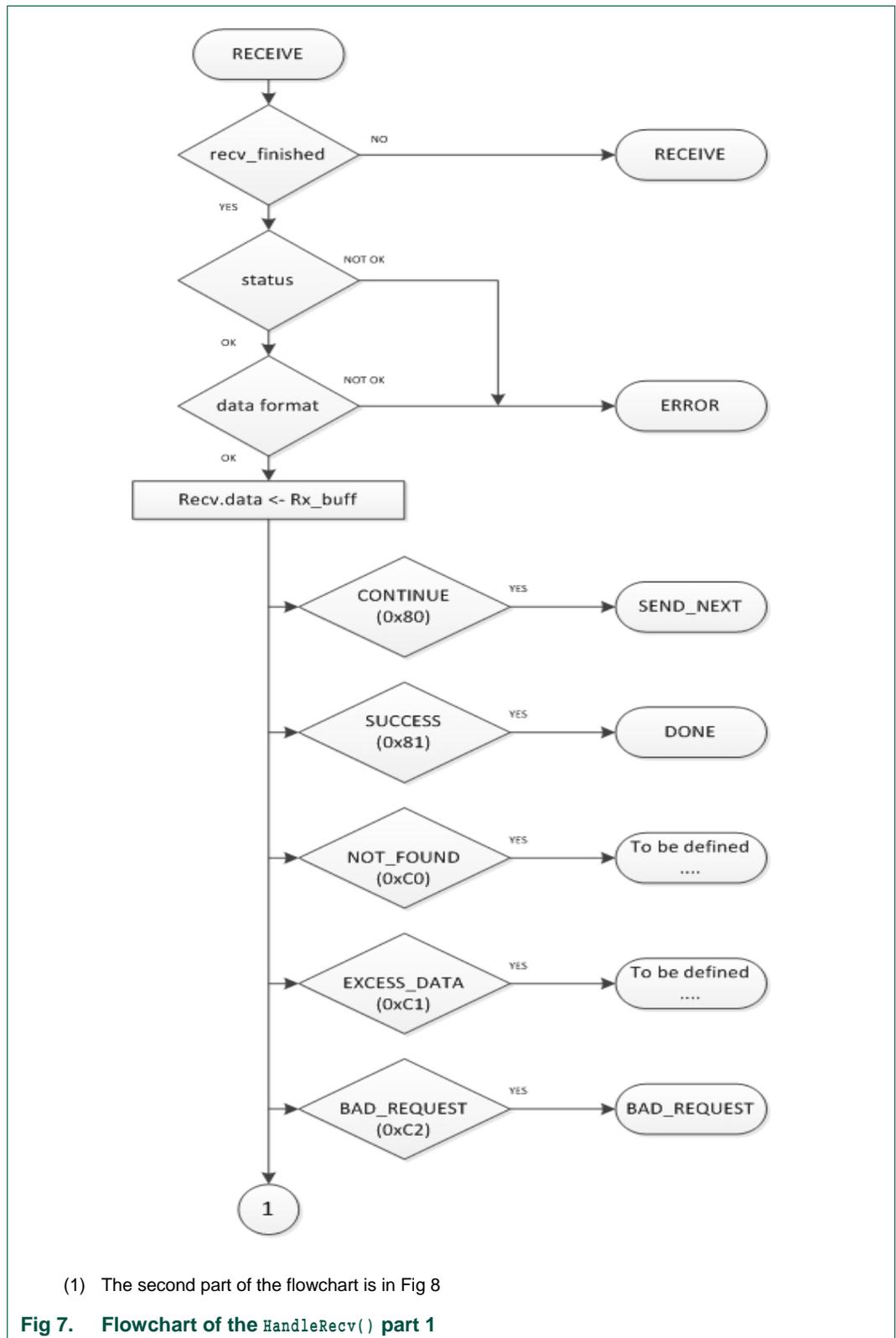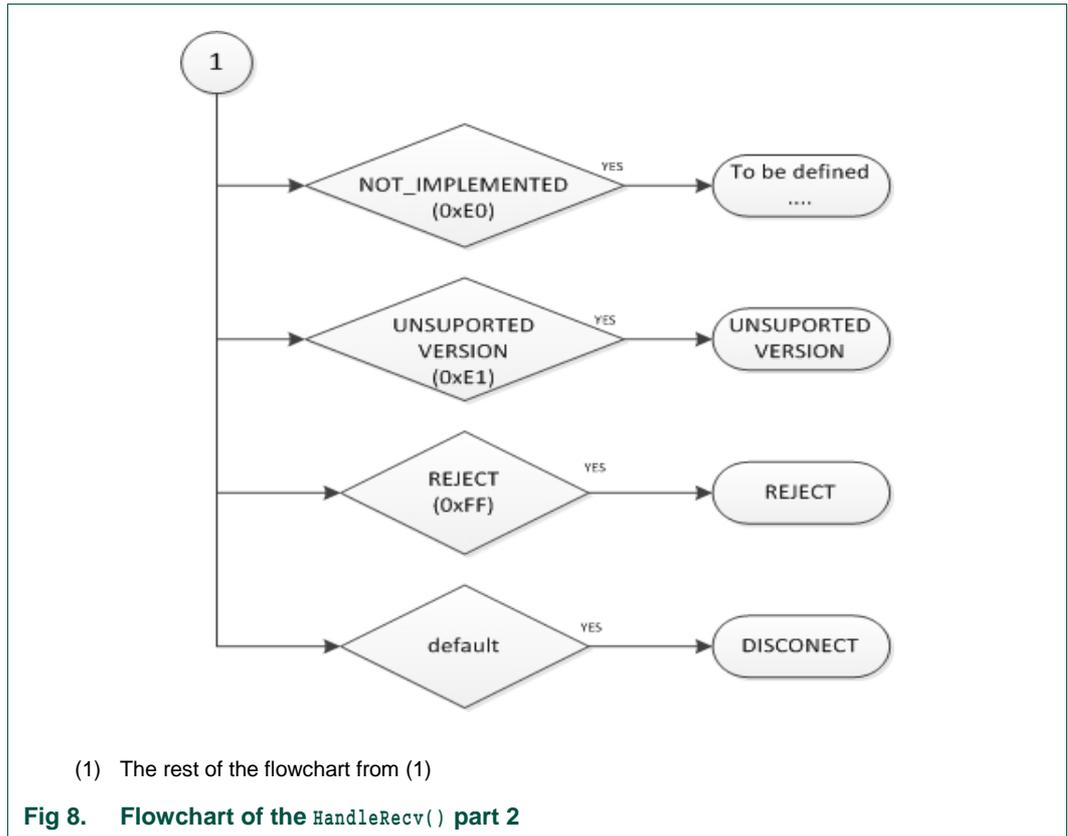
4.2.9 Sending the rest of the SNEP message

```
phStatus_t HandleSendNext(SnepClientData_t *pClientData);
```

Once the Continue response from the SNEP server has been received the SNEP client can send the rest of the SNEP message in sequence of 128 byte long SNEP fragments (the last one may be shorter). This handler shall be called repeatedly in loop until the last SNEP fragment sent. During sending the SNEP client does not expect any response from the SNEP server thus the sending sequence shall not be interrupted. As soon as the last fragment of the SNEP message has been transmitted, reception of the SNEP Success response from the SNEP server is expected. Reception procedure of SNEP response is the same as in `HandleSendFirst()` (see section 4.2.7).

UM10721

User manual
COMPANY PUBLIC

Rev. 1.2 — 5 August 2018
270112

65 of 77
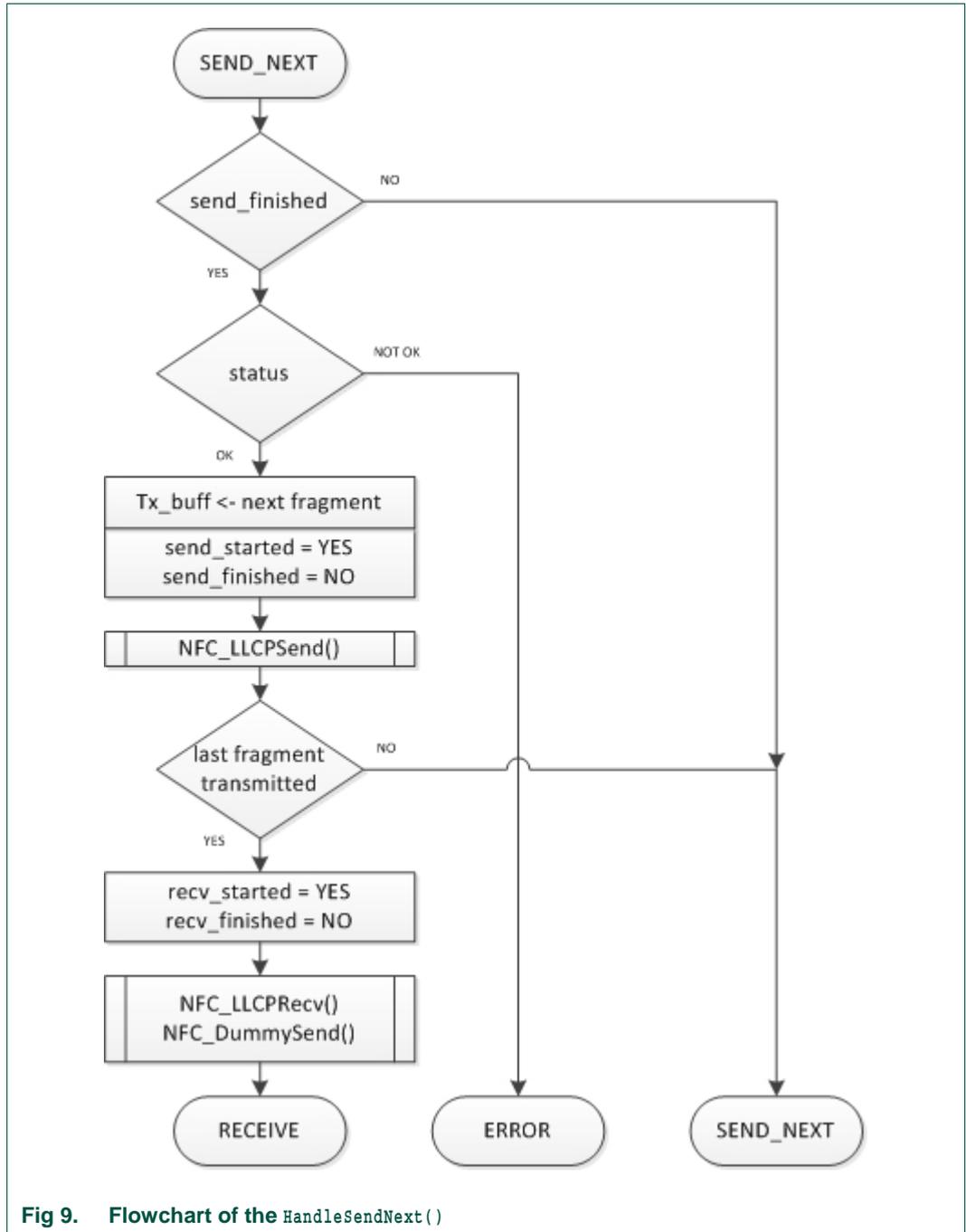
Fig 9. Flowchart of the `HandleSendNext()`

The first decision *sends_finished* - NO option avoids sending the same SNEP fragment multiple times. YES option is run when sending of the SNEP fragment completed. The second decision *status* goes OK when sending on the LLCP layer succeeded, otherwise the sending canceled moreover the SNEP client disconnected and the program escapes from `SNEPClientDemo()`. The decision point *last fragment transmitted* goes always over NO branch but when the last fragment transmitted the client shall be set for listening to SNEP response (same as in `HandleSendFirst()` 4.2.7).

UM10721

User manual
COMPANY PUBLIC

Rev. 1.2 — 5 August 2018
270112

66 of 77

4.2.10 SNEP transmission successful

```
phStatus_t HandleDone(SnepClientData_t *pClientData);
```

This handler is called as soon as the SNEP Success response from the SENP server received. It indicates the reception of the entire SNEP message has been received on the SNEP server side. At this moment the transmitted picture should be displayed on the mobile device screen. The SNEP client shall be disconnected by `HandleDisconnect()` handler (see Fig 10).

4.2.11 Disconnect from SNEP server

```
phStatus_t HandleDisconnect(SnepClientData_t *pClientData);
```

This handler closes the LLC link connection between the SNEP client and server on the LLCP layer by calling the `NFC_LlcpClose()` function (see section 4.1.12). The SNEP client shall reach this handler in all circumstances either SNEP entire message successfully received on SNEP server side or transmission fail (indicated by any SNEP non Continue or Success message) as shown in Fig 10.
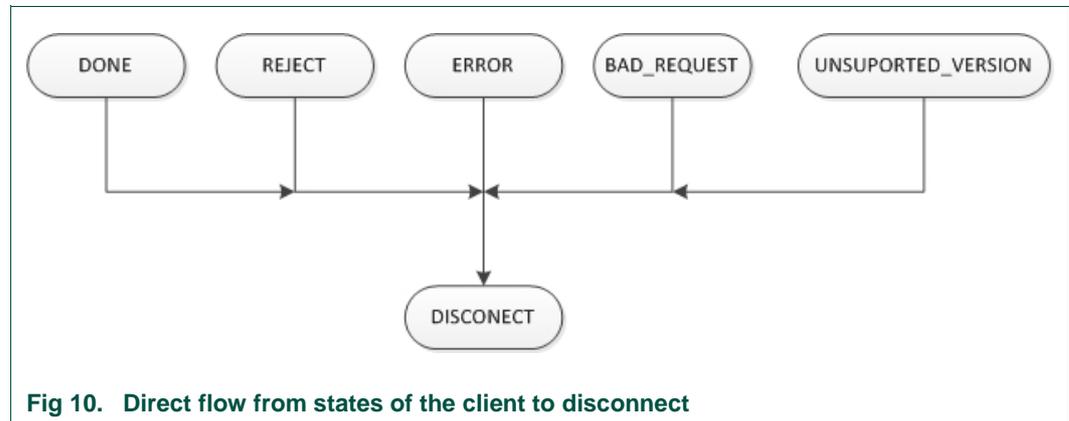
Fig 10.   Direct flow from states of the client to disconnect

`HandleDisconnect()` always disconnects `SnepClientDemo()` regardless the previous states as shown in Fig 11.
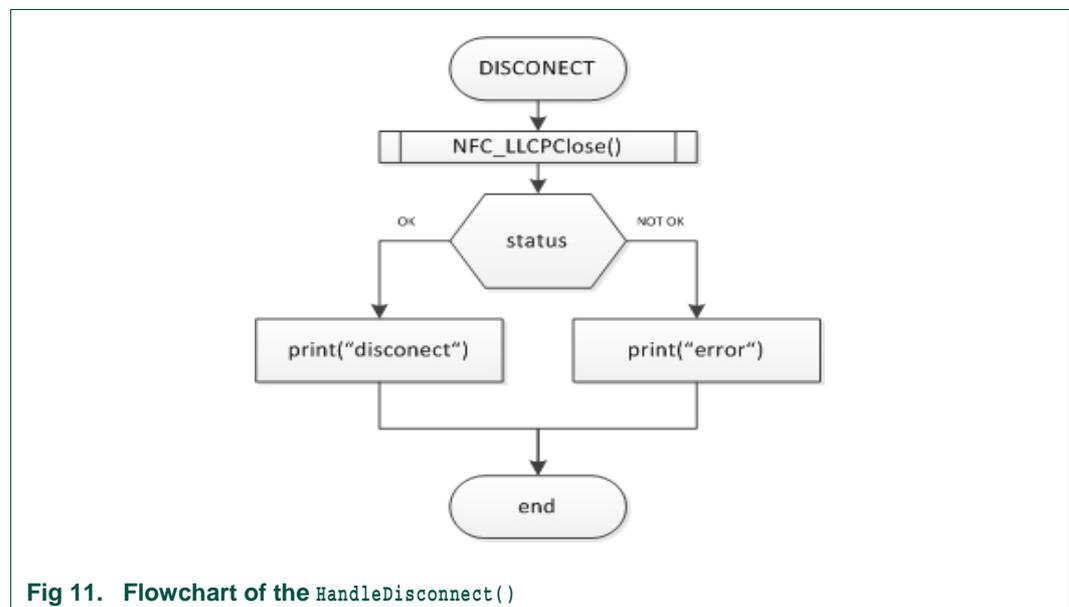
Fig 11.   Flowchart of the `HandleDisconnect()`

UM10721

© NXP B.V. 2018. All rights reserved.

User manual
COMPANY PUBLIC

Rev. 1.2 — 5 August 2018
270112

67 of 77

4.2.12 PUT request Rejected

`phStatus_t HandleReject(SnepClientData_t *pClientData);`

This handler manages situation when the SNEP Reject message from the SNEP server received. In compliance with the SNEP specification [15] the SNEP Reject response informs the client about server's inability to receive the SNEP message declared in the SNEP header. The SNEP client shall not send the rest of the SNEP message. Afterward `HandleDisconnect()` handler (see section 4.2.11) closes the LLC link connection.

4.2.13 Error on LLCP layer

`phStatus_t HandleError(SnepClientData_t *pClientData);`

Error on LLCP layer has occurred – client creation, send or receive operation failed.

5. Abbreviations

Table 9. **Abbreviations**

Acronym	Description
AL	Application Layer
ACK	Acknowledgement
ATQA	Answer To Request, type A
BAL	Bus Abstraction Layer
CB	Callback
CC	Connection Complete (in LLCP)
CRC	Cyclic Redundancy Check
DEP	Data Exchange Protocol
DID	Device Identifier
DISC	Disconnect (in LLCP)
DM	Disconnected Mode (in LLCP)
EEPROM	Electrically Erasable Programmable Read-Only Memory
FRMR	Frame Reject (in LLCP)
GPIO	General Purpose Input Output
HAL	Hardware Abstraction Layer
I	Information (in LLCP)
I2C	Inter-Interchanged Circuit
IC	Integrated Circuit
LLC	Logical Link Control
LLCP	Logical Link Control Protocol
LTO	Link Timeout (in LLCP)
MAC	Medium Access Control (in LLCP)
MCU	Microcontroller Unit
MF	MIFARE
MIU	Maximum Information Unit (in LLCP)
MIUX	Maximum Information Unit Extension (in LLCP)
NAD	Node Address
NAK	Negative Acknowledgement
NDEF	NFC Data Exchange Format
NFC	Near Field Communication
NFCIP	NFC Interface and Protocol
OPT	Option link parameter (in LLCP)
PAL	Protocol Abstraction Layer
PAX	Parameter Exchange (in LLCP)
PCD	Proximity Coupling Device (Contactless Reader)
PICC	Proximity Integrated Circuit Card (Contactless Card)
PDU	Protocol Data Unit (in LLCP)

UM10721

All information provided in this document is subject to legal disclaimers.

© NXP B.V. 2018. All rights reserved.

User manual
COMPANY PUBLIC

Rev. 1.2 — 5 August 2018
270112

69 of 77

Acronym	Description
PTYPE	PDU Type (in LLCP)
RNR	Receive Not Ready (in LLCP)
RR	Receive Ready (in LLCP)
SAM	Secure Access Module
SAP	Service Access Point
DSAP	Destination Service Access Point
SSAP	Source Service Access Point
SNEP	Simple NDEF Exchange Protocol
SPI	Serial Peripheral Interface
SYMM	Symmetry token (in LLCP)
UID	Unnumbered Information (in LLCP)
UID	Unique Identifier
WKS	Well Known Services

UM10721

All information provided in this document is subject to legal disclaimers.

© NXP B.V. 2018. All rights reserved.

User manual
COMPANY PUBLIC

Rev. 1.2 — 5 August 2018
270112

70 of 77

6. References

[1] Direct link to the NXP Reader Library Public Release
https://www.nxp.com/products/identification-and-security/nfc/nfc-reader-ics/nfc-reader-library

[2] **User Manual** NXP Reader Library User Manual based on CLRC663, BU-ID Doc. No. 2574**1, available on
http://www.nxp.com/demoboard/CLEV663B.html#documentation

[3] **Technical Specification** Logical Link Control Protocol, NFCForum-TS-LLCP_1.1, available on https://nfc-forum.org/our-work/specifications-and-application-documents/specifications/specification-releases/

[4] **Data Sheet** MF1S503X MIFARE Classic EV1 1K - Mainstream contactless smart card IC for fast and easy solution development, available on
https://www.nxp.com/docs/en/data-sheet/MF1S50YYX_V1.pdf

[5] **Data Sheet** - MIFARE Ultralight MF0ICU1, MIFARE Ultralight contactless single-ticket IC, BU-ID Doc. No. 0286**, available on
http://www.nxp.com/documents/data_sheet/MF0ICU1.pdf

[6] **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

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

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

[9] Standard ECMA – Near Field Communication Interface and Protocol (NFCIP-1)
https://www.ecma-international.org/publications/files/ECMA-ST/Ecma-340.pdf

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

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

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

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

[14] **Technical Specification**-NFC Digital Protocol, NFC Forum-TS-DigitalProtocol-1.0, available on https://nfc-forum.org/our-work/specifications-and-application-documents/specifications/specification-releases/

[15] **Technical Specification** – Simple NDEF Exchange Protocol, NFCForum-TS-SNEP_1.0, available on https://nfc-forum.org/our-work/specifications-and-application-documents/specifications/specification-releases/

---

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

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. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

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

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

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

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

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

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

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

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

Translations — A non-English (translated) version of a document is for reference only. The English version shall prevail in case of any discrepancy between the translated and English versions.

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 Semiconductors IC does not include a license to any NXP patent (or other IP right) covering combinations of those products with other products, whether hardware or software.

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

UM10721

All information provided in this document is subject to legal disclaimers.

© NXP B.V. 2018. All rights reserved.

**User manual
COMPANY PUBLIC**

**Rev. 1.2 — 5 August 2018
270112**

72 of 77

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 — is a trademark of NXP B.V.

DESFire — is a trademark of NXP B.V.

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

UM10721

User manual
COMPANY PUBLIC

Rev. 1.2 — 5 August 2018
270112

73 of 77

8. List of figures

9. List of tables

10. Contents

UM10721

User manual **Rev. 1.2 — 5 August 2018** **76 of 77**
COMPANY PUBLIC **270112**