UM10721

NXP NFC Reader Library User Manual

Rev. 2.1 — 07 April 2014
270121

User Manual
COMPANY PUBLIC

Document information

Info	Content
Keywords	NFC Reader Library, P2P, CLRC663, PN512, LPC1769, ISO18092, Discovery Loop, LLCP, SNEP, NFC Forum Tag Type Operation, NFC Forum, MIFARE, ISO14443.
Abstract	This document describes the implementation of the NFC Reader Library and how to use it.

Revision history

Rev	Date	Description
2.1	20140407	Description of API is moved to the new document – UM10802
2.0	20140221	Second release
1.3	20140205	Revision check. Minor changes.
1.2	20140131	Updated UM for the NFC Reader Library v3.010 software release
1.1	20130724	Change of descriptive title
1.0	20130613	First release

1. Audience

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

2. Abstract

This document describes the implementation of the NFC Reader Library and how to use it. This user manual is intended to help software developers, implementers and integrators to get familiar with the NFC Reader Library and to learn how to work with it. The document is divided in sections: after the introductory Sections 1 and 2, Section 3 provides an overview of the NFC Reader Library and its layered architecture. Section 4 describes the functionality of the sample projects included in the NFC Reader Library release. In Sections 5, 6, 7 and 8 sample code examples are explained in depth. Section 9 describes the memory footprint of the NFC Reader Library components and how to optimize the memory consumption. Section 10 provides the guidelines to port a project into a different MCU. Section 11 provides a tutorial on how to create a new project from scratch. Finally, Section 12 compiles the FAQs and Section 13 (Appendix) depicts all the NFC Reader Library error codes.

Detailed description of the NFC Reader Library API is explained in a user manual UM10802 - NXP NFC Reader Library API [37].

3. Introduction

3.1 Overview of the NXP NFC Reader Library

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

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

- Bus Abstraction Layer (BAL): Implements the communication interface between the host device and the contactless reader IC.
- Hardware Abstraction Layer (HAL): Implements the hardware specific elements of the contactless reader IC and executes native commands of the chip.
- Protocol Abstraction Layer (PAL): Implements the functions for contactless card activation and contactless card protocols.
- Application Layer (AL): Implements the commands to work with several contactless smart card technologies.
- NFC Forum Tag Type Operations (TOP): Implements an API for developers to perform read and write operations on top of the four Tag Types defined in the NFC Forum specifications.

- NFC Activity: Implements a routine for sensing the RF field to detect the presence of contactless smart cards, NFC tags or other NFC-enabled devices in close proximity.
- NFC P2P Package: Implements P2P functionality based on the NFC Forum defined P2P protocol stack allowing two NFC devices to exchange data when they are brought into proximity.

The NFC Reader Library also includes an additional layer named:

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

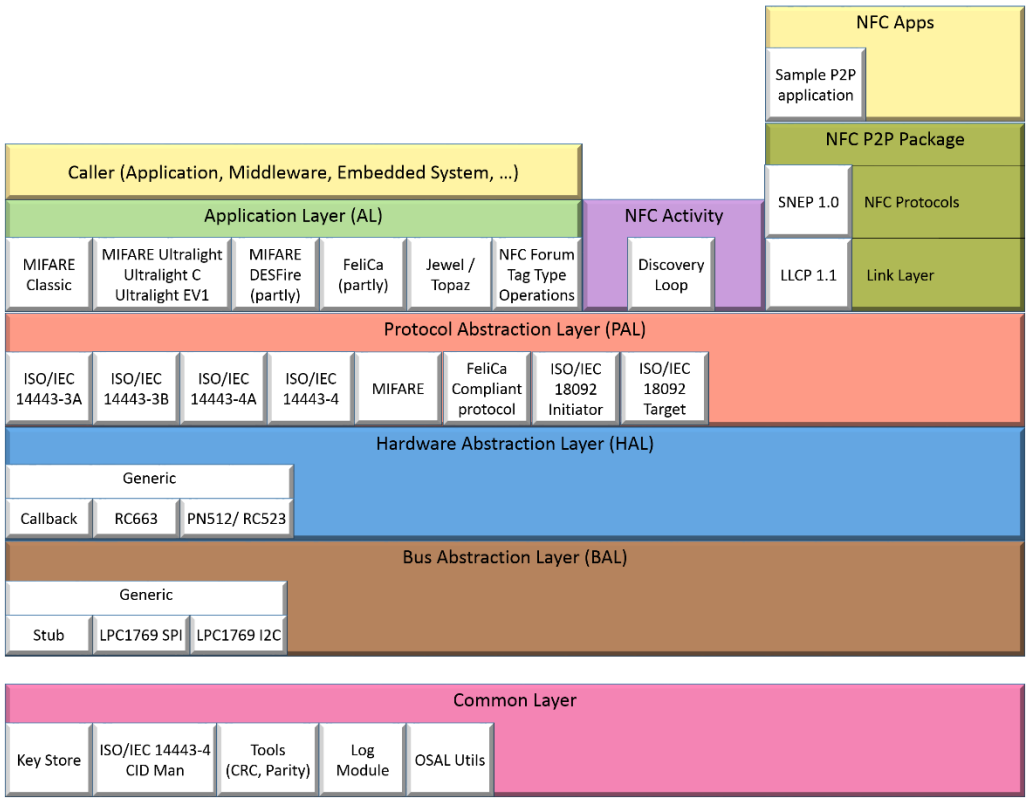

Fig 1. Layered Structure of the NFC Reader Library

3.2 NFC Reader Library Software Release Versioning Rule

The name of each of the releases of the NFC Reader Library includes information about its version, allowing to differentiate and identify them easily. This versioning information has the format of ***Trk.MAJ.min.patch+WkNumber***.

- **Trk**: It is a one-digit number that represents the NFC Reader Library track.
- **MAJ**: It is a three-digit number. Digits 1 and 2 represent the Major number of the release (e.g. the new release includes a Card Emulation API). Digit 3 means:
  - **'0'**: The NFC Reader Library does not contain Crypto components.
  - **'1'**: The NFC Reader Library contains Crypto components.

  - **‘2’**: The NFC Reader Library contains Crypto components and is only for internal use.
- **min**: It is a two-digit number that represents the Minor number of the release. It is incremented when existing features have been enhanced and for intermediate planned releases with bug fixes (e.g. optimization of specific layers such as LLCP/SNEP).
- **patch+WkNumber**: It is a six-digit number. Digits 1 and 2 are the patch number. Digits 3 to 6 represent the year and the week at which the version was released (yyww). It is incremented when bugs from the field have been fixed.

For instance, the ***NxpRdLib_PublicRelease_V_3_010_00_001407*** software release version stands for:

- **Trk**: Its Track Number is (3).
- **MAJ**: Its Major version number is (01) and the release does not include Crypto components (0).
- **min**: Its Minor version number is (00).
- **patch+WkNumber**: Its patch number is (00) and it was released the 7th week of 2014.

3.3 NFC Reader Library Software Stack

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

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

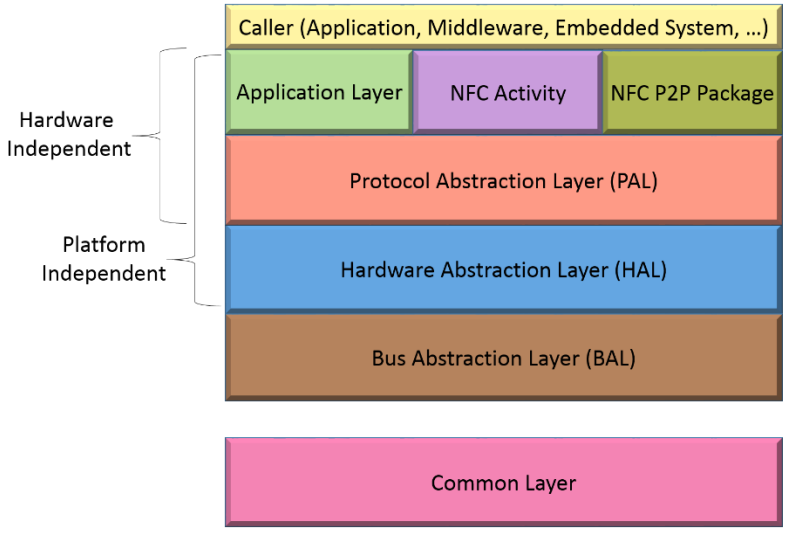

Fig 2. Hardware and Platform independent layers of the NFC Reader Library

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

3.3.1 Bus Abstraction Layer

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

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

3.3.2 Hardware Abstraction Layer

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

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

The NFC Reader Library currently supports the following contactless readers:

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

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

3.3.3 Protocol Abstraction Layer

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

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

3.3.4 Application Layer

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

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

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

3.3.5 NFC Activity

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

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

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

3.3.6 NFC P2P Package

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

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

3.3.7 Common Layer

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

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

3.3.8 Building a Project from bottom to top

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

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

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

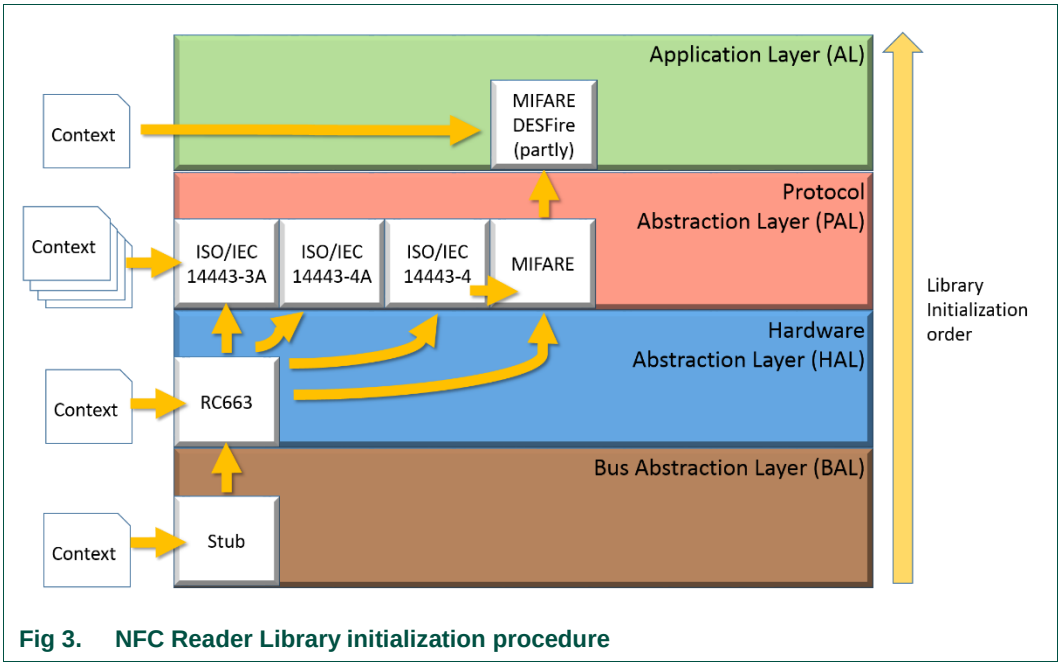

Fig 3. NFC Reader Library initialization procedure

3.4 NFC Reader Library and NFC Operating Modes

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

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

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

3.4.1 Read/Write Mode

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

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

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

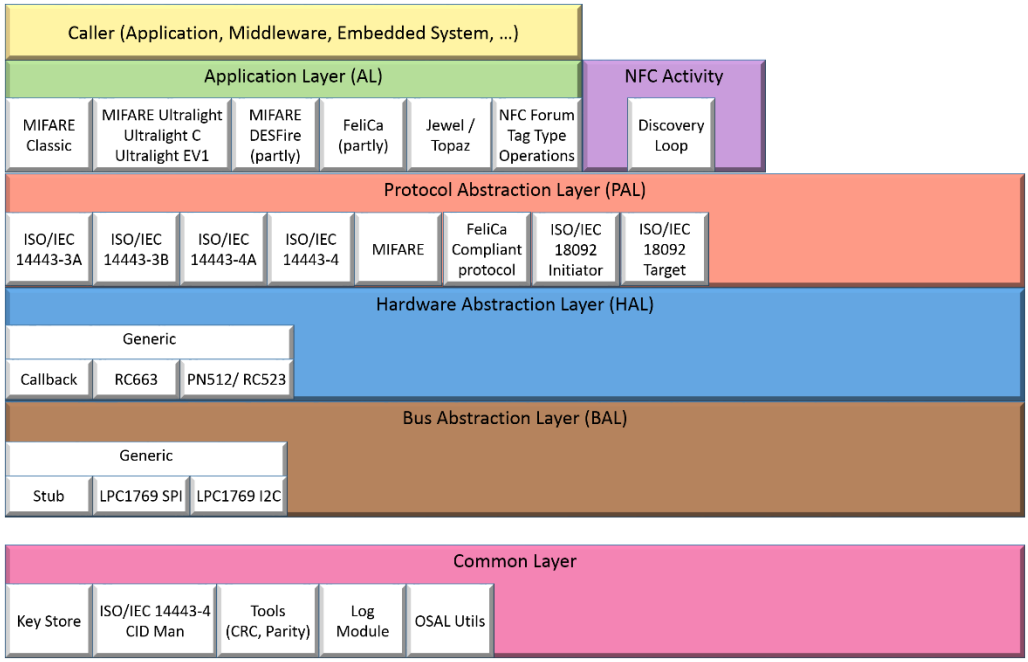

Fig 4. NFC Reader Library - Read and Write relevant modules

3.4.2 Peer-to-Peer Mode

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

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

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

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

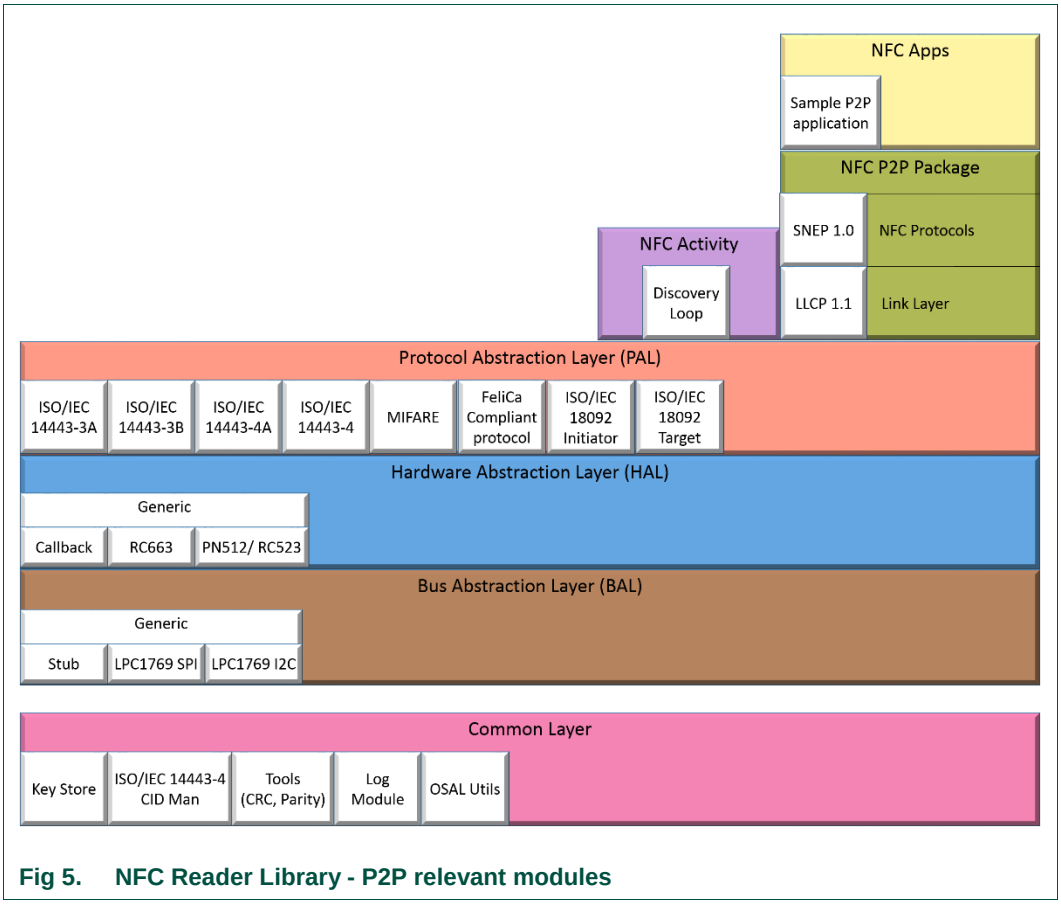

Fig 5. NFC Reader Library - P2P relevant modules

3.4.3 Card Emulation

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

3.5 NXP Export Controlled Reader Library

The Export Controlled version of the Reader Library [2] is an extension of the NXC Reader Library [1] which provides full support for MIFARE Plus and MIFARE DESFire cards and enables the usage of Secure Application Module (SAM), designed to support secure storage of cryptographic keys and the implementation of cryptographic functions in the transactions between the contactless smart card and the contactless reader.

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

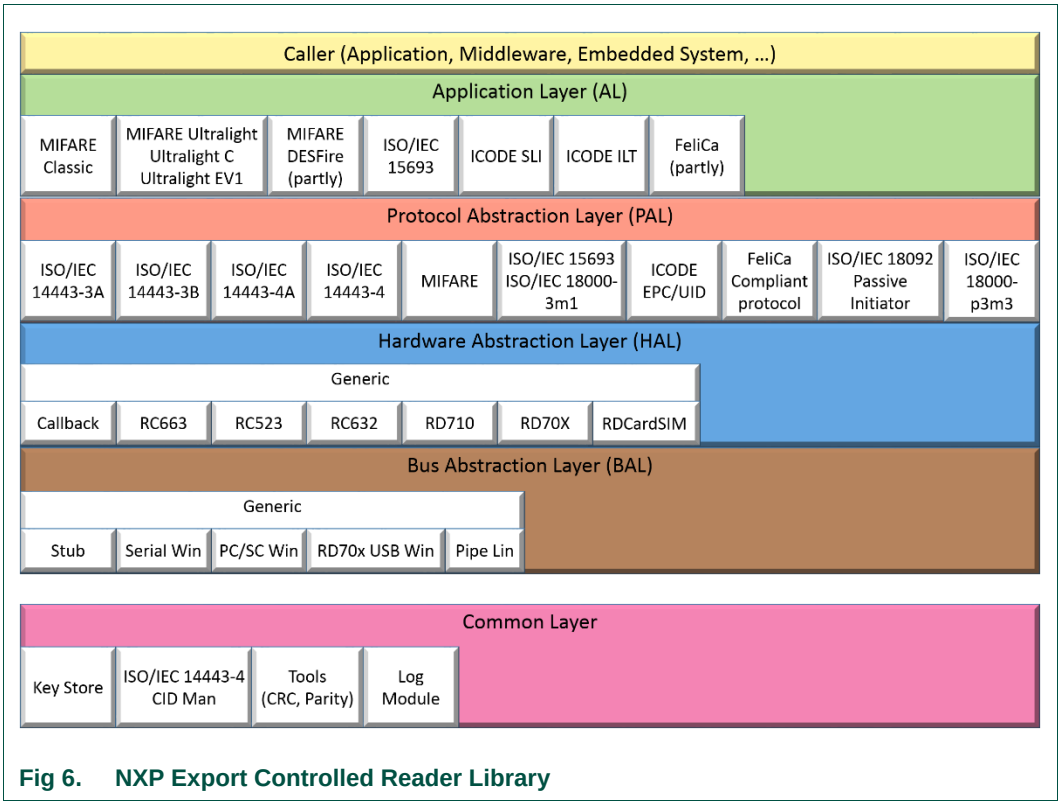

Fig 6. NXP Export Controlled Reader Library

4. Sample projects included in the software release

The NFC Reader Library v3.010 release [3] includes four sample projects:

- PN512_LPC17xx_P2P_Active_Initiator
- PN512_LPC17xx_P2P_Initiator
- PN512_LPC17xx_P2P_Target
- RC663_LPC17xx_P2P_Initiator

These sample projects are prepared to be used with either PN512 or CLRC663 NXP reader ICs (accordingly with the sample project name) and LPC1769 target board. They implement a Discovery Loop that is permanently scanning for NFC tags and P2P devices (for the Initiator projects) or waiting for a NFC Initiator device (for the Target project).

The four sample projects mentioned have configurable flags within the source code that can be enabled or disabled to provide different behaviors. The configurable flags are:

- `#define SNEP_SERVER`: This flag sets the application to act as SNEP Server. The application awaits incoming requests from the other peer.
- `#define SNEP_CLIENT`: This flag sets the application to act as SNEP Client. The application sends a URI or a text message to the other peer.
- `#define DEFAULT_SERVER`: This flag sets the SNEP Default Server as the SNEP Server type to be used.
- `#define NON_DEFAULT_SERVER`: This flag sets the SNEP Non-Default Server as the SNEP Server type to be used.
- `#define SNEP_PUT_REQUEST`: This flag enables the SNEP PUT request message.

- `#define SNEP_GET_REQUEST`: This flag enables the SNEP GET request message.
- `#define DISCOVERY_MODE`: This flag configures the reader IC communication role:
  - Initiator: `PHAC_DISCLOOP_SET_POLL_MODE | PHAC_DISCLOOP_SET_PAUSE_MODE`
  - Target: `PHAC_DISCLOOP_SET_LISTEN_MODE`
- `#define POLL_TYPE`: This flag configures the reader IC communication mode:
  - Active: `PHAC_DISCLOOP_CON_POLL_ACTIVE`
  - Passive: `PHAC_DISCLOOP_CON_POLL_A | PHAC_DISCLOOP_CON_POLL_B |PHAC_DISCLOOP_CON_POLL_F`
- `#define URIMESSAGE`: This flag makes the SNEP Client to send a URI Message to the other peer.
- `#define TEXTMESSAGE`: This flag makes the SNEP Client to send a Text Message to the other peer.

Most of the source code is shared between the four projects, but they enable and disable different macros in order to perform different operations.

Note: the reader IC used for the development of a project limits its NFC capabilities as it is explained in UM10802 – NXP NFC Reader Library API [37].

4.1 PN512_LPC17xx_P2P_Active_Initiator Project

The *PN512_LPC17xx_P2P_Active_Initiator* project configures the reader IC to act as an Initiator device in Active Communication mode:

- `#define DISCOVERY_MODE PHAC_DISCLOOP_SET_POLL_MODE | PHAC_DISCLOOP_SET_PAUSE_MODE`
- `#define POLL_TYPE PHAC_DISCLOOP_CON_POLL_ACTIVE`

Using this configuration, the reader IC can only communicate with another NFC device supporting active communication mode. Tags are not detected since they are passive elements.

4.2 PN512_LPC17xx_P2P_Initiator Project

The *PN512_LPC17xx_P2P_Initiator* project configures the reader IC to act as an Initiator device in Passive Communication mode:

- `#define DISCOVERY_MODE PHAC_DISCLOOP_SET_POLL_MODE | PHAC_DISCLOOP_SET_PAUSE_MODE`
- `#define POLL_TYPE PHAC_DISCLOOP_CON_POLL_A | PHAC_DISCLOOP_CON_POLL_B |PHAC_DISCLOOP_CON_POLL_F`

Using this configuration, the reader IC can communicate with Type A, B and F tags and P2P devices supporting passive communication mode.

4.3 RC663_LPC17xx_P2P_Initiator Project

The *RC663_LPC17xx_P2P_Initiator* project configures the reader IC to act as an Initiator device in Passive Communication mode. Note that the CLRC663 reader IC is a NFC-ready device, so it does only support Passive communication mode.

- `#define DISCOVERY_MODE PHAC_DISCLOOP_SET_POLL_MODE | PHAC_DISCLOOP_SET_PAUSE_MODE`

- `#define POLL_TYPE PHAC_DISCLOOP_CON_POLL_A | PHAC_DISCLOOP_CON_POLL_B |PHAC_DISCLOOP_CON_POLL_F`

Using this configuration, the reader IC can communicate with Types A, B and F tags and P2P device supporting passive communication mode.

4.4 PN512_LPC17xx_P2P_Target Project

The *PN512_LPC17xx_P2P_Target* project configures the reader IC to act as a Target device in Passive Communication mode:

- `#define DISCOVERY_MODE PHAC_DISCLOOP_SET_LISTEN_MODE`

Using this configuration, the reader IC waits for another NFC active device to start the communication.

5. Example: P2P Application

The four sample projects included in the NFC Reader Library software release (see Section 5) share the major part of the source code. For this reason, the *PN512_LPC17xx_P2P_Initiator* sample project (see Section 4.2) is used as reference to explain how to build a P2P application. The code fragments presented in the following subsections are extracted from the sample project source code.

The sample P2P application explained in this section has the following development workflow:

1. Initialization of the NFC Reader Library lower layer components.
2. Configuration and start of the discovery polling loop.
3. In case a P2P device is detected: Initialization of the LLCP and SNEP components. The SNEP component implements either a SNEP Client or a SNEP Server.
4. In case no P2P device is detected, the loop is started again.

The Fig 7 illustrates the P2P application workflow:

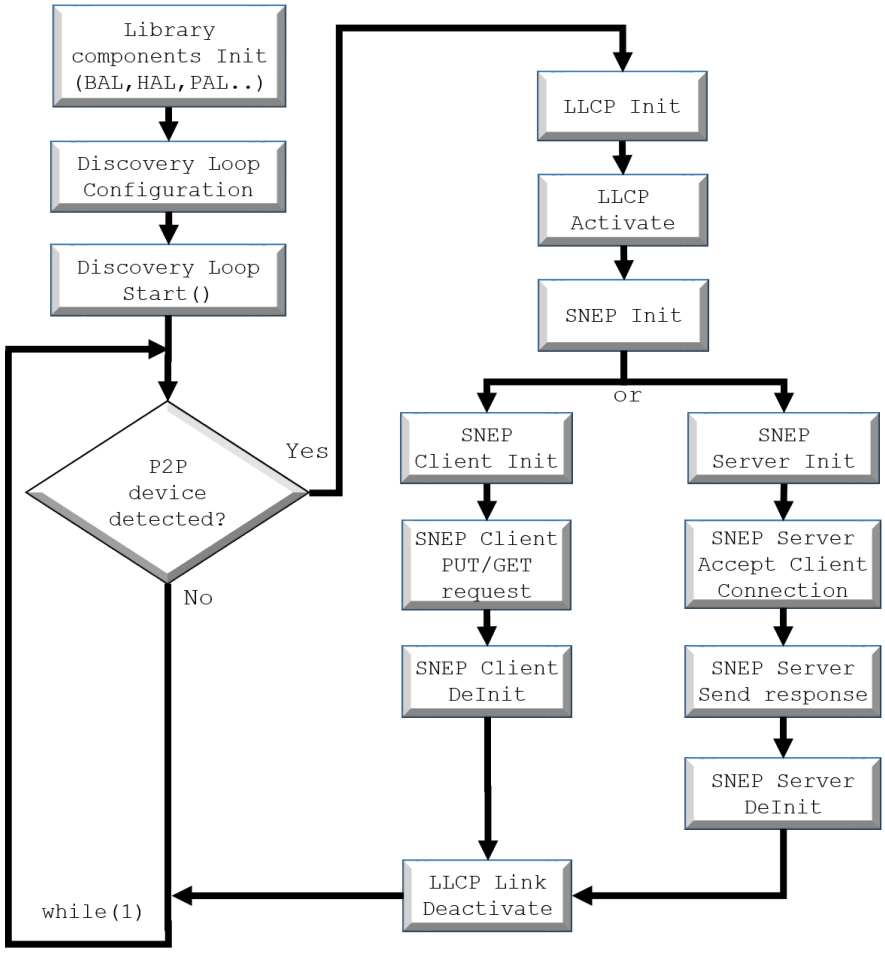

Fig 7. P2P sample application development workflow

The explanation of this sample P2P application is divided into subsections. Section 5.1 initializes the NFC Reader Library components from BAL to PAL layers and the OSAL layer. Section 5.2 details how to configure and start the Discovery Loop. Section 5.3 details how to implement a SNEP Client or a SNEP Server to enable the P2P communication. Finally, Section 5.4 shows a sample application logic that selects the message to be transmitted to the other peer.

Note: Some of the functions explained in NXP NFC Reader Library API user manuals are not used in the following examples since they are called internally by upper layer services in the stack.

5.1 NFC Reader Library Initialization

The first step to be completed in any project is the initialization of the NFC Reader Library components required by the application. The set of components to be initialized depends on the hardware in use and on the application to be developed.

The BAL layer is configured in accordance with the MCU and the communication interface to be used. The project taken as reference (*PN512_LPC17xx_P2P_Initiator*) uses the LPC1769 MCU.

The HAL layer initializes the component that refers to the reader IC to be used. In this case, PN512 Blueboard is used.

The PAL layer sets up the contactless protocols that are going to be used in the application. This sample P2P application implements a Discovery Loop which is permanently scanning for NFC P2P devices (A P2P device can use Type A or Type F contactless protocols). Therefore, ISO/IEC 14443-A, FeliCa and ISO/IEC 18092 contactless protocols are initialized.

Finally, the OSAL component is also initialized as it is required for the Discovery Loop for the calculation of time intervals between the sensing of different contactless protocols. The OSAL is also required in P2P application to calculate time intervals for the communication establishment and data exchange.

Fig 8 highlights in yellow the components that are going to be initialized in this section.

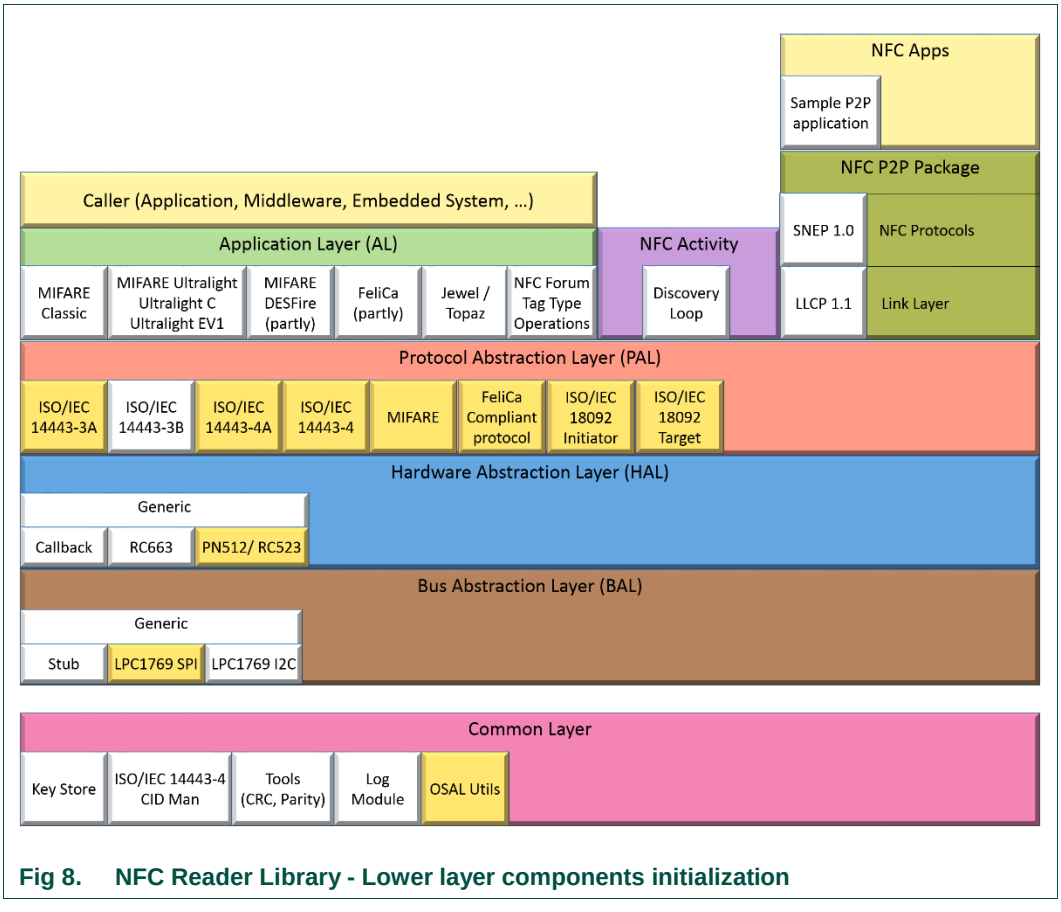

Fig 8. NFC Reader Library - Lower layer components initialization

Therefore, the following data parameter components shall be declared:

```
1    phbalReg_Lpc1768Spi_DataParams_t   balReader;      /* LPC1769 BAL component */
2    phhalHw_Rc523_DataParams_t         hal;           /* PN512 HAL componen */
3    phpalI14443p3a_Sw_DataParams_t     palI14443p3a;  /* PAL I14443-A component */
4    phpalI14443p4a_Sw_DataParams_t     palI14443p4a;  /* PAL I14443-4A component */
5    phpalI14443p4_Sw_DataParams_t      palI14443p4;   /* PAL I14443-4 component */
6    phpalFelica_Sw_DataParams_t        palFelica;     /* PAL Felica component */
7    phpalI18092mPI_Sw_DataParams_t     palI18092mPI;  /* PAL mPI component */
8    phpalMifare_Sw_DataParams_t        palMifare;     /* PAL Mifare component */
9    phOsal_Lpc17xx_DataParams_t        osal;          /* OSAL component holder */
```

5.1.1 BAL Layer Initialization

The BAL Layer is in charge of setting up the communication between the MCU and the contactless reader. In this example, the LPC1769 SPI component is initialized.

```
10   /* Initialize the Reader BAL (Bus Abstraction Layer) component */
11   phbalReg_Lpc1768Spi_Init(&balReader, sizeof(phbalReg_Lpc1768Spi_DataParams_t));
```

5.1.2 HAL Layer Initialization

The next layer to be initialized is the HAL. The HAL layer is in charge of initializing the contactless reader specifics. Therefore, the structure related with the PN512 contactless reader (`phhalHw_Rc523_Init`) shall be used.

```
12   /* Initialize the Reader HAL (Hardware Abstraction Layer) component */
13   status = phhalHw_Rc523_Init(
14      &hal,
15      sizeof(phhalHw_Rc523_DataParams_t),
16      &balReader,
17      0,
18      bHalBufferTx,
19      sizeof(bHalBufferTx),
20      bHalBufferRx,
21      sizeof(bHalBufferRx));
```

The `phhalHw_Rc523_Init()` function selects the RS232 interface by default. As the SPI interface is set up in the BAL layer, the developer needs to manually configure the HAL layer to use SPI interface:

```
22   /* Set the parameter to use the SPI interface */
23   hal.bBalConnectionType = PHHAL_HW_BAL_CONNECTION_SPI;
```

Finally, the `phbalReg_OpenPort()` is called to establish a communication channel between the MCU and the reader IC.

```
24   status = phbalReg_OpenPort(&balReader);
25   CHECK_SUCCESS(status);
```

5.1.3 PAL Layer Initialization

In the PAL Layer, the contactless protocols are initialized depending on which card or tag we aim to establish a communication with. The Discovery Loop is configured to be permanently scanning for NFC P2P devices (Type A and Type F contactless protocols). Therefore, the ISO/IEC 14443-A, FeliCa and ISO/IEC 18092 contactless protocols are initialized.

```
26   /* Initialize the I14443-A PAL layer */
27   status = phpalI14443p3a_Sw_Init(&palI14443p3a,
28      sizeof(phpalI14443p3a_Sw_DataParams_t), &hal);
29   CHECK_SUCCESS(status);
30
31   /* Initialize the I14443-A PAL component */
```

```
32   status = phpalI14443p4a_Sw_Init(&palI14443p4a,
33      sizeof(phpalI14443p4a_Sw_DataParams_t), &hal);
34   CHECK_SUCCESS(status);
35
36   /* Initialize the I14443-4 PAL component */
37   status = phpalI14443p4_Sw_Init(&palI14443p4,
38      sizeof(phpalI14443p4_Sw_DataParams_t), &hal);
39   CHECK_SUCCESS(status);
40
41   /* Initialize the Mifare PAL component */
42   status = phpalMifare_Sw_Init(&palMifare, sizeof(phpalMifare_Sw_DataParams_t),
43      &hal, &palI14443p4);
44   CHECK_SUCCESS(status);
45
46   /* Initialize PAL Felica PAL component */
47   status = phpalFelica_Sw_Init(&palFelica, sizeof(phpalFelica_Sw_DataParams_t),
48      &hal);
49   CHECK_SUCCESS(status);
50
51   /* Init 18092 PAL component */
52   status = phpalI18092mPI_Sw_Init(&palI18092mPI,
53      sizeof(phpalI18092mPI_Sw_DataParams_t), pHal);
54   CHECK_SUCCESS(status);
55
```

5.1.4 OSAL Layer Initialization

The Operating System Abstraction Layer provides abstraction of the MCU's features to the embedded software. In our case, the MCU in use is the LPC1769. The timers components are also initialized as they will be used in P2P-based applications.

```
56       /* Initialize the LPC17xx timers component */
57       status = phOsal_Lpc17xx_Init(&osal);
58       CHECK_SUCCESS(status);
59
60      /* Initialize the timer component */
61       status = phOsal_Timer_Init(&osal);
62       CHECK_SUCCESS(status);
```

5.2 Discovery Loop

This section details the initialization and configuration of the Discovery Loop routine for the detection of P2P devices. In Fig 9, the NFC Activity component is highlighted in yellow and in grey the components that have already been initialized in the previous section.

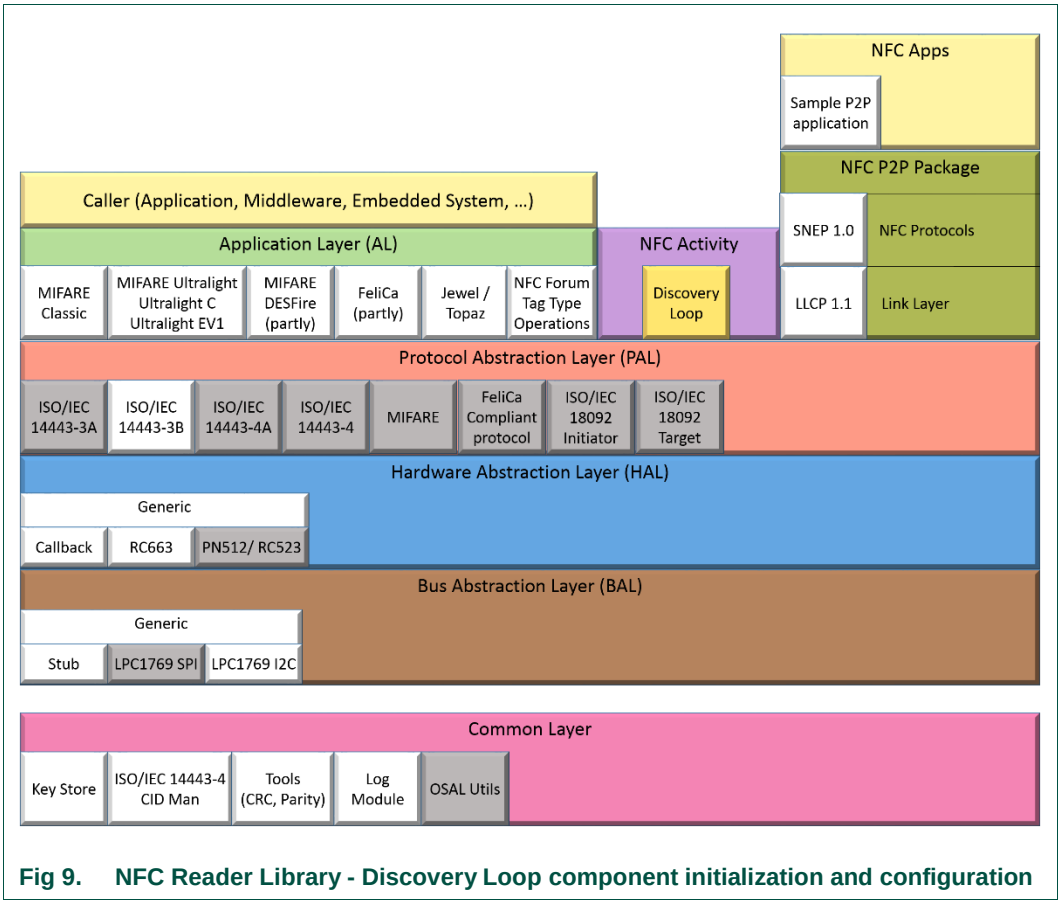

Fig 9. NFC Reader Library - Discovery Loop component initialization and configuration

5.2.1 Discovery Loop Initialization

The first step is to declare and initialize the Discovery Loop component. The Discovery Loop component initialization is done using the `phacDiscLoop_Sw_Init` function. This function sets all the structure parameters to zero or to their default values.

```
63    /* Discovery Loop component declaration */
64    phacDiscLoop_Sw_DataParams_t discLoop;

65    /* Initialize the Discovery Loop component */
66    phacDiscLoop_Sw_Init(
67       &discLoop,
68       sizeof(phacDiscLoop_Sw_DataParams_t),
69       &hal,
70       &osal);
```

The Discovery Loop is a routine that sequentially sets the reader IC in different functional configurations so it can discover the presence of tags or NFC devices in the field. Therefore, it is required to set the pointer to the corresponding PAL component for each technology to be sensed into the Discovery Loop structure. The PAL components have ben initialized previously (see Section 5.1). The pointers to be set for the P2P application are:

```
71    discLoop.pPal1443p3aDataParams  = &palI14443p3a; //ISO/IEC 14443-3A PAL component
72    discLoop.pPal1443p4aDataParams  = &palI14443p4a; //ISO/IEC 14443-4A PAL component
```

```
73    discLoop.pPal18092mPIDataParams = &palI18092mPI; //ISO/IEC 18092 PAL component
74    discLoop.pPalFelicaDataParams   = &palFelica;    //FeliCa PAL component
```

After this, it is required to configure the Discovery Loop according to the P2P application requirements. The Discovery Loop parameters can be configured using the `phacDiscLoop_SetConfig()` function.

5.2.2 Discovery Loop Configuration

The Discovery Loop configuration identifiers are listed in the *NxpRdLib_PublicRelease/intfs/phacDiscLoop.h* file and the developer can recognize them because they use the **`PHAC_DISCLOOP_CONFIG_`**`XXX_XXX` naming scheme. Each identifier can be configured using the `phacDiscLoop_SetConfig()` function.

The `PHAC_DISCLOOP_CONFIG_DETECT_TAGS`, `PHAC_DISCLOOP_CONFIG_MODE` and `PHAC_DISCLOOP_CONFIG_NUM_POLL_LOOPS` configurations are exaplined in this section.

5.2.2.1 Communication Mode Configuration

In NFC technology, both Active and Passive communication modes are possible. The developer can configure the reader IC communication mode using the Discovery Loop settings. The identifier to configure the reader IC communication mode is: `#define PHAC_DISCLOOP_CONFIG_DETECT_TAGS`

The macros defined to set up the reader IC in Active or Passive communication modes are the following:

Table 1. reader IC communication mode configuration

Communication mode	Macro
Active Communication	`PHAC_DISCLOOP_CON_POLL_ACTIVE`

Passive Communication	`PHAC_DISCLOOP_CON_POLL_A`
	`PHAC_DISCLOOP_CON_POLL_B`
	`PHAC_DISCLOOP_CON_POLL_F`

The Active communication mode can be configured in the Discovery Loop using the `phacDiscLoop_SetConfig()` function in the following way:

```
75    /*Enable Technology type */
76    #define POLL_TYPE  PHAC_DISCLOOP_CON_POLL_ACTIVE
77
78    /* Set for detection of TypeA and Type F P2P devices */
79    status = phacDiscLoop_SetConfig(
80       pDataParams,
81       PHAC_DISCLOOP_CONFIG_DETECT_TAGS,
82       POLL_TYPE);
```

Using this configuration, the reader IC senses the field for P2P active peers.

The Passive communication mode can be configured in the Discovery Loop using the `phacDiscLoop_SetConfig()` function in the following way:

```
83    /*Enable Technology type */
84    #define POLL_TYPE
85       PHAC_DISCLOOP_CON_POLL_F | PHAC_DISCLOOP_CON_POLL_A
86
```

```
87    /* Set for detection of TypeA, TypeB and Type F tags */
88    status = phacDiscLoop_SetConfig(
89       pDataParams,
90       PHAC_DISCLOOP_CONFIG_DETECT_TAGS,
91       POLL_TYPE);
```

Using this configuration, the reader IC senses the field looking for P2P devices with Passive communication mode capabilities using ISO 14443-A (Type A) or FeliCa (Type F) contactless protocols.

5.2.2.2 Communication Role Configuration

Two communication roles are defined in NFC: Initiator and Target. The developer can configure the reader IC to work either as an Initiator or as a Target using the Discovery Loop settings. The identifier to configure the reader IC communication mode is: `#define PHAC_DISCLOOP_CONFIG_MODE`

The macros defined to set up the reader IC as Initiator or Target are the following:

Table 2. reader IC communication role configuration

Communication role	Macro
Initiator	`PHAC_DISCLOOP_SET_POLL_MODE` `PHAC_DISCLOOP_SET_PAUSE_MODE*`
Target	`PHAC_DISCLOOP_SET_LISTEN_MODE`

The `PHAC_DISCLOOP_SET_PAUSE_MODE*` flag enables a waiting time interval at the end of the Discovery Loop procedure. This flag is used to stop the field scanning for a certain time between two consecutive loops.

The Initiator role can be configured in the Discovery Loop using the `phacDiscLoop_SetConfig()` function in the following way:

```
92    /*Define Poll and Pause mode  */
93    #define DISCOVERY_MODE PHAC_DISCLOOP_SET_POLL_MODE | PHAC_DISCLOOP_SET_PAUSE_MODE
94
95    /* Set for poll and listen mode */
96    status = phacDiscLoop_SetConfig(
97       pDataParams,
98       PHAC_DISCLOOP_CONFIG_MODE,
99       DISCOVERY_MODE );
```

The `PHAC_DISCLOOP_SET_POLL_MODE` flag makes the Discovery Loop to poll through the detection sequence defined in `PHAC_DISCLOOP_CONFIG_DETECT_TAGS` flag (see 5.2.2.1).

The Target role can be configured in the Discovery Loop using the `phacDiscLoop_SetConfig()` function in the following way:

```
100   /*Define Poll and Pause mode  */
101   #define DISCOVERY_MODE PHAC_DISCLOOP_SET_LISTEN_MODE
102
103   /* Set for poll and listen mode */
104   status = phacDiscLoop_SetConfig(
105      pDataParams,
106      PHAC_DISCLOOP_CONFIG_MODE,
107      DISCOVERY_MODE );
```

5.2.2.3 Configuring the number of loop iterations

The number of iterations of the Discovery Loop can be configured using the `phacDiscLoop_SetConfig()` function in the following way (e.g. five iterations):

```
108  /* Set number of polling loops to 5 */
109  status = phacDiscLoop_SetConfig(
110     pDataParams,
111     PHAC_DISCLOOP_CONFIG_NUM_POLL_LOOPS,
112     5);
```

5.2.3 Discovery Loop: Start

Once the Discovery Loop has been configured, the developer can start it. The function to start the Discovery Loop is:

```
113  /* Start the Discovery Loop */
114  status = phacDiscLoop_Start(pDataParams);
```

5.2.4 Discovery Loop: P2P Device Detection

The detection of a P2P device shall be done after one loop iteration is completed. This can be done using the `phacDiscLoop_GetConfig()` function and the `PHAC_DISCLOOP_CONFIG_TAGS_DETECTED` identifier:

```
115  /* Get the Type tags or P2P devices detected info */
116  status = phacDiscLoop_GetConfig(pDataParams,
117     PHAC_DISCLOOP_CONFIG_TAGS_DETECTED,
118     &wTagsDetected);
```

There are bitmasks defined that can be used to check whether a particular Type Tag or NFC device has been detected. To verify if a P2P device using ISO 14443-A protocol was detected, the `PHAC_DISCLOOP_TYPEA_DETECTED_TAG_P2P` bitmask shall be used is:

```
119  if(PHAC_DISCLOOP_CHECK_ANDMASK(wTagsDetected, PHAC_DISCLOOP_TYPEA_DETECTED_TAG_P2P)){
120      printf ("Type A P2P device detected ");
121
122      //
123      // Your application code to work with P2P devices
124      //
125  }
```

`PHAC_DISCLOOP_CHECK_ANDMASK`: Macro that logically ANDs two values. If the corresponding bit is set, then non-zero value is returned. Otherwise, zero is returned.

`&wTagsDetected`: The binary map indicating which Type tags were found.

`PHAC_DISCLOOP_TYPEA_DETECTED_TAG_P2P`: P2P device using ISO 14443-A protocol macro.

5.3 NFC P2P Package

This section explains how to implement a P2P application which establishes a bidirectional channel between the reader IC and another NFC device to exchange data when they are brought into close proximity.

In Fig 10, NFC P2P package components are highlighted in yellow and in grey, the components that have already been initialized.

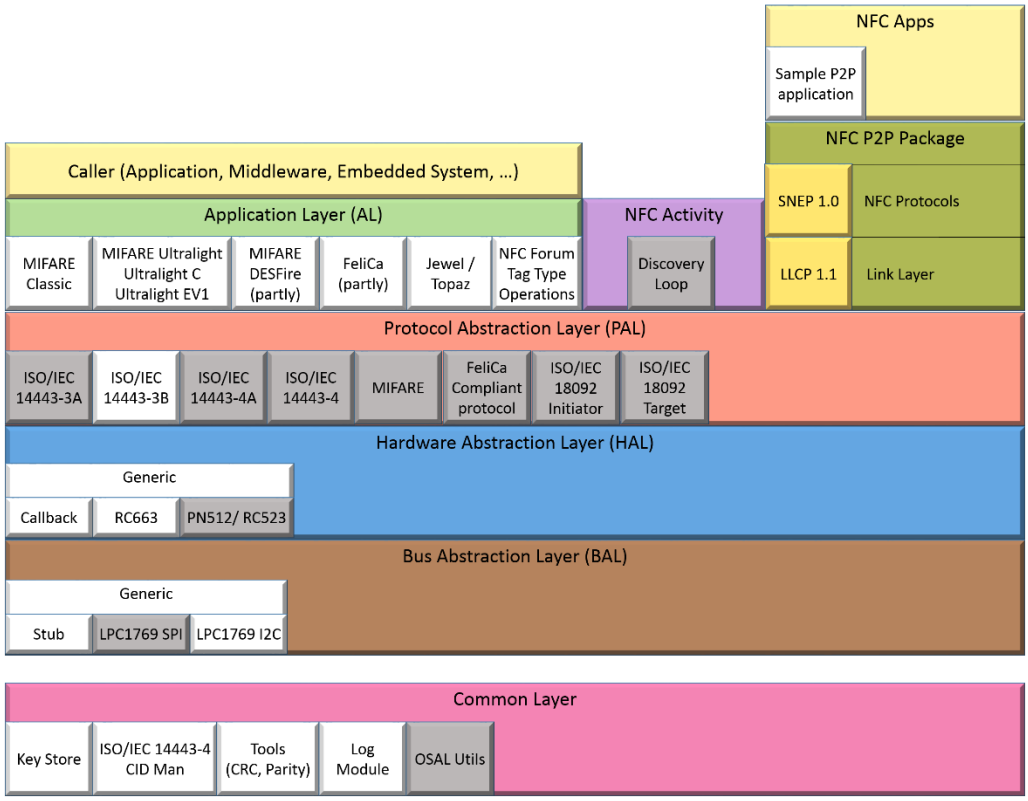

Fig 10. NFC Reader Library – NFC P2P package components

The NFC P2P package consists of two layers, the LLCP and the SNEP. Both of them need to be initialized and used for setting up the P2P communication.

5.3.1 LLCP

In this section, how to initialize and how to configure the LLCP Component for enabling the transmission of upper layer data units is explained.

The different tasks to be done by the developer in order to create a link connection with the remote peer device, transmit upper layer data units and finally close the link connection are described in this section.

5.3.1.1 LLCP Component Initialization

The LLCP component makes use of several structures (described in UM10802 [37]) for the management of the connection between peers.

```
126  phlnLlcp_Fri_Transport_t LlcpTransport; /* LLCP transport layer component */
127  phlnLlcp_Fri_sLinkParameters_t LinkParam; /* LLCP link parameter */
128  phlnLlcp_Fri_t Llcp; /* LLCP pointer */
129  phHal_sRemoteDevInformation_t RemoteInfo; /* Remote Info component */
130  phlnLlcp_Fri_DataParams_t lnLlcpDataparams; /* LLCP Data Parameters */
```

`LlcpTransport`: LLCP Transport component management structure.

`LinkParam`: LLCP Link component management structure.

`Llcp`: LLCP component parameter structure.

`RemoteInfo`: Structure that stores remote peer device info.

`lnLlcpDataparams`: Structure that stores pointers to all the LLCP related structures.

In addition, a set of buffers and variables for the management of the component behaviour are declared. These variables are out of the scope of this explanation.

When a valid remote peer device is detected by the Discovery Loop, the local peer is ready for the initialization and configuration of the LLCP component. The LLCP initialization involves that both the LLCP Link component and the LLCP Transport component must be initialized.

The `phHal_sRemoteDevInformation_t` structure that handles the information of the remote peer device must be manually initialized by the developer (described in UM10802 [37]). This structure is defined within the `phlnLlcp_Fri_Mac_t` structure of the LLCP Mac Mapping component, which is part of the LLCP component main structure.

```
131  if (PHAC_DISCLOOP_CHECK_ANDMASK(wTagsDetected, PHAC_DISCLOOP_TYPEA_DETECTED))  {
132      if (wTagsDetected & PHAC_DISCLOOP_TYPEA_DETECTED_TAG_P2P) {
133          printf("Type A device has been detected \n");
134
135          /* push the data to the LLCP protocol data structure */
136          RemoteInfo.SessionOpened = 1;
137          RemoteInfo.RemDevType = phlnLlcp_Fri_eNfcIP1_Target;
138          RemoteInfo.RemoteDevInfo.NfcIP_Info.ATRInfo_Length =
139             (discLoop.sTypeATargetInfo.sTypeA_P2P.bAtrResLength - 17);
140
141          memcpy(RemoteInfo.RemoteDevInfo.NfcIP_Info.ATRInfo,
142             &discLoop.sTypeATargetInfo.sTypeA_P2P.pAtrRes[17],
143             (discLoop.sTypeATargetInfo.sTypeA_P2P.bAtrResLength - 17));
```

The initialization of the LLCP component requires a set of callings to different functions defined by the LLCP Component API, as well as the configuration of several structures. The correct flow of calls to be made for both the LLCP Link component and the LLCP Transport component is depicted in UM10802 [37].

```
144  phStatus_t NFC_LLCPInitialize(void) {
145        uint32_t DummyContext;
146        Llcp_running = true;
```

The LLCP link parameter values are initialized according to the default values defined by the NFC Forum LLCP specification.

```
147        LinkParam.miu = 128; /* The remote Maximum Information Unit
148        LinkParam.lto = 100; /* The remote Link TimeOut (in 1/100s) */
149        LinkParam.wks = 0x0001; /* The remote Well-Known Services */
150        LinkParam.option = 0x00; /* The remote options */
151        bChecking = 0; /* Sync variable used for checkLlcp Callback */
```

The LLCP component initialization involves:

- Calling the LLCP component initialization function, which sets the pointers to the structures handled by the main LLCP structure.

- Calling the LLCP reset functions, this sets the default values of those structures.

```
152         status = phlnLlcp_Fri_Init(&lnLlcpDataparams, sizeof(lnLlcpDataparams),
153              &Llcp, &LinkParam, &LlcpTransport, &RemoteInfo, &pTxBuffer,
154              sizeof(pTxBuffer), &pRxBuffer, sizeof(pRxBuffer), &palI18092mPI);
155
156         status = phlnLlcp_Reset(&lnLlcpDataparams, &LinkCB, &DummyContext);
157         status = phlnLlcp_Transport_Reset(&lnLlcpDataparams);
```

The pointer to the OSAL component structure that is referenced by the `phlnLlcp_Fri_t` structure for the correct management of the LTO timeout defined by the LLCP component shall be manually added by the developer.

```
158         /* Assign the osal pointer to the LLCP after reset*/
159         Llcp.osal = &osal;
```

5.3.1.2 Link Activation

After the LLCP component initialization has been completed, the developer should proceed to complete the link activation. This is a needed step before being able to create the link connection at a later stage.

First of all the validity of the link to be activated is checked. To do that, the NFC Device checks the LLCP link parameters received in the ATR from the remote peer device. The execution of the source code that completes the initialization of the LLCP link shall be stopped until the link is validated. The completion of this task is announced via a callback function.

```
160         status = phlnLlcp_ChkLlcp(&lnLlcpDataparams, &ChkCb, (void*)
      &DummyContext);
161
162         while (bChecking == 0);
163         CHECK_SUCCESS(status);
```

Once the link is already validated by the local peer, the link needs to be activated for the transmission of upper layer services data units.

```
164         status = phlnLlcp_Activate(&lnLlcpDataparams);
165         return status;
166     }
```

LLCP Check Link Validity Callback function

The `ChkCb()` callback function defined by the `phlnLlcp_ChkLlcp()` function is executed when the link validity check is completed. The `bChecking` variable announces the validity of the link to the rest of the source code.

```
167     static void ChkCb ( void *pContext, phStatus_t status) {
168         if(status != PH_ERR_SUCCESS)
169             printf("phlnLlcp_Fri_ChkLlcp callback function status = %d \n", status);
170
171         bChecking = 1;
172     }
```

LLCP Link Status Change Callback function

The `LinkCb()` callback function defined by the `phlnLlcp_Reset()` function informs about the status change of the LLCP link.

```
173  static void LinkCB (void *pContext, phlnLlcp_Fri_eLinkStatus_t eLinkStatus) {
174      if (eLinkStatus == phlnLlcp_Fri_Mac_eLinkDeactivated)
175          Llcp_running = false;
176  }
```

5.3.1.3 Message Transmission and Reception

The LLCP component API defines a couple of functions for the transmission and the reception of LLCP PDUs. These functions are not called directly by developers as they are executed internally by upper layer services, in this case the SNEP protocol.

5.3.1.4 Link Closure

The LLCP link should remain active as long as the communication exists and should be closed when the communication is finished.

The LLCP link is deactivated by calling the `phlnLlcp_Deactivate()` function, which disconnects LLCP link connections sending DISC PDU data units to the remote peer device. Optionally, the transmission of a Deselect Request packet in the ISO/IEC 18092 layer might be performed.

```
177  phStatus_t NFC_LlcpClose()
178  {
179      /* Deactivate the LLCP layer link */
180      status = phlnLlcp_Deactivate(&lnLlcpDataparams);
181      CHECK_SUCCESS(status);
182
183      /*  De-select ISO18092 */
184      // status = phpalI18092mPI_Deselect(&palI18092mPI,
185            PHPAL_I18092MPI_DESELECT_DSL);
186
187      /* Developers code */
188
189      return status;
190  }
```

The LLCP link deactivation is notified to the application through the `LinkCB()` callback function.

5.3.2 SNEP

The SNEP protocol enables the exchange of NDEF messages between two NFC devices using P2P communication. The SNEP application can be divided into three categories: the *Session Establishment,* the *Data Exchange and* the *Session Release*.

- The *Session Establishment* includes the creation of a SNEP Client or a SNEP Server and establishing a connection between them.
- The *Data Exchange* requires sending PUT or GET requests from the SNEP Client to the SNEP Server and receiving the corresponding responses from the SNEP Server.

- The *Session release* disconnects and closes the communication channel.

5.3.2.1 SNEP Client

The SNEP Client application sends a PUT or a GET request to a SNEP Server peer in order to either push data or retrieve data from the remote peer device. The following data structures and variables will be used in the underlying sample code as part of the SNEP Client implementation:

```
191  phnpSnep_Fri_Config_t pConfigInfo;
192  phnpSnep_Fri_DataParams_t npSnepDataParams;
193  uint32_t gSocketHandle = 0;
194  phnpSnep_Fri_ClientSession_t ClientSession;
195  phNfc_sData_t sGetData;
196  phNfc_sData_t sMessage;
```

`pConfigInfo`: Defines the SNEP Server type, Service name and Socket options.

`npSnepDataParams`: The SNEP Data Parameters structure.

`gSocketHandle`: Integer that uniquely identifies the connection session.

`ClientSession`: The SNEP Client Session data structure.

`sGetData`: Buffer data sent to the SNEP Server as part of a GET request.

`sMessage`: Buffer for Text message to be sent in a PUT request.

Additionally, the following callback functions must be implemented:

```
197  static void SnepClientConnection_CB();
198  static void SnepClientPutReq_CB();
199  static void SnepClientGetReq_CB();
```

`SnepClientConnection_CB`: Pointer to the SNEP Client connection callback function

`SnepClientPutReq_CB`: Pointer to the SNEP Client PUT request callback function.

`SnepClientGetReq_CB`: Pointer to the SNEP Client PUT request callback function.

SNEP Component Initialization

The first step is to declare and initialize the SNEP component. The SNEP Component initialization is common for both the SNEP Client and the SNEP Server:

```
200  /* SNEP Fri Initialization  */
201  status = phnpSnep_Fri_Init( &npSnepDataParams,
202     sizeof(npSnepDataParams),
203     &lnLlcpDataparams ); // Pointer to the LLCP data parameter structure
```

The type of SNEP Server (Default or non-default server) to use has to be defined. The NFC Forum compatible devices mandatorily support the SNEP Default server. The use of a Default SNEP Server can be defined with:

```
204  /* Data parameter which defines the SNEP Server type and options.
205  phnpSnep_Fri_Config_t pConfigInfo;
206
207  /* For Default Server connection Service Name and length  */
208  pConfigInfo.SnepServerType = phnpSnep_Fri_Server_Default;
```

The use of a non-Default SNEP Server can be defined with:

```
209  /*  For Non Default Server connection Service Name and length  */
210  pConfigInfo.SnepServerType = phnpSnep_Fri_Server_NonDefault;
211  pConfigInfo.SnepServerName = &ServerName_NonDef;
```

SNEP Client Initialization

The `phnpSnep_Client_Init()` function initializes the SNEP Client application and tries to setup a connection channel with the SNEP Server. Once the connection is established successfully, the SNEP Client can perform the action to either push (PUT) or retrieve data (GET) from the SNEP Server. Note that the function also includes a reference to the SNEP Client connect callback function.

```
212  status = phnpSnep_Client_Init( &npSnepDataParams,
213     &pConfigInfo,
214     gSocketHandle,
215     SnepClientConnection_CB,
216     &ClientSession,
217     (void*) &DummyContext_Client ); //Client context to be passed (if any)
```

SNEP Client Connection Callback function

The SNEP Client connection callback function is triggered when the SNEP Client receives the SNEP Server response on its connection request. The first step is to look at the `status` variable value. The `status` variable stores the SNEP Server request result. If the SNEP Server has accepted the connection request, then `status==PHNPSNEP_FRI_CONNECTION_SUCCESS`

```
218  static void SnepClientConnection_CB(
219     void *pContext,      /* Context of the connect call back */
220     uint32_t ConnHandle, /* ConnHandle */
221     phStatus_t status    /* status code */)
222  {
223     if(status != PHNPSNEP_FRI_CONNECTION_SUCCESS)
224     {
225         printf("SNEP Client Connection Failed 0x=%x \n", status);
226     }
227     else
228     {
229        /* Set Flag indicating connection is Request Completed */
230        bSocketconnected = true;
231        gSocketHandle = ConnHandle;
232      }
233  }
```

SNEP Client PUT Request

The SNEP Client sends a PUT request to push data into the SNEP Server using the `phnpSnep_ClientReqPut()` function. Note that the function also includes a reference to the SNEP Client PUT request callback function.

```
234  status =  phnpSnep_ClientReqPut(
235     &npSnepDataParams,
236     gSocketHandle,
```

```
237     &sMessage,
238     SnepClientPutReq_CB,
239     (void*) &DummyContext_Client ); //Client context to be passed (if any).
```

SNEP Client PUT Request Callback function

The SNEP Client PUT request callback function is triggered after the SNEP Server responds to the client previous PUT request. This callback function implements how the SNEP Client application has to process the SNEP Server response. The `status` variable stores the SNEP Server request result. The data sent by the SNEP Server as a response is stored in the `pReqResponse` variable.

```
240  static void SnepClientPutReq_CB(
241     ph_NfcHandle ConnHandle,
242     void *pContext,
243     phStatus_t Status,
244     phNfc_sData_t *pReqResponse)
245  {
246     if (PH_ERR_SUCCESS == Status && NULL != pReqResponse)
247     {
248        sPutResponse.length = pReqResponse->length;
249        /* Copy the Response Data to the Local Buffer */
250        memcpy(sPutResponse.buffer, pReqResponse->buffer, pReqResponse->length);
251     }
252     else
253     {
254        /* No Data Received */
255        sPutResponse.length = 0;
256     }
257     /* Set Flag indicating SNEP Put Request is Completed */
258     bSnep_Put = true;
259  }
```

SNEP Client GET Request

The SNEP Client sends a GET request to retrieve data from the SNEP Server using the `phnpSnep_ClientReqGet()` function. Note that the function also includes a reference to the SNEP Client GET request callback function.

```
260  status = phnpSnep_ClientReqGet(
261     &npSnepDataParams,
262     gSocketHandle,
263     &sGetData,
264     iAcceptable_length,
265     SnepClientGetReq_CB,
266     (void*) &DummyContext_Client ); //Client context to be passed (if any)
```

Note: The default SNEP Server does not support the SNEP Client GET Requests.

SNEP Client GET Request Callback function

The SNEP Client GET request callback function is triggered after the SNEP Server responds to the client previous GET request. This callback function implements how the

SNEP Client application has to process the SNEP Server response. The `status` variable stores the SNEP Server request result. The data sent by the SNEP Server as a response is stored in the `pReqResponse` variable.

```
267  static void SnepClientGetReq_CB(
268     ph_NfcHandle ConnHandle,
269     void *pContext,
270     phStatus_t Status,
271     phNfc_sData_t *pReqResponse)
272  {
273      //your callback function code
274  }
```

SNEP Client de-Initialization

After the data exchange is completed, the communication channel has to be closed. This is done with the following function:

```
275  /*  SNEP Client De-Initialization  */
276  status = phnpSnep_Client_DeInit(
277     &npSnepDataParams,
278     gSocketHandle );
```

5.3.2.2 SNEP Server

The SNEP Server application listens for incoming SNEP Client connection requests to receive (PUT) or push (GET) application data. The SNEP Server processes the requests and responds back to the SNEP Client. The following data structures and variables will be used in the underlying sample code as part of the SNEP Server implementation:

```
279  phnpSnep_Fri_Config_t pConfigInfo;
280  phnpSnep_Fri_DataParams_t npSnepDataParams;
281  uint32_t gServer_SocketHandle ;
282  uint32_t gIncomingConnHandle[2];
283  phnpSnep_Fri_ServerSession_t ServerSession; /* One Server Session */
284  phnpSnep_Fri_ServerConnection_t pServerConnection[PHNPSNEP_MAX_SNEP_SERVER_CNT];
```

`pConfigInfo`: Defines the SNEP Server type, Service name and Socket options.

`npSnepDataParams`: The SNEP Data Parameters structure.

`gServer_SocketHandle`: The SNEP Server Socket handler.

`ServerSession`: The SNEP Server Session data structure.

Additionally, the following callback functions must be implemented:

```
285  static void SnepServerConnection_CB()
286  static void  SnepServerPutNtf_CB()
287  static void SnepServerGetNtf_CB()
288  static void SnepServerRspNtf_CB()
```

`SnepServerConnection_CB`: Pointer to the SNEP Server connection callback function.

`SnepServerPutNtf_CB`: Pointer to the SNEP Server PUT request callback function.

`SnepServerGetNtf_CB`: Pointer to the SNEP Server GET request callback function.

`SnepServerRspNtf_CB`: Pointer to the SNEP Server send response callback.

SNEP Component Initialization

The first step is to declare and initialize the SNEP component. The SNEP Component initialization is common for both the SNEP Client and the SNEP Server cases as already shown in previous Section 5.3.2.1.

SNEP Server Initialization

The `phnpSnep_Server_Init()` function initializes the SNEP Server application in order to listen for incoming requests (`pServerSession->Server_state = phnpSnep_Fri_Server_Initialized`). Note that the function also includes a reference to the SNEP Server connect callback function.

```
289  /* SNEP Server Initialization */
290  status = phnpSnep_Server_Init(
291     &npSnepDataParams,
292     &pConfigInfo,
293     SnepServerConnection_CB,
294     &gServer_SocketHandle,
295     &ServerSession,
296     (void*) &DummyContext_server );
```

SNEP Server Connect Callback function

The SNEP Server Connect callback function implements the operations to be performed by the SNEP Server application when it receives an incoming connection request from a SNEP Client.

The first step is to look at the `status` variable value. For an incoming connection request, the `status==PHNPSNEP_FRI_INCOMING_CONNECTION`. The SNEP Server can accept the incoming connection requests calling the `phnpSnep_Server_Accept()` function.

The `phnpSnep_Server_Accept()` function creates a connection context between the SNEP Server and the SNEP Client. Note that the function also includes a reference to the SNEP Server PUT request and to the SNEP Server GET request callback functions.

```
297  static void SnepServerConnection_CB(
298     void *pContext,
299     uint32_t ConnHandle,
300     phStatus_t status )
301  {
302    if( (NULL!= pContext) && (status == PHNPSNEP_FRI_INCOMING_CONNECTION))
303    {
304        gIncomingConnHandle[0] = ConnHandle;
305
306        status = phnpSnep_Server_Accept(
307           &npSnepDataParams,
308           &sAppReceiveBuffer,
309           &pConfigInfo.sOptions,
310           gServer_SocketHandle,
311           gIncomingConnHandle[0],
312           SnepServerPutNtf_CB,
```

```
313           SnepServerGetNtf_CB,
314           pContext );
315     }
316     else if((status != PHNPSNEP_FRI_INCOMING_CONNECTION) &&
317        (status != PHNPSNEP_FRI_CONNECTION_SUCCESS))
318     {
319        printf("SnepServerConnection_CB callback function Failedx \n", status);
320     }
321     /* Set Flag indicating Socket connection is completed */
322     bSocketconnected = true; }
```

SNEP Server PUT Request Callback function

The SNEP Server PUT request callback function implements the the operations to be performed by the SNEP Server application when it receives a PUT request from the SNEP Client.

The SNEP Client uses the PUT request to push data to the SNEP Server. This callback function processes the incoming request, generates a response and sends it back to the SNEP Client.

The SNEP Server sends the response to the SNEP Client using the `phnpSnep_ServerSendResponse()` function. Note that the function also includes the response status code (e.g. `PH_ERR_SUCCESS)` and a reference to the SNEP Server Send Response callback function.

```
323  static void  SnepServerPutNtf_CB(
324     void *pContext,
325     phStatus_t Status,
326     phNfc_sData_t *pDataInbox,
327     ph_NfcHandle ConnHandle)
328  {
329     phnpSnep_Fri_DataParams_t *pDataParams = &npSnepDataParams;
330     uint8_t Data[] = {'S','N','E','P',' ','D','A','T','A',' ','P','U','T','\0'};
331     phNfc_sData_t   sPutData;
332     PutReqRecvdSize = 0;
333
334     /* Check pContext */
335     if (pContext== NULL)
336        return;
337
338     /* Reset the buffer length */
339     sPutResponse.length = 0;
340
341     if (PH_ERR_SUCCESS == Status )
342     {
343        if (NULL != pDataInbox &&  0!= pDataInbox->length)
344        {
345           /* Local Buffer to store the data received for PUT Request */
346           sPutResponse.buffer = pAppTempBuffer;
347
348           sPutResponse.length = pDataInbox->length;
349           /* Copy Data to the Local Buffer */
```

```
350           memcpy(sPutResponse.buffer, pDataInbox->buffer, pDataInbox->length);
351
352           /* Data length Received for PUT request */
353           PutReqRecvdSize = pDataInbox->length;
354        }
355
356        /* NEXT STEP : SNEP Server response */
357        sPutData.buffer = &Data[0];
358        sPutData.length = sizeof(Data);
359
360        /* Clear Flag Indicating Put Request */
361        bPutReq_Complete = false;
362
363        phnpSnep_ServerSendResponse(
364          DataParams,
365          gIncomingConnHandle[0],
366           &sPutData,
367           PH_ERR_SUCCESS,
368           (ph_NfcHandle) SnepServerRspNtf_CB,
369           pContext );
370        /* Set Flag Indicating Put Request is completed */
371        bPutReq_Complete = true; }}
```

SNEP Server GET Request Callback function

The SNEP Server GET request callback function implements the operations to be performed by the SNEP Server application when it receives a GET request from the SNEP Client.

The SNEP Client uses the GET request to retrieve data from the SNEP Server. This callback function processes the incoming request, generates a response and sends it to the SNEP Client.

The SNEP Server sends the response to the SNEP Client using the `phnpSnep_ServerSendResponse()` function. Note that the function also includes the response status code (e.g. `PH_ERR_SUCCESS)` and a reference to the SNEP Server Send Response callback function.

```
372  static void SnepServerGetNtf_CB(
373     void *pContext,
374     phStatus_t Status,
375     phNfc_sData_t *pDataInbox,
376     ph_NfcHandle ConnHandle)
377  {
378     phnpSnep_Fri_DataParams_t     *pDataParams = &npSnepDataParams;
379
380     uint8_t Data[] = {'D', 'e', 'm', 'o', 'n', 's', 't', 'r', 'a', 't', 'i', 'o',
381                       'n','\0'};
382     if (PH_ERR_SUCCESS == Status)
383     {
384         if( NULL != pDataInbox && 0 != pDataInbox->length )
385         {
386         /* NEXT STEP : SNEP Server response */
```

```
387
388             sPutGetData.buffer = pAppTempBuffer;
389             sPutGetData.length = PutReqRecvdSize;
390
391             /* Get the Response Data to the Local Buffer */
392             sGetResponse = *pDataInbox;
393         }
394         /* Send Default response message, if only GET request from client */
395         if( 0== PutReqRecvdSize )
396         {
397            sPutGetData.buffer = Data;
398            sPutGetData.length = sizeof(Data);
399         }
400         phnpSnep_ServerSendResponse(
401           pDataParams,
402            gIncomingConnHandle[0],
403            &sPutGetData,
404            PH_ERR_SUCCESS,
405            SnepServerRspNtf_CB,
406            pContext );
407
408         /* Set Flag Indicating Get Request is completed */
409         bGetReq_Complete = true;
410      }
411   }
```

SNEP Server Response Callback function

The SNEP Server Response callback function implements the operations to be performed by the SNEP Server application when a GET or PUT request has been completed.

```
412   void SnepServerRspNtf_CB(
413      void *pContext,
414      phStatus_t Status,
415      ph_NfcHandle ConnHandle)
416   {
417      /* Check pContext */
418      if (pContext == NULL)
419      {
420          return;
421      }
422      bPutReq_Complete = true;
423      if(1==bGetReq_Complete)
424      {
425         /* Set Flag Indicating Put and/or Get Request is completed */
426         bSnep_RspComplete = true;
427       }
428   }
```

5.4 Application Logic

The application logic is the piece of code where the developer shall implement its application functionality. This logic can be as simple or as complex as the project requires. The *PN512_LPC17xx_P2P_Initiator* example is taken as reference. This project transmits a NDEF message to the remote peer device, where two different messages can be transmitted:

- NFC Well-known RTD Text Type message.
- NFC Well-known RTD URI Type message.

The NDEF message to be sent can be selected using their respective programming defines.

```
#define URIMESSAGE      /**< Enable URI message */
#define TEXTMESSAGE     /**< Enable for 1024 bytes text message */
```

The NDEF message is stored in form of a `phNfc_sData_t` buffer type, which is defined by the NFC Reader Library.

```
phStatus_t NPPClientDemo() {
   phNfc_sData_t sMessage; /* Buffer for Text message */

   #ifdef TEXTMESSAGE
   static const uint8_t message[] = { 0xC1, 0x01, 0x00, 0x00, 0x04, 0x01,
     0x54, 0x02, 0x65, 0x6E,
   /* TEXT TO BE TRANSMITTED */ };
   #endif /* TEXTMESSAGE */

   #ifdef URIMESSAGE
   static const uint8_t message[] = { 0xC1, 0x01, 0x00, 0x00, 0x00, 0x08,
      0x55, 0x01,
   'n', 'x', 'p', '.', 'c','o','m'    };
   #endif /* URIMESSAGE */

   sMessage.buffer = (uint8_t *) message;
   sMessage.length = sizeof(message);
```

The `sMessage` buffer containing the NDEF message to be transmitted, is passed as an input parameter to the `phnpSnep_ClientReqPut()` function.

```
status = phnpSnep_ClientReqPut(&npSnepDataParams, gSocketHandle, &sMessage,
      SnepClientPutReq_CB, (void*) &DummyContext_Client);
```

6. Example: Writing NDEF Application

This sample application explains how to use the NFC Forum Tag Type Operations API to write a NDEF message in any of the four Type Tags defined by the NFC Forum specifications (described in UM10802 [37]). The *PN512_LPC17xx_P2P_Initiator* sample project (see Section 4.2) is used as reference to explain how to use the NFC Forum Tag Type Operations API. The code fragments presented in the following subsections are extracted from the sample project source code.

The sample application explained in this section has the following development workflow:

1. Initialization of the NFC Reader Library lower layer components.

2. Configuration and start of the discovery polling loop.
3. In case a tag is detected: Initialization of the NFC Forum Tag Type Operations API to write a NDEF message in the detected tag.
4. In case no tag is detected, the loop is started again.

The Fig 11 illustrates the application workflow:

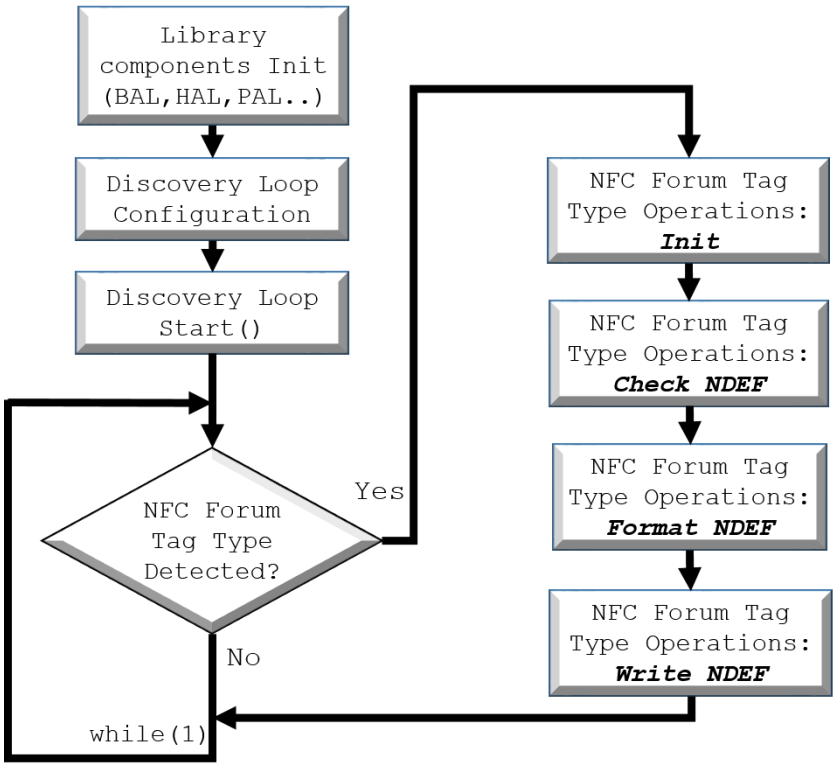

Fig 11. Writing NDEF sample application development workflow

The explanation this sample application is divided into subsections. Section 6.1 initializes the NFC Reader Library components from BAL to PAL and OSAL layers. Section 6.2 details how to configure and start the Discovery Loop for the detection of Type A, Type B and Type F tags. Section 6.3 explains how to initialize the required components of the Application Layer (AL). Finally, Section 6.4 shows a sample application logic that uses the NFC Forum Tag Type Operations API to write a NDEF message into the detected tag.

Note: Some of the functions explained in NXP NFC Reader Library API user manuals are not used in the following examples since they are called internally by upper layer services in the stack.

6.1 NFC Reader Library Initialization

The first step to be completed in any project is the initialization of the NFC Reader Library components required by the application. The set of components to be initialized depends on the hardware in use and on the application to be developed.

The project taken as reference (*PN512_LPC17xx_P2P_Initiator*) uses the LPC1769 MCU (BAL) and the PN512 Blueboard (HAL).

The PAL layer sets up the contactless technologies that are going to be used in the application. This application implements a Discovery Loop which is permanently sensing for Type A, Type B and Type F tags. Therefore, ISO/IEC 14443-A, ISO/IEC 14443-B, FeliCa contactless protocols are initialized.

Finally, the OSAL component is also initialized as it is required for the Discovery Loop to define time intervals between the sensing of the field for different contactless protocols. The Fig 12 highlights in yellow the components that are going to be initialized in this section.

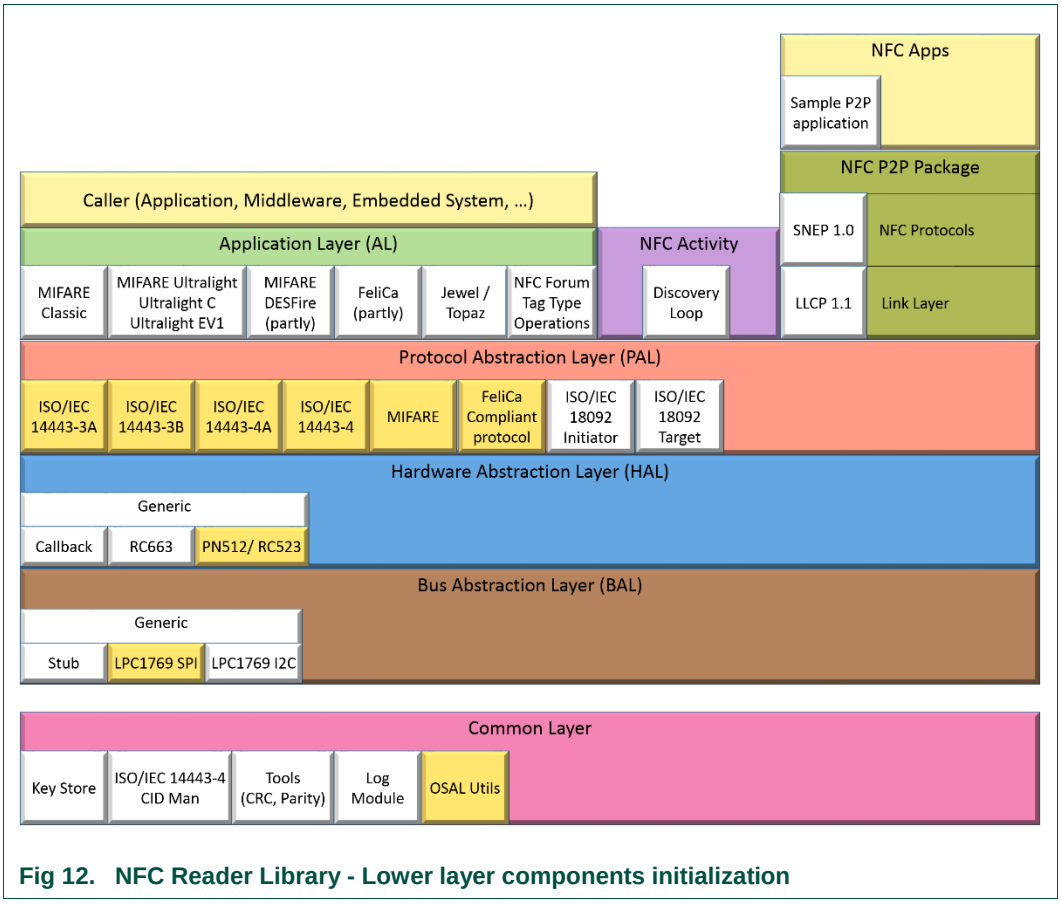

Fig 12. NFC Reader Library - Lower layer components initialization

Therefore, the following data parameter components shall be declared:

```
450  phbalReg_Lpc1768Spi_DataParams_t   balReader;      /* LPC1769 BAL component */
451  phhalHw_Rc523_DataParams_t         hal;            /* PN512 HAL componen */
452  phpalI14443p3a_Sw_DataParams_t     palI14443p3a;   /* PAL I14443-A component */
453  phpalI14443p4a_Sw_DataParams_t     palI14443p4a;   /* PAL I14443-4A component */
454  phpalI14443p3b_Sw_DataParams_t     palI14443p3b;   /* PAL I14443-B component */
455  phpalI14443p4_Sw_DataParams_t      palI14443p4;    /* PAL I14443-4 component */
456  phpalFelica_Sw_DataParams_t        palFelica;      /* PAL Felica component */
457  phpalMifare_Sw_DataParams_t        palMifare;      /* PAL Mifare component */
458  phOsal_Lpc17xx_DataParams_t        osal;           /* OSAL component holder */
```

6.1.1 BAL Layer Initialization

The BAL Layer is in charge of setting up the communication between the MCU and the contactless reader. Further information about the BAL Layer initialization for the LPC1769 SPI component be found in Section 5.1.1.

6.1.2 HAL Layer Initialization

The HAL layer is in charge of initializing the contactless reader specifics. Further information about the HAL Layer initialization for the PN512 reader IC can be found in Section 5.1.2.

6.1.3 PAL Layer Initialization

In the PAL Layer, specific contactless protocol components are initialized depending on which card or tag we aim to establish a communication with. Further information about the PAL Layer initialization for ISO/IEC 14443-A, ISO/IEC 14443-B and FeliCa protocols can be found in Section 5.1.3

6.1.4 OSAL Layer Initialization

The Operating System Abstraction Layer provides abstraction of the MCU's features to the embedded software. Further information about the OSAL Layer initialization can be found in Section 5.1.4.

6.2 Discovery Loop

This section details the configuration of the Discovery Loop routine for the detection of the four NFC Forum Type Tags. In Fig 13, the Discovery Loop components are highlighted in yellow and in grey the components that have already been initialized.

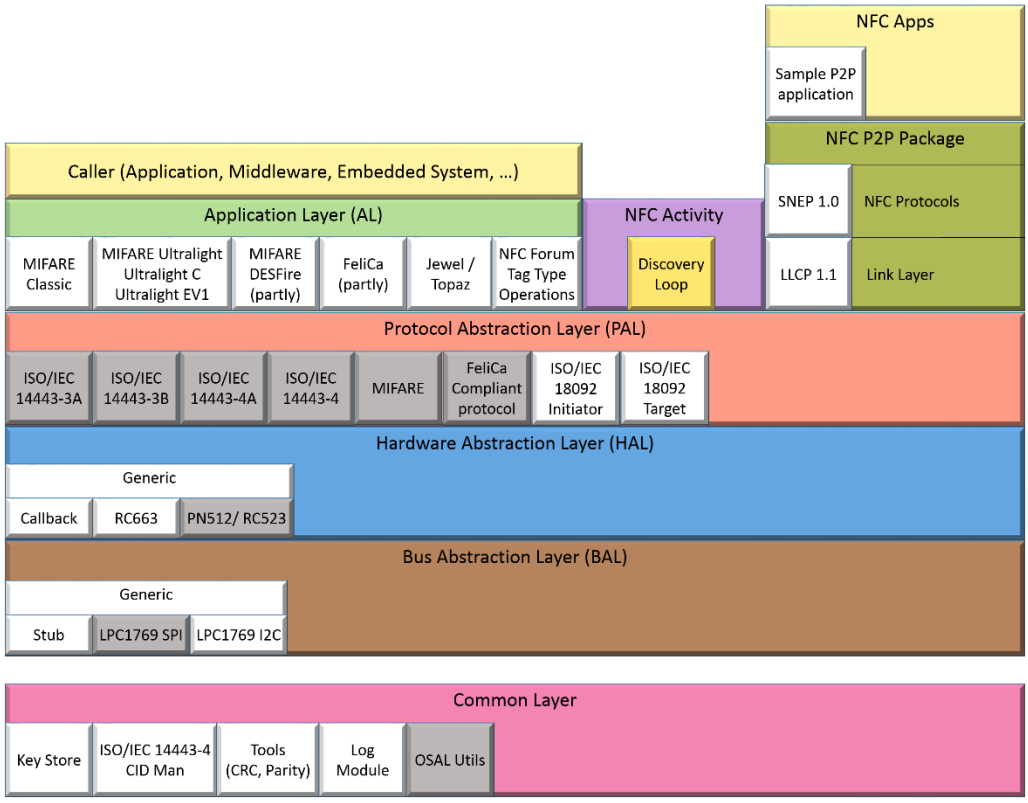

Fig 13. NFC Reader Library - Discovery Loop component initialization and configuration

The Discovery Loop section provides the specific configuration details for this example. An extended version of the Discovery Loop initialization and configuration can be found in Section 5.2.

6.2.1 Discovery Loop Initialization

The Discovery Loop component initialization is done using the `phacDiscLoop_Sw_Init` function.

```
459  /* Discovery Loop component declaration */
460  phacDiscLoop_Sw_DataParams_t discLoop;

461  /* Initialize the Discovery Loop component */
462  phacDiscLoop_Sw_Init( &discLoop,  sizeof(phacDiscLoop_Sw_DataParams_t), &hal,
463     &osal);
```

The Discovery Loop pointers to the corresponding contactless protocols components from the PAL layer for the sensing of Type A, Type B and Type F tags are:

```
464  discLoop.pPal1443p3aDataParams  = &palI14443p3a; //ISO/IEC 14443-3A PAL component
465  discLoop.pPal1443p4aDataParams  = &palI14443p4a; //ISO/IEC 14443-4A PAL component
466  discLoop.pPal1443p3bDataParams  = &palI14443p3b; //ISO/IEC 14443-3B PAL component
467  discLoop.pPalFelicaDataParams   = &palFelica;    //FeliCa PAL component
```

The Discovery Loop parameters can be configured using the `phacDiscLoop_SetConfig()` function.

6.2.2 Discovery Loop Configuration

For the sensing of Type A, Type B and Type F tags, the Passive communication mode shall be configured in the Discovery Loop. The Passive communication mode shall be configured since tags are not powered devices. The `phacDiscLoop_SetConfig()` function shall be used in the following way:

```
/*Enable Technology type */
#define POLL_TYPE
  PHAC_DISCLOOP_CON_POLL_A |PHAC_DISCLOOP_CON_POLL_B | PHAC_DISCLOOP_CON_POLL_F

/* Set for detection of TypeA, TypeB and Type F tags */
status = phacDiscLoop_SetConfig(
   pDataParams,
   PHAC_DISCLOOP_CONFIG_DETECT_TAGS,
   POLL_TYPE);
```

Additionally, the reader IC shall be configured as Initiator. The Initiator role can be configured in the Discovery Loop using the `phacDiscLoop_SetConfig()` function in the following way:

```
/*Define Poll and Pause mode  */
#define DISCOVERY_MODE PHAC_DISCLOOP_SET_POLL_MODE | PHAC_DISCLOOP_SET_PAUSE_MODE

/* Set for poll and listen mode */
status = phacDiscLoop_SetConfig(
   pDataParams,
   PHAC_DISCLOOP_CONFIG_MODE,
   DISCOVERY_MODE );
```

6.2.3 Discovery Loop: Start

After the setting up of the Discovery Loop parameters is completed, the developer can start it. The function to be used to start the Discovery Loop is:

```
/* Start the Discovery Loop */
status = phacDiscLoop_Start(pDataParams);
```

6.2.4 Discovery Loop: NFC Type Tag detection

The detection of a NFC Forum Type Tags shall be done after one loop iteration is completed. This can be done using the `phacDiscLoop_GetConfig()` function and the `PHAC_DISCLOOP_CONFIG_TAGS_DETECTED` identifier:

```
/* Get the Type tags or P2P devices detected info */
status = phacDiscLoop_GetConfig(pDataParams,
   PHAC_DISCLOOP_CONFIG_TAGS_DETECTED,
   &wTagsDetected);
```

There are bitmasks defined that can be used to check whether a particular Type Tag or NFC device has been detected. To verify if any NFC Forum Type Tags were detected, the following bitmask shall be used:

```
491     if (PHAC_DISCLOOP_CHECK_ANDMASK(wTagsDetected,
492        PHAC_DISCLOOP_TYPEA_DETECTED_TAG_TYPE1)) {
493           printf ("Type A T1 tag detected ");
494           // Your application code
495     } else if (PHAC_DISCLOOP_CHECK_ANDMASK(wTagsDetected,
496        PHAC_DISCLOOP_TYPEA_DETECTED_TAG_TYPE2)) {
497           printf ("Type A T2 tag detected ");
498           // Your application code
499     } else if (PHAC_DISCLOOP_CHECK_ANDMASK(wTagsDetected,
500        PHAC_DISCLOOP_TYPEF_DETECTED_TAG_TYPE3)) {
501           printf ("Type 3 tag detected ");
502           // Your application code
503     } else if (PHAC_DISCLOOP_CHECK_ANDMASK(wTagsDetected,
504           PHAC_DISCLOOP_TYPEA_DETECTED_TAG_TYPE4A)) {
505           printf ("Type 4A tag detected ");
506           // Your application code
507      }
```

`PHAC_DISCLOOP_CHECK_ANDMASK`: Macro that logically ANDs two values. If the corresponding bit is set, then non-zero value is returned. Otherwise, zero is returned.

`&wTagsDetected`: The binary map indicating which Type tags were found.

`PHAC_DISCLOOP_TYPEA_DETECTED_TAG_TYPE1`: Type 1 Tag detection macro.

`PHAC_DISCLOOP_TYPEA_DETECTED_TAG_TYPE2`: Type 2 Tag detection macro.

`PHAC_DISCLOOP_TYPEA_DETECTED_TAG_TYPE3`: Type 3 Tag detection macro.

`PHAC_DISCLOOP_TYPEA_DETECTED_TAG_TYPE4A`: Type 4 Tag detection macro.

6.3 AL Layer Initialization

The AL layer provides specific implementations of various contactless products. In this example, the application operates with any of the four types of NFC Forum tags. The Fig 14 highlights in yellow the components that are going to be initialized in the Application layer and in grey the components that have already been initialized.

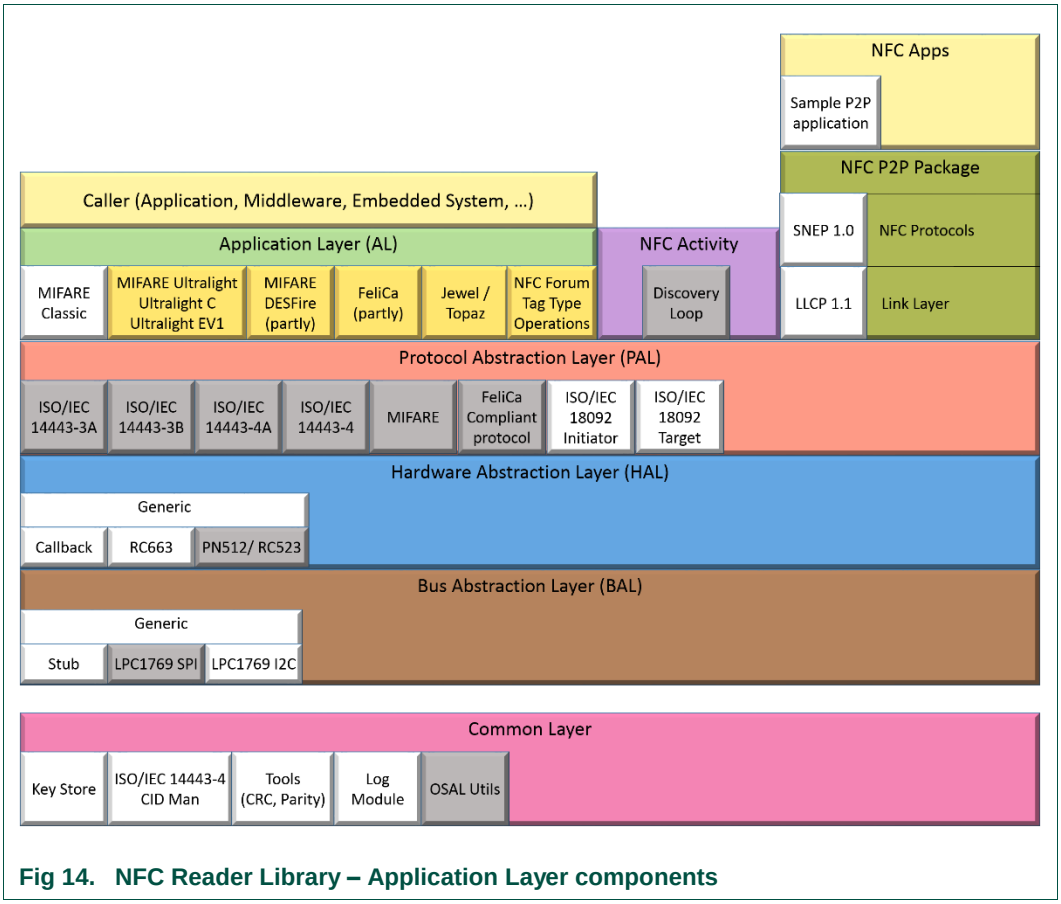

Fig 14. NFC Reader Library – Application Layer components

The following components must be initialized:

- Type 1 Tag: Initialize Jewel/Topaz (T1T AL) component.
- Type 2 Tag: Initialize MIFARE Ultralight component.
- Type 3 Tag: Initialize FeliCa component.
- Type 4 Tag: Initialize MIFARE DESFire component.
- NFC Tag Type Operations: An API to perform Read/Write operations on top of NFC Forum Type Tags.

Therefore, the following data parameter components shall be declared:

```
508  phalMful_Sw_DataParams_t          alMful;         /* AL Ultralight component */
509  phalMfdf_Sw_DataParams_t          alMfdf;         /* AL Desfire component */
510  phalFelica_Sw_DataParams_t        alFelica;       /* AL Felica component */
511  phalT1T_Sw_DataParams_t           alT1T;          /* AL T1T component */
512  phalTop_Sw_DataParams_t           tagtop;         /* AL TOP component */
513  phalTop_T1T_t                     t1tparam;       /* AL T1T TOP component */
514  phalTop_T2T_t                     t2tparam;       /* AL T2T TOP component */
515  phalTop_T3T_t                     t3tparam;       /* AL T3T TOP component */
516  phalTop_T4T_t                     t4tparam;       /* AL T4T TOP component */
```

The initialization of these components can be done in the following way:

```
/* Initialize the T1T AL component */
status = phalT1T_Sw_Init(&alT1T, sizeof(phalT1T_Sw_DataParams_t), &palI14443p3a);
CHECK_SUCCESS(status);

/* Initialize the Mful AL component */
status = phalMful_Sw_Init(&alMful, sizeof(phalMful_Sw_DataParams_t), &palMifare,
   NULL, NULL, NULL);
CHECK_SUCCESS(status);

/* Initialize the Felica AL component */
status = phalFelica_Sw_Init(&alFelica, sizeof(phalFelica_Sw_DataParams_t),
   &palFelica);
CHECK_SUCCESS(status);

/* Initialize the MF DesFire EV1 component */
status = phalMfdf_Sw_Init(&alMfdf, sizeof(phalMfdf_Sw_DataParams_t), &palMifare,
   NULL, NULL, NULL, &hal);
CHECK_SUCCESS(status);
```

After these components are declared and initialized, the NFC Forum Tag Type Operations (TOP) component shall be initialized.

```
/* Initialize the NFC Forum Tag Type Operations component */
status = phalTop_Sw_Init(&tagop, sizeof(phalTop_Sw_DataParams_t), &t1tparam,
   &t2tparam, &t3tparam, &t4tparam, NULL);
CHECK_SUCCESS(status);
```

The NFC Forum Tag Type Operations component relies on the AL components API for the execution of Read / Write operations. Therefore, the corresponding AL component associated to each the NFC Forum Tag Type must be previously initialized and has to be referenced into the NFC Forum Tag Type Operations (TOP) component.

```
((phalTop_T1T_t *)(tagop.pT1T))->phalT1TDataParams = &alT1T;
((phalTop_T2T_t *)(tagop.pT2T))->phalT2TDataParams = &alMful;
((phalTop_T3T_t *)(tagop.pT3T))->phalT3TDataParams = &alFelica;
((phalTop_T4T_t *)(tagop.pT4T))->phalT4TDataParams = &alMfdf;
```

6.4 Application Logic

This example uses the Discovery Loop component for the detection and initialization of tags in the field. Therefore, the operations to be performed on the tag shall be programmed after the Discovery Loop detection procedure. After the detection and activation of a tag, a NDEF message will be written into the detected tag.

```
status = phacDiscLoop_Start(pDataParams);

if ((status & PH_ERR_MASK) == PH_ERR_SUCCESS) {
   /* Get the Type tags detected info */
   status = phacDiscLoop_GetConfig(pDataParams,
      PHAC_DISCLOOP_CONFIG_TAGS_DETECTED, &wTagsDetected);

   if (PHAC_DISCLOOP_CHECK_ANDMASK(wTagsDetected,
      PHAC_DISCLOOP_TYPEA_DETECTED_TAG_TYPE1)) {
         printf ("Type A T1 tag detected ");
```

```
553             status = WriteNDEF(PHAL_TOP_TAG_TYPE_T1T_TAG);
554         } else if (PHAC_DISCLOOP_CHECK_ANDMASK(wTagsDetected,
555            PHAC_DISCLOOP_TYPEA_DETECTED_TAG_TYPE2)) {
556             printf ("Type A T2 tag detected ");
557             status = WriteNDEF(PHAL_TOP_TAG_TYPE_T2T_TAG);
558         } else if (PHAC_DISCLOOP_CHECK_ANDMASK(wTagsDetected,
559            PHAC_DISCLOOP_TYPEF_DETECTED_TAG_TYPE3)) {
560             printf ("Type 3 tag detected ");
561             status = WriteNDEF(PHAL_TOP_TAG_TYPE_T3T_TAG);
562         } else if (PHAC_DISCLOOP_CHECK_ANDMASK(wTagsDetected,
563             PHAC_DISCLOOP_TYPEA_DETECTED_TAG_TYPE4A)) {
564             printf ("Type 4A tag detected ");
565             status = WriteNDEF(PHAL_TOP_TAG_TYPE_T4T_TAG);
566          }
```

The *PN512_LPC17xx_P2P_Initiator* example implements a `WriteNDEF()` function. This function declares the NDEF message to be written and uses the NFC Forum Tag Type Operations API to deal with the underlying card technology. The Type tag is passed as reference to the `WriteNDEF()` function.

First, the NDEF message to be written is declared:

```
567   phStatus_t WriteNDEF(uint8_t TopTagType){
568      phStatus_t    status;
569      uint8_t bNdefData[16] = {0xD1, 0x01, 0x08, 0x55, 0x01, 0x6E, 0x78, 0x70,
570                               0x2E, 0x63, 0x6F, 0x6D};
```

The identifier to configure the Type tag to be used by the NFC Forum Tag Type Operations component is:

```
571   #define PHAL_TOP_CONFIG_TAG_TYPE
572   status = phalTop_SetConfig(&tagop, PHAL_TOP_CONFIG_TAG_TYPE, TopTagTypeDetected);
573   CHECK_SUCCESS(status);
```

Later, the `CheckNdef()` function is called to gather information about tag specific configuration and to check the correct format of the tag. It also verifies whether there is any previous NDEF message stored.

If the tag is not properly formatted for the storage of NDEF messages, the `phalTop_FormatNdef()` function shall be called.

```
574   status = phalTop_CheckNdef(&tagop, &bNdefPresence);
575   CHECK_SUCCESS(status);
576
577   status = phalTop_GetConfig(&tagop, PHAL_TOP_CONFIG_TAG_FORMATTABLE,
578      &TagFormattable);
579
580   if(TagFormattable == PH_SUPPORTED && (bNdefPresence == false))  {
581      status = phalTop_FormatNdef(&tagop);
582      CHECK_SUCCESS(status);
583   }
```

Finally, the NDEF message is written in the tag memory by calling the `phalTop_WriteNdef()` function. The NFC Reader Library core takes care of using the proper Type Tag commands for this purpose.

```
584   if((TagFormattable == PH_SUPPORTED) || (bNdefPresence == true)) {
```

```
585     status = phalTop_WriteNdef(&tagop, bNdefData, wNdefLen);
586     CHECK_SUCCESS(status);
587   } else {
588     printf("\n Not an valid NDEF Tag\n");
589     return PH_ERR_SUCCESS;
590   }
591   return status;
592   }
```

7. Example: MIFARE Classic

The MIFARE Classic example explains the initialization of the NFC Reader Library layers (from bottom to top) and the establishment of the communication with a MIFARE Classic card. This example activates a MIFARE Classic card, retrieves its UID and writes one data block on the MIFARE Classic IC memory. The Key Store component in the Common Layer is also initialized as it is required to store MIFARE keys for cryptographic and authentication operations

7.1 NFC Reader Library Initialization

The first step is to initialize the NFC Reader Library in accordance with the hardware and the application to be developed. Similarly as with the previous examples, the hardware used for this example is a LPC1769 MCU (BAL) and a PN512 reader IC (HAL). For a MIFARE Classic application, the ISO/IEC 14443-3A and MIFARE PAL components are required. Finally, the MIFARE Classic component in the AL layer shall be initialized in order to use the MIFARE Classic command set. The Fig 15 highlights in yellow the different components that are going to be initialized in this example.

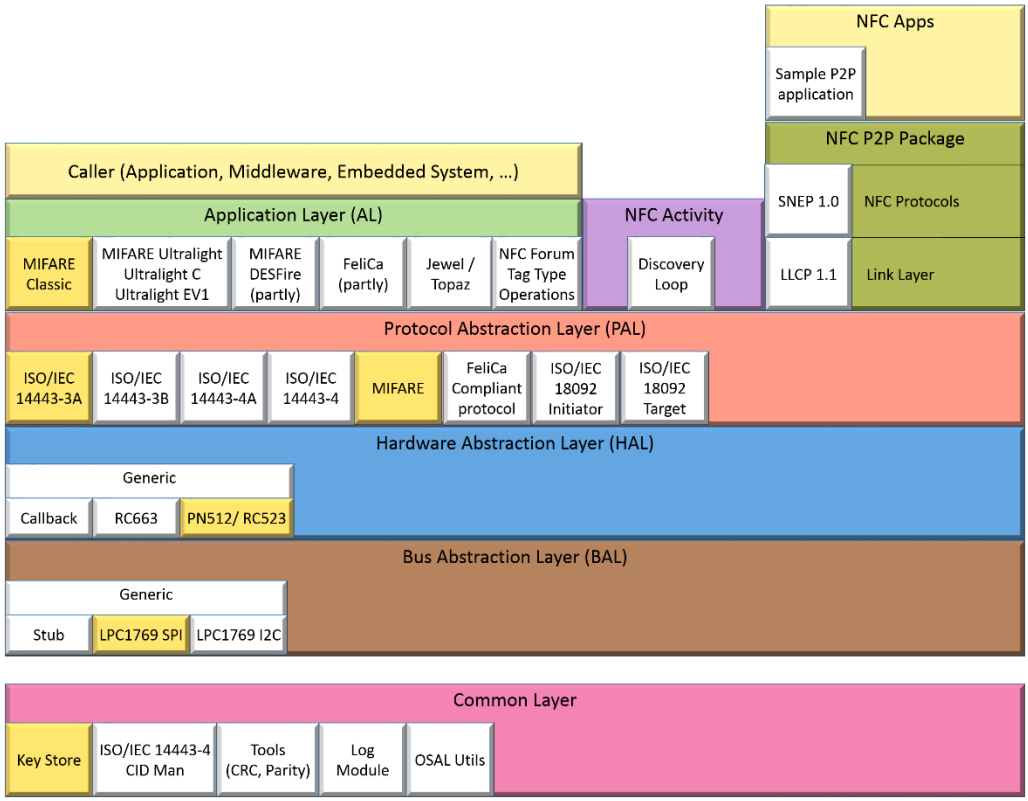

Fig 15. NFC Reader Library – MIFARE Classic application components initialization

Once the required components are identified, the data parameter structures used for all the layers context initialization shall be created:

```
593  phbalReg_Lpc1768Spi_DataParams_t balReader;      // LPC1769 BAL component
594  phhalHw_Rc523_DataParams_t hal;                  /* PN512 HAL componen */
595  phpalI14443p3a_Sw_DataParams_t I14443p3a;        //ISO/IEC 14443-3A PAL LAYER
596  phpalMifare_Sw_DataParams_t palMifare;           //MIFARE PAL LAYER
597  phalMfc_Sw_DataParams_t alMfc;                   //MIFARE Classic AL LAYER
598  phKeyStore_Sw_DataParams_t SwkeyStore;           //Key Store Common LAYER
```

Further details on the initialization of the BAL, HAL and PAL layers can be found on the examples Section 5.1 and Section 6.1. However, a brief summary of the initialization of BAL, HAL and PAL components is shown here for convenience:

```
599  /* Initialize the Reader BAL (Bus Abstraction Layer) component */
600  phbalReg_Lpc1768Spi_Init(&balReader, sizeof(phbalReg_Lpc1768Spi_DataParams_t));
601
602  /* Initialize the Reader HAL (Hardware Abstraction Layer) component */
603  status = phhalHw_Rc523_Init(&hal, sizeof(phhalHw_Rc523_DataParams_t), &balReader,
604     0, bHalBufferTx, sizeof(bHalBufferTx), bHalBufferRx, sizeof(bHalBufferRx));
605
606  /* Set the parameter to use the SPI interface */
607  hal.bBalConnectionType = PHHAL_HW_BAL_CONNECTION_SPI;
608
609  /* Open the communication channel between the MCU and the reader IC
610  status = phbalReg_OpenPort(&balReader);
```

```
611   CHECK_SUCCESS(status);
612
613   /* Initialize the I14443-A PAL layer */
614   status = phpalI14443p3a_Sw_Init(&palI14443p3a,
615      sizeof(phpalI14443p3a_Sw_DataParams_t), &hal);
616   CHECK_SUCCESS(status);
```

Finally, the Application Layer (AL) is the top layer of the software stack, providing specific implementations of various contactless technologies. To initialize the MIFARE Classic component:

```
617   /* Initialize the Mifare Classic AL component
618   phalMfc_Sw_Init(&alMfc,
619         sizeof(phalMfc_Sw_DataParams_t), &palMifare, NULL);
```

7.2 Key Store Initialization

To perform any operation with a MIFARE Classic, the card needs to be activated and authenticated in advance. For instance, before using any command or authentication function, the software Key Store has to be initialized. For that purpose, the following function can be used:

```
620   /* Initialize the keystore component */
621   PH_CHECK_SUCCESS_FCT(status, phKeyStore_Sw_Init(&SwkeyStore,
622      sizeof(phKeyStore_Sw_DataParams_t),
623      &pKeyEntries[0],NUMBER_OF_KEYENTRIES,&pKeyVersionPairs[0],
624      NUMBER_OF_KEYVERSIONPAIRS, &pKUCEntries[0], NUMBER_OF_KUCENTRIES));
```

Moreover, the key storage file system has to be formatted to a MIFARE key type

```
625   /* load a Key to the Store ;-) */
626   status = phKeyStore_FormatKeyEntry(&SwkeyStore, 1, PH_KEYSTORE_KEY_TYPE_MIFARE);
```

Then, we can store a MIFARE Crypto key in the Key Store. In this example, we store the default MIFARE Classic key at delivery (0xFF 0xFF 0xFF 0xFF 0xFF 0xFF).

```
627   /* Mifare Classic card, set Key Store */
628   Status= phKeyStore_SetKey(&SwkeyStore, 1, 0, PH_KEYSTORE_KEY_TYPE_MIFARE,
629   &Key[0], 0);
```

7.3 MIFARE Classic Application Code

Once the NFC Reader Library components are initialized, the next step is to setup the contactless reader to turn on the RF field, configure the protocol settings and activate MIFARE Classic card. Firstly, we reset the PN512 configuration:

```
630   /* SoftReset the IC*/
631   phhalHw_Rc523_Cmd_SoftReset(&halReader);
```

Then the PN512 field generation is also reset:

```
632   /* Reset the Rf field */
633   phhalHw_FieldReset(&halReader);
```

The PN512 is reconfigured with the required register settings to work on ISO/IEC 14443 Type A card detection.

```
634  /* Apply the type A protocol settings and activate the RF field. */
635  phhalHw_ApplyProtocolSettings(&halReader,
636         PHHAL_HW_CARDTYPE_ISO14443A);
```

Then, the `phpalI14443p3a_ActivateCard` function shall be called. The `phpalI14443p3a_ActivateCard` function senses the field to detect the presence of MIFARE Classic cards in the RF field and it proceeds to activate it (ReqA or WupA and Anticollision/Select).

```
637  /* Activate the communication layer part 3 of the ISO 14443A standard. */
638  phpalI14443p3a_ActivateCard(&I14443p3a, NULL, 0x00, bUid, &bLength,
639              bSak, &bMoreCardsAvailable);
```

The type of card detected on the field can be determined by examining the value of the `bSak` variable. For the MIFARE Classic, the SAK value is: 0x08.

```
640  /* Check if we have a card in the RF field. If so, check what card it is. */
641  if (PH_ERR_SUCCESS == status)
642  {
643        /* Check if there is a Mifare Classic card in the RF field */
644        if (0x08 == (*bSak & 0x08))
645        {
646          debug_printf_msg("Mifare Classic card detected");
647
648          //** YOUR MIFARE CLASSIC APPLICATION CODE HERE **//
649        }
```

At this point, the MIFARE Classic has been successfully detected and activated, and the application code can be defined.

The first task is to get the card UID. The card UID is stored in the Block 0 of the Sector 0 of the IC. This manufacturer block is programmed and write-protected during the production test. Authentication against Sector 0 before reading the UID value is required. The authentication against the MIFARE Classic can be done in the following way:

```
650  /* Mifare Classic card, send authentication for sector 0 */
651  status = phalMfc_Authenticate(&alMfc, 0, PHHAL_HW_MFC_KEYA, 1, 0, bUid, bLength);
652  if(status)
653  {
654        debug_printf_msg("\n!!! Authentication was not successful.\n"
655        debug_printf_msg("\n/****** Abort of execution ******/");
656        return 0;
657  }
658  debug_printf_msg("\n**** Authentication successful");
```

After the authentication against the Sector 0 is successfully done, the block 0 can be read in order to retrieve the UID. The UID can be either four bytes or seven bytes length. To read any MIFARE Classic data block, the following function can be used:

```
659  /* Check the UID of the Classic card in the field */
660  phalMfc_Read(&alMfc, 0,&bBufferReader[0]));
```

The second part of this exercise is about writing some data on Block 4 of the memory structure. Before writing, we need to authenticate with the corresponding sector. Block 4

belongs to Sector 1. As we have just discussed, an authentication request to Sector 1 is sent:

```
661  /* Mifare Classic card, send authentication for sector 1 */
662  phalMfc_Authenticate(&alMfc, 4, PHHAL_HW_MFC_KEYA, 1, 0, bUid, bLength);
```

Afterwards, the data to be written into the block needs to be generated. For that purpose, a function generating 16 bytes string is defined:

```
663  /* Fill block with data */
664  Fill_Block(bBufferReader, 15);
```

Finally, the MIFARE Write command requires a block address to store the 16 bytes. To write a Block data into a MIFARE Classic, the following `phalMfc_Write` function can be used:

```
665  /* Write data @ block 4 */
666  phalMfc_Write(&alMfc, 4, bBufferReader);
667  debug_printf_msg("\nWrite successful 16 bytes");
```

8. Example: MIFARE Ultralight

The MIFARE Ultralight example explains the initialization of the NFC Reader Library layers (from bottom to top) and the establishment of the communication with a MIFARE Ultralight card. The example writes data in a MIFARE Ultralight page and protects the memory to avoid any further writing operations by making use the lock bytes.

8.1 NFC Reader Library Initialization

Similar to previous examples, the first step is to initialize the NFC Reader Library in accordance with the hardware and the application to be developed. Again, the hardware used for this example is a LPC1769 MCU (BAL) and a PN512 reader IC (HAL). For any MIFARE Ultralight application, the ISO/IEC 14443-3A and MIFARE PAL components are required. Finally, the MIFARE Ultralight component in the AL layer shall be initialized in order to use the MIFARE Ultralight command set. The Fig 16 highlights in yellow the different components that are going to be initialized in this example.

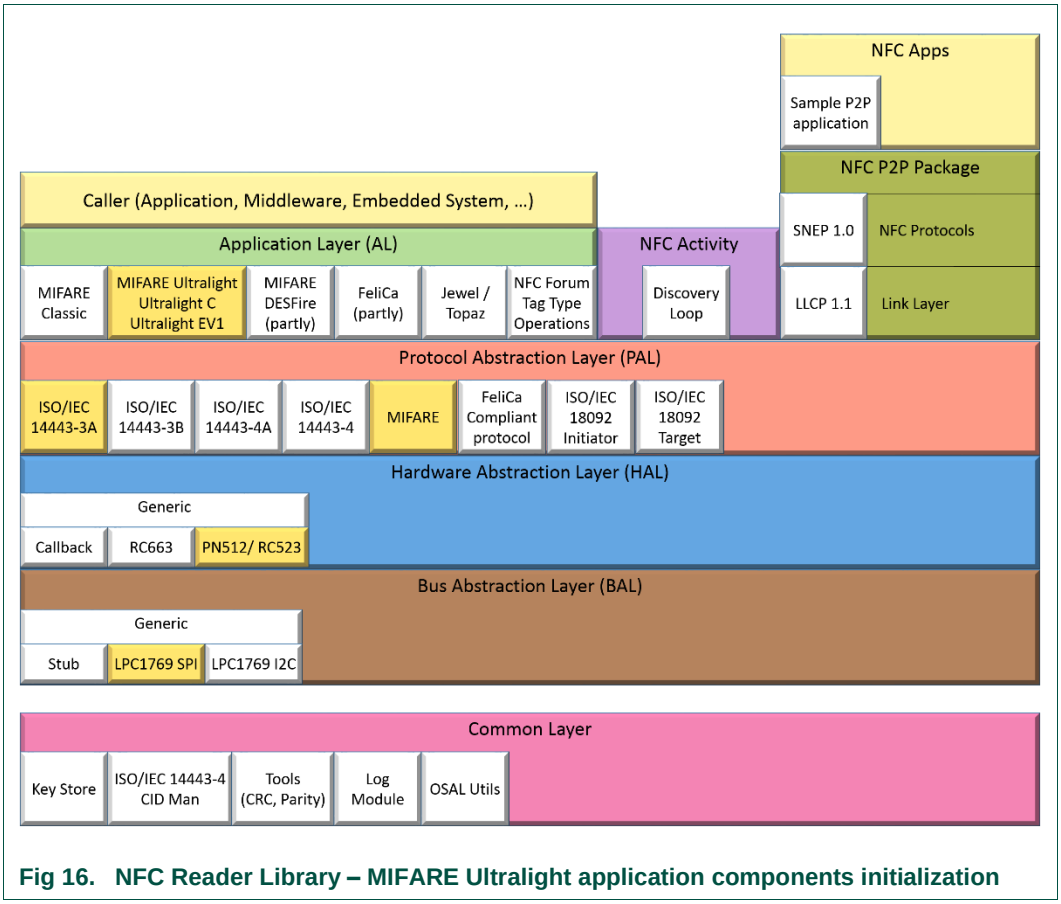

Fig 16. NFC Reader Library – MIFARE Ultralight application components initialization

Once the required components are identified the data parameter structures used for all the layers context initialization are created:

```
668   phbalReg_Lpc1768Spi_DataParams_t balReader;      // LPC1769 BAL component
669   phhalHw_Rc523_DataParams_t hal;                  /* PN512 HAL componen */
670   phpalI14443p3a_Sw_DataParams_t I14443p3a;        //ISO/IEC 14443-3A PAL LAYER
671   phpalMifare_Sw_DataParams_t palMifare;           //MIFARE PAL LAYER
672   phalMfc_Sw_DataParams_t alMful;                  //MIFARE Ultralight AL LAYER
```

Further details on the initialization of the BAL, HAL and PAL layers can be found on the examples Section 5.1 and Section 6.1. However, a brief summary of the initialization of BAL, HAL and PAL components is shown here for convenience:

```
673   /* Initialize the Reader BAL (Bus Abstraction Layer) component */
674   phbalReg_Lpc1768Spi_Init(&balReader, sizeof(phbalReg_Lpc1768Spi_DataParams_t));
675
676   /* Initialize the Reader HAL (Hardware Abstraction Layer) component */
677   status = phhalHw_Rc523_Init(&hal, sizeof(phhalHw_Rc523_DataParams_t), &balReader,
678      0, bHalBufferTx, sizeof(bHalBufferTx), bHalBufferRx, sizeof(bHalBufferRx));
679
680   /* Set the parameter to use the SPI interface */
681   hal.bBalConnectionType = PHHAL_HW_BAL_CONNECTION_SPI;
682
683   /* Open the communication channel between the MCU and the reader IC
684   status = phbalReg_OpenPort(&balReader);
685   CHECK_SUCCESS(status);
```

```
686  /* Initialize the I14443-A PAL layer */
687  status = phpalI14443p3a_Sw_Init(&palI14443p3a,
688     sizeof(phpalI14443p3a_Sw_DataParams_t), &hal);
689  CHECK_SUCCESS(status);
```

Finally, the application layer is the top layer of the software stack, providing specific implementations of various contactless technologies. To initialize the MIFARE Ultralight component:

```
1    /* Initialize Ultralight(-C) AL component */
2    PH_CHECK_SUCCESS_FCT(status, phalMful_Sw_Init(&alMful,
3         sizeof(phalMful_Sw_DataParams_t), &palMifare, NULL,
4         NULL, NULL));
```

8.2 MIFARE Ultralight Application Code

Firstly, we write some user data on page 4. This is done using the `phalMful_Write()` function.

```
5    uint8_t data_to_write[4] = {0x12U, 0x34U, 0x56U, 0x78U};
6    // we write page 4 of the ultralight
7    Status=phalMful_Write(&alMful, 0x04, data_to_write);
8    printf(" Written in page 4 0x12345678");
```

The byte 02h and 03h of page 02h represent the field programmable read-only locking mechanism. Each page from 03h (OTP) to 0Eh can be individually locked by setting the corresponding locking bit to logic 1 to prevent further write access. After locking, the page becomes read-only memory. The content of bytes 0 and 1 of page 2 (BCC1 and Internal bytes) are unaffected by the corresponding data bytes of the WRITE command. For instance, to block all the MIFARE Ultralight memory, the following blocking command shall be sent:

```
9    uint8_t blockingCommand[4]={0,0,0xFF,0xFF};
10   status= phalMful_Write(&alMful, 2, blockingCommand);
11   printf(" Blocked page 4");
```

Finally, to activate the new locking configuration, a REQA or WUPA command must be carried out.

```
12   /* Reset the RF field */
13   status= phhalHw_FieldReset(pHal);
14
15   /* Apply the type A protocol settings and activate the RF field. */
16   status=phhalHw_ApplyProtocolSettings(pHal, PHHAL_HW_CARDTYPE_ISO14443A);
17
18   /* Activate the communication layer part 3 of the ISO 14443A standard. */
19   status = phpalI14443p3a_ActivateCard(&I14443p3a, NULL, 0x00, bUid, &bLength,bSak,
     &bMoreCardsAvailable);
```

After sending the blocking command and carry out a REQA or WUPA command, any further WRITE operation on the MIFARE Ultralight card will be refused and will return an error.

9. NFC Reader Library Memory Management

9.1 MCU Memory Size

Desktop computers and mobile platforms provide a large memory capacity and the code size is not usually critical. However, on embedded programing development, the memory is a limited resource with a well-defined storage size that shall be taken into account.

Each MCU has an amount of programmable memory space. Please refer to its datasheet to know the amount of programmable memory space available in a certain MCU, or consult it directly in the LPCXpresso IDE by clicking on the project properties and check the memory details at "*C/C++ Build / MCU Settings".* For instance, Fig 17 shows the Flash memory size for the commercial LPCXpresso LPC1227/301 MCU (128Kbytes).

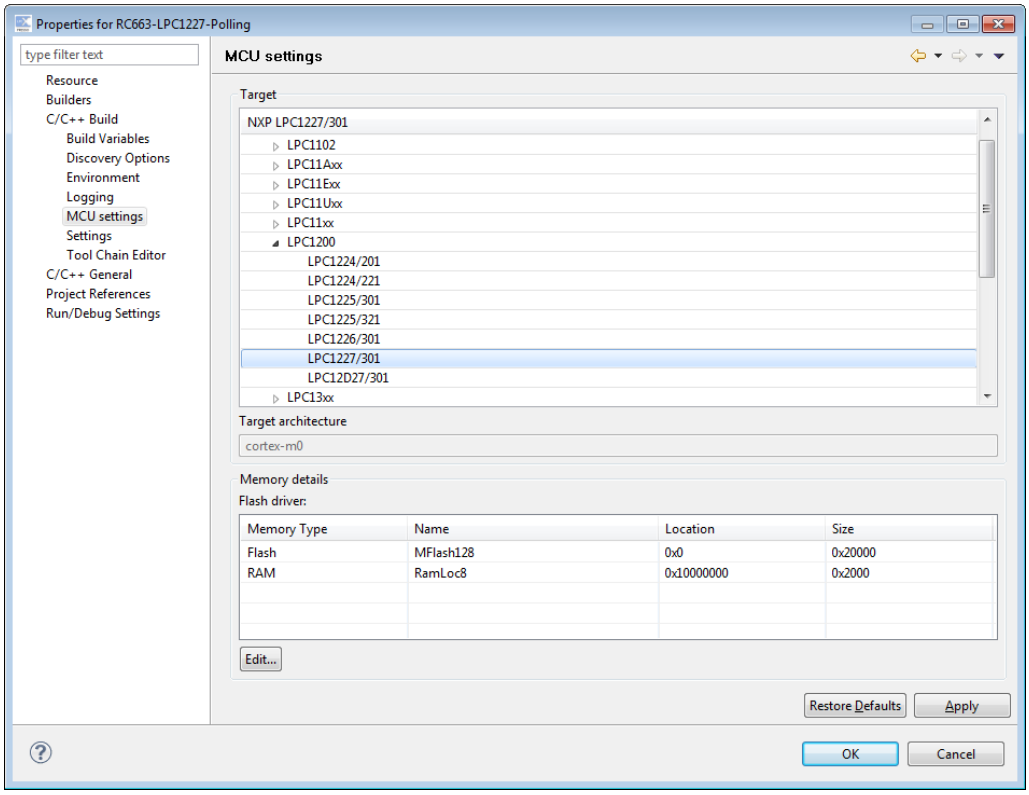

Fig 17. MCU Memory size

9.2 Project Memory Consumption

The developers shall ensure the project fits in the available MCU programmable memory. Otherwise, the project cannot be executed and a buffer overflow error will be shown by the LPCXpresso during project execution. The project size is displayed after the code has been compiled and built.

Fig 18. Project memory size

The total memory size of a project is the sum of the following three segments:

- Text: It holds space for everything that ends up in the Flash memory such as coding functions.
- Data: It holds space for the initialized data such as static variables.
- Bss: It holds space for the uninitialized data that is represented by zero-valued bits in memory.

9.3 NFC Reader Library Memory consumption

Importing the NFC Reader Library into a LPCXpresso IDE project does not occupy memory space by itself. Only those modules that have been loaded in the application will occupy space when the project is compiled and built. For instance, the P2P application of Section 5, where several Application Layer (AL) modules are loaded, will consume a larger memory than the MIFARE Ultralight example of Section 8, where only MIFARE related modules are loaded. However, note that the NFC Reader Library imported is the same.

9.3.1 Memory Footprint of NFC Reader Library Components

This section provides an approximate memory footprint of all the components that are part of the NFC Reader Library. These values should be taken as a reference as they might vary depending on the NFC Reader Library version.

Note: All the values indicated in this document have been measured in the release build configuration (see section 12.7 for further details regarding debug and release modes).

Table 3. Memory footprint of NFC Reader Library components

Layer	Component	Code (byte)	RAM (byte)
Application L	Type 1 Tag – Jewel / Topaz	4796	12
Tag Operation L	Type 2 Tag – MIFARE UL (EV1)	1132	18
	Type 3 Tag – FeliCa	2012	36
	Type 4 Tag – MIFARE DESFire	1274	20
	Tap Operation	21824	228
	MIFARE Classic	1328	10
	MIFARE DESFire	17124	96

Layer	Component	Code (byte)	RAM (byte)
NFC Activity	Discovery Loop	4390	516
NFC P2P Package	SNEP	4943	3196
	LLCP	13864	
Protocol AL	ISO/IEC14443-3A & MIFARE	3250	28
	ISO/IEC14443-3B	2316	35
	ISO/IEC14443-4A & ISO/IEC14443-4	4114	31
	FeliCa	1172	24
	ISO/IEC18092 Initiator	4496	27
	ISO/IEC18092 Target	4479	68
Hardware AL	Hal common	846	-
	Callback	390	60
	RC663	15268	96
	PN512/RC523	11524	88
Bus AL	BAL common	358	-
	SPI for LPC1769	414	4
	I2C for LPC1769	246	4
Common	Key Store Common	719	-
	Key store RC663	492	8
	Key Store SW	914	24
	ISO/IEC 14443-4 CID Man	170	16
	Tools (CRC, Parity)	1575	-
	Log Module	379	-
	OSAL Utils	1696	52

A LPCXpresso project includes the CMSIS source code, the MCU drivers on which the project will be executed, and the application logic (see Section 11). Therefore, the project code size is always larger than the sum of the NFC Reader Library components footprint.

For instance, the project from scratch that is created in Section 11, which only contains the drivers for the communication with the LPC17xx MCU, leads to a 12kbyte size binary file.

```
make --no-print-directory post-build
Performing post-build steps
arm-none-eabi-size "lpc17xx_scratch.axf"; # arm-none-eabi-objcopy -O binary "lpc17xx_scratc
   text    data     bss     dec     hex filename
  11704       0     336   12040    2f08 lpc17xx_scratch.axf
```

Fig 19. Memory footprint of a project from scratch

Considering the project size depicted in Fig 19 and the memory footprint of NFC Reader Library components shown in Table 3, customers are able to foresee the memory size of their projects.

9.3.1.1 Memory footprint of a sample MIFARE Ultralight Read/Write project

The approximate memory consumption of a MIFARE Ultralight application is calculated in this section. In order to optimize the memory consumption, all those modules that are not necessary have been excluded from the build operation. Fig 20 depicts the components required to communicate with a MIFARE Ultralight card.

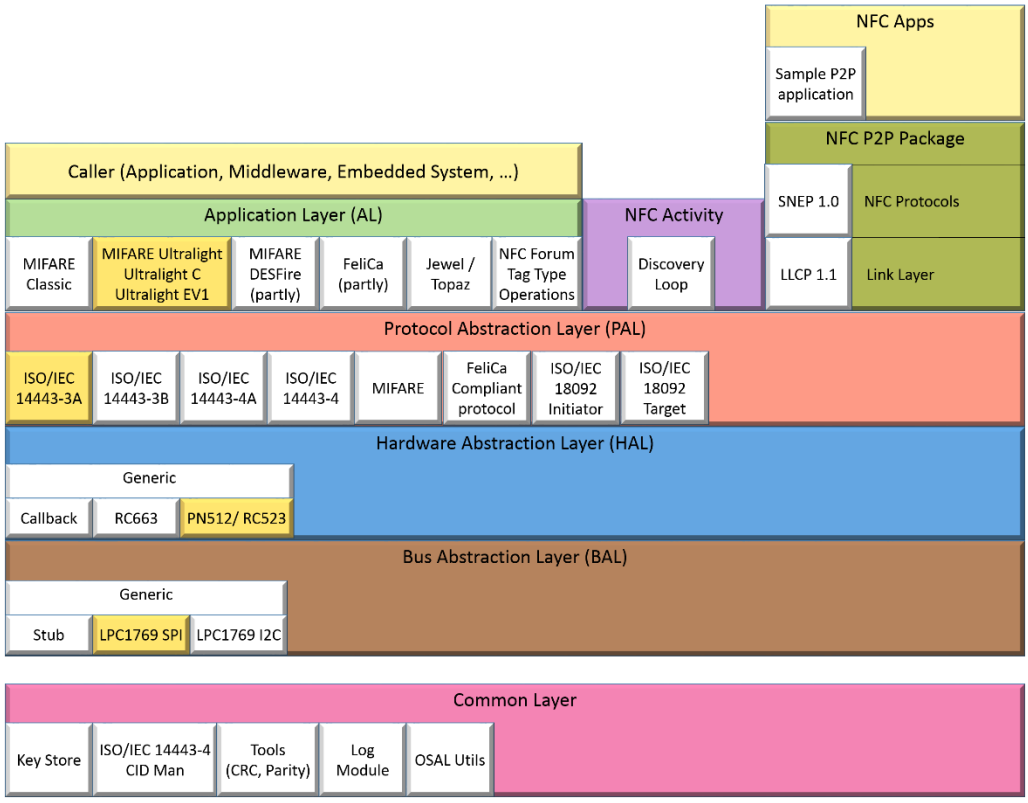

Fig 20. MIFARE Ultralight Read/Write project layer architecture

According to values defined in Fig 19 and Table 3, the expected minimum project size for a MIFARE Ultralight project is calculated in Table 4.

Table 4. Memory footprint of NFC Reader Library components

Layer	Component	Code (byte)
Application L	Type 2 Tag – MIFARE UL (EV1)	1132
Protocol AL	ISO/IEC14443-3A & MIFARE	3250
Hardware AL	PN512/RC523	11524
Platform AL	LPC1769SPI	414
Project from scratch memory footprint		12040
Expected memory footprint		**28360**

The expected memory footprint value calculated in Table 4 is referred to the NFC Library components that are enabled during the build procedure. The value memory footprint of

the project includes also the application logic which is part of the "*main.c*" file of the LPCXpresso project.

Fig 21 depicts the total memory size of a sample MIFARE Ultralight project developed in LPCXpresso IDE occupies. Please note that the memory consumed is similar to the one calculated in Table 4.

```
make --no-print-directory post-build
Performing post-build steps
arm-none-eabi-size "Pn512_Lpc17xx_MUL.axf"; # arm-none-eabi-objcopy -O binary "Pn512_Lpc17xx_MUL.ax
   text    data     bss     dec     hex filename
  27440      24     968   28432    6f10 Pn512_Lpc17xx_MUL.axf
```

Fig 21. Sample MIFARE Ultralight Read/Write project binary file

9.3.1.2 Memory footprint of a sample P2P Initiator project

The sample project addressed in this section implements a P2P Initiator that transmits a NDEF message to the remote peer device detected. In order to optimize the memory consumption, all those modules that are not necessary have been excluded from the build operation. Fig 22 depicts the components required to implement a P2P application in Initiator mode.

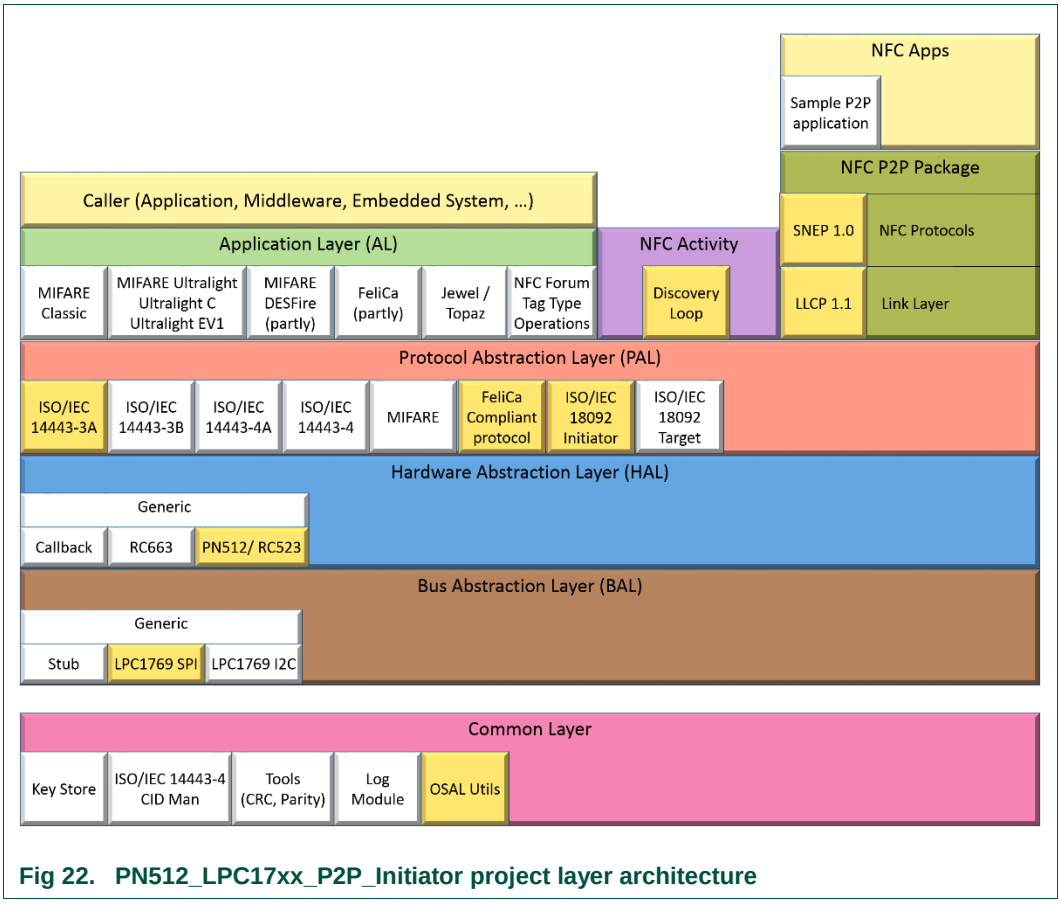

Fig 22. PN512_LPC17xx_P2P_Initiator project layer architecture

According to the values defined in Fig 22 and Table 4, the expected minimum project size for this P2P Initiator is calculated in Table 5.

Table 5. Memory footprint of NFC Reader Library components

Layer	Component	Code (byte)
NFC Activity	Discovery Loop	4390
NFC P2P Package	SNEP	4943
	LLCP	13864
Protocol AL	ISO/IEC14443-3A & MIFARE	3250
	FeliCa	1172
	ISO/IEC18092 Initiator	4496
Hardware AL	PN512/RC523	11524
Bus AL	LPC1769SPI	414
Common	OSAL Utils	1696
Project from scratch memory footprint		12040
Expected memory footprint		**57789**

As it is explained in section 9.3.1.1, this expected memory footprint value corresponds to the memory footprint occupied by the NFC Reader Library components. The application logic implemented in "*main.c*" file for this particular example is much larger as there are more components to initialize and manage.

Fig 23 depicts the total memory size of a sample P2P Initiator project developed in LPCXpresso IDE. Please note that the memory consumed is similar to the one calculated in Table 5.

```
make --no-print-directory post-build
Performing post-build steps
arm-none-eabi-size "Pn512_Lpc17xx_P2P_Active_Initiator.axf"; # arm-none-eabi-objcopy -O binary "Pn512_Lpc
LPC1769 -d "Pn512_Lpc17xx_P2P_Active_Initiator.bin";
   text    data     bss     dec     hex filename
  51312      20    5980   57312    dfe0 Pn512_Lpc17xx_P2P_Active_Initiator.axf
```

Fig 23. PN512_LPC17xx_P2P_Initiator project binary file.

9.3.2 Scaling Down Memory Consumption

In order to optimize the memory consumption of a project, developers shall only load the modules that are strictly mandatory in order to meet their application requirements. The NFC Reader Library provides an easy way to load or unload modules. The complete list of modules that can be included or excluded in the project compilation are defined in the "*NxpRdLib_PublicRelease/types/ph_NxpBuild.h*" file.

`#define NXPBUILD__PH_LOG` – Includes the Log Component during project build.

`// #define NXPBUILD__PH_LOG` – Excludes the Log Component during project build.

For a comprehensive view of all the technical possibilities that should be considered by developers, please see the AN1132 – How to Scale Down the NFC Reader Library [36].

10. Porting

10.1 NFC Reader Library

The NFC Reader Library has been implemented using a modular and a multi-layered approach, where all its layers except the Bus Abstraction Layer are platform independent. Therefore, upper layers of the software stack can be used independently on the specific underlying communication interface with the host MCU.

The Bus Abstraction Layer implements the communication interface between the host MCU and the contactless reader IC. The MCU sends reader IC specific commands and the reader IC responds to the MCU with data received from contactless cards or related information stored in requested registers.

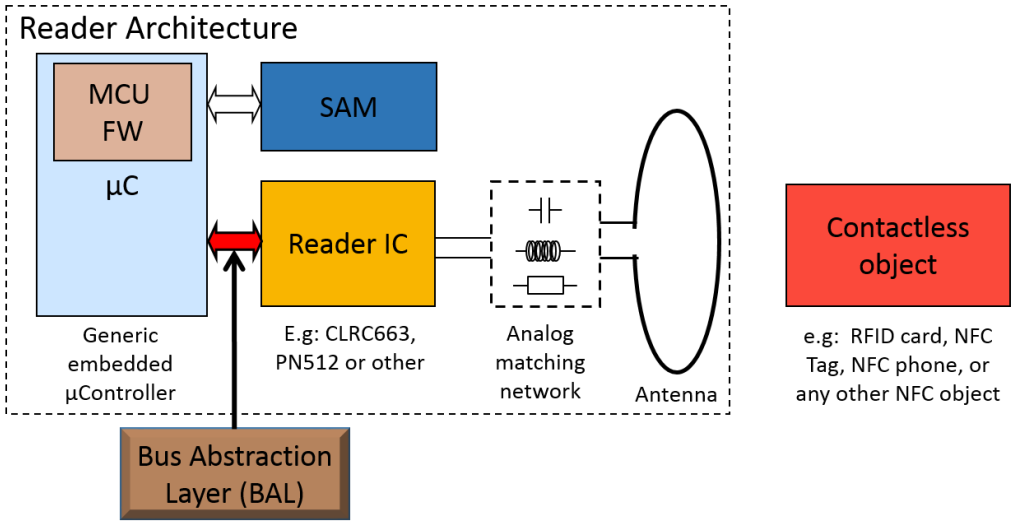

Fig 24. Typical Reader Architecture

The current NFC Reader Library implements support for LPC1769 MCU with both SPI and I2C communication interfaces.

The Bus Abstraction Layer components are bound to specific MCU functions. As a result, to migrate to another MCU, a specific BAL component which supports this particular MCU has to be developed.

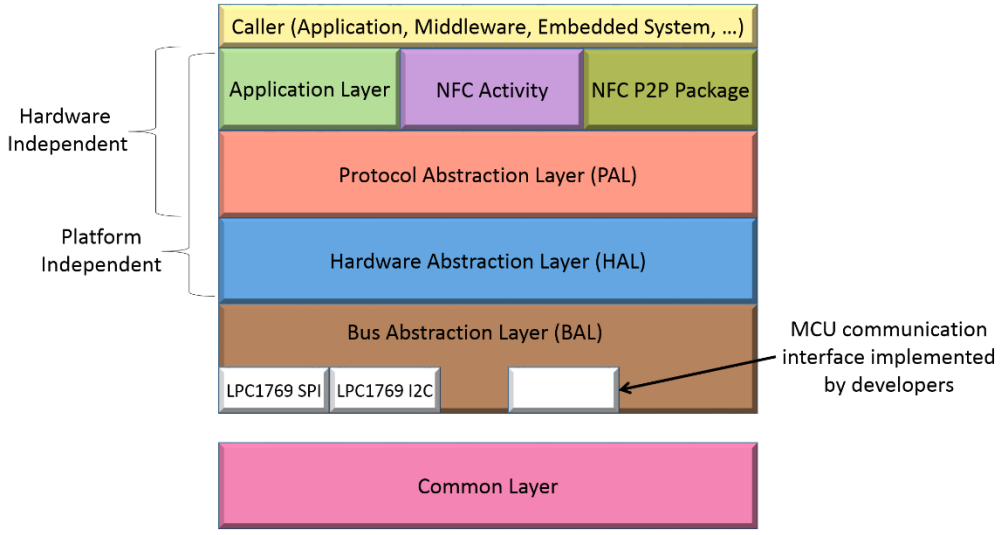

Fig 25. NFC Reader Library - BAL component extension

10.2 ARM Architecture based MCU Drivers

In addition to the modification of the Bus Abstraction Layer, MCU related files that are part of the LPCXpresso project should be replaced. These files include the '*CMSIS'* folder and hardware specific source files that are part of the '*include'* folder.

CMSIS stands for *Cortex Microcontroller Software Interface Standard* and is a specification defined by ARM for Cortex-M based systems which provide a vendor-independent hardware abstraction layer for the Cortex-M processor series. The CMSIS enables consistent and simple software interfaces to the processor and peripherals registers. A summary of the source code files you can find within the CMSIS library is:

- ***Core_cmX.h***: Provides access to Cortex-M core's built-in peripherals
- ***Core_cmFunc.h***: Provides inline helper functions for accessing register and peripherals within the Cortex-M core.
- ***Core_cmInst.h***: Provides inline helper functions for accessing instructions not directly generated by compiler
- ***System_<mcu>.h***: Contain the prototype/implementations for the *SystemInit()* and *SystemCoreClockUpdate()*.

Within the LPCXpresso IDE examples subdirectory (by default installed in *C:\nxp\LPCXpresso_6.0.2_151\lpcxpresso\Examples\CMSIS_CORE*), several CMSIS library projects can be found. Each of these CMSIS library projects contain the appropriate CMSIS header files and source code for that specific MCU family.

10.3 Non-ARM Architecture based MCU Drivers

If a customer aims to port the NFC Reader Library to another MCU based on a non-ARM architecture, the MCU specific drivers should be included as part of the LPCXpresso project.

This solution is out of the scope of this document.

11. How to create a new Project from Scratch

This tutorial guides developers on how to create a new project from scratch using LPCXpresso v6 IDE and how to import the NFC Reader Library into the workspace. After completing this process, the developers will have a clean project for start developing NFC applications.

The hardware used to prepare this tutorial is a PN512 Blueboard connected to a LPC1769 target board. In case another MCU is used, the steps for the creation of the project remain equal but specific MCU drivers shall be used.

The developer shall follow three main steps in order to create a new project from scratch with the NFC Reader Library:

5. Import the CMSIS Library.
6. Create a new project.
7. Import the NFC Reader Library and MCU drivers into the project.

11.1 Importing the CMSIS Library

CMSIS (*Cortex Microcontroller Software Interface Standard)* is a specification defined by ARM for Cortex-M based systems which provides a vendor-independent hardware abstraction layer for the Cortex-M processor series. CMSIS enables consistent and simple software interfaces to the processor and peripherals registers.

Within the LPCXpresso IDE examples subdirectory (by default installed in *C:\nxp\LPCXpresso_6.0.2_151\lpcxpresso\Examples\CMSIS_CORE*), the developers can find several CMSIS library projects. Each of these CMSIS library projects contain the appropriate CMSIS source code files for that specific MCU family.

In order to import the CMSIS library, open LPCXpresso IDE, click on “*Import Project*” and then click “*Browse*” into the project archive (zip) label.

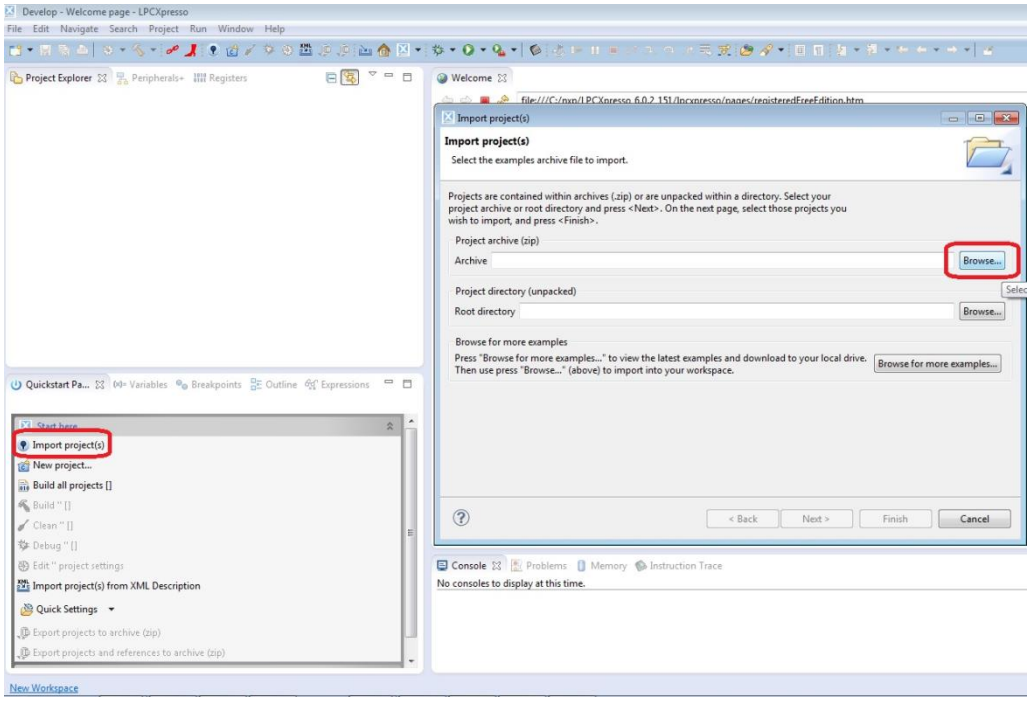

Fig 26. Importing CMSIS Library into workspace (1)

Browse the "*Examples*" folder in your LPCXpresso IDE installation directory and select "*LPC17xx_LatestCMSISLibraries.zip"* file and press "*Next*".

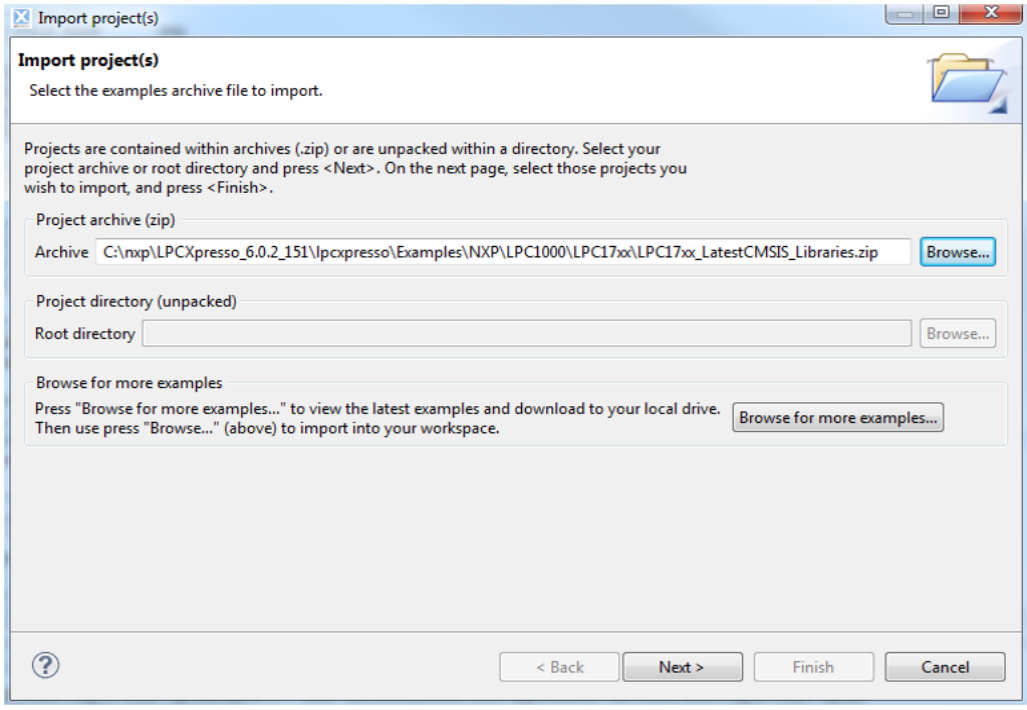

Fig 27. Importing CMSIS Library into workspace (2)

Pick the CMSIS project according to the MCU used in the project. In this example, the "*CMSIS_CORE_LPC17xx"* is picked. Finally, click "*Finish*".

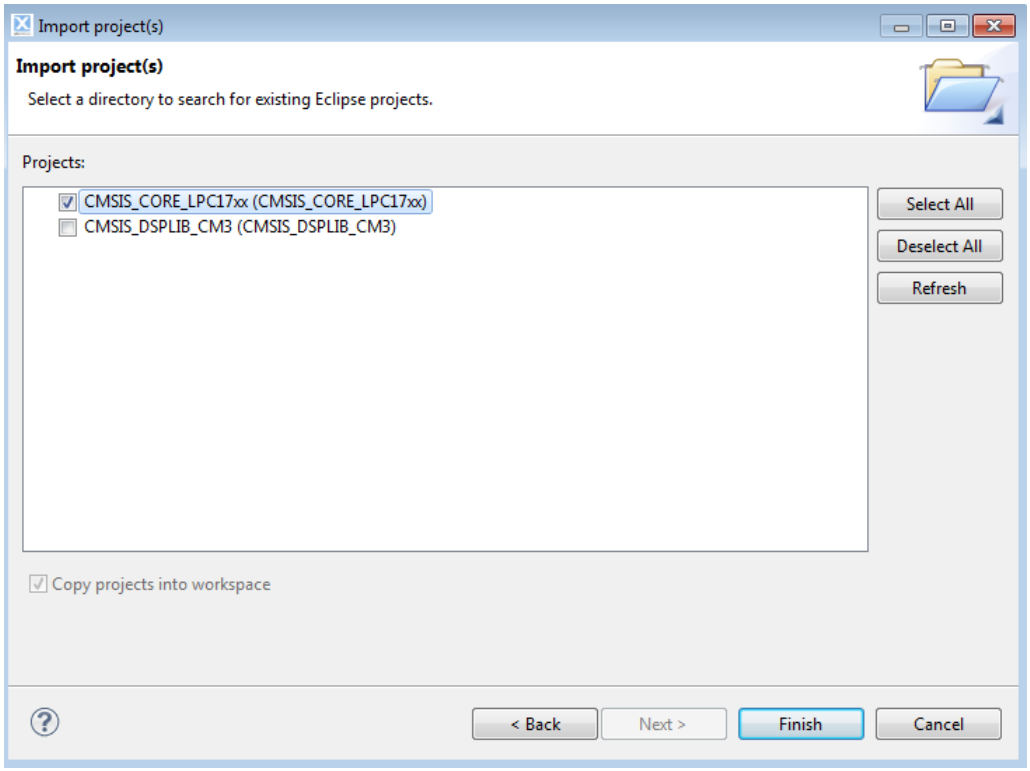

Fig 28. Importing CMSIS Library into workspace (3)

This operation imports the CMSIS source code into the LPCXPresso workspace. The most important source code files that can be found within the *CMSIS_CORE_LPC17xx* are:

- ***Core_cmX.h***: Provides access to Cortex-M core's built-in peripherals
- ***Core_cmFunc.h***: Provides inline helper functions for accessing register and peripherals within the Cortex-M core.
- ***Core_cmInst.h***: Provides inline helper functions for accessing instructions not directly generated by compiler
- ***System_<mcu>.h***: Contain the prototype/implementations for the *SystemInit()* and *SystemCoreClockUpdate()*.

Once the CMSIS library is successfully imported, the LPCXpresso project explorer should look this way:

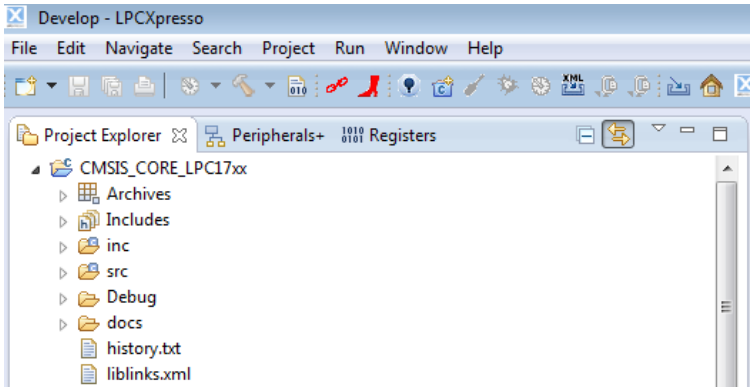

Fig 29. Importing CMSIS Library into workspace (4)

11.2 Creating a new Project

After the CMSIS libraries are imported, a new project can be created. The new project shall add the references to the CMSIS libraries that have just been imported. Click "*New Project*" on the Quick Start Panel and select the LPC 175X_6x MCU family, then choose "*C Project (Semihosted)".* The explanation of what is semihosting can be found on Section 12.8.

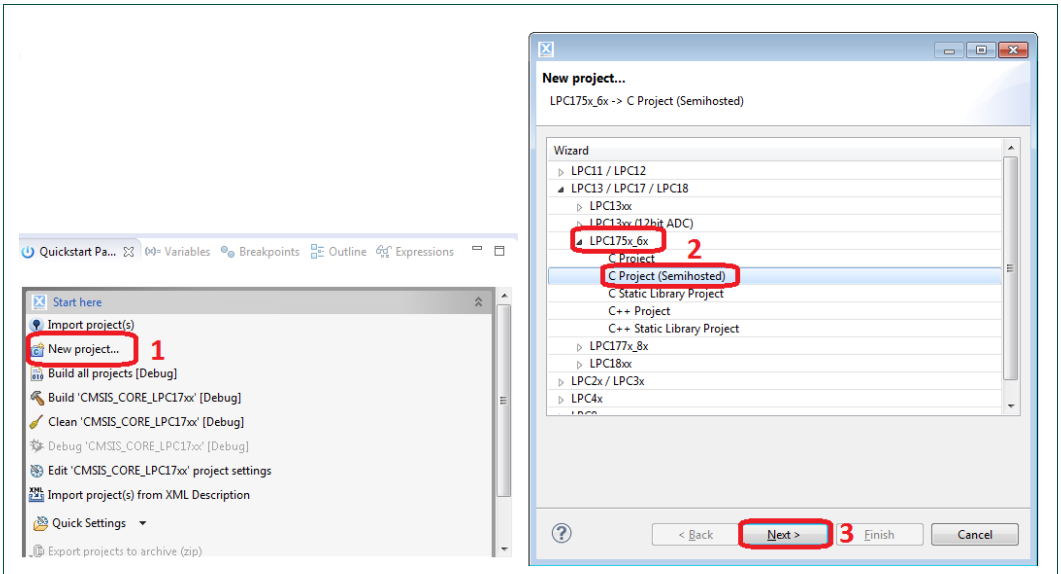

Fig 30. Creating a new Project (1)

In the next window, choose a name for the new project and click "*Next".*

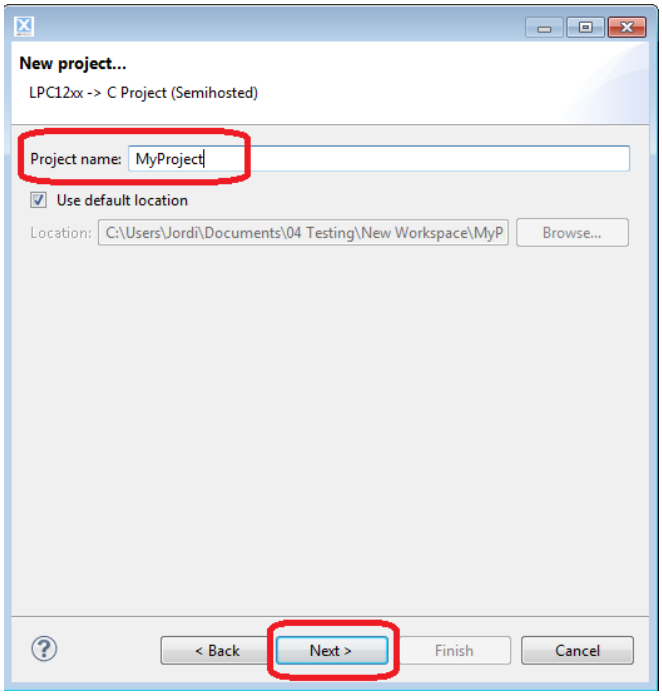

Fig 31. Creating a new Project (2)

Then, the setup assistant guides the developer through a sequence of configurations until the project is created. In most cases, the default configurations can be accepted.

The first assistant window requests the developer to select the target MCU to be used. In this guide, the LPC1769 MCU is selected, then click "*Next*".

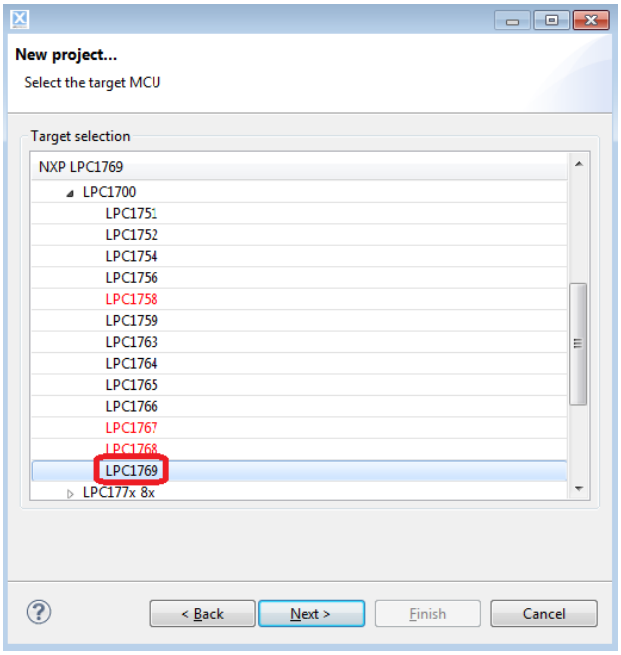

Fig 32. Creating a new Project (3)

The CMSIS library option within the LPCXpresso IDE project wizard provides the possibiliy to link the CMSIS library to the new project under creation. Note that the appropriate CMSIS library project must have been imported into the workspace before starting the "*New Project*" wizard, otherwise an error will be given when the wizard attempts to create the project.

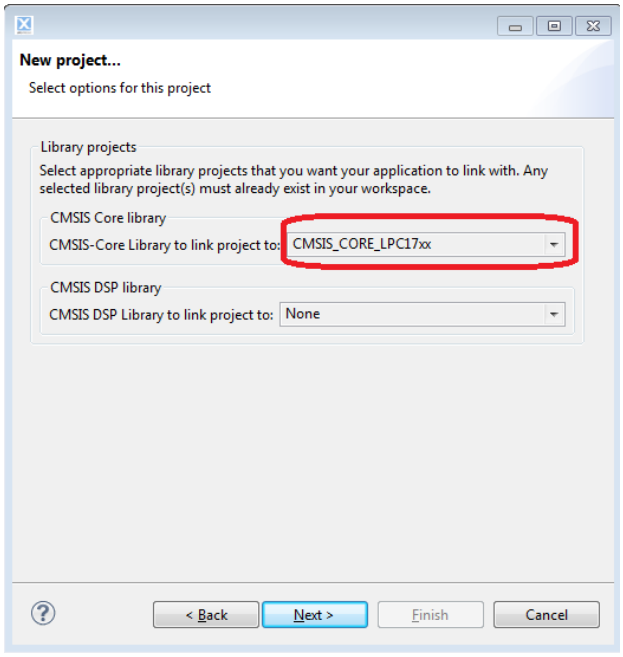

Fig 33. Creating a new Project (4)

Then, the assistant asks the developer to enable or disable "*Code Read Protect*" functionality. NXP Cortex MCUs provide a "*Code Read Protect*" mechanism to prevent certain types of access to internal flash memory by external tools. This option can be left by default. Then, click "*Next*" to continue.

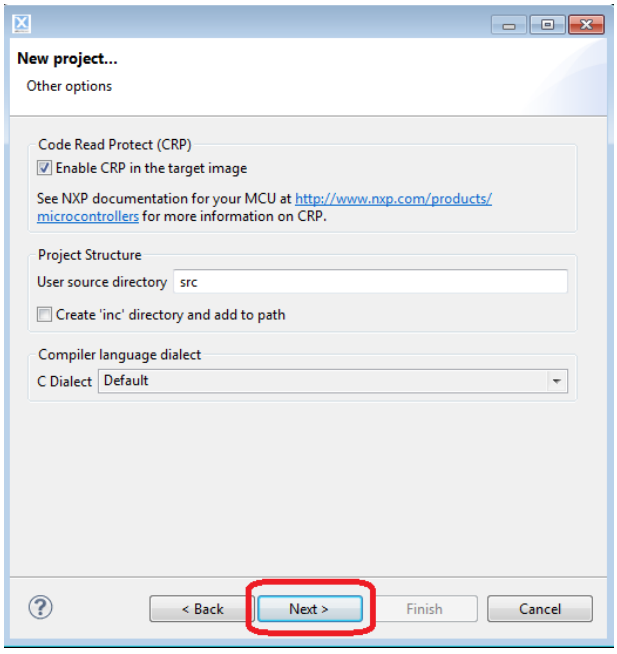

Fig 34. Creating a new Project (5)

The last setup assistant window configures the "*printf*" options of the project. The "*semihosting C project*" wizard provides two options to configure the implementation of "*printf*" that will get pulled in from the "*Redlib C*" library:

- ***Use non-floating-point of printf***: If the application does not pass floating point numbers to *printf*(), the developer can select a non-floating-point variant of *printf*. This will help to reduce the code size of your application.
- ***Use character-rather than string-based printf***: The default *printf*() and *puts*() functions make use of *malloc*() function to provide a temporary buffer on the heap in order to generate the string to be displayed. Enable this option to switch to "character-by-character" versions of these functions (which do not require additional heap space).

These options can be left by default.

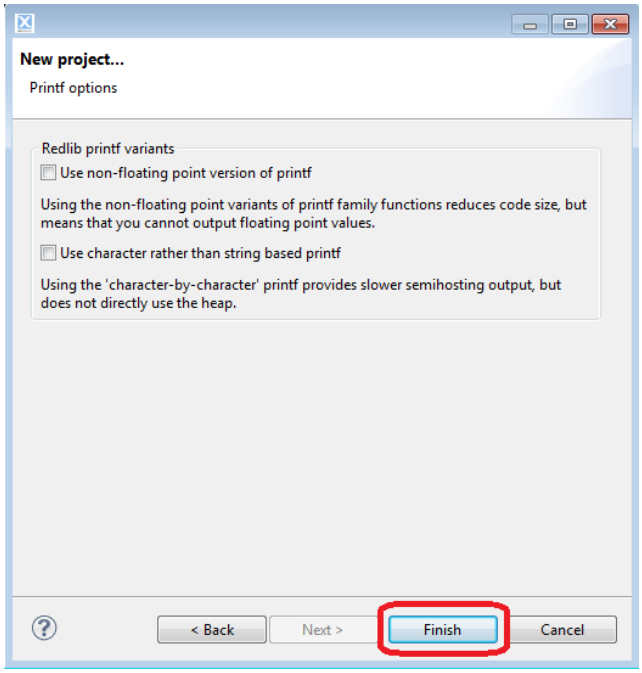

Fig 35. Creating a new Project (6)

After this setup, the "*MyProject*" project has been created. In order to ease the development, LPCXpresso automatically generates an example application code in the project main file, "main.c".

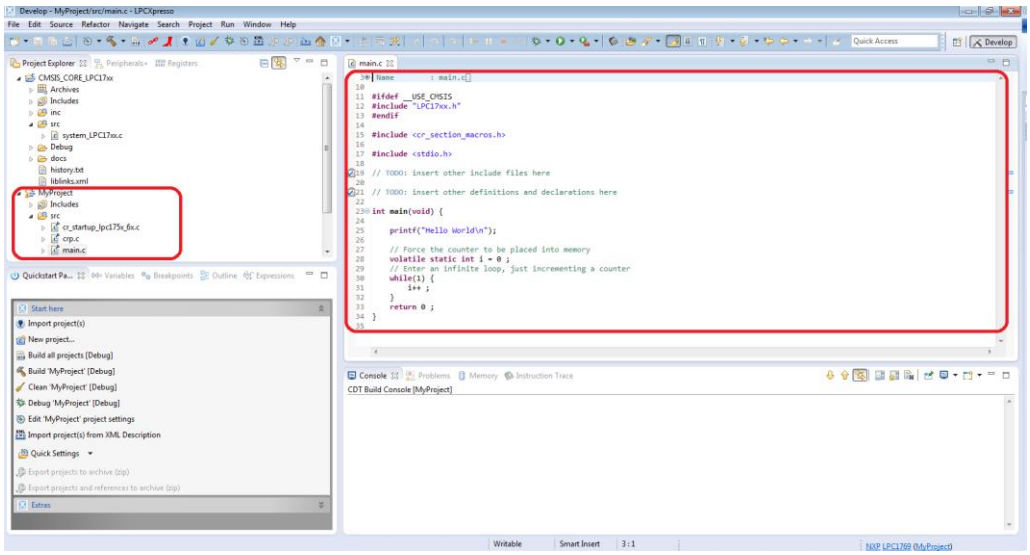

Fig 36. Creating a new Project (7)

The developer can check if the project has been successfully created by compiling and executing the project. To do so, connect the PN512 Blueboard, click first on "*Build"* and later on "*Debug"* on the QuickStart panel. If the operation is performed successfully, the developer will see the "*Hello World message*" in the Console Window.

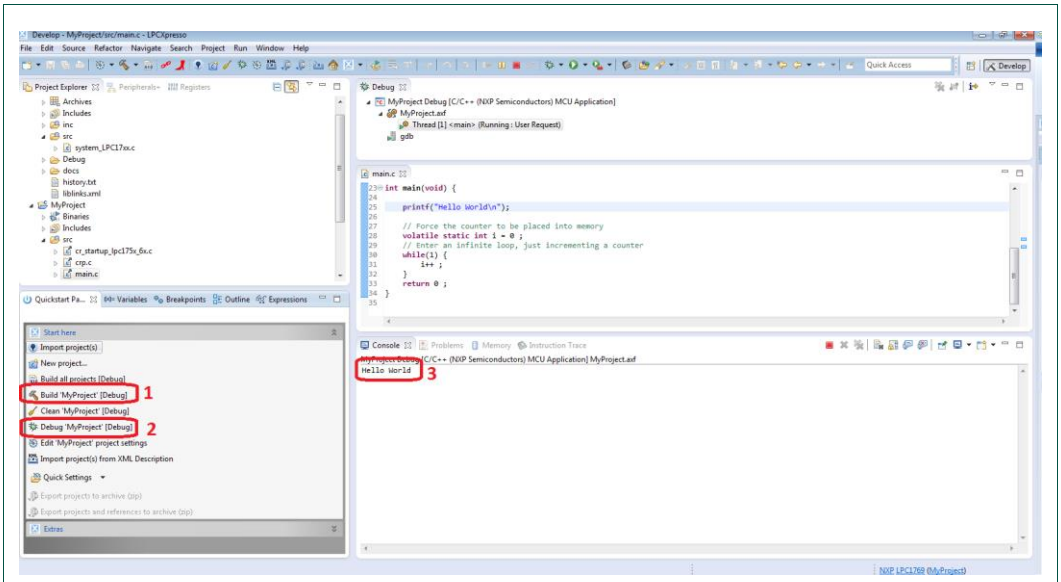

Fig 37. Compiling & Executing the project

11.3 Importing the NFC Reader Library and MCU drivers

After the *MyProject* project is created, the NFC Reader Library, the LPC1769 drivers and LPC1769 configuration files should be imported into the project workspace. These components can be imported by copying the source code files inside the *src* folder of the

project workspace. The LPC1769 drivers and the LPC1769 configuration files (include folder) can be found in any of the sample projects (see Section 4) provided in the NFC Reader Library release [3].

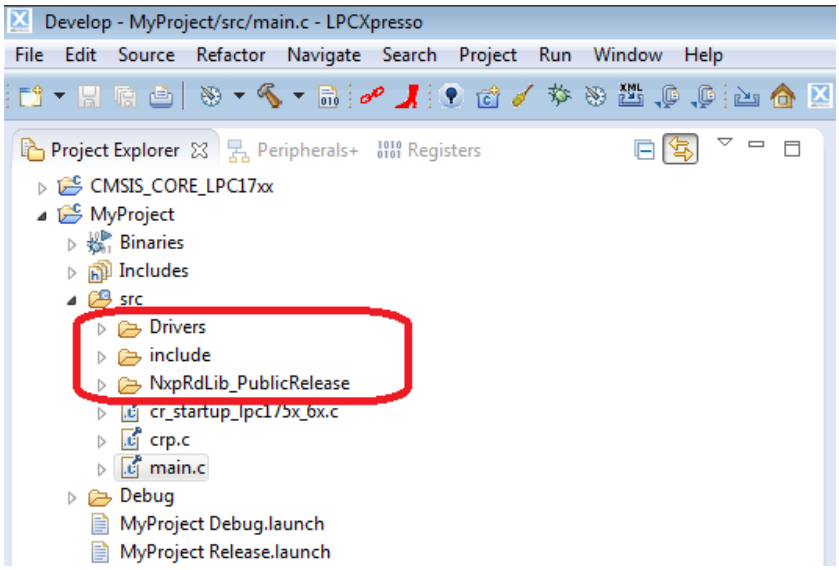

Fig 38. Import the NFC Reader Library and LPC1769 driver and LPC1769 configuration.

The *MyProject* project shall be properly configured for allowing the compilation and link of the NFC Reader Library and the LPC1769 driver folder that have been just imported. This step is completed by clicking on the project *Properties* and configuring the files to be included during project build on the *C/C++ Build Settings*:

```
"${workspace_loc:/${ProjName}/src/NxpRdLib_PublicRelease/intfs}"
"${workspace_loc:/${ProjName}/src/NxpRdLib_PublicRelease/types}"
"${workspace_loc:/${ProjName}/src/include}"
"${workspace_loc:/${ProjName}/src/Drivers/include}"
```

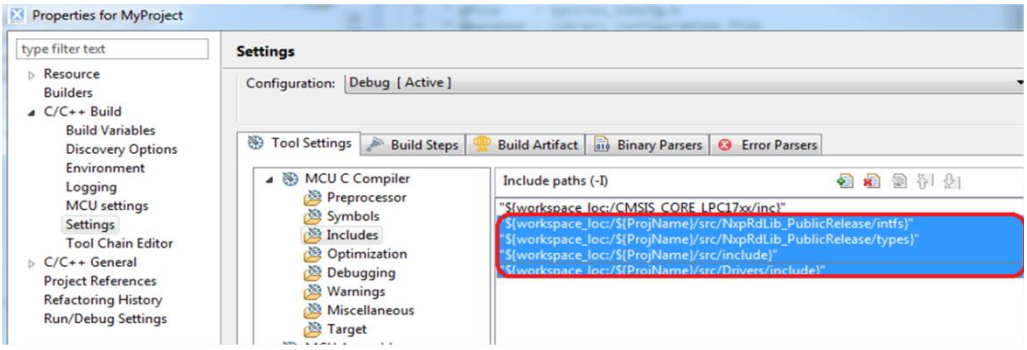

Fig 39. Project properties configuration

The developers can quickly check that the configuration has successfully been completed by compiling and executing the project and verify that the "*Hello World*" message is written in the Console.

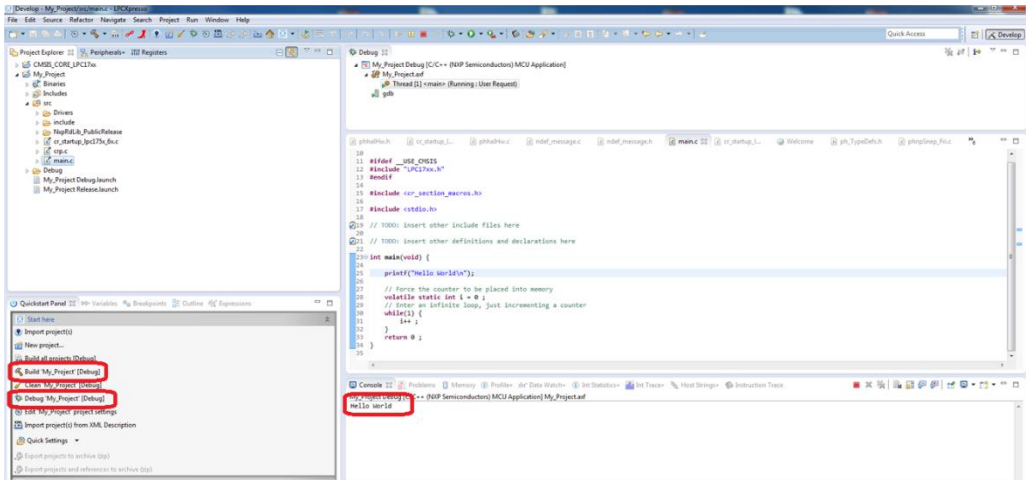

Fig 40. Compiling & Executing the project

11.4 Developing Customer Solutions

Once the whole project has been successfully created and configured, the developers are able to program their own application logic using the API provided by the NFC Reader Library.

12. FAQ

12.1 Does the NFC Reader Library allow the communication with existing NFC-enabled phones?

The NFC Reader Library implements the NFC Forum reference protocol stack for the Peer-to-Peer operating mode. Therefore, contactless solutions built according to the protocols that have been explained in this document are able to communicate with any NFC Forum Device.

Android Operating System is NFC Forum compliant since its version 4.0, also known as Ice Cream Sandwich. Windows Phone Operating System is NFC Forum compliant since its version 8.0 and Blackberry OS is NFC Forum compliant since its version 7.1.

12.2 Is it mandatory to use SNEP protocol for the P2P data exchange?

According to the protocol stack defined by the Peer-to-Peer operating mode, it is not mandatory to use the application-level SNEP protocol. A developer could implement its own communication logic on top of LLCP link-level protocol.

However, we strongly recommend to use the SNEP protocol. Firstly, because it is already implemented as part of the NFC Reader Library. Secondly, to ensure interoperability when communicating with other NFC-enabled devices such as smartphones, tablets, laptops, infrastructure readers, and other devices.

12.3 Why my Android phone does not respond to SNEP GET requests?

The current Android Operating System implementation is compliant with the SNEP Default Server defined by the NFC Forum. The SNEP protocol specification mandates the SNEP Default Server to only implement PUT requests and to return "NOT_IMPLEMENTED" response code to GET requests.

12.4 Can I use any other LPC MCU rather than LPC1769 MCU with the NFC Reader Library?

The implementation of the Peer-to-Peer protocol stack in the NFC Reader Library leads to a project binary file that requires a large memory size. The LPC1769 MCU, which is part of the LPC ConnectPlus family of NXP Semiconductors, perfectly fits the memory requirements imposed by the NFC Reader Library thanks to its large memory capacity. If there is a strong requirement to use another LPC MCU, the NFC Reader Library can be ported.

12.5 Can I port the NFC Reader Library to other MCU platforms?

The NFC Reader Library has been implemented as a modular layer based protocol stack. As it has been explained in the Introduction section, all the NFC Reader Library layers except the Bus Abstraction Layer are platform independent. Therefore, anyone can develop its own solution on any MCU platform as long as the Bus Abstraction Layer is implemented accordingly.

In addition to the modification of the Bus Abstraction Layer, MCU related files, part of the LPCXpresso project, should be adapted. These files include the '*CMSIS'* folder and hardware specific source files that are part of the '*include'* folder.

12.6 What are the differences between I²C and SPI communication protocols?

I²C and SPI protocols are well-suited for communications between integrated circuits and for slow communication with on-board peripherals. Most modern microcontrollers have hardware support for both protocols.

I²C is a multi-master protocol that uses 2 signal lines. The two I²C signals are called 'serial data' (SDA) and 'serial clock' (SCL). The I2C protocol specification states that the IC that initiates a data transfer on the bus is considered the Bus Master. Consequently, at that time, all the other ICs are regarded to be Bus Slaves. The protocol defines 7-bit unique slave addresses. Each device connected to the bus has a unique address.

SPI is a single-master communication protocol. SPI is a protocol on 4 signal lines: a clock signal named SCLK, sent from the bus master to all slaves, a slave select signal for

each slave, a data line from the master to the slaves, named MOSI (Master Out-Slave In) and a data line from the slaves to the master, named MISO (Master In-Slave Out). This means that one central device initiates all the communications with the slaves.

When the SPI master wishes to send data to a slave and/or request information from it, it selects a slave by pulling the corresponding line low and it activates the clock signal at a clock frequency usable by the master and the slave.

12.7 What are the differences between the debug and the release mode?

Debug and Release are different configurations for building your project. Debug mode is generally used for debugging your project, and the Release mode for the final build for end users.

The Debug mode does not optimize the binary it produces (as optimizations can greatly complicate debugging), and generates additional data to aid debugging. The Release mode enables optimizations and generates less (or no) extra debug data.

12.8 What is semihosting?

The main objective of semihosting is to allow I/O operations to be performed in a target system. This mechanism is especially interesting during debugging phase where output debug messages are shown on the screen in order to trace the execution of the program. A representative application of semihosting is for strings being depicted on the IDE screen by using printf function.

Semihosting basically indicates that part of the functionality is carried out by the host (the PC with the debug tools running on it), and partly by the target board. This task is done by piping messages over a serial cable to the PC where the IDE is running.

The performance achieved by programs performing operations is limited. Every time an I/O operation is required, the processor is stopped until the data is delivered, making our program to run slower than non semihosting enabled programs.

13. Appendix

13.1 Error Codes

The NFC Reader Library functions return codes that indicate success or failure of a certain operation. The error codes are 2 bytes long (type `uint16_t`) and provide information about the component on which the error has occurred (the first byte) and the error code number itself (the second byte). If no error has occurred during the function execution, 0x0000 (`PH_ERR_SUCCESS`) code is returned. Two masks ease the identification of the error received. The formulas to process it are as follows:

- (`0xFF00 &` *retunValue*) or (`PH_COMP_MASK &` *retunValue*) for the component code
- (`0x00FF &` *retunValue*) or (`PH_COMPID_MASK &` *retunValue*) for the error code

The defined error codes can be found in ph_*Status.h* file placed in *NxpRdLib_PublicRelease/types* folder.

13.1.1 Error Code Examples

Some concrete examples on how to read error codes are provided. The developer should be able to double check the error codes with the tables described in sections below.

- Error code: 0xEF21
  - 0xEF: Log component.
  - 0x21: Invalid parameter supplied.
- Error code: 0x0502
  - 0x05: ISO14443-4A PAL component.
  - 0x02: Integrity error, wrong CRC or parity detected.
- Error code: 0x1007
  - 0x10: MIFARE Classic AL component.
  - 0x07: Authentication error.

13.1.2 Component Error Code

The component error code identifies on which component the error has occurred. For instance, an error code in the form of 0x01xx means that some error has occurred in the BAL layer, as shown in the following table.

Table 6. Component error codes

Component Code	Component
0x0000U	Generic component code
0x0100U	BAL component
0x0200U	HAL component
0x0300U	ISO14443-3A PAL component
0x0400U	ISO14443-3B PAL component
0x0500U	ISO14443-4A PAL component
0x0600U	ISO14443-4 PAL component
0x0700U	MIFARE PAL component
0x0800U	FeliCa PAL component
0x0900U	ICode EPC/UID PAL component
0x0A00U	ICode SLI/ISO15693 PAL component
0x0B00U	ISO18000-3 Mode3 PAL component
0x0C00U	ISO18092 Initiator mode PAL component
0x0D00U	ISO18092 target mode PAL component
0x1000U	MIFARE Classic AL component
0x1100U	MIFARE Ultralight AL component
0x1200U	MIFARE Plus AL component
0x1300U	Virtual Card Architecture AL component
0x1400U	FeliCa AL component
0x1500U	ISO15693 AL component
0x1600U	ICode SLI AL component

Component Code	Component
0x1800U	ISO18000-3 Mode3 AL component
0x1900U	MIFARE DESFIRE EV1 AL component
0x1C00U	Type 1 Tag AL component
0x4000U	Discovery Loop NFC Activity component
0x6000U	LLCP P2P Package component
0x6100U	LLCP Core P2P Package component
0x6200U	LLCP Mac Mappings P2P Package component
0x6300U	LLCP Transport P2P Package component
0x6400U	LLCP OVR HAL P2P Package component
0x7000U	SNEP P2P Package component
0xE000U	Cid Manager component
0xE100U	CryptoSym component
0xE200U	KeyStore component
0xE300U	Tools component
0xE400U	CryptoRng component
0xEF00U	Log component
0xF000U	OSAL component

13.1.3 Error Code

The error code describes what kind of error occurred. It provides the user with information on why the error has happened and should help the user to identify the root cause. The NFC Reader Library differentiates between errors related to the communication with other components and errors derived from a wrong call of a function of the API.

Table 7. Communication error codes

Error Code	Communication error
0x0001U	IO TIMEOUT – No reply received –
0x0002U	INTEGRITY ERROR – Wrong CRC or parity detected –
0x0003U	COLLISION ERROR – A collision occurred –
0x0004U	BUFFER OVERFLOW – Buffer overflow –
0x0005U	FRAMING ERROR – Invalid frame format –
0x0006U	PROTOCOL ERROR – Received response violates protocol –
0x0007U	AUTH ERROR – Authentication error –
0x0008U	READ WRITE ERROR – A Read or Write error occurred in RAM/ROM or Flash –
0x0009U	TEMPERATURE ERROR – The RC sensors signal overheating –
0x000AU	RF ERROR – Error on RF Interface –
0x000BU	INTERFACE ERROR – An error occurred in RC communication –
0x000CU	LENGTH ERROR – A length error occurred –
0x000DU	RESOURCE ERROR – A resource error –
0x007FU	INTERNAL ERROR- An internal error occurred -

Table 8. Parameter and command error codes

Error code	Parameter and command error
0x0020U	INVALID DATA_PARAMS – Invalid data parameters supplied –
0x0021U	INVALID PARAMETER – Invalid parameter supplied–
0x0022U	PARAMETER OVERFLOW – Reading/Writing a parameter would produce an overflow –
0x0023U	UNSUPPORTED PARAMETER – Parameter not supported –
0x0024U	UNSUPPORTED COMMAND – Command not supported –
0x0025U	USE CONDITION – Condition of use not satisfied –
0x0026U	KEY – A key error occurred –

Table 9. NFC error codes

Error code	Parameter and command error
0x0003U	BUFFER TOO SMALL – Buffer provided by the caller is too small –
0x0006U	INVALID DEVICE – Device specified value is invalid for the operation –
0x000CU	INSUFFICIENT RESOURCES – Not enough resources available –
0x000DU	PENDING – Returned by a non-blocking function to indicate that an internal operation is in progress –
0x0011U	INVALID STATE – Invalid state of the particular state machine –
0x0031U	NOT INITIALIZED – The component has not been initialized –
0x0034U	NOT REGISTERED – Fail during unregistration command on a non-registered component –
0x0035U	ALREADY REGISTERED – Fail during registration command on an already registered component –
0x006FU	BUSY – The system is busy with the previous operation –

14. References

[1] NXP Generic Reader Library, http://www.nxp.com/documents/software/200312.zip

[2] NXP Export Controlled Library. (Available in DocStore [30]).

[3] NFC Reader Library (To be published)

[4] **Data Sheet** MF1S503X MIFARE Classic 1K - Mainstream contactless smart card IC for fast and easy solution development, available on http://www.nxp.com/documents/data_sheet/MF1S503x.pdf

[5] **Data Sheet** - MIFARE Ultralight ; MF0ICU1, MIFARE Ultralight contactless single-ticket IC, BU-ID Doc. No. 0286**[1], available on http://www.nxp.com/documents/data_sheet/MF0ICU1.pdf

[6] **Data Sheet** – MIFARE Ultralight EV1- contactless ticket IC, available on http://www.nxp.com/documents/data_sheet/MF0ULX1.pdf

[7] **Data Sheet** – MIFARE MF0ICU2 – MIFARE Ultralight C , available on http://www.nxp.com/documents/short_data_sheet/MF0ICU2_SDS.pdf

[8] **Data Sheet** - MIFARE DESFire; MF3ICDx21_41_81, MIFARE DESFire EV1 contactless multi-application IC, BU-ID Doc. No. 1340**, available on http://www.nxp.com/documents/short_data_sheet/MF3ICDX21_41_81_SDS.pdf

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

[10] **Data Sheet** – Innovision Topaz, http://downloads.acs.com.hk/drivers/en/TDS_TOPAZ.pdf

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

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

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

[14] **Data sheet** – PN512; Transmission module, BU-ID Doc. No. 1113**, available on http://www.nxp.com/documents/data_sheet/PN512.pdf

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

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

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

[18] **ISO/IEC Standard** - ISO/IEC 14443 Identification cards - Contactless integrated circuit cards - Proximity cards

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

[20] **Technical Specification** Logical Link Control Protocol, NFCForum-TS-LLCP_1.1, available on www.nxp.com/redirect/nfc-forum.org/specs/spec_license

[21] **Technical Specification** – Simple NDEF Exchange Protocol, NFCForum-TS-SNEP_1.0, available on www.nxp.com/redirect/nfc-forum.org/specs/spec_license

[22] **Technical Specification** – Type 1 Tag Operation, NFCForum-TS-Type-1-Tag_1.1, available on www.nxp.com/redirect/nfc-forum.org/specs/spec_license

[23] **Technical Specification** – Type 2 Tag Operation, NFCForum-TS-Type-2-Tag_1.1, available on www.nxp.com/redirect/nfc-forum.org/specs/spec_license

[24] **Technical Specification** – Type 3 Tag Operation, NFCForum-TS-Type-3-Tag_1.1, available on www.nxp.com/redirect/nfc-forum.org/specs/spec_license

[25] **Technical Specification** – Type 4 Tag Operation, NFCForum-TS-Type-4-Tag_2.0, available on www.nxp.com/redirect/nfc-forum.org/specs/spec_license

[26] **Technical Specification** – NFC Data Exchange Format, NFCForum-TS-NDEF_1.0, available on www.nxp.com/redirect/nfc-forum.org/specs/spec_license

[27] **Application note** - AN11211 Quick Start Up Guide RC663 Blueboard, available on http://www.nxp.com/documents/application_note/AN11211.pdf

[28] **Application note** – AN11308 Quick Start Up Guide PNEV512B, available on http://www.nxp.com/documents/application_note/AN11308.pdf

[29] LPCZone, http://www.nxp.com/techzones/microcontrollers-techzone/news.html

[30] NXP DocStore, https://www.docstore.nxp.com/flex/DocStoreApp.html#/l

[31] LPCXPresso IDE, http://www.lpcware.com/lpcxpresso/code-red

[32] LPCXpresso target boards, http://www.nxp.com/techzones/microcontrollers-techzone/tools-ecosystem/lpcxpresso.html

[33] **Application note** - AN11211 CLEV663B Blueboard Quick Start Guide, http://www.nxp.com/documents/application_note/AN11211.pdf

[34] **Application note** - AN11308 PNEV512B Blueboard Quick Start Guide, http://www.nxp.com/documents/application_note/AN11308.pdf

[35] NXP Contactless reader IC Demoboards ordering, http://www.nxp.com/products/identification_and_security/#demoboards

[36] **Application note** - AN11342 How to Scale Down the NXP Reader Library, http://www.nxp.com/documents/application_note/AN11342.pdf

[37] **Application note –** AN10802 NXP NFC Reader Library API

15. Legal information

15.1 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.

15.2 Disclaimers

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

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

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

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

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

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

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

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

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

Evaluation products — This product is provided on an "as is" and "with all faults" basis for evaluation purposes only. NXP Semiconductors, its affiliates and their suppliers expressly disclaim all warranties, whether express, implied or statutory, including but not limited to the implied warranties of non-infringement, merchantability and fitness for a particular purpose. The entire risk as to the quality, or arising out of the use or performance, of this product remains with customer.

In no event shall NXP Semiconductors, its affiliates or their suppliers be liable to customer for any special, indirect, consequential, punitive or incidental damages (including without limitation damages for loss of business, business interruption, loss of use, loss of data or information, and the like) arising out the use of or inability to use the product, whether or not based on tort (including negligence), strict liability, breach of contract, breach of warranty or any other theory, even if advised of the possibility of such damages.

Notwithstanding any damages that customer might incur for any reason whatsoever (including without limitation, all damages referenced above and all direct or general damages), the entire liability of NXP Semiconductors, its affiliates and their suppliers and customer's exclusive remedy for all of the foregoing shall be limited to actual damages incurred by customer based on reasonable reliance up to the greater of the amount actually paid by customer for the product or five dollars (US$5.00). The foregoing limitations, exclusions and disclaimers shall apply to the maximum extent permitted by applicable law, even if any remedy fails of its essential purpose.

15.3 Licenses

Purchase of NXP ICs with NFC technology

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

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

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

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

RATP/Innovatron Technology

15.4 Trademarks

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

MIFARE — is a trademark of NXP B.V.

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

16. Contents

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

Date of release: 07 April 2014
270121
Document identifier: UM10721

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

Date of release: 07 April 2014
270121
Document identifier: UM10721