PTX130R NCI-Reader API Stack Integration (SDK v2.0.0)

This document describes the “PTX NCI-Reader API” software stack (further referenced as “NCIRD API”) for the PTX130R [1] and the steps needed to build and integrate it into a target platform.

Contents

Figures

Tables

1. Introduction

This document describes the “PTX NCI-Reader API” software stack (further referenced as “NCIRD API”) for the PTX130R [1] and the steps needed to build and integrate it into a target platform.

The NCIRD API implements the “NFC Controller Interface (NCI)” protocol version 2.0 as specified by the NFC Forum [2]. The API represents a set of library functions which allows to implement generic NFC Forum reader applications or to integrate it into a mobile platform hosting a mobile OS such as Android or iOS.

In addition to the NCIRD API, the delivery also contains a demo implementation which shows the correct usage of the API and some typical examples like starting the RF discovery, perform RF data exchanges etc.

1.1 Audience

This document is intended to be used by:

- Software architects
- Software engineers
- Software integrator

1.2 Requirements

1.2.1. Building the NCIRD API Library

For building the NCIRD API stand-alone and / or the demo, the following tools are required:

- CMake 3.15 or higher (see cmake.org)
- Any C-compiler (depending on target platform) (*)
  (*) tested with gcc (Raspbian 8.3.0-6+rpi1) 8.3.0
- PTX1xxR IOT Config Tool
  (to build RF- and System-configuration)

1.2.2. Running the NCIRD API Library

To use the NCIRD API and / or execute the demo, the target platform must fulfill the following requirements:

- A general-purpose or real-time Operating System like Windows, (embedded) Linux, FreeRTOS, ...
- Permission to access the file system of the Operating System to:
  - Read configuration file(s) for the PTX130R
  - Read / write general configuration parameters from / to files
- Driver access to physical host interfaces

1.3 Terminology and Abbreviations

Term	Abbreviation
HW	Hardware
Integrator	Developer who builds and / or integrates the NCI-Reader API into a target application
NCI	NFC Controller Interface
NFC	Near Field Communication
NSC	NFC Soft Controller

Term	Abbreviation
RF	Radio Frequency
RTOS	Real-time Operating System
SDK	Software Development Kit
SW	Software

2. NCIRD API Software Architecture

The NCIRD API is based on the generic “PTX NSC Software Stack” and the generic “PTX NCI Software Stack” (both components are further referenced as “Core Stack”) which consists of multiple layers and (sub-) components.

The NCIRD API and the Core Stack are implemented in ANSI C and are therefore independent of the underlaying target platform.

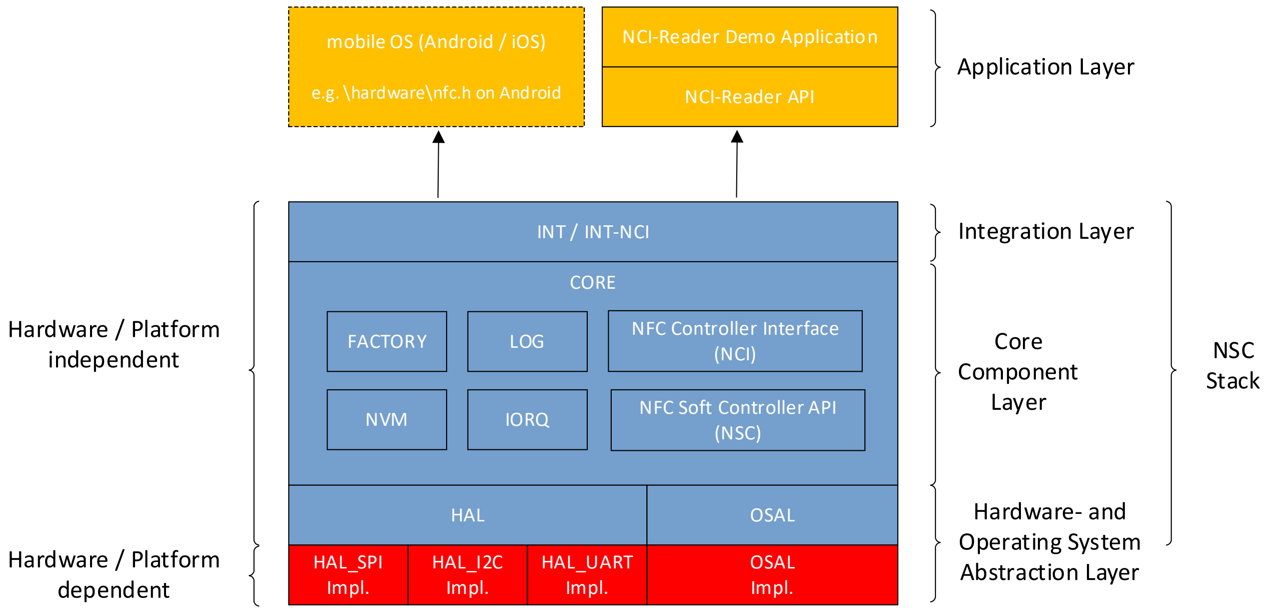

mobile OS (Android / iOS)
e.g. \hardware\nfc.h on Android
NCI-Reader Demo Application
NCI-Reader API
Application Layer
INT / INT-NCI
Integration Layer
CORE
FACTORY
LOG
NFC Controller Interface (NCI)
NVM
IORQ
NFC Soft Controller API (NSC)
Core Component Layer
NSC Stack
Hardware / Platform independent
HAL
OSAL
HAL_SPI Impl.
HAL_I2C Impl.
HAL_UART Impl.
OSAL Impl.
Hardware- and Operating System Abstraction Layer
Hardware / Platform dependent

Figure 1. NCIRD API and PTX Software Stack Architecture

2.1 Layer Description

2.1.1. Application Layer

2.1.1.1. Standard Application

This layer implements the actual NCIRD API which is the main interface to the target application as shown in Figure 1.

The NCIRD API provides a “NCI-Write”-function which allows to send NCI CMD- and (Tx-)DATA-packets as specified in [2]. A user-defined callback-routine handles all the synchronous events (NCI RSP-packets) and asynchronous events (NCI NTF- and (Rx-)DATA-packets).

The NCIRD API itself is described in detail in *NCIRD API Description* and an example flow is described in *NCIRD API States*.

2.1.1.2. Mobile Application / Integration

To integrate the "PTX NCI Software Stack" into a mobile OS, the integrator can either use the NCIRD API or can directly access the functions from the "Integration Layer" (see section 2.1.2). The implementation of the NCIRD API is an actual wrapper for a specific set of functions from the "Integration Layer".

2.1.2. Integration Layer

This layer implements a collection of all APIs of the Core Stack including its sub-components. The APIs from this layer can be used directly for the integration into a target application.

2.1.3. Core Component Layer

This layer represents the actual core of the Renesas (formerly Panthronics) NSC Stack including the NCI-component. It provides the following functionalities:

- PTX130R chip configuration and initialization
- RF- and SYSTEM configuration
- RF-communication including call-back functions for asynchronous events and error handling
- Simplified NSC Stack initialization and parametrization via Factory sub-component
- Extensive Logging-capabilities to ease system integration and debug support
- Internal HW access dispatcher ("IORQ")
- Abstracted file access ("NVM"), etc.

All these functionalities can either be accessed via the APIs of the "Integration Layer" or are encapsulated within an NCI-command.

2.1.4. Supported Features / Limitations of the NCI-Reader Stack

2.1.4.1. NCI Control Packets

Control Packet Type	Status	Comment
CORE_RESET_CMD	🟩	
CORE_RESET_RSP	🟩	
CORE_RESET_NTF	🟩	
CORE_INIT_CMD	🟩	
CORE_INIT_RSP	🟩	
CORE_GET_CONFIG_CMD	🟩	
CORE_GET_CONFIG_RSP	🟩	
CORE_SET_CONFIG_CMD	🟩	
CORE_SET_CONFIG_RSP	🟩	
CORE_CONN_CREATE_CMD	🟩	
CORE_CONN_CREATE_RSP	🟩	
CORE_CONN_CLOSE_CMD	🟩	
CORE_CONN_CLOSE_RSP	🟩	
CORE_CONN_CREDITS_NTF	🟩	
CORE_GENERIC_ERROR_NTF	🟩	

Control Packet Type	Status	Comment
CORE_INTERFACE_ERROR_NTF	🟩	
CORE_SET_POWER_SUB_STATE_CMD	🟩	
CORE_SET_POWER_SUB_STATE_RSP	🟩	
RF_DISCOVER_MAP_CMD	🟩	
RF_DISCOVER_MAP_RSP	🟩	
RF_SET_LISTEN_MODE_ROUTING_CMD	🟨	Implemented but disabled for NCI-Reader stack (answered with “STATUS_REJECTED”)
RF_SET_LISTEN_MODE_ROUTING_RSP	🟨	See RF_SET_LISTEN_MODE_ROUTING_CMD
RF_GET_LISTEN_MODE_ROUTING_CMD	🟨	Implemented but disabled for NCI-Reader stack (answered with “STATUS_REJECTED”)
RF_GET_LISTEN_MODE_ROUTING_RSP	🟨	See RF_GET_LISTEN_MODE_ROUTING_CMD
RF_GET_LISTEN_MODE_ROUTING_NTF	🟨	See RF_GET_LISTEN_MODE_ROUTING_CMD
RF_DISCOVER_CMD	🟩	Listen-Mode parameters ignored
RF_DISCOVER_RSP	🟩	
RF_DISCOVER_NTF	🟩	
RF_DISCOVER_SELECT_CMD	🟩	
RF_DISCOVER_SELECT_RSP	🟩	
RF_INTF_ACTIVATED_NTF	🟩	
RF_DEACTIVATE_CMD	🟩	
RF_DEACTIVATE_RSP	🟩	
RF_DEACTIVATE_NTF	🟩	
RF_FIELD_INFO_NTF	🟨	Implemented but disabled for NCI-Reader stack
TF_T3T_POLLING_CMD	🟩	
TF_T3T_POLLING_RSP	🟩	
TF_T3T_POLLING_NTF	🟩	
RF_NFCEE_ACTION_NTF	🟨	Implemented but disabled for NCI-Reader stack (not sent)
RF_NFCEE_DISCOVERY_REQ_NTF	🟨	Implemented but disabled for NCI-Reader stack (not sent)
RF_PARAMETER_UPDATE_CMD	🟦	Optional feature
RF_PARAMETER_UPDATE_RSP	🟦	Optional feature
RF_INTF_EXT_START_CMD	🟦	Optional feature
RF_INTF_EXT_START_RSP	🟦	Optional feature
RF_INTF_EXT_STOP_CMD	🟦	Optional feature
RF_INTF_EXT_STOP_RSP	🟦	Optional feature

Control Packet Type	Status	Comment
RF_EXT_AGG_ABORT_CMD	🟦	Optional feature
RF_EXT_AGG_ABORT_RSP	🟦	Optional feature
RF_NDEF_ABORT_CMD	🟦	Optional feature
RF_NDEF_ABORT_RSP	🟦	Optional feature
RF_ISO_DEP_NAK_PRESENCE_CMD	🟩	
RF_ISO_DEP_NAK_PRESENCE_RSP	🟩	
RF_ISO_DEP_NAK_PRESENCE_NTF	🟩	
RF_SET_FORCED_NFCEE_ROUTING_CMD	🟨	Implemented but disabled for NCI-Reader stack (answered with “STATUS_REJECTED”)
RF_SET_FORCED_NFCEE_ROUTING_RSP	🟨	See RF_SET_FORCED_NFCEE_ROUTING_CMD
NFCEE_DISCOVER_CMD	🟨	Implemented but disabled for NCI-Reader stack (answered with “STATUS_REJECTED”)
NFCEE_DISCOVER_RSP	🟨	See NFCEE_DISCOVER_CMD
NFCEE_DISCOVER_NTF	🟨	See NFCEE_DISCOVER_CMD
NFCEE_MODE_SET_CMD	🟨	Implemented but disabled for NCI-Reader stack (answered with “STATUS_REJECTED”)
NFCEE_MODE_SET_RSP	🟨	See NFCEE_MODE_SET_CMD
NFCEE_MODE_SET_NTF	🟨	See NFCEE_MODE_SET_CMD
NFCEE_STATUS_NTF	🟨	Implemented but disabled for NCI-Reader stack (not sent)
NFCEE_POWER_AND_LINK_CNTRL_CMD	🟨	Implemented but disabled for NCI-Reader stack (answered with “STATUS_REJECTED”)
NFCEE_POWER_AND_LINK_CNTRL_RSP	🟨	See NFCEE_POWER_AND_LINK_CNTRL_CMD
PROPR_RUN_SYSTEM_CMD_CMD	🟩	Proprietary command – implemented and enabled in the stack. Starts specific system actions.
PROPR_RUN_SYSTEM_CMD_RSP	🟩	See PROPR_RUN_SYSTEM_CMD
PROPR_RUN_SYSTEM_CMD_NTF	🟩	See PROPR_RUN_SYSTEM_CMD

2.1.4.2. RF Technologies and Modes

RF-Technology & Mode	Status	Comment
NFC_A_PASSIVE_POLL_MODE	🟩	
NFC_B_PASSIVE_POLL_MODE	🟩	
NFC_F_PASSIVE_POLL_MODE	🟩	
NFC_V_PASSIVE_POLL_MODE	🟩	
NFC_ACTIVE_POLL_MODE	🟦	Optional feature
NFC_A_PASSIVE_LISTEN_MODE	🟨	Implemented but disabled for NCI-Reader stack

RF-Technology & Mode	Status	Comment
NFC_B_PASSIVE_LISTEN_MODE	🟨	Implemented but disabled for NCI-Reader stack
NFC_F_PASSIVE_LISTEN_MODE	🟨	Implemented but disabled for NCI-Reader stack
NFC_ACTIVE_LISTEN_MODE	🟦	Optional feature

2.1.4.3. RF Interfaces

RF-Interface	Status	Comment
NFCEE Direct RF Interface	🟦	Optional feature
Frame RF Interface	🟩	
ISO-DEP RF Interface	🟩	Optional feature
NFC-DEP RF Interface	🟩	Optional feature
NDEF RF Interface	🟦	Optional feature
ExtMode RF Interface	🟦	Optional feature (proprietary)

2.1.4.4. RF Interface Extensions

RF-Interface Extension	Status	Comment
Frame Aggregated RF Interface Extension	🟦	Optional feature
LLCP Symmetry RF Interface Extension	🟦	Optional feature

2.1.4.5. RF Protocols

RF-Protocol	Status	Comment
PROTOCOL_UNDETERMINED	🟩	
PROTOCOL_T1T	🟥	Support dropped by NFC Forum
PROTOCOL_T2T	🟩	
PROTOCOL_T3T	🟩	
PROTOCOL_ISO_DEP	🟩	
PROTOCOL_NFC_DEP	🟩	
PROTOCOL_T5T	🟩	
PROTOCOL_NDEF	🟦	Optional feature
PROTOCOL_EXTMODE	🟦	Optional feature (proprietary)

2.1.4.6. NCI Data Messages

Data messages are used to exchange data over Logical Connections between a Device Host and NFCC target (NFCEE or Remote NFC Endpoint) [2]. The NCI-Reader Stack supports full-set of features defined by the NCI standard and related to data message exchange (e.g., chaining, credit-based flow control, payload sizes, etc).

Specific case are empty NCI data messages (with the payload length 0).

If an empty NCI data message is received by the host and the underlying RF interface and protocol is ISO-DEP, the Empty I-Block Presence Check will be started and the response will be sent to the host, as is done for any other data message.

If an empty NCI data message is received by the host and the underlying RF interface and protocol is NFC-DEP, the Attention command will be sent to the tag and the response will sent to the host.

For other interfaces and protocols, an empty NCI data message will issue sending an empty data message back to the host and nothing will be sent to NFC endpoint via transport layer.

2.1.4.7. Hardware Abstraction Layer

This component is dependent on the used physical HW interface (SPI, I2C or UART) between the application processor of the target platform and the PTX130R chip.

The Software interface requires function implementations to

- Open / close an HW interface
- Configure an HW interface (e.g. speed, timeouts etc.)
- Exchange data via the HW interface
- Cancel operations / access

Attention: The delivered SDK contains a reference implementation for SPI for the Linux OS. If an implementation for a different target platform is required, the reference implementation needs to be adapted (see “NCIRD API Target System Integration” or refer to Renesas.com to check for available reference implementations for other target platforms).

2.1.4.8. Operating System Abstraction Layer

This component is dependent on the Operating System of target platform. The SW interface requires function implementations to

- Create / close / suspend threads
- Allocate / free dynamic memory
- Initialize / destroy / lock / unlock Mutexes
- Initialize / close / post / wait Semaphores
- Create / close / start / stop / measure timers

Attention: The delivered SDK contains a reference implementation for the Linux OS. If an implementation for a different target platform is required, the reference implementation needs to be adapted (see “NCIRD API Target System Integration” or refer to Renesas.com to check for available reference implementations for other target platforms).

3. NCIRD API Description

This chapter contains an overview of the functions provided by the NCIRD API.

Note: A detailed description of all functions including parameters and types can be found in the “DOCS”-folder of the delivery (see \DOCS\index.html).

3.1 ptxNCIRD_Allocate_Stack

Declaration	`void *ptxNCIRD_Allocate_Stack (void);`	
Description	Allocates the main NCIRD API stack component	
Input Parameters	-	
Return Value	Pointer to stack component	

3.2 ptxNCIRD_Init_Stack

Declaration	`uint16_t ptxNCIRD_Init_Stack ( void *stackComp, ptxNCIRd_InitStack_Params_t *initParams);`	
Description	Initializes the NCIRD API stack component with given parameters	
Input Parameters	stackComp	Pointer to stack component
	initParams	Initialization parameters (host-interface type, host-interface name, path to RF- and SYS-config, user-defined callback-function etc.)
Return Value	Status of operation	

3.3 ptxNCIRD_Write

Declaration	`uint16_t ptxNCIRD_Write ( void *stackComp, uint8_t *nciTxData, size_t nciTxDataLen);`	
Description	Writes / Sends an NCI-Control or Data-packet	
Input Parameters	stackComp	Pointer to stack component
	nciTxData	Buffer containing NCI CMD- or DATA-packet
	nciTxDataLen	Length of NCI CMD- or DATA-packet
Return Value	Status of operation	

3.4 ptxNCIRD_Close_Stack

Declaration	`uint16_t ptxNCIRD_Close_Stack ( void *stackComp, char *logFile);`	
Description	Uninitializes the NCIRD API stack and provides an optional log-file	
Input Parameters	stackComp	Pointer to stack component
	logFile	Name of log file
Return Value	Status of operation	

4. NCIRD API States

Figure 2 below shows an example flow of how the NCIRD API should be used.

The NCIRD API itself follows an object-oriented approach where a so called “Stack Component” needs to be allocated first. This “Stack Component” serves as the main object which is used for every further call to any of the NCIRD API functions.

Figure 2 shows a typical example flow of how to use the NCIRD API assuming all functions return successfully. If an error occurs, the system remains in the current state.

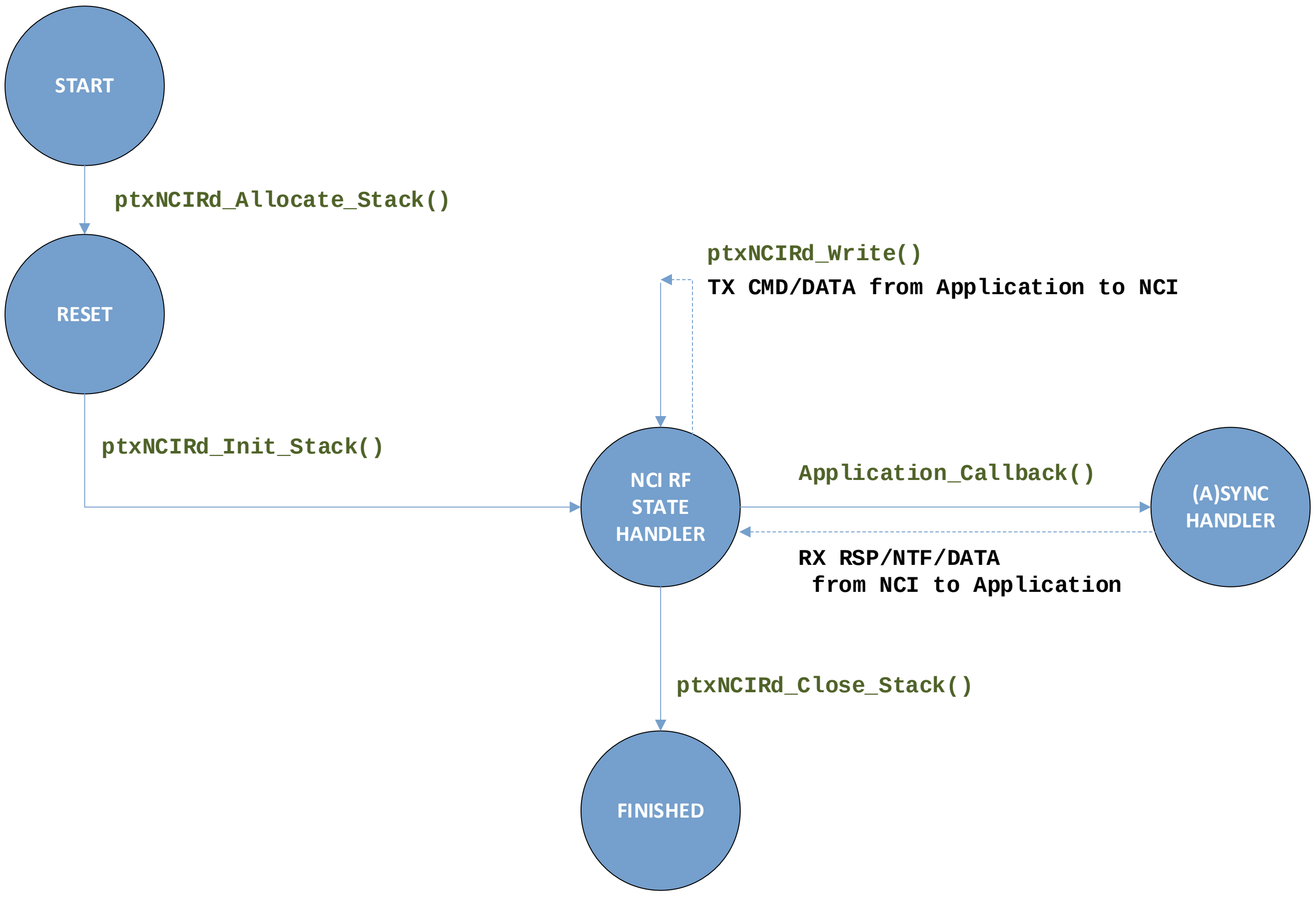


Figure 2. NCIRD API Flow Example

Attention: With exception of function “**ptxNCIRD_Allocate_Stack**”, all NCIRD API functions return a 16-bit status word indicating the status of the requested operation.

If an operation succeeded, the status word is set to 0x0000 (= SUCCESS). In any other case the upper 8 bit of the status word indicate the (sub-)component identifier of where the error occurred and the lower 8 bit indicate the exact error code.

Details of the status word definition can be found in the API-documentation (see chapter 5) or directly in the source file “**\SRC\COMPS\ptx_Status.h**”.

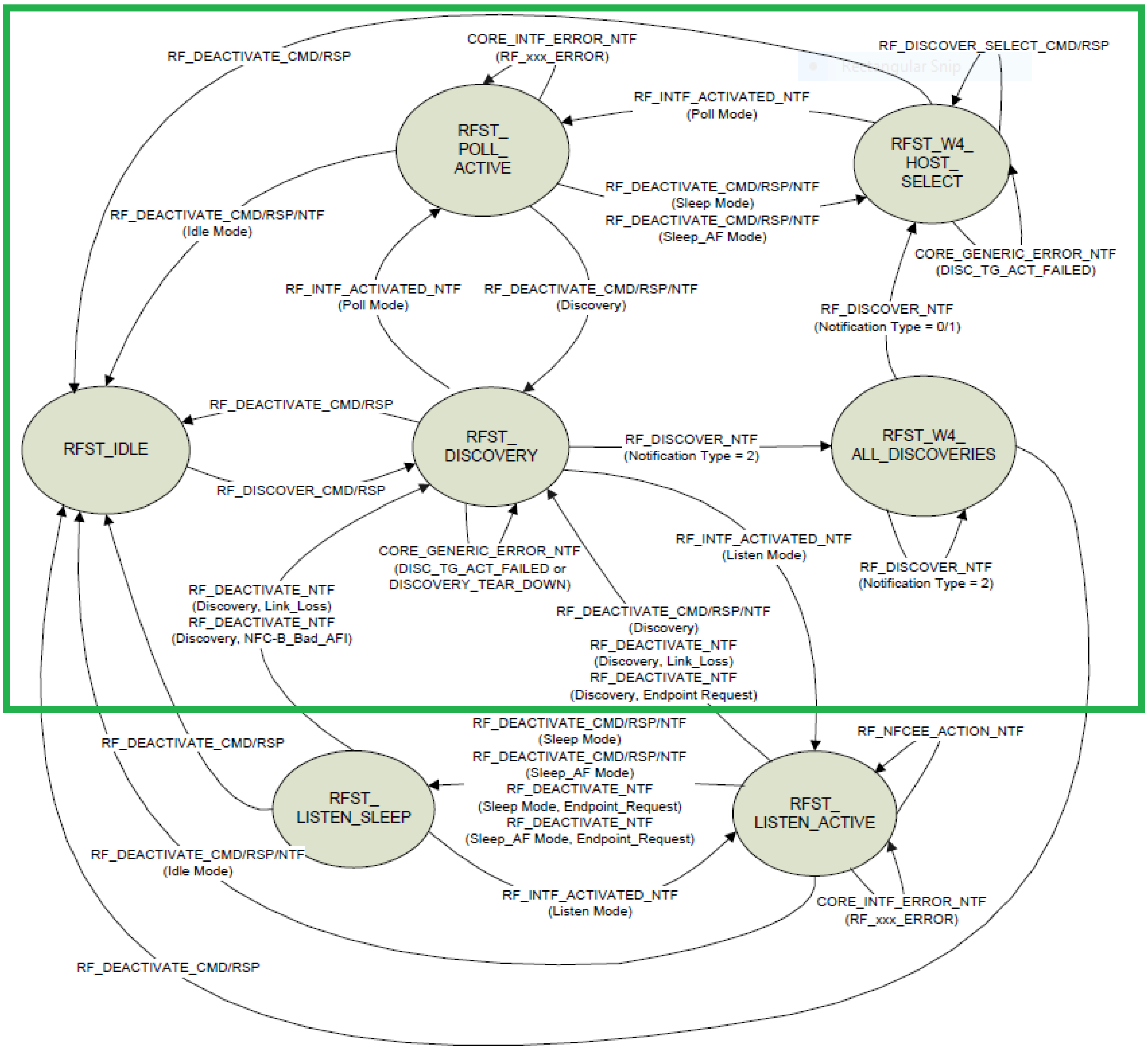


Figure 3. NCI RF State Machine [2]

State Descriptions:

- **State: START** – The "Stack Component" object needs to be allocated first via a call to "**ptxNCIRD_Allocate_Stack**".
- **State: RESET** – Initializes the NCIRD API and the internal NSC- and NCI components via a call to "**ptxNCIRD_Init_Stack**". This call takes parameters to initialize:
  - the intended host-interface including characteristics,

- the file path to the mandatory RF-configuration files "**NSC_RF_CONFIG.dat**" and "**NSC_SYS_CONFIG.dat**",
- the user-defined callback-application and context parameter structure.

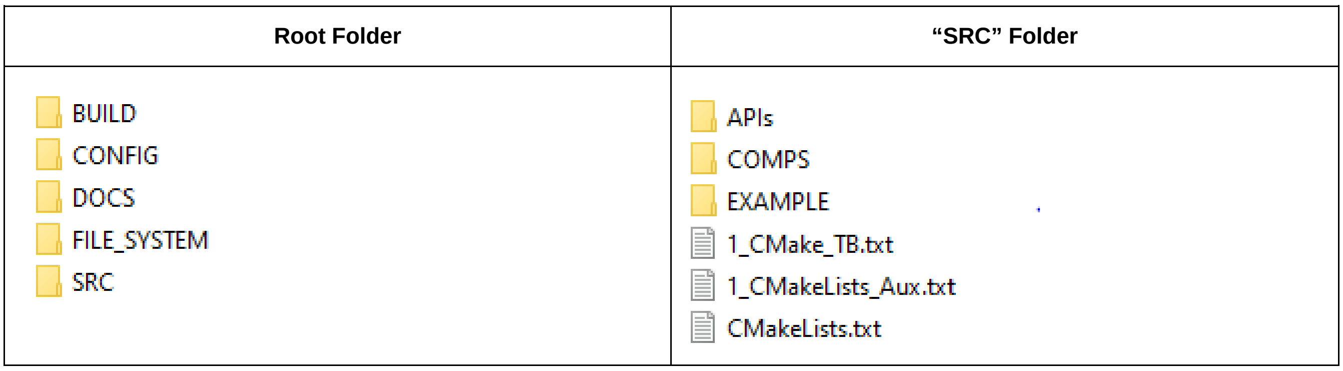
- **State: NCI RF STATE HANDLER** – After the Stack-initialization, the NCIRD API is ready to process the Poll-mode related “RF Communication State Machine” as defined in [2] and shown in Figure 3 below.
  NCI CMD- and Data-packets can be continuously exchanged via calls to “**ptxNCIRd_Write()**”.
- **State: (A)SYNC HANDLER** – This state is entered when synchronous or asynchronous events i.e. NCI RSP-, NTF- or Data-packet get received from the NCI-stack. When a packet arrives, the user-defined callback-routine gets called and the complete packet content is passed as input parameter via the application context parameter structure for further processing.
- **State: FINISHED** – Once the complete application shall be stopped or shut-down, it is required to call function "**ptxNCIRD_Close_Stack**" to free previously allocated system resources like memory, drivers etc.

5. NCIRD API SDK Deliverable

The NCIRD API SDK delivery contains the source code and API documentation for the NCIRD API, the NSC Stack and a demo example implementation.

The SDK also contains configuration file(s) for RF-settings a as well as build scripts to build the APIs and examples for the Linux OS based on a HAL reference implementation for a specific host interface.

The SDK is structured as shown in the following figure.



Root Folder	“SRC” Folder
BUILD	APIs
CONFIG	COMPS
DOCS	EXAMPLE
FILE_SYSTEM	1_CMake_TB.txt
SRC	1_CMakeLists_Aux.txt
	CMakeLists.txt

Figure 4. NCIRD API SDK Folder Structure

Note: If not otherwise stated, folders containing source code includes the corresponding .c and .h files.

- **\BUILD**
  Output folder of the build process containing NCIRD API as shared library and / or the demo application as executable.
- **\CONFIG**
  Configuration-files and -scripts for RF- and System-configuration for PTX130R chip.
- **\DOCS**
  HTML-based description of the NCIRD API.
  Note: The landing page for the description is “**\DOCS\html\index.html**”.
- **\FILE_SYSTEM**
  Contains the various configuration files for NSC Stack and PTX130R chip.

  Content:

\NSC_RF_CONFIG.dat	RF-configuration for PTX130R chip
\NSC_SYS_CONFIG.dat	System-configuration for PTX130R chip

- **\SRC\APIs**
  Contains the source code for the NCIRD API.

Content:

\NCI_READER\ptxNCI_READER.*	NCIRD API

- **\SRC\COMPS**
  Source code of NSC Stack including (sub-)components and HAL and OSAL.

  Content:

\FACTORY*.*	Factory component
\HAL*.*	Hardware Abstraction Layer (HAL) including a reference implementation based on either UART, I2C or SPI for Linux
\INT*.*	Integration Layer
\IORQ*.*	Hardware access dispatcher for PTX130R
\LOG\	Logging component
\NSC\	NSC Stack Core component
\NCI\	NCI Stack Core component
\NVM\	NVM access component (file access)
\OSAL\	Operating System Abstraction Layer (OSAL) including reference implementation for Linux
*.h	Generic headers (status information, compile switches etc.)

- **\SRC\EXAMPLE**
  Contains a demo example implementation how to use the NCIRD API.

  Content:

\ptxNCI_READER_Demo.*	Demo application
\ptxNCI_READER_Demo_Support.*	Support functions for demo application

- **\SRC\CMake*.txt**
  CMake-based scripts to build the NCIRD API and the demo application.

6. NCIRD API Target System Integration

This chapter describes the required steps for an SW integrator to

- compile the NCIRD API together with an example application as binary to work stand-alone
- integrate the source code of the NCIRD API as (sub-)component into an existing application
- implement the abstraction layers for HAL and OSAL
- use the CMake build system
- enable/disable logging

6.1 Introduction

As described in chapter 5, the NCIRD API and all the other components are available as source code of the delivered SDK.

Attention: The SDK contains a reference implementation for the target platform dependent abstraction layers HAL and OSAL based on the Linux OS and a specific host interface.
If another target platform and / or host interface is used, HAL and OSAL need to be adapted accordingly as described in section 6.5.

While the source code of the NCIRD API can be directly integrated into existing applications (see section 6.4), the SDK contains ready-to-use build-scripts based on CMake which supports quick creation of the following in order to allow fast prototyping and integration:

- The NCIRD API as stand-alone shared library or
- Combined with the demo application as executable binary

6.2 Build System

The build system which is delivered with the SDK is based on CMake, a cross-platform independent tool to build software.

CMake-based projects have the advantage that they can be imported into a variety of well-known development tools like Eclipse, Visual Studio, Visual Studio Code and many others. In addition, CMake supports automatic compiler detection by searching for typical executables like “cc”, “gcc”, “clang” etc. (as defined and available in PATH-variable). Automatic compiler detection is an optional feature and can be overwritten by a manual choice. For more details on how to set up specific generators for compilers, see CMake or invoke “**`cmake /?`**” (*) from the command line.

(*) If CMake is not registered in a system-wide environment variable, please invoke command directly from the installation folder of CMake.

The SDK for the NCIRD API provides the following configuration CMake-scripts:

- 1_CMake_TB.txt
  - Main entry script file defining build-targets
- 1_CMakeLists.Aux.txt
  - Defines which source files are included in the build

Attention: If the existing CMake-scripts are used and new files get added to the application or existing files get renamed, these changes must be added / changed in this file!

- CMakeLists.txt
  - Defines which source files / lists (see 1_CMakeLists.Aux.txt) belong to which build-target.

Currently the following build targets are defined:

- “NCIRD_EXAMPLE_EXE”
  This target builds the NCIRD API together with the demo application as executable binary. The resulting executable is named “NCIRDExample” and uses the corresponding file extension of the target platform (e.g. .exe on Windows, .a on Linux etc.).

  Build-steps:
  ```
  1. “cmake -g .”
  2.“cmake --build . --target NCIRD_EXAMPLE_EXE --config BUILD_TYPE”
  ```

- **“NCIRD_LIB”**
  This target builds the NCIRD API stand-alone as dynamic library (*). The resulting library is named “libTestBench” and uses the corresponding file extension of the target platform (e.g. .dll on Windows, .so on Linux etc.)

  Build-steps:
  ```
  1. “cmake -g .”
  2.“cmake --build . --target NCIRD_LIB --config BUILD_TYPE”
  ```

Attention: Build-step 1 of each target is only needed once after installation of SDK or once every time one of the CMake-scripts get changed!

Note: “BUILD_TYPE” is an optional parameter and defines whether the executable is a release version (BUILD_TYPE to be replaced with “Release”) or a debug version (BUILD_TYPE to be replaced with “Debug”).

6.3 Integration Flow – NCIRD API (Stand-alone)

Figure 5 shows the integration flow for the NCIRD API as stand-alone library (Path 1) and the NCIRD API combined with the demo application as executable (Path 2) based on the CMake-build process.

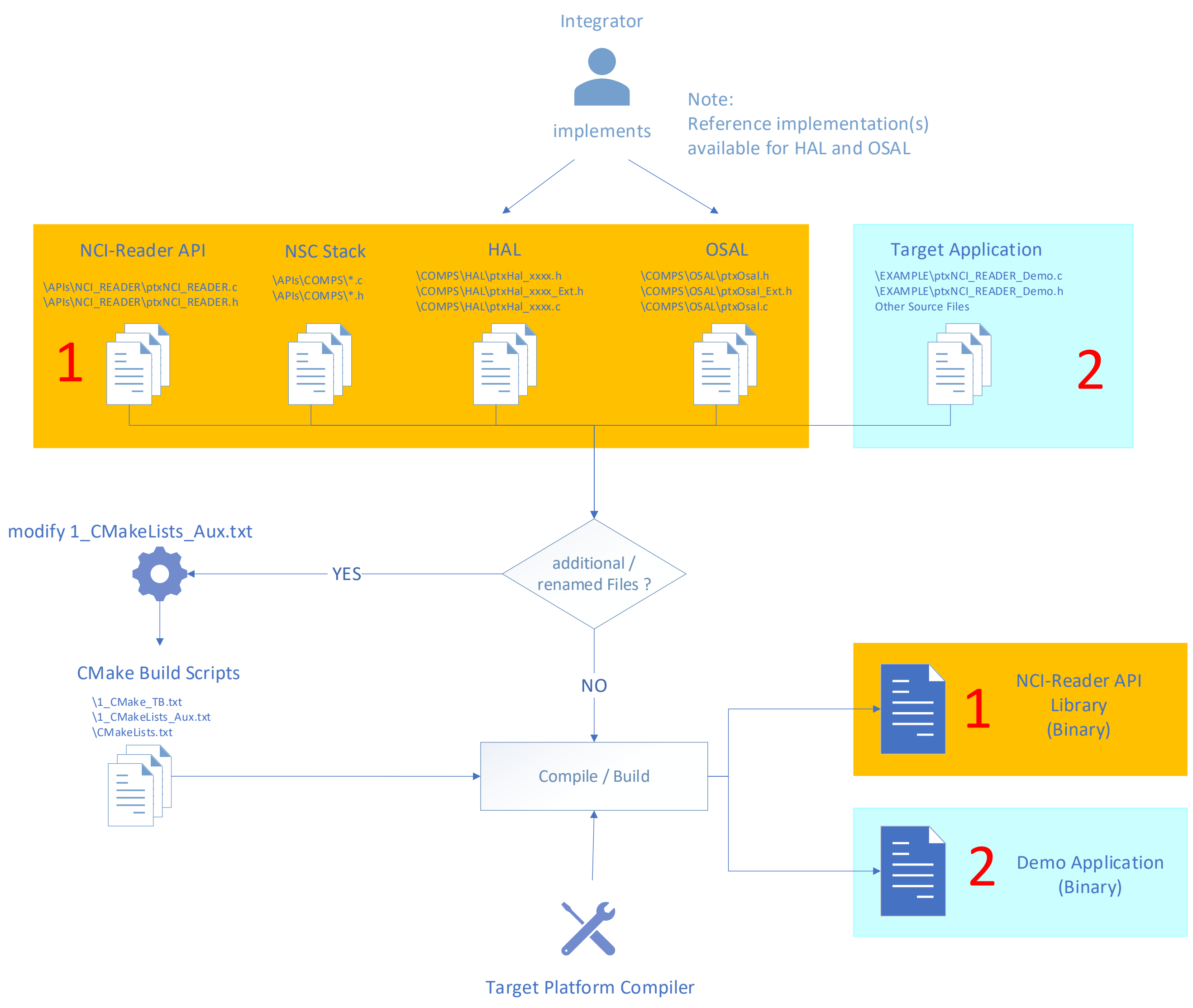

Integrator
implements
Note:
Reference implementation(s)
available for HAL and OSAL
NCI-Reader API
\APIs\NCI_READER\ptxNCI_READER.c
\APIs\NCI_READER\ptxNCI_READER.h
1
NSC Stack
\APIs\COMPS\*.c
\APIs\COMPS\*.h
HAL
\COMPS\HAL\ptxHal_xxxx.h
\COMPS\HAL\ptxHal_xxxx_Ext.h
\COMPS\HAL\ptxHal_xxxx.c
OSAL
\COMPS\OSAL\ptxOsal.h
\COMPS\OSAL\ptxOsal_Ext.h
\COMPS\OSAL\ptxOsal.c
Target Application
\EXAMPLE\ptxNCI_READER_Demo.c
\EXAMPLE\ptxNCI_READER_Demo.h
Other Source Files
2
modify 1_CMakeLists_Aux.txt
YES
additional / renamed Files ?
NO
CMake Build Scripts
\1_CMake_TB.txt
\1_CMakeLists_Aux.txt
\CMakeLists.txt
Compile / Build
Target Platform Compiler
1
NCI-Reader API Library (Binary)
2
Demo Application (Binary)

Figure 5. NCIRD API Integration Flow: Stand-alone System

Common for both paths are the implementation and / or adaptions for HAL and OSAL which need to be done by the integrator at the very beginning. The remaining components like the actual NCIRD API and the NSC Stack can be used “as-is”.

The provided CMake-scripts already use the correct list of source-files. If there are any modifications necessary (e.g., renamed files or added / deleted files), they must be added to the file “1_CMakeLists_Aux.txt” in the root folder of the SDK.

6.4 Integration Flow – NCIRD API (Component)

Figure 6 shows the example flow when the source code of the NCIRD API and the other components get directly integrated into an existing system / application.

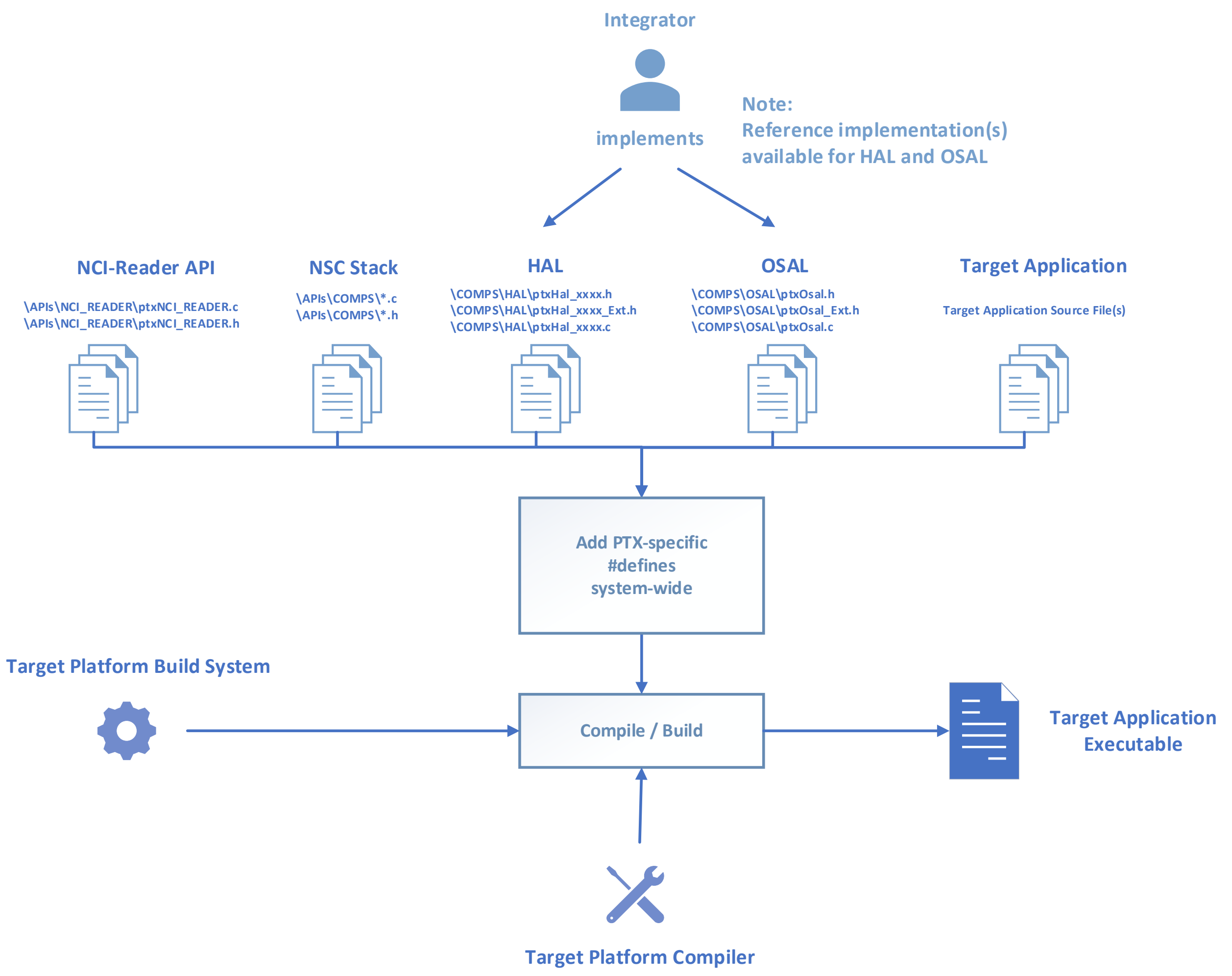

Integrator
implements
Note:
Reference implementation(s)
available for HAL and OSAL
NCI-Reader API
\APIs\NCI_READER\ptxNCI_READER.c
\APIs\NCI_READER\ptxNCI_READER.h
NSC Stack
\APIs\COMPS\*.c
\APIs\COMPS\*.h
HAL
\COMPS\HAL\ptxHal_xxxx.h
\COMPS\HAL\ptxHal_xxxx_Ext.h
\COMPS\HAL\ptxHal_xxxx.c
OSAL
\COMPS\OSAL\ptxOsal.h
\COMPS\OSAL\ptxOsal_Ext.h
\COMPS\OSAL\ptxOsal.c
Target Application
Target Application Source File(s)
Add PTX-specific #defines system-wide
Target Platform Build System
Compile / Build
Target Application Executable
Target Platform Compiler

Figure 6. NCIRD API Integration Flow: (sub-)Component within Existing Application

The approach for the NCIRD API, the NSC Stack and the abstraction layers HAL and OSAL is the same as described in chapter 6.3.

Attention: To build the NCIRD API part, the following #defines must be added to the build-environment:

- “PTX_FEATURES_NSC_READER_ONLY”
- “PTX_FEATURES_NCI_INCLUDED”
- “PTX_FEATURES_HAL_YYY” (YYY = SPI, I2C or UART)

6.5 Target Platform Abstraction Layers

The NSC Stack contains the two components

- Hardware Abstraction Layer (HAL)
- Operating System Abstraction Layer (OSAL)

which are both split into a target platform independent - and dependent part.
The target platform independent part is directly used by the NSC Stack and should therefore not be changed.
The target platform dependent needs to be adapted for the specific platform.

Note: The base for a target platform specific OSAL- or HAL implementation is the reference implementation of OSAL for the Linux OS and a reference HAL implementation for a specific host interface provided in the SDK-package.

6.5.1. Hardware Abstraction Layer (HAL)

The HAL serves as the abstraction layer for the dedicated physical host interface between the application processor and the PTX130R chip which can be either SPI, I2C or UART.

The source code including the SW interface for the HAL can be found in \SRC\COMPS\HAL\ and is structured as follows:

Target platform independent Part:

- **\SRC\COMPS\HAL\ptxHal.h**
  HAL SW interface directly used by NSC Stack
- **\COMPS\HAL\ptxHal_Ext.h**
  HAL data structures and support functions for implementation
- **\SRC\COMPS\HAL\ptxHal.c**
  Host interface independent HAL implementation

Target platform dependent Part:

Note: “xxxx” in the following paragraphs stands for either “UART”, “I2C” or “SPI”.

- **\SRC\COMPS\HAL\ptxHal_xxxx.h**
  HAL SW interface defining the API functions to be implemented by the integrator.
- **\SRC\COMPS\HAL\ptxHal_xxxx_Ext.h**
  Customizable HAL data structure and helpers for reference implementation.

Attention: The file “**ptxHal_xxxx_Ext.h**” defines the data structure type “**ptxHal_xxxx**” which is used by the target platform independent part.
The current definition of this structure including the content is used by the reference implementation (see below) but can be customized depending on the requirements and / or implementation of the target platform.

- **\SRC\COMPS\HAL\ptxHal_xxxx_Linux.c**
  - Linux OS reference implementation for specific host interface.

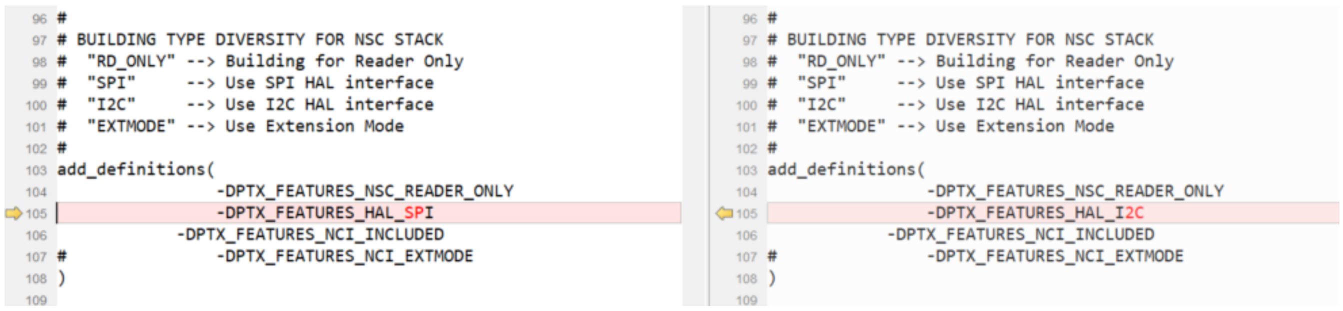
Attention: When the physical host interfaces SPI or I2C are used, the PTX130R chip use the pin “IRQ” to indicate if it wants to send data to the host application processor. When using the UART as physical host interface, the pin “IRQ” is not used i.e. data is sent asynchronously to the host application processor.

6.5.1.1. Hardware Interface Selection

Default HAL interface of the SDK delivery is SPI. This means all the source files relevant for SPI are included in the “1_CMakeLists_Aux.txt” in the root folder of the SDK. In addition, the file “1_CMake_TB.txt“ contains the definition of PTX_FEATURES_HAL_SPI.

NCI SDK v2.0.0 also adds the support for HAL I2C interface – source files for SPI and I2C are present in the SDK. Please follow the steps below to quickly set up/select the I2C target interface:

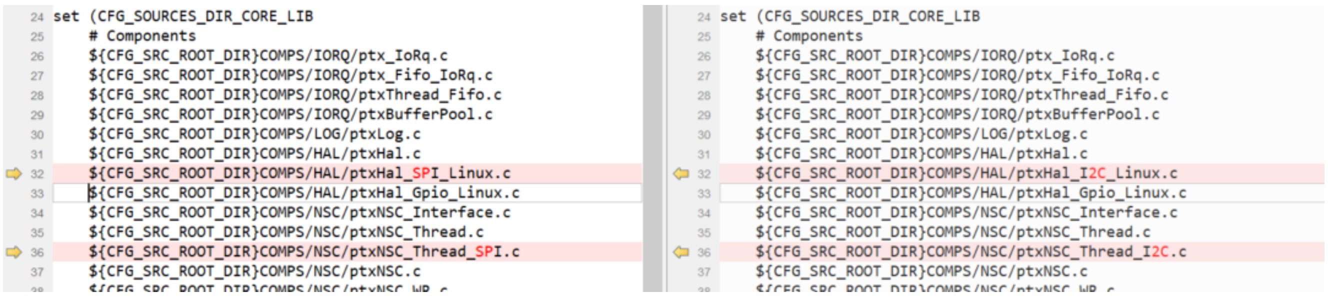
- Define PTX_FEATURES_HAL_I2C (replace the definition for SPI with the one for I2C) in “1_CMake_TB.txt“



```
#
# BUILDING TYPE DIVERSITY FOR NSC STACK
#  "RD_ONLY" --> Building for Reader Only
#  "SPI"     --> Use SPI HAL interface
#  "I2C"     --> Use I2C HAL interface
#  "EXTMODE" --> Use Extension Mode
#
add_definitions(
                    -DPTX_FEATURES_NSC_READER_ONLY
                    -DPTX_FEATURES_HAL_SPI
               -DPTX_FEATURES_NCI_INCLUDED
#                   -DPTX_FEATURES_NCI_EXTMODE
)
```

```
#
# BUILDING TYPE DIVERSITY FOR NSC STACK
#  "RD_ONLY" --> Building for Reader Only
#  "SPI"     --> Use SPI HAL interface
#  "I2C"     --> Use I2C HAL interface
#  "EXTMODE" --> Use Extension Mode
#
add_definitions(
                    -DPTX_FEATURES_NSC_READER_ONLY
                    -DPTX_FEATURES_HAL_I2C
               -DPTX_FEATURES_NCI_INCLUDED
#                   -DPTX_FEATURES_NCI_EXTMODE
)
```

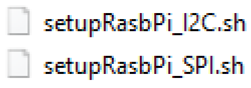
- Replace the following source files in “1_CMakeLists_Aux.txt”



```
set (CFG_SOURCES_DIR_CORE_LIB
    # Components
    ${CFG_SRC_ROOT_DIR}COMPS/IORQ/ptx_IoRq.c
    ${CFG_SRC_ROOT_DIR}COMPS/IORQ/ptx_Fifo_IoRq.c
    ${CFG_SRC_ROOT_DIR}COMPS/IORQ/ptxThread_Fifo.c
    ${CFG_SRC_ROOT_DIR}COMPS/IORQ/ptxBufferPool.c
    ${CFG_SRC_ROOT_DIR}COMPS/LOG/ptxLog.c
    ${CFG_SRC_ROOT_DIR}COMPS/HAL/ptxHal.c
    ${CFG_SRC_ROOT_DIR}COMPS/HAL/ptxHal_SPI_Linux.c
    ${CFG_SRC_ROOT_DIR}COMPS/HAL/ptxHal_Gpio_Linux.c
    ${CFG_SRC_ROOT_DIR}COMPS/NSC/ptxNSC_Interface.c
    ${CFG_SRC_ROOT_DIR}COMPS/NSC/ptxNSC_Thread.c
    ${CFG_SRC_ROOT_DIR}COMPS/NSC/ptxNSC_Thread_SPI.c
    ${CFG_SRC_ROOT_DIR}COMPS/NSC/ptxNSC.c
```

```
set (CFG_SOURCES_DIR_CORE_LIB
    # Components
    ${CFG_SRC_ROOT_DIR}COMPS/IORQ/ptx_IoRq.c
    ${CFG_SRC_ROOT_DIR}COMPS/IORQ/ptx_Fifo_IoRq.c
    ${CFG_SRC_ROOT_DIR}COMPS/IORQ/ptxThread_Fifo.c
    ${CFG_SRC_ROOT_DIR}COMPS/IORQ/ptxBufferPool.c
    ${CFG_SRC_ROOT_DIR}COMPS/LOG/ptxLog.c
    ${CFG_SRC_ROOT_DIR}COMPS/HAL/ptxHal.c
    ${CFG_SRC_ROOT_DIR}COMPS/HAL/ptxHal_I2C_Linux.c
    ${CFG_SRC_ROOT_DIR}COMPS/HAL/ptxHal_Gpio_Linux.c
    ${CFG_SRC_ROOT_DIR}COMPS/NSC/ptxNSC_Interface.c
    ${CFG_SRC_ROOT_DIR}COMPS/NSC/ptxNSC_Thread.c
    ${CFG_SRC_ROOT_DIR}COMPS/NSC/ptxNSC_Thread_I2C.c
    ${CFG_SRC_ROOT_DIR}COMPS/NSC/ptxNSC.c
```

Note: Default target platform of the SDK delivery is RaspberryPi. Since the SDK also contains a demo application for the target platform, a RaspberryPi device needs to be properly configured to use SPI or I2C interface, and to use GPIO pin as defined by the SDK. There are 2 scripts in the \BUILD folder, to configure each of the HAL interfaces to be used by the demo application. A corresponding script should be run (twice) before the application executable.



setupRasbPi_I2C.sh
setupRasbPi_SPI.sh

6.5.2. Operating System Abstraction Layer (OSAL)

The OSAL serves as the abstraction layer for the underlying Operating System.

The source code including the SW interface for the HAL can be found in SRC\COMPS\OSAL\ and is structured as follows:

Target platform independent Part:

- **\SRC\COMPS\OSAL\ptxOsal.h**
  OSAL SW interface directly used by NSC Stack
- **\SRC\COMPS\OSAL\ptxOsal_Ext.h**
  OSAL data structures and support functions for implementation

Target platform dependent Part:

- **\SRC\COMPS\OSAL\ptxOsal_Linux.c**
  Linux OS reference implementation.

7. RF Interface Extensions

RF Interface Extensions extend the functionality of an RF Interface. Beside the extensions described in chapter 9 of NCI Technical Specification [2], the **"NCIRD API"** implements in addition one proprietary RF interface - Extension Mode RF Interface (further referenced as **"ExtMode"**). This interface provides specific features to enable compatibility to Mifare Classic cards incl. authentication, encryption and decryption.

Attention: **ExtMode** support is optional and can be opted out by the means of (un)commenting the preprocessor definition PTX_FEATURES_NCI_EXTMODE. This option is provided as a target compile definition option in the SDK CMakeLists.txt file (*-DPTX_FEATURES_NCI_EXTMODE*).

If **ExtMode** is used in the project, it is needed to obtain the 3rd party code as explained in the chapter 6.6.3 External Dependencies.

7.1 Startup and Stop Conditions

After activation of the Frame RF Interface, the **ExtMode** is automatically started (activated) under the following conditions:

- The Frame RF Interface is activated in POLL mode (**RFST_POLL_ACTIVE** state).
- The Remote NFC Endpoint is using **PROTOCOL_T2T** RF Protocol
- The Remote NFC Endpoint is using a custom **specific SEL_RES**

ExtMode is stopped automatically after RF_DEACTIVATE_CMD. The following values have been assigned for **ExtMode** RF Protocol and Interface (values are chosen from the range allocated for proprietary use in the Table 133. and 134. of NCI specification [2]):

EXTMODE_RF_PROTOCOL	0x80
EXTMODE_RF_INTERFACE	0x80

7.2 Functionality

ExtMode uses the same data mapping as Frame RF Interface in [2]. However, data messages payload has specific, custom defined structure. All the functionality is provided/executed by using NCI data packets, with the payload containing MFC commands or responses, as shown in Figure 7.

Commands are sent from device host (DH) to NFC Controller (NFCC). Responses are sent from NFCC to DH.

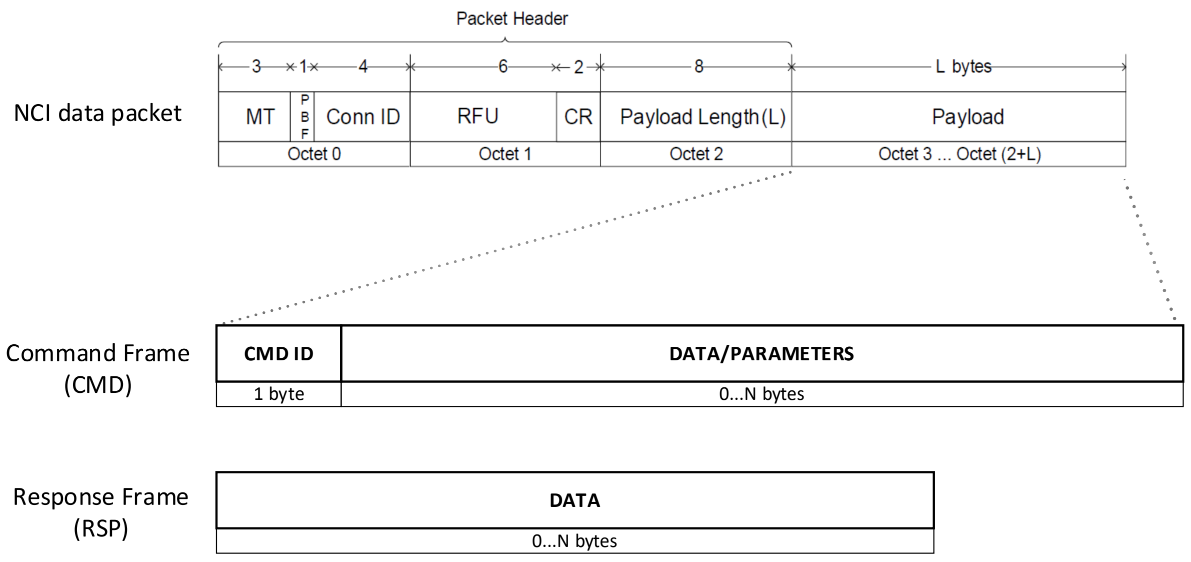


Figure 7. ExtMode RF Interface Data Messages Structure

Currently, **ExtMode** supports the commands and responses specified in Table 1 and 2. Their specific structures are further described in Tables 3 to 8.

Table 1. ExtMode Commands

ExtMode Commands		
Command name	**CMD ID**	**Description**
EXTM_AUTH_KEYA	0x60	DH requests that NFCC starts executing authentication procedure with the connected RF endpoint by using the KeyA.
EXTM_AUTH_KEYB	0x61	DH requests that NFCC starts executing authentication procedure with the connected RF endpoint by using the KeyB.
EXTM_AUTH_READ	0x30	DH requests that NFCC sends encrypted read block command.
EXTM_AUTH_WRITE	0xA0	DH requests that NFCC sends encrypted write block command.

Table 2. ExtMode Responses

ExtMode Responses		
Type	**Value**	**Description**
EXTM_RSP_ACK	0x00	NFCC sends in case of successful authentication or write command.
EXTM_RSP_NAK	0xFF	NFCC sends in case of authentication fail.

Table 3. EXTM_AUTH_KEYA/B Command

EXTM_AUTH_KEY			
CMD ID	**BLOCK NUMBER**	**UID**	**KEY**
1 By 0x60 (KeyA)/ 0x61 (KeyB)	1 By	4 By	6 By

Table 4. Response to EXTM_AUTH_KEY Command

ExtMode Response	
Value	**Description**
0x00	Returned only in case all authentication steps complete successfully. Any other value means error. Length: 1By

Table 5. EXTM_AUTH_READ Command

EXTM_READ	
CMD ID	**BLOCK NUMBER**
1 By 0x30	1 By

Table 6. Response to EXTM_AUTH_READ Command

ExtMode Response	
Value	**Description**
-	In case of success: 16 By of data are returned. Otherwise, no data returned.

Table 7. EXTM_AUTH_WRITE Command

EXTM_WRITE		
CMD ID	**BLOCK NUMBER**	**DATA**
1 By 0xA0	1 By	1 – 16 By

Table 8. Response to EXTM_AUTH_WRITE Command

ExtMode Response	
Value	**Description**
0x00	Write operation completed successfully. Any other value means error. Length: 1By

Attention: After the data transfer has successfully started, it might still happen that no data is received from NFCC, for example if a card is suddenly removed from the RF field. In such cases internal data exception handlers will be started and appropriate NCI error notification sent to DH.

7.3 External Dependencies

The authentication procedure implemented in **ExtMode** is based on the proprietary “Crypto-1” cipher algorithm, which is used as a 3rd-party SW implementation.

NCIRD API uses the publicly available “**Crapto-1**” implementation which is for example part of the following projects hosted on GitHub:

- MIFARE Classic Offline Cracker "MFOC" [1]
- MIFARE Classic Universal Toolkit "MFCUK"

The exact list of files required is given in the Figure 8.

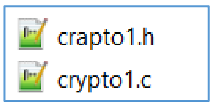

crapto1.h
crypto1.c

Figure 8. Third-Party Crypto-1 Implementation Files

The files simply need to be added to build-environment of the target system / application.

1. NCIRD API SDK uses the Crypto-1 implementation from the MIFARE Classic Offline Cracker "MFOC" project, accessed on November 6th, 2020.

8. Logging System

The internal stack contains a powerful logging-system which is very helpful for system integration and debugging. The logging-system is based on a ring-buffer implementation in RAM with a configurable size. It can be optionally written to a given file once the “**ptxNCIRD_Close_Stack**” API function gets called.

Even though the Logger itself uses only RAM, a lot of entries can impact the system performance. Additionally, the size i.e., the number of logging-entries has direct impact on overall RAM-consumption.

The size itself is configurable via the define “NCIRD_LOG_DEPTH”. Configuring a size of 0 disables the Logger completely.

9. RF- and System-Config Updates

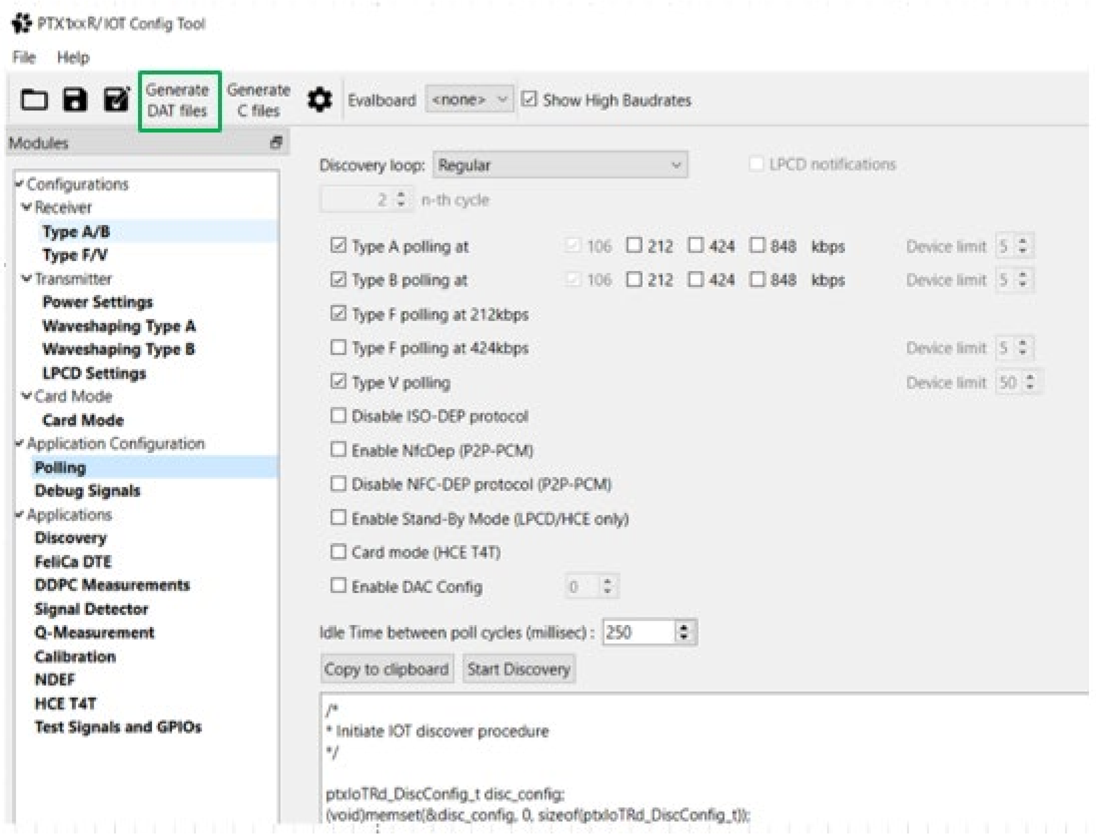
The PTX130R allows to configure various RF- and System-configuration settings which are applied during the initialization phase of the NCI protocol during the execution of a “**NCI_CORE_INIT_CMD**”.

9.1 RF Configuration

The default RF configuration is stored in a binary file called “**NSC_RF_CONFIG.dat**” and is stored inside the NCIRD API SDK in the folder “**FILE_SYSTEM**”. The location of the files can be changed by setting the parameter “**DeviceFsPath**” of the API function “**ptxNCIRD_Init_Stack”.**

The default RF configuration i.e., the binary file itself can be generated by using the “PTX130x IOT Config Tool” from Renesas.

Figure 9 shows a screenshot of the “PTX1xxR IOT Config Tool”. It allows to configure the RF-configuration parameters and to generate the required .dat-files accordingly.



Evalboard: <none> ☑ Show High Baudrates

Discovery loop: Regular ☐ LPCD notifications

2 n-th cycle

☑ Type A polling at ☑ 106 ☐ 212 ☐ 424 ☐ 848 kbps Device limit 5

☑ Type B polling at ☑ 106 ☐ 212 ☐ 424 ☐ 848 kbps Device limit 5

☑ Type F polling at 212kbps

☐ Type F polling at 424kbps Device limit 5

☑ Type V polling Device limit 50

☐ Disable ISO-DEP protocol

☐ Enable NfcDep (P2P-PCM)

☐ Disable NFC-DEP protocol (P2P-PCM)

☐ Enable Stand-By Mode (LPCD/HCE only)

☐ Card mode (HCE T4T)

☐ Enable DAC Config 0

Idle Time between poll cycles (millisec): 250

```
/*
* Initiate IOT discover procedure
*/

ptxIoTRd_DiscConfig_t disc_config;
(void)memset(&disc_config, 0, sizeof(ptxIoTRd_DiscConfig_t));
```

Figure 9. PTX1xxR IOT Config Tool

Attention: The folder "**FILE_SYSTEM**" is the default path where the NCI-stack looks up for the RF (and System) -configuration. If a different path should be used, this can be set by adapting the initialization parameters for the call to "**ptxNCIRd_Init_Stack**".

The SDK contains default parameters for the RF configuration. These parameters need to be adapted for the final target application as the RF configuration is dependent on various factors like actual application requirements, antenna size / form / matching etc.

To update RF configuration of the NCIRD API SDK, click the toolbar button "Generate DAT files" and select the folder "**FILE_SYSTEM**". If a custom folder has been specified using the "**DeviceFsPath**" parameter select that folder instead. Pressing the "Choose" button will replace the binary file called "**NSC_RF_CONFIG.dat**" in the selected folder with one that contains the RF configuration parameters from the "PTX1xxR IOT Config Tool".

9.2 System Configuration

The default system configuration is stored in a binary file called "**NSC_SYS_CONFIG.dat**" and is stored inside the NCIRD API SDK in the folder "**FILE_SYSTEM**". The location of the files can be changed by setting the parameter "**DeviceFsPath**" of the API function "**ptxNCIRD_Init_Stack".**

Configuring the parameters and generating the .dat-file works the same way as described in *RF Configuration*.

10. Proprietary Actions

To enable access and configuration of specific PTX130R system features, there is a set of NCI proprietary control messages provided to the user.

Those proprietary control messages (command, response, and notification) have a TLV-based parameter structure, which defines the actual system function and parameters. Currently supported control messages are listed in the following table.

Table 9. Proprietary Control Messages

PROPR_RUN_SYSTEM_CMD_CMD				
Payload Field	**Length**	**Value/Description**		
TLV structure defining system specific action	m+2 Bytes	ID	1 Byte	Identification of the system specific action to be executed.
		Len	1 Byte	Length of the Val field
		Val	m Bytes	Parameters/Input arguments for the system specific action.

PROPR_RUN_SYSTEM_CMD_RSP		
Payload Field	**Length**	**Value/Description**
Status	1 Byte	PTX_STATUS_OK (0x00) if command syntax is OK and the command is supported. Otherwise, an error code is set. In case of PTX_STATUS_OK, the requested system action will take place and the PTX_RUN_SYSTEM_CMD_NTF notification will follow.

PROPR_RUN_SYSTEM_CMD_NTF				
Payload Field	**Length**	**Value/Description**		
TLV structure containing result of the executed system action	m+2 Bytes	ID	1 Byte	Identification of the system specific action that has been executed.
		Len	1 Byte	Length of the Val field
		Val	m Bytes	Result of the system action execution. First byte is always status (PTX_STATUS_OK or an error code).

Currently supported system actions:

- Type / ID: 0x00 Temperature-Sensor Calibration
- Type / ID: 0x01 Setting Temperature-Sensor Shut-down
- Others are RFU

10.1 Temperature Sensor Calibration

PTX130R features an on-chip temperature sensor that continuously monitors the die temperature. In case the temperature exceeds a configurable threshold, the transmitter is automatically disabled.

To get expected accuracy, temperature sensor requires calibration. This should be done **once for a given PTX130R** and is available to the user via the proprietary PROPR_RUN_SYSTEM_CMD_CMD**.**

Along with sensor calibration, the compensated temperature threshold will be calculated, and the resulting value returned through PROPR_RUN_SYSTEM_CMD_NTF. That value should be then used as the input parameter to Set Temperature Sensor Shut-down proprietary action.

Temperature compensation steps:

- Perform core reset
    - call CORE_RESET_CMD
    - get CORE_RESET_RSP and NTF
- Perform temperature calibration:
    - Set ambient temperature to a desired value e.g. 25°C. Provide this value as **“Tambient”** input parameter**.**
    - Set the value of expected temperature shutdown threshold in **“Tshutdown”** parameter (e.g. 100°C, the value is provided in [1]).
    - Call PROPR_RUN_SYSTEM_CMD_CMD (ID = 0x00) with provided parameters**.**
    - Store the value returned in **PROPR_RUN_SYSTEM_CMD_NTF** for future use.

PROPR_RUN_SYSTEM_CMD_CMD Temperature Calibration TLV			
ID	**Length**	**Value**	
0x00	0x02	[Tambient]	[Tshutdown]

PROPR_RUN_SYSTEM_CMD_NTF Temperature Calibration TLV			
ID	**Length**	**Value**	
0x00	0x02	STATUS	[Tshutdown]

Set shut-down temperature:

- Perform temperature compensation
- Call **PROPR_RUN_SYSTEM_CMD_CMD (ID=0x01)** with the stored value from temperature calibration step

PROPR_RUN_SYSTEM_CMD_CMD Set Temperature Shutdown TLV		
ID	Length	Value
0x01	0x01	[Tshutdown]

PROPR_RUN_SYSTEM_CMD_NTF Set Temperature Shutdown TLV		
ID	**Length**	**Value**
0x01	0x01	STATUS

The set value is then automatically used during system initialization procedure (CORE_INIT).

Attention: Temperature sensor calibration (temperature threshold compensation) must be executed once per PTX130R in controlled environment conditions. Once done, it does not need to be started all over again before NFCC initialization.

11. References

[1] Renesas (formerly Panthronics), *PTX130R Datasheet*.

[2] NFC Forum NFC Controller Interface (NCI) Technical Specification Version 2.1, 2018

[-] NFC Forum

12. Revision History

Revision	Date	Description
1.7	Oct 16, 2023	▪ Completed minor updates; no technical changes were made.
1.6	Oct 5, 2023	▪ Updated the document to the latest template. ▪ Completed minor updates throughout; however, no technical changes were made.
1.5	May 2022	▪ Updated ExtMode description. ▪ Updated Core component description - added T3T polling command. ▪ Added Hal interface selection.
1.4	Feb 2022	▪ Added logging chapter. ▪ Updated RF/System configuration description.
1.3	Nov 2021	▪ Added RF Interface Extensions description. ▪ Added proprietary commands.
1.2	Sep 2021	▪ Added ISO-DEP Presence Check R(NAK) method. ▪ Added NCI Data Messages chapter.
1.1	Jul 2020	▪ Added Linux reference version
1.0	Jan 2020	▪ First version including API description

Corporate Headquarters

TOYOSU FORESIA, 3-2-24 Toyosu,
Koto-ku, Tokyo 135-0061, Japan
www.renesas.com

Contact Information

For further information on a product, technology, the most up-to-date version of a document, or your nearest sales office, please visit:
www.renesas.com/contact/

Trademarks

Renesas and the Renesas logo are trademarks of Renesas Electronics Corporation. All trademarks and registered trademarks are the property of their respective owners.