PTX2xxR/W RUL (Reader Universal Library) SDK (v1.2.1)

Introduction

This document describes the Renesas Reader Universal Library (further referred to as RUL API or RUL SDK) for the following Renesas NFC Reader products:

- Renesas PTX2xxR
- Renesas RRQ35230

Unless specified otherwise, any SDK content (for example. source code, file names, etc.) and / or deliverables (for example, this User Manual and other related documents) may use the combined term "**PTX2xxR/W**" to refer to all Renesas NFC Reader products listed above.

Contents

1. Overview 6
1.1 Audience 7
1.2 Abbreviations and Terminology 7
2. RUL Software Architecture 8
2.1 Architecture Overview 8
2.1.1. Platform Dependence 8
2.1.2. API Function Behavior (Blocking vs. Non-blocking) 9
2.1.3. Communication PTX2xxR/W Hardware – Host 10
2.2 Layer and Component Description 10
2.2.1. Platform Abstraction Layer 10
2.2.2. Core Component Layer 11
2.2.3. Application Layer 12
2.2.4. Demos / Examples and Applications 14
3. RUL API Overview 16
3.1 RUL API Description 16
3.1.1. RUL Initialization and De-initialization 16
3.1.2. Calibrating RUL 18
3.1.3. Configuring RUL 19
3.1.4. Communication RUL Functions 20
3.1.5. Stack Information 29
3.1.6. Card Registry 30
3.1.7. Card Activation and Deactivation 32
3.1.8. PTX2xxR/W GPIOs 34
3.1.9. Extensions 36
3.1.10. Utility and Measurement APIs 37
4. RUL API States 39
4.1 State Descriptions 39
5. RUL SDK Deliverable 44

6. RUL SDK Target System Integration and Porting 45
6.1 Build System 45
6.1.1. Build Requirements 45
6.1.2. Configuring and Building the Stack with CMake 46
6.1.3. Configuring and Building the Stack with Renesas e^2 Studio 48
6.2 Adding Add-On Libraries to the Build System 49
6.2.1. Additional Compile Switches for Add-On Libraries 49
6.3 Hardware Abstraction Layer (HAL) 50
6.3.1. Introduction 50
6.3.2. Hardware Port Configuration 51
6.3.3. Required HAL Driver API 52
6.4 Logging Mechanism (LOG) 57
6.4.1. Introduction 57
6.4.2. LOG Architecture 57
6.4.3. Reference LOG Implementations 59
6.5 Integration Notes and Hints 59
6.5.1. Endianness 59
6.5.2. FTDI DLL-Path 60
6.5.3. FTDI Hardware Pin Connections 60
6.5.4. FTDI Pull-Down Resistors for Standby Mode 60
6.5.5. Raspberry Pi 4 Hardware Pin Configuration 62
6.5.6. Raspberry Pi 4 UART Remapping 62
6.5.7. HAL Driver Default Communication Speeds 62
6.5.8. RRQ35230EB Board 63
6.5.9. Reference SDK Module Flash / RAM Consumption 63
7. Add-On Libraries/APIs 65
7.1 Native Tag API 65
7.1.1. API Architecture 65
7.1.2. Native Tag Type 2 Tag API 66
7.1.3. Native Tag Type 3 Tag API 68
7.1.4. Native Tag Type 4 Tag API 72
7.1.5. Native Tag Type 5 Tag API 77
7.1.6. Native Tag API States 89
7.1.7. Implementation Guidelines/Suggestions 90
7.2 MIFARE Classic® Compatibility (MFCC) 91
7.2.1. API Architecture 91
7.2.2. API Description 92
7.2.3. MFCC API States 97
7.2.4. Implementation Guidelines/Suggestions 98
7.3 NDEF API 100
7.3.1. API Architecture 100
7.3.2. Tag-Specific NDEF Operations 101
7.3.3. Generic NDEF Operations 110
7.3.4. NDEF Message Creator/Parser 113
7.3.5. NDEF API States 151

7.3.6. Implementation Guidelines/Suggestions 152
7.4 Logical Link Control Protocol (LLCP) API 153
7.4.1. API Architecture 153
7.4.2. API Description 153
7.4.3. LLCP API States 160
7.4.4. Implementation Guidelines/Suggestions 161
7.5 Simple NDEF Exchange Protocol (SNEP) API 161
7.5.1. SNEP Operations 161
7.5.2. API Architecture 162
7.5.3. API Description 163
7.5.4. SNEP API States 167
7.5.5. Implementation Guidelines/Suggestions 168
7.6 Transparent Mode API 168
7.6.1. API Architecture 169
7.6.2. API Description 169
7.6.3. Transparent Mode API States 172
7.6.4. Implementation Guidelines/Suggestions 173
7.7 RF Test API 175
7.7.1. API Architecture 175
7.7.2. API Description 176
7.7.3. RF Test API States 177
7.7.4. Implementation Guidelines/Suggestions 177
7.8 POS DTE API 177
7.8.1. API Architecture 178
7.8.2. API Description 178
7.8.3. POS DTE API States 185
7.8.4. Implementation Guidelines/Suggestions 186
7.9 NFC Forum DTA API 187
7.9.1. API Architecture 188
7.9.2. API Description 189
7.9.3. NFC Forum API States 191
7.9.4. Implementation Guidelines/Suggestions 191
7.10 FeliCa DTE API 192
7.10.1. API Architecture 192
7.10.2. API Description 193
7.10.3. FeliCa DTE API States 194
7.10.4. Implementation Guidelines/Suggestions 195
7.11 HCE API 195
7.11.1. API Architecture 196
7.11.2. API Description 196
7.11.3. HCE API States 198
7.11.4. Implementation Guidelines/Suggestions 199
7.12 PSL API 199
7.12.1. API Architecture 199
7.12.2. API Description 200
7.12.3. PSL API States 206

7.12.4. Implementation Guidelines/Suggestions 206
7.13 Wireless Charging API 207
7.13.1. API Architecture 207
7.13.2. General WLC API 207
7.13.3. State Machine WLC API 209
7.13.4. Proprietary WLC API 213
7.13.5. WLC API States 215
7.13.6. Implementation Guidelines/Suggestions 219
8. RF Configuration Updates 220
8.1 Static RF Configuration 220
8.2 Dynamic at Runtime 221
8.3 Platform Specific 221
9. System Features 221
9.1 Power Management 221
9.1.1. Stand-by Mode: NFC RF Discovery 221
9.1.2. Stand-by Mode: API Call 223
9.1.3. Low Power Card Detection (LPCD) 224
9.2 System Errors and Recovery 224
9.2.1. Over-Temperature System Error 224
9.2.2. Current-Limiter System Error 224
9.2.3. Out-of-Memory System Error 224
9.3 System Configuration 225
9.3.1. Static System Configuration 226
9.3.2. Dynamic (at Runtime) System Configuration 226
9.4 Tx-ESD Prevention Calibration 226
10. General Integration Hints 227
10.1 Compliance / Certification Testing 227
11. References 228
11.1 Standards and Regulations 228
12. Revision History 229

Figures

Figure 1. RUL API Software Architecture Overview 8
Figure 2. RUL Stack State Overview 40
Figure 3. Standard RF-Discovery Procedure Sequence 41
Figure 4. HAL Overview 50
Figure 5. HAL Driver RA MCU SPI Reference Implementation Port Configuration Type 51
Figure 6. HAL Driver RA MCU Reference Implementation Port Configuration 52
Figure 7. LOG Component Architecture 57
Figure 8. FTDI-DLL Path 60
Figure 9. FTDI Pull-Down Resistor for Standby Mode 61
Figure 10. Native Tag API Component Architecture 65
Figure 11. Native Tag Type API States 89

Figure 12. MFCC API Component Architecture 91
Figure 13. MFCC API States 97
Figure 14. crapto1.c Changes for Memory Allocation Purposes 98
Figure 15. crapto1.h Changes for Memory Allocation Purposes and Definition for Custom Validation Helper Function 98
Figure 16. crypto1.c Changes for Memory Allocation Purposes and Performance Improvement 98
Figure 17. crapto1.c Changes for Performance Purposes 99
Figure 18. crapto1.c Changes for Custom Validation Helper Function 99
Figure 19. crapto1.c Changes for Performance Purposes 99
Figure 20. crapto1.c Changes for Performance Purposes 99
Figure 21. NDEF API Component Architecture 100
Figure 22. NDEF API States 151
Figure 23. Message API States 151
Figure 24. NDEF Record API States 151
Figure 25. LLCP Module API Component Architecture 153
Figure 26. LLCP API State 160
Figure 27. SnepClient Module API Component Architecture 162
Figure 28. SnepClient API State 167
Figure 29. SnepServer API State 167
Figure 30. Transparent Mode API Component Architecture 169
Figure 31. Transparent Mode API States 172
Figure 32. RF Test API Component Architecture 175
Figure 33. RF Test API states 177
Figure 34. POS DTE API Component Architecture 178
Figure 35. POS DTE API States 185
Figure 36. NFC Forum DTA API Component Architecture 188
Figure 37. NFC Forum DTA API States 191
Figure 38. FeliCa DTE API Component Architecture 192
Figure 39. FeliCa DTE API States 194
Figure 40. HCE API Component Architecture 196
Figure 41. HCE API States 198
Figure 42. PSL API Component Architecture 199
Figure 43. PSL API States 206
Figure 44. WLC API Component Architecture 207
Figure 45. WLC API States 215
Figure 46. WLC State Machine API States 216
Figure 47. PTX2xxR RUL Config Tool Overview 220
Figure 48. RF Discovery Procedure with Standby Mode 222

1. Overview

The Renesas Reader Universal Library SDK (RUL) is an enhanced software package designed to deliver user-friendly and scalable software for NFC designs using the Renesas PTX2xxR/W family of NFC products.

The RUL SDK includes the RUL API and a large collection of optional add-on APIs to support fast integration and implementation of different target applications such as IoT, Payment (POS), NFC Wireless Charging (WLC), Transport and many more using these standard NFC operating modes: Reader, Card Emulation and P2P. The fast integration is supported by various code examples and demo applications included in the SDK which demonstrates the correct usage of the APIs or can serve as a reference for specific target applications.

For more details on the RUL SDK architecture, the RUL API and the optional add-on APIs, refer to the following sections:

- RUL Software Architecture
- RUL API Overview
- RUL API States
- Add-On Libraries/APIs

The RUL SDK is designed to be built/compiled for various target platforms such as the Renesas RA-family of MCUs, Windows and Linux (embedded) systems. The entire RUL SDK code base is implemented using the C (C-99) language which simplifies integration and porting to different target platforms. The necessary software layers which need to be ported are encapsulated and abstracted in a dedicated Hardware Abstraction Layer (HAL). The RUL SDK includes various HAL reference implementations for the Renesas RA-family of MCUs, Windows and Linux. The integration and porting process is supported by an optionally available Logging mechanism which can be used for debugging and monitoring purposes.

For more details on the target integration and porting of the SDK and the SDK package content, refer to the following sections:

- RUL SDK Target System Integration and Porting
- RUL SDK Deliverable

The compile/build process of the RUL SDK is implemented based on CMake (cmake.org) (Windows, Linux) or the Renesas e2 Studio IDE (RA MCU family). All necessary CMake-scripts and e2 Studio project files are also included in the SDK.

For more details on the build process, compile options/flags, etc., refer to the following section:

- Build System

To achieve the best possible NFC RF performance for the desired target application, it may be necessary to adapt the RF settings for the target hardware and environment. The SDK includes default RF-settings for the standard Reader and Wireless Charging Poller EVKs which may need to be overwritten using the *PTX2xxR RUL Config Tool*. Another aspect of performance is system features such as power management.

For more details on the *PTX2xxR RUL Config Tool*, how to update the RF settings and various system features such as Power Management, refer to the following sections:

- RF Configuration Updates
- System Features

1.1 Audience

This document is intended to be used by:

- SW architects
- SW engineers
- SW integrators

1.2 Abbreviations and Terminology

Abbreviations	Terminology
I2C	Inter IC Communication
SPI	Serial Peripheral Interface
UART	Universal Asynchronous Receiver Transmitter
TMR	Timer
RX	Receive
TX	Transmit
GPIO	General Purpose Input Output
HAL	Hardware Abstraction Layer
DRV	Driver
DEV	Device
MCU	Microcontroller Unit
API	Application Programming Interface
NDEF	NFC Data Exchange Format
MFCC	MIFARE Classic® Compatibility
NSC	NFC Soft Controller
NFC	Near Field Communication

2. RUL Software Architecture

2.1 Architecture Overview

The software architecture of the RUL Stack is comprised of multiple layers where each layer contains one or more component(s). A high-level overview of the RUL software architecture is shown in Figure 1 below.

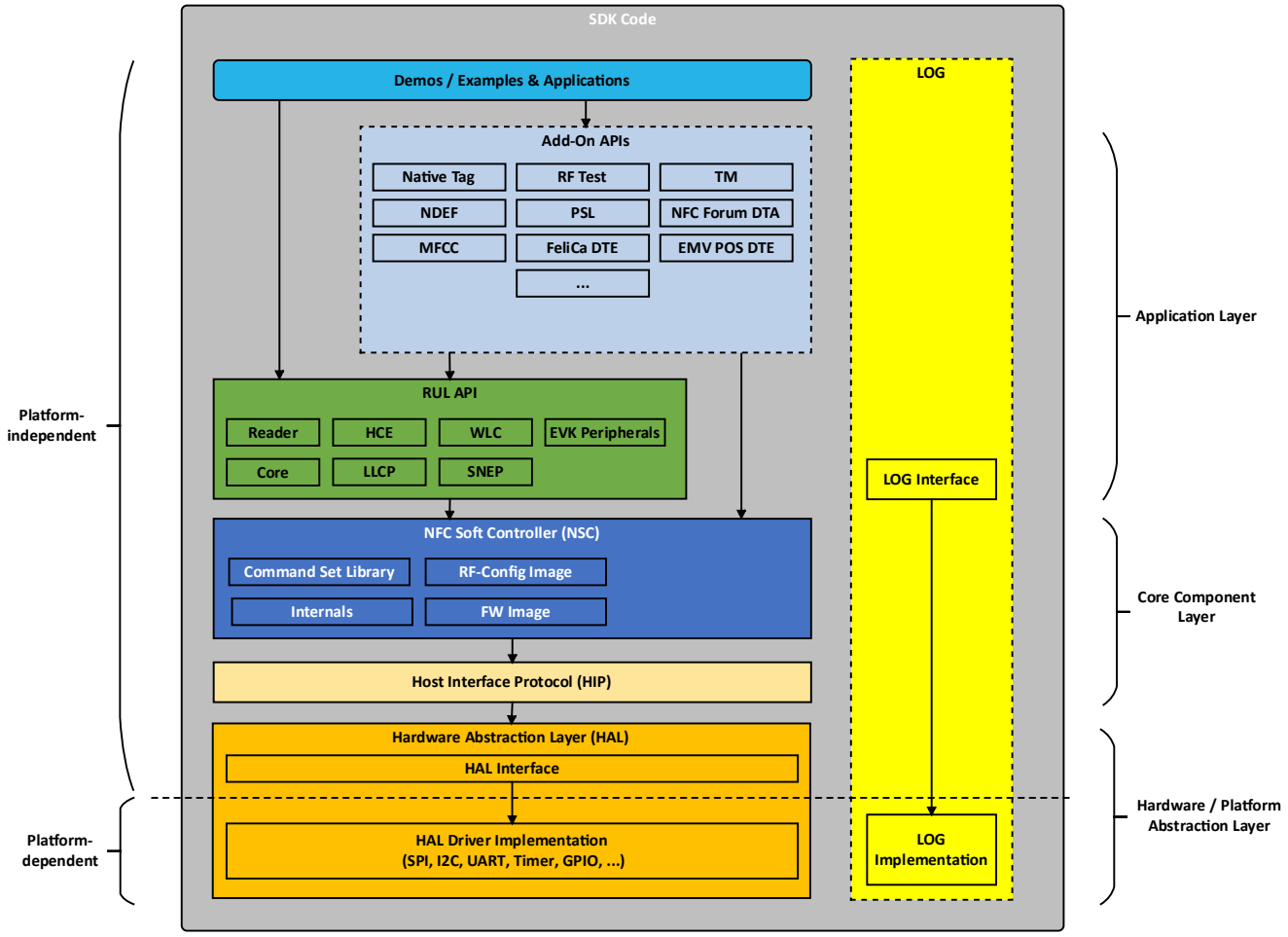

Figure 1. RUL API Software Architecture Overview

2.1.1. Platform Dependence

The entire RUL Stack is implemented in C and is therefore target-platform independent, except for these two components:

- Hardware Abstraction Layer (HAL) and
- LOG (optional)

The RUL SDK contains reference implementations for various platforms such as Renesas RA-family of MCUs, Windows and (embedded) Linux. Due to the large number of possible ways to implement concrete abstracted components, the provided reference implementations serve as examples using a specific set of available platform libraries and drivers. Should the final target application/platform differ from the provided reference implementations or different platform libraries and drivers, the HAL and the LOG component (if used) need to be adapted/ported.

The upper software layers of the RUL Stack work the target-platform independently and do not require or use any operating system features and resources such as Threads, Semaphores, Mutexes, dynamic memory allocation, etc. Despite not needing any of these operating system resources, the RUL Stack can still be integrated/executed in operating system environments like Windows or Linux (embedded) operating systems, but also any embedded RTOS environment such as FreeRTOS.

***Important*:** Depending on the target platform, some of the provided HAL reference implementations may use operating system specific features and resources such as Semaphores, Mutexes, dynamic memory allocation or 3rd party library usages.

2.1.2. API Function Behavior (Blocking vs. Non-blocking)

The API functions of the RUL API and the optional add-on APIs all work synchronously, in other words, in a blocking manner. However, functions that supposedly block the system for a longer or even unknown time (for example, events based on user interaction) are split into Trigger/Start functions and Query Status functions. This is useful and partially required when the RUL Stack is integrated into RTOS environments so as not to block resources and task execution.

Typical examples for such functions are:

- **NFC RF Discovery**
  The RUL API provides a dedicated API function to start the NFC RF Discovery process with given discovery parameters. Once successfully started, the API function returns immediately to the calling level. The status of the RF Discovery process can then be queried by another API function which returns whether something was discovered or not (simplified).
- **POS/EMV Loopback and TRANSAC_X**
  The POS DTE add-on API implements several functions to perform the so called EMV Loopback and EMV TRANSAC_X applications; both used for EMV type approval.

  Due to the specification of both functionalities, these applications are supposed to run endlessly. Both functionalities are implemented in a non-blocking, but re-entrant manner in the POS-DTE API (in other words, the API functions can be called repeatedly) – only internal state-machines are updated.

Some of the API functions implement callback functions to notify upper layers about certain events or to provide mechanisms to cancel/stop an ongoing operation.

Typical examples for such functions are:

- **POS / EMV Interoperability Loopback**
  Besides the standard EMV Loopback application, the POS DTE API also implements the interoperability variant of this application which may be subject during type approval testing.

  Certain external test houses may require visual or auditory notifications about certain triggered events, for example, events such as card discovered, a transaction passed or failed, etc.
- **NFC Forum DTA Execution**
  The NFC Forum DTA add-on API implements a dedicated function to start a given DTA request with the given Pattern Number and test configuration according to the NFC Forum. Due to the complexity of this function, a DTA request is implemented in a blocking manner. However, a callback function can be registered during component initialization allowing the DTA request to be interrupted/stopped, if necessary.

All options and possibilities for all the individual API functions and their containing components are described in later sections of this manual.

2.1.3. Communication PTX2xxR/W Hardware – Host

When the PTX2xxR/W needs to send data to the host running the RUL Stack, it generates an IRQ via one of the physical GPIO-pins. The RUL Stack does not implement platform-dependent IRQ handling involving Interrupt Service Routines (ISRs) but polls the level of the IRQ pin instead. This is implicitly achieved whenever certain RUL API functions are called.

If needed, custom HAL Driver implementations and (see following sections) may implement IRQ handling including ISRs. The usage is up to the desired target integration/porting.

2.2 Layer and Component Description

This section provides a high-level introduction of all architecture layers and individual software components. The RUL API, the add-on APIs as well as components which may require integration and porting effort are described in more detail in the following sections.

2.2.1. Platform Abstraction Layer

2.2.1.1. Hardware Abstraction Layer (HAL)

This layer implements the access to hardware resources such as host-interfaces (for example, SPI, I^2C, UART, etc.), timers, GPIOs, interrupt handling, etc.

The RUL SDK includes HAL reference implementations for the following hardware platforms:

- Renesas RA MCU Family
- Microsoft Windows
- Native or embedded Linux

2.2.1.1.1. HAL Interface

The HAL Interface separates the platform-dependent and platform-independent layers/components by implementing a unique software interface towards the upper software layers. The upper software layers only use this dedicated interface and is therefore agnostic to the actual underlying hardware-blocks, host-interfaces, drivers etc.

2.2.1.1.2. HAL Driver Implementation(s)

HAL drivers implement the minimum required functionality such as sending/receiving bytes via a physical host interface, accessing GPIOs and sensing IRQ-signals.

The RUL SDK includes reference implementations of HAL drivers for various platforms based on Renesas FSP drivers for the RA MCU family, standard Linux drivers and Windows drivers based on virtual COM port or FTDI libraries, etc.

HAL drivers can be implemented for more than one hardware configuration internally. For this purpose, Port Configurations are used. These are statically defined internally used C-arrays, which define the available hardware configurations. How this configuration looks depends on the specific driver implementation. The RUL SDK includes several port configurations for each platform and communication interface reference implementation. One of these configurations always exist and is called "Default". These are used to interact with the PTX2xxR/W NFC hardware. Another use case for a different HAL Driver Port Configuration is to set a specific SPI HAL driver configuration to Mode-0 for the CS-pin, while another may require Mode-1 for the CS-pin, but still using the same underlying HAL Driver Code. This additional abstraction allows to combine different HAL configurations and settings by keeping the implementation effort and potentially memory requirements low.

2.2.1.2. Logging (LOG)

This component implements logging capabilities which can be used for debugging and monitoring purposes. The log output is highly customizable via dedicated user-callback functions allowing to forward the log entries to any desired output. The implementation is based on the third-party logging library “zf_log”.

Except for the core LOG component, the remaining code parts of the LOG component can be completely disabled/excluded from the target build.

2.2.1.2.1. LOG Interface

The LOG Interface is platform-independent and consists of the core LOG component for initialization/de-initialization and user-callback registration as well as various macros to print LOG-messages and buffers/memories using different logging levels.

The following print-macros and logging-levels are supported:

- PTX_LOG_I // Log/Print Message, Logging Level = Information
- PTX_LOG_V // Log/Print Message, Logging Level = Verbose
- PTX_LOG_D // Log/Print Message, Logging Level = Debug
- PTX_LOG_W // Log/Print Message, Logging Level = Warning
- PTX_LOG_E // Log/Print Message, Logging Level = Error
- PTX_LOG_F // Log/Print Message, Logging Level = Fatal
- PTX_LOG_I_MEM // Log/Print Memory, Logging Level = Information
- PTX_LOG_V_MEM // Log/Print Memory, Logging Level = Verbose
- PTX_LOG_D_MEM // Log/Print Memory, Logging Level = Debug
- PTX_LOG_W_MEM // Log/Print Memory, Logging Level = Warning
- PTX_LOG_E_MEM // Log/Print Memory, Logging Level = Error
- PTX_LOG_F_MEM // Log/Print Memory, Logging Level = Fatal

2.2.1.2.2. LOG Implementation(s)

The LOG Implementation component contains the user-specific callback functions to forward Log messages to a desired output.

The RUL SDK contains the following reference LOG implementations:

- Console (Windows and Linux)
- Serial Terminal (Renesas RA MCU Family)
- File (Windows and Linux)

2.2.2. Core Component Layer

2.2.2.1. Host Interface Protocol (HIP)

The HIP component implements the PTX2xxR/W NFC hardware specific:

- Transport mappings for SPI, I^2C and UART, and
- Low-level commands to download the FW image into the code memory, accessing command buffers, reading/writing from/to memory and registers.

2.2.2.2. NFC Soft Controller (NSC)

2.2.2.2.1. Command Set Library

The PTX2xxR/W NFC hardware implements a dedicated on-chip controller which hosts a Firmware/FW image. The PTX2xxR/W FW interacts with the RUL Stack using a Command/Responses/Data/Notifications as Message-types and implements the following functionalities:

- NFC RF Discovery/Polling-loop for NFC Forum-mode, EMV-mode, and ISO-mode including collision-resolution and device activation and deactivation.
- Higher level NFC protocols such as ISO-DEP and NFC-DEP
- RF data exchange including handling of chaining

- Power Management (Stand-by, Low Power Card Detection (LPCD), …)
- …

The Command Set Library component implements all functions from the FW as dedicated API functions towards the upper software layers.

If needed, certain functions such as Host Card Emulation and Wireless Charging can be excluded from the build.

2.2.2.2.2. RF Config Image

The RF settings for the on-chip analog and digital blocks and peripherals are optimized for the PTX2xxR/W Reader and Wireless Charging Poller EBs. As mentioned in the Introduction, it is necessary to update the RF settings for the given target application to achieve the best possible NFC-performance.

The RF settings are stored as a C-array in binary form and can be updated statically via the *PTX2xxR RUL Config Tool* or dynamically at runtime through a dedicated RUL API function call.

2.2.2.2.3. FW Image

The actual FW image is stored as a C-array in binary form.

Important: To ensure proper NFC functionality of the RUL SDK, the content of the **included** FW image *must not* be changed or modified. Any problem with the FW image will be reported by returning the status code **PTX_STATUS_E_GEN_FW_INVALID.**

2.2.2.2.4. Internals

This general block/component contains various internal functionalities such as state machine handling of the RUL Stack and the FW, as well as various system configuration settings.

2.2.3. Application Layer

2.2.3.1. Reader Universal Library (RUL) API Components

The Reader Universal Library (RUL) API is the main component of the RUL SDK and is the main SW interface towards the application. It implements different core functionalities for Reader/Card-Emulation, P2P and Wireless Charging to implement various applications for IoT, POS, NFC WLC-Poller.

If needed, certain functions such as Host Card Emulation and Wireless Charging can be excluded from the build.

2.2.3.1.1. RUL Reader API

Specific set of RUL API functions implementing standard NFC Reader/Writer-Poll mode covering:

- RF Discovery in NFC Forum-mode, EMV-mode and ISO-mode supporting NFC technologies A, B, F and V,
- Card activation and deactivation,
- NFC/RF data exchange,
- …

2.2.3.1.2. RUL Core API

Specific set of additional RUL API functions to query various system and status information, updating various parameters and configurations, handling power management, etc.

2.2.3.1.3. RUL Host Card Emulation (HCE) API

Specific set of RUL API functions implementing standard NFC Host Card Emulation/Listen mode supporting NFC technologies A, B and F. Host Card Emulation functionality is implemented based on events triggered by the remote NFC Reader. An application can query if a new event is available and can process these events as needed.

2.2.3.1.4. RUL Logical Link Control Protocol (LLCP) API

This component implements the Logical Link Control Protocol (LLCP) as specified by the NFC Forum used in NFC P2P mode on top of the NFC-DEP protocol. The LLCP API can be used in Initiator and Target modes.

2.2.3.1.5. RUL Simple NDEF Exchange Protocol (SNEP) API

This component implements the Simple NDEF Exchange Protocol (SNEP) as specified by the NFC Forum used in NFC P2P mode on top of the LLCP protocol to exchange NDEF message between NFC P2P devices. The RUL SNEP API is dependent on the RUL LLCP API.

2.2.3.1.6. RUL Wireless Charging (WLC) API

Specific set of RUL API functions implementing standard NFC Forum Wireless Charging Poller (WLC-P) functionality as specified by the NFC Forum. It also contains generic WLC API functions to create and use custom protocols as well as proprietary Renesas WLC features with compatible Wireless Charging Listeners (WLC-L).

2.2.3.1.7. EVK Peripherals

The EVK Peripherals component implements a set of functions for various external components, such as temperature sensors as well as controlling external power supplies. This component is only needed for the RRQ35230EB Board. The implementation may be re-used for application-specific purposes but can be excluded otherwise.

2.2.3.2. Add-on APIs

All add-on APIs are optional and can be individually included/excluded in a build using compile switches.

The exceptions are:

- Native Tag and NDEF API
  The NDEF API is dependent on the Native Tag API.
- NFC Forum DTA API and most other NFC Forum APIs
  The NFC Forum DTA API is dependent on the following add-on APIs: Native Tag, NDEF, LLCP, SNEP

2.2.3.2.1. Native Tag API

The Native Tag API component implements all the native command sets for the NFC Forum Type Tags 2, 3, 4 and 5 as specified by the NFC Forum.

2.2.3.2.2. NDEF API

The NDEF API component implements all the NDEF mappings and commands (Read/Write/Lock) for the NFC Forum Type Tags 2, 3, 4 and 5 as specified by the NFC Forum. Besides that, this component also contains NDEF record and message modules to create and parse NDEF records and messages such as Text, URI, Device Information, Wireless Charging, etc.

2.2.3.2.3. MFCC API

The MIFARE Classic® API component implements the standard MIFARE Classic entirely in software, in other words, without additional HW support for the Crypto-1 implementation. The Crypto-1 implementation is abstracted in the RUL SDK and needs to be implemented by the target application integrator. It is possible to re-use one of the many publicly available third-party libraries.

This component belongs also to the Native Tag API.

Note: MIFARE Classic® and MIFARE® are registered trademarks of NXP Semiconductors.

2.2.3.2.4. RF Test API

The RF Test API implements various RF-related tests and sequence-generators such as Carrier/Field On, PRBS 9/15.

2.2.3.2.5. PSL API

The Product Support Library (PSL) API implements various system features such as RSSI measurement, Antenna-Q measurement, exposing various test-signals via GPIOs, etc.

2.2.3.2.6. FeliCa DTE API

The FeliCa DTE API component implements various tests such as the “FeliCa Reader/Writer Digital Protocol Requirements” and the “FeliCa RF Performance” tests.

2.2.3.2.7. Transparent Mode (TM) API

The Transparent Mode (TM) API implements low-level NFC access to the PTX2xxR/W. This allows an application to implement a variety of low-level NFC frames (for example, bit-oriented frames, frames with/without CRC or parity, etc.) in Reader mode to support custom implementations or to implement proprietary protocols which are not supported by default in the RUL Stack.

2.2.3.2.8. NFC Forum DTA API

The NFC Forum DTA API implements the functionality of the “Device Test Application” specification from the NFC Forum. The API exposes various functions to execute a specific test scenario with given test parameters and pattern-numbers.

Note: The NFC Forum DTA API can only be used to execute a specific test scenario as specified by the NFC Forum. It does not implement the actual testcases.

2.2.3.2.9. EMV POS DTE API

The EMV POS DTE implements the functionality of the “Device Test Environment” specification from EMVCo and the EMV Loopback functionality in standard or interoperability mode.

2.2.4. Demos / Examples and Applications

The RUL SDK includes various demos/code examples and full applications which may be (partially) included in a target application or simply used as reference.

The current RUL SDK version contains:

- **NFC Forum Basic Discovery**
  Demonstrates card/tag detection for all supported RF technologies and protocols according to NFC Forum and exemplary raw data exchange for every detected card/tag.
- **NFC Forum Device Test Application**
  Demonstrates the usage of the NFC Forum Device Test Application by trying different test patterns.
- **NFC Forum Native Tag**
  Similar to NFC Forum Basic Discovery but using the commands from the Native Tag API for the individual NFC Forum Type Tags to replace the raw data exchanges.
- **NFC Forum NDEF**
  Similar to NFC Forum Basic Discovery but only using the NDEF API to create/parse and to read/write NDEF messages from/to NFC Forum Type Tags.
- **Host Card Emulation Type 4 Tag**
  Demonstrates emulation of an NFC Forum Type 4 Tag in Host Card Emulation mode based on RF technologies A and B. Emulated Tag stores a URL which is readable by an NFC-enabled mobile phone (or any other NDEF-capable NFC Reader).

Host Card Emulation Type 3 Tag
Demonstrates emulation of an NFC Forum Type 3 Tag in Host Card Emulation mode based on RF technology F.

Emulated Tag stores a URL which is readable by an NFC-enabled mobile phone (or any other NDEF-capable NFC Reader).

- **POS/EMV Reader**
  Demonstrates how a typical EMV L1 application can be implemented using the standard RUL API including card detection, collision detection, data exchange and invoking the EMV removal procedure.
- **POS/EMV Basic Commands**
  Demonstrates how to use the low-level commands of the POS DTE API by implementing a very simple card detection for Type A and B cards and exchanging WUPA/WUPB/RATS and ATTRIB commands.
- **POS/EMV Loopback Application**
  Demonstrates how to execute the standard or interoperability EMV Loopback application based on RF technologies A and B using the POS DTE API.
- **POS/EMV TRANSAC_X Functionality**
  Demonstrates how to execute the TRANSAC_A or TRANSAC_B functionality using the POS DTE API.
- **EMV PICC Digital Device Application**
  Demonstrates the use of the EMV Digital Device PICC application using the HCE add-on API. The application is used for compliance / type-approval testing.
- **Felica DTE Performance Tests**
  Demonstrates how to execute FelicaDTE Performance Tests using the FeliCaDTE add-on API.
- **Felica DTE Protocol Tests**
  Demonstrates how to execute FelicaDTE Protocol Tests using the FeliCaDTE add-on API.
- **ISO Loopback**
  Demonstrates the use of the ISO Loopback application using the Reader Universal Library (RUL) and the HCE add-on API. The application is used for compliance / type-approval testing.
- **Mifare Classic Compatibility**
  Demonstrates the setup and usage of the MFCC API to communicate with MFCC cards
- **RF Test**
  Demonstrates how to trigger various test scenarios such as enabling a continuous carrier/field signal or to continuously generate PRBS9/15 sequences. These tests may be required during EMC testing in an external test house.
- **Product Support Library**
  Demonstrates the usage of the Product Support Library (PSL) API, which is designed to assist customers in seamlessly integrating the PTX2xxR/W into the final target application, by performing several helpful measurements.
- **SNEP Client**
  Demonstrates the implementation of a Simple NDEF Exchange Protocol (SNEP) client that performs a PUT request to a SNEP server.
- **SNEP Server**
  Demonstrates the usage of a Simple NDEF Exchange Protocol (SNEP) server that listens for and processes incoming requests from SNEP clients.

- **Transparent Mode**
  Demonstrates how to implement a simple polling loop using the low-level NFC commands for the RF technologies A, B, F and V in Reader-mode/Poller-mode using the Transparent Mode API. This example also demonstrates how to send bit-oriented frames as needed during Type A activation/anti-collision as well as reception of multiple card-responses during Type F activation/anti-collision.
- **Logging**
  A stand-alone example only using the LOG component to demonstrate how to use the various logging macros and logging levels.
- **Wireless Charging**
  Demonstrated how to use the Wireless Charging (WLC) API to charge other WLC listener NFC devices.

3. RUL API Overview

The Reader Universal Library (RUL) is the main API for the contactless NFC world. It allows the user to configure the PTX2xxR/W device and interface with tags, devices, etc. supporting NFC protocols.

The component contains all the necessary functions to communicate with the PTX2xxR/W. The functions are exposed to the upper layers, the user application. RUL provides functions to:

- Initialize RUL Stack
- Initialize the PTX2xxR/W chip
- Discover cards according to NFC Forum, EMV or ISO polling scheme
- Activate cards
- Retrieve card details like technical and activation parameters
- Exchange RF data and RF bitstreams
- Stop RF communication
- Access to Card Registry
- Update RF and System configuration parameters at runtime
- Extension APIs and custom APIs

The basic RUL API is capable to communicate with tags in reader mode, where the API configures the device to read from, and write to a tag. It is also possible to configure RUL to function as a listener and do host card emulation or LLCP/SNEP. However, it is advised to use the already existing add-on libraries for such a purpose.

3.1 RUL API Description

This section covers the basic RUL API. Basic RUL API contains all essential functions to create a working reader. The following subsections describe the functions in detail.

3.1.1. RUL Initialization and De-initialization

This section describes the initialization and De-initialization routines for the RUL.

3.1.1.1. ptx_RUL_Init

This function initializes the RUL Stack. It allocates the required buffers for RUL and initializes its subcomponents which are required to interact with the PTX2xxR/W.

Declaration	`ptx_Status_t` **`ptx_RUL_Init`** `(` **`ptx_RUL_t`** `*rul,` **`ptx_RUL_InitParams_t`** `*initParams);`	
Description	Initialization of the RUL component.	
Parameters	rul	Pointer to allocated RUL component
	initParams	Pointer to initialization parameter structure
Return Value	Status, indicating whether the operation was successful.	

3.1.1.2. ptx_RUL_InitHardware

This function initializes the PTX2xxR/W chip. The first function waits until the PTX2xxR/W sends a reset notification. The reset notification indicates that the PTX2xxR/W has successfully reset itself. The next function downloads the static FW image (if required). After a successful verification of the FW, the function starts transmitting the system configuration to the PTX2xxR/W. Finally, the RF configuration is transmitted to the PTX2xxR/W.

Declaration	`ptx_Status_t` **`ptx_RUL_InitHardware`** `(` **`ptx_RUL_t`** `*rul);`	
Description	Initialization of the PTX2xxR/W Hardware.	
Parameters	rul	Pointer to allocated RUL component
Return Value	Status, indicating whether the operation was successful.	

3.1.1.3. ptx_RUL_Reset

This function resets the PTX2xxR/W. A reset can either be a hardware or a soft reset. A soft reset sends a reset command to the PTX2xxR/W which resets its internal states. A hardware reset toggles a hardware pin and resets the PTX2xxR/W. The hardware pin is defined in **PTX_NSC_RESET_PIN**.

Declaration	`ptx_Status_t` **`ptx_RUL_Reset`** `(` **`ptx_RUL_t`** `*rul` **`ptx_RUL_ResetTypes_t`** `resetType);`	
Description	Reset of the PTX2xxR/W	
Parameters	rul	Pointer to allocated RUL component
	resetType	Enum indicating the reset type
Return Value	Status, indicating whether the operation was successful.	

3.1.1.4. ptx_RUL_DeInit

This function deinitializes the RUL Stack. All previously acquired buffers are released and subcomponents are deinitialized.

Declaration	`ptx_Status_t` **`ptx_RUL_DeInit`** (**`ptx_RUL_t`** `*rul);`	
Description	De-initialization of the RUL-component.	
Parameters	rul	Pointer to allocated RUL component
Return Value	Status, indicating whether the operation was successful.	

3.1.2. Calibrating RUL

Certain parts of the PTX2xxR/W HW and/or FW requires an initial calibration depending on the target application. This section describes the API functions used for such configurations.

3.1.2.1. ptx_RUL_CalibrateTxEsdPreventionThreshold

This function executes the calibration routine to measure the Tx-ESD prevention threshold through RSSI measurements. This calibration needs to be done during design-in phase to determine the threshold at which the output power must be limited to prevent Tx-ESD clamping during operation.

The results are applicable to all devices using the same antenna system configuration and it is therefore not required to perform per-device calibration!

The usage of the resulting threshold value is described in 9.4 Tx-ESD Prevention Calibration.

Note: Calling this calibration routine is only required if the RUL-SDK is used for WLC Poller applications.

Declaration	`ptx_Status_t` **`ptx_RUL_CalibrateTxEsdPreventionThreshold`** (**`ptx_RUL_t`** `*rul,` **`uint8_t`** `*results,` **`uint8_t`** `resultsLength);`	
Description	Performs calibration routine to measure Tx-ESD prevention threshold.	
Parameters	rul	Pointer to allocated RUL component
	results	Pointer to an array, which stores all measurements of the calibration routine.
	resultsLength	Length of the array to store the calibration measurements. Attention: When calling the function, this parameter is interpreted as input value and must be set to the actual length of the passed array “results”.
Return Value	Status, indicating whether the operation was successful.	

3.1.3. Configuring RUL

Sometimes it may be helpful to configure the RUL differently, update the RF configuration, or use the GPIO pins provided by the PTX2xxR/W. This section describes the API functions used for such configurations.

3.1.3.1. ptx_RUL_UpdateHardwareConfiguration

This function allows to update the RF and System configuration. Both RF and System configurations can be updated at the same time, but it is not necessary as the function can be called multiple times.

Declaration	`ptx_Status_t` **`ptx_RUL_UpdateHardwareConfiguration`** `(` **`ptx_RUL_t`** `*rul,` **`uint32_t`** `nrConfigs,` **`ptx_RUL_HWConfig_t`**`* configs);`	
Description	Update of the RUL Hardware Configuration	
Parameters	rul	Pointer to allocated RUL component
	nrConfigs	Number of configurations passed in configs
	configs	Data structure holding the configuration. The number of configurations is indicated in the nrConfigs parameter.
Return Value	Status, indicating whether the operation was successful.	

3.1.3.2. ptx_RUL_RoutingTableConfig

This function allows the user to config the Routing Table in Listen Mode.

Important: The current implementation of this API function (valid up to and including RUL SDK v1.2.0) is a prototype implementation and shall not be actively used at Application-level. The current implementation automatically forwards all incoming RF-traffic from a remote Reader-device automatically to the Host when operating in Card-Emulation / Listen mode. The API function itself is subject to change in a future SDK release.

Declaration	`ptx_Status_t` **`ptx_RUL_RoutingTableConfig`** `(` **`ptx_RUL_t`** `*rul,` **`uint32_t`** `length,` **`ptx_RUL_RoutingList_t`** `*rt_list);`	
Description	Set a Routing Table List.	
Parameters	rul	Pointer to allocated RUL component
	length	Routing table entries to be set
	rt_list	Data structure holding the routing table configuration. The number of entries is indicated in the length parameter.
Return Value	Status, indicating whether the operation was successful.	

3.1.4. Communication RUL Functions

3.1.4.1. ptx_RUL_InitiateDiscovery

This function starts the RF discovery procedure after a prior initialization of the PTX2xxR/W. It supports various settings and modes to configure the behavior of the procedure. The system can be configured to poll for cards, or to listen for remote devices/Readers, as well as a combination of both. If no configuration is passed, the previous configuration will be used. This is only possible when "**ptx_RUL_InitiateDiscovery**" has been called with valid parameters before.

Declaration	`ptx_Status_t` **`ptx_RUL_InitiateDiscovery`** `(` **`ptx_RUL_t`** `*rul,` **`ptx_RUL_DiscoverConfig_t`** `*cfg);`	
Description	Initializes the RF discovery using the configuration provided, or the previous configuration, if no configuration was provided.	
Parameters	rul	Pointer to allocated RUL component
	cfg	Pointer to the configuration structure
Return Value	Status, indicating whether the operation was successful.	

3.1.4.1.1. ptx_RUL_DiscoverConfig_t Parameters

General Configurations

Name	Data Type	Description
PollMode	ptx_RUL_PollMode_t	This option describes the configuration of the poll mode. The following options are allowed: ▪ ptx_RUL_PollMode_Disabled – Disables polling ▪ ptx_RUL_PollMode_NfcForum – Enables NFC Forum Polling ▪ ptx_RUL_PollMode_Emvco – Enables EMVCO polling ▪ ptx_RUL_PollMode_Iso – Enables ISO polling ▪ ptx_RUL_PollMode_ContinuousWave – Enables the field, does not poll at all
GuardTime	uint8_t	This option allows to define an additional guard time in milliseconds after the RF field has been activated and before the first RF command is sent.
EnableDDPC	uint8_t	Enables the Dynamic Digital Power Control (DDPC) ▪ 0 = default value, DDPC is disabled ▪ 1 = DDPC enabled
EnableLPCD	uint8_t	Enables Low Power Card Detection (LPCD) ▪ 0 = default value, LPCD is disabled, and the regular RF discovery cycle is entered ▪ 1 = LPCD is used in every discovery cycle ▪ 2 = 255 – regular discovery is used in every 2–255 discovery cycles, while LPCD is used in every other cycle LPCD is not supported when IdleTime is set to 0.
EnableLPCDNtf	uint8_t	Enable Notifications whenever LPCD gets triggered. ▪ 0 = LPCD Ntf disabled ▪ 1 = LPCD Ntf enabled
StandbyCfg	ptx_RUL_StandbyCfg_t	Standby and wake-up configuration. 0 = disable configuration parameter 1 = enable configuration parameter ▪ EnableWuByTimer – Enable wake-up by timer enable, uint8_t ▪ EnableWuByLpfd – Enable wake-up by LPFD (Low Power Field Detect) enable, uint8_t ▪ EnableWuByHif – Enable wake-up by host interface enable, uint8_t
IdleTime	uint32_t	Configures the idle time between two consecutive discovery cycles in 1 ms units. If IdleTime is set to 0, LPCD cannot be enabled.

Poll Mode Configurations

Name	Datatype	Description
EnablePolling	uint8_t	Enables the polling mode. If polling is not enabled, the PTX2xxR/W will not start polling for any technologies, even if they are configured. ▪ 0 = default, polling is disabled ▪ 1 = polling is enabled
PollTypeA	uint8_t	Enables polling for Type A cards. ▪ 0 = Polling for Type A cards is disabled ▪ 1 = Polling for Type A cards is enabled
PollTypeAPollFrequency	uint8_t	Determines the polling frequency for Type A cards. ▪ 0 = Do not poll for Type A technology ▪ 1 = Poll for Type A technology in every discovery cycle ▪ 2–10 = Poll for Type A technology every 2 through10 cycles
PollTypeADeviceLimit	uint8_t	Configure the maximum Type A cards that can be detected by the PTX2xxR/W. ▪ 0 = Default value, the limit is set to a preconfigured limit by the RUL automatically ▪ 1–255 = Number of Type A cards to be detected during discovery
PollTypeABailOut	uint8_t	Configures the discovery loop to bail out after NFC-A technology is found. ▪ 0 = No Bail out when NFC-A technology is found ▪ 1 = Bail out when NFC-A technology is found
PollTypeB	uint8_t	Enables polling for Type B cards. ▪ 0 = Polling for Type B cards is disabled ▪ 1 = Polling for Type B cards is enabled
PollTypeBPollFrequency	uint8_t	Determines the polling frequency for Type B cards. ▪ 0 = Do not poll for Type B technology ▪ 1 = Poll for Type B technology in every discovery cycle ▪ 2–10 = Poll for Type B technology every 2 through 10 cycles
PollTypeBDeviceLimit	uint8_t	Configure the maximum Type B cards that can be detected by the PTX2xxR/W. ▪ 0 = Default value, the limit is set to a preconfigured limit by the RUL automatically ▪ 1–255 = Number of Type B cards to be detected during discovery
PollTypeBBailOut	uint8_t	Configures the discovery loop to bail out after NFC-A or NFC-B technology is found. ▪ 0 = No Bail out when NFC-A, or NFC-B technology is found ▪ 1 = Bail out when NFC-A, or NFC-B technology is found
PollTypeF212	uint8_t	Enables polling for Type F cards with a speed of 212kbps. ▪ 0 = Polling for Type F cards is disabled ▪ 1 = Polling for Type F cards is enabled
PollTypeF424	uint8_t	Enables polling for Type F cards with a speed of 424kbps. ▪ 0 = Polling for Type F cards is disabled ▪ 1 = Polling for Type F cards is enabled

Name	Datatype	Description
PollTypeFPollFrequency	uint8_t	Determines the polling frequency for Type F cards. ▪ 0 = Do not poll for Type F technology ▪ 1 = Poll for Type F technology in every discovery cycle ▪ 2–10 = Poll for Type F technology every 2 through 10 cycles
PollTypeFDeviceLimit	uint8_t	Configure the maximum Type F cards that can be detected by the PTX2xxR/W. ▪ 0 = Default value, the limit is set to a preconfigured limit by the RUL automatically ▪ 1–255 = Number of Type F cards to be detected during discovery
PollTypeFBailOut	uint8_t	Configures the discovery loop to bail out after NFC-A, NFC-B or NFC-F technology is found. ▪ 0 = No Bail out when NFC-A, NFC-B or NFC-F technology is found ▪ 1 = Bail out when NFC-A, NFC-B or NFC-F technology is found
PollTypeV	uint8_t	Enables polling for Type V cards. ▪ 0 = Polling for Type V cards is disabled ▪ 1 = Polling for Type V cards is enabled
PollTypeVPollFrequency	uint8_t	Determines the polling frequency for Type B cards. ▪ 0 = Do not poll for Type V technology ▪ 1 = Poll for Type V technology in every discovery cycle ▪ 2–10 = Poll for Type V technology every 2 through 10 cycles
PollTypeVDeviceLimit	uint8_t	Configure the maximum Type V cards that can be detected by the PTX2xxR/W. ▪ 0 = Default value, the limit is set to a preconfigured limit by the RUL automatically ▪ 1–255 = Number of Type V cards to be detected during discovery
DisablePollIsoDep	uint8_t	Disables the auto usage of the poll mode ISO-DEP protocol. ▪ 0 = default, ISO DEP poll mode is enabled ▪ 1 = ISO DEP poll mode is disabled
DisableNfcDepProtocol	uint8_t	Disables the auto usage of the poll mode NFC-DEP protocol. ▪ 0 = default, NFC DEP poll mode is enabled ▪ 1 = NFC DEP poll mode is disabled
PollIsoDepHbr	ptx_RUL_PollIsoDepHBR_t	Enables Higher Bit Rates (HBR) for the ISO-DEP protocol. ▪ ptx_RUL_PollIsoDepHBR_Maximum – Default setting. Maximum possible bit-rate ▪ ptx_RUL_PollIsoDepHBR_Standard – Standard bit-rate (106kbps) ▪ ptx_RUL_PollIsoDepHBR_212 – Higher bit-rates, up to 212kbps may be used ▪ ptx_RUL_PollIsoDepHBR_424 – Higher bit-rates, up to 424kbps may be used ▪ ptx_RUL_PollIsoDepHBR_848 – Higher bit-rates, up to 848kbps may be used ▪ ptx_RUL_PollIsoDepHBR_UndefinedSafeguard

Name	Datatype	Description
PollNfcDepHbr	ptx_RUL_PollNfcDepHBR_t	Enables Higher Bit Rates (HBR) for the NFC-DEP protocol. ▪ ptx_RUL_PollNfcDepHBR_Maximum – Default setting. Maximum possible bit-rate should be used ▪ ptx_RUL_PollNfcDepHBR_Standard – Activation Bit-rate should be used ▪ ptx_RUL_PollNfcDepHBR_212 – Higher bit-rates, up to 212kbps may be used ▪ ptx_RUL_PollNfcDepHBR_424 – Higher bit-rates, up to 424kbps may be used ▪ ptx_RUL_PollNfcDepHBR_Undefined – Safeguard
PollNfcDepAtrRes	uint8_t	Devices in Peer-to-Peer Mode use the NFC-DEP to exchange information. PollNfcDepAtrRes is used by the Initiator to activate a Target.
PollNfcDepAtrResLen	uint8_t	Length of PollNfcDepAtrRes

Listen Mode Configurations

Name	Datatype	Description
ListenMode	uint8_t	Activates listen mode. If listen mode is not enabled, then the configuration for different technologies is ignored. ▪ 0 = default, Listen Mode is disabled. ▪ 1 = Listen Mode is enabled ▪ 3 = Listen Mode is enabled, and the RF-Field notification is enabled. ***Important***: If listen mode is enabled, RF Field notifications per default are not received. This may be crucial for some applications. Therefore, enable listen mode with RF-Field notifications.
ListenTypeA	uint8_t	Enables listening for Type A technology. ▪ 0 = Polling for Type A technology is disabled ▪ 1 = Polling for Type A technology is enabled
ListenTypeANfcId	uint8_t	Devices in listen mode may exchange a unique ID named NFC-ID for Type-A. The NFC-ID may consist of 4, 7, or 10 bytes to be valid.
ListenTypeANfcIdLen	uin8_t	Length of ListenTypeANfcId. ▪ 0 = default, Random NFC-ID is generated ▪ 4 = The NFC-ID is 4 bytes long ▪ 7 = The NFC-ID is 7 bytes long ▪ 10 = The NFC-ID is 10 bytes long
ListenTypeASensRes	uint8_t	Devices in listen mode may exchange information. ListenTypeASensRes defines the response to a SENS_REQ and must be 2 bytes long.
ListenTypeASensResLen	uint8_t	Length of ListenTypeASensRes. If this parameter is 0, ListenTypeASensRes is ignored.
ListenTypeB	uint8_t	Enables listening for Type B technology. ▪ 0 = Polling for Type B technology is disabled ▪ 1 = Polling for Type B technology is enabled
ListenTypeBNfcId	uint8_t	Devices in listen mode may exchange a unique ID named NFC-ID. The NFC-ID must be 4 bytes long.

Name	Datatype	Description
LsitenTypeBNfcIdLen	uint8_t	Length of the ListenTypeBNfcId. If this parameter is 0, LsitenTypeBNfcId is ignored.
ListenTypeBAppInfo	uint8_t	Devices in listen mode may exchange information. ListenTypeBAppInfo defines the AppInfo contained in SENSB_RES and must be 4 byte long.
ListenTypeBAppInfoLen	uint8_t	Length of the ListenTypeBAppInfoLen. If this parameter is 0, ListenTypeBAppInfo is ignored.
ListenTypeBProtInfo	uint8_t	Devices in listen mode may exchange information. ListenTypeBProtInfo defines the ProtInfo contained in SENSB_RES and must be 3 bytes long.
ListenTypeBProtInfoLen	uint8_t	Length of the ListenTypeBProtInfo. If this parameter is set to 0, ListenTypeBProtInfo is ignored.
ListenTypeF212	uint8_t	Enables listening for Type F212 technology. ▪ 0 = Polling for Type F212 technology is disabled ▪ 1 = Polling for Type F212 technology is enabled
ListenTypeF424	uint8_t	Enables listening for Type F424 technology. ▪ 0 = Polling for Type F424 technology is disabled ▪ 1 = Polling for Type F424 technology is enabled
ListenTypeFNfcId	uint8_t	Devices in listen mode may exchange a unique ID named NFC-ID. The NFC-ID must be 8 bytes long.
ListenTypeFNfcIdLen	uint8_t	Length of ListenTypeFNfcId. If this parameter is set to 0, ListenTypeFNfcId is ignored.
ListenTypeFRequestData	uint8_t	Devices in listen mode may exchange information. ListenTypeFRequestData defines the Request Data contained in SENSF_RES and must be 2 bytes long
ListenTypeFRequestDataLen	uint8_t	Length of ListenTypeFRequestData. If this parameter is set to 0, ListenTypeFRequestData is ignored.
ListenIsoDep	uint8_t	Activates listening for ISO-DEP protocol. ▪ 0 = default, Listening for ISO-DEP is deactivated ▪ 1 = Listening for ISO-DEP is activated, and the device listens for T4AT ▪ 2 = Listening for ISO-DEP is activated, and the device listens for T4BT ▪ 3 = Listening for ISO-DEP is activated, and the device listens for both T4AT, and T4BT.
ListenIsoDepAts	uint8_t	Devices in listen mode use ISO-DEP to exchange information. ListenIsoDepAts defines the response to Request for Answer to Select (RATS) command. This is used for the Type 4A Tag platform
ListenIsoDepAtsLen	uint8_t	Length of the ListenIsoDepAts.
ListenIsoDepAttribRes	uint8_t	Devices in listen mode use ISO-DEP to exchange information. ListenIsoDepAttribRes defines the response to ATTRIB command. This is used for the Type 4 B Tag platform.
ListenIsoDepAttribResLen	uint8_t	Length of the ListenIsoDepAttribRes.

Name	Datatype	Description
ListenNfcDep	uint8_t	Activates listening for NFC-DEP protocol. ▪ 0 – default, Listening for NFC-DEP is deactivated ▪ 1 = Listening for NFC-DEP is activated, and the device listens for NFC-A ▪ 2 = Listening for NFC-DEP is activated, and the device listens for NFC-F ▪ 3 = Listening for NFC-DEP is activated, and the device listens for both NFC-A, and NFC-F.
ListenNfcDepAtrRes	uint8_t	Devices in Peer-to-Peer Mode use the NFC-DEP to exchange information. ListenNfcDepAtrRes defines the response to an Attribute Request Command (ATR_REQ) sent by the Initiator.
ListenNfcDepAtrResLen	uint8_t	Length of ListenNfcDepAtrRes

3.1.4.2. ptx_RUL_Exchange

This function exchanges data with the endpoint (card or initiator). The function requires to pass a transmit buffer plus information about the size of the buffer to transmit data and a receive buffer plus information about the receive buffers size. If a buffer is not provided, the function will not execute the function the buffer is required for. This means that if you do not provide a transmit buffer the function will not send any data and only receive data. If no receive buffer is provided, then the function will not try to receive data from the device.

Declaration	`ptx_Status_t ptx_RUL_Exchange ( ptx_RUL_t *rul, uint8_t *tx, uint32_t txLen, uint8_t *rx, uint32_t *rxLen, uint32_t msTimeout);`	
Description	Performs an exchange of data with the RF endpoint, depending on the buffer(s) provided. If a transmit buffer is provided, data will be sent to the endpoint. If a receive buffer is provided, data will be received from the endpoint.	
Parameters	rul	Pointer to allocated RUL component.
	tx	Pointer to the buffer to be transmitted.
	txLen	Length of the transmit buffer tx.
	rx	Pointer to the receive buffer.
	rxLen	Pointer containing the length of the receive buffer rx. After a successful execution of the pointer contains the value of bytes received in the receive buffer rx.
	msTimeout	Timeout of the function in ms.
Return Value	Status, indicating whether the operation was successful.	

3.1.4.3. ptx_RUL_Send

The send function can be used instead of the exchange function to send data to an endpoint. It sends the data provided in the buffer.

Declaration	`ptx_Status_t` **`ptx_RUL_Send`** (**`ptx_RUL_t`** *rul, **`uint8_t`** *tx, **`uint32_t`** txLen);	
Description	Works similarly to ptx_RUL_Exchange but only transmits data to the RF endpoint.	
Parameters	rul	Pointer to allocated RUL component.
	tx	Pointer to the buffer to be transmitted.
	txLen	Length of the transmit buffer tx.
Return Value	Status, indicating whether the operation was successful.	

3.1.4.4. ptx_RUL_Receive

This function receives data from an endpoint if data is available.

Declaration	`ptx_Status_t` **`ptx_RUL_Receive`** (**`ptx_RUL_t`** *rul, **`uint8_t`** *rx, **`uint32_t`** *rxLen, **`uint32_t`** msTimeout);	
Description	Works similarly to ptx_RUL_Exchange but only receives data from the RF endpoint.	
Parameters	rul	Pointer to allocated RUL component.
	rx	Pointer to the receive buffer.
	rxLen	Pointer containing the length of the receive buffer rx. After a successful execution of the pointer contains the value of bytes received in the receive buffer rx.
	msTimeout	Timeout of the function in ms.
Return Value	Status, indicating whether the operation was successful.	

3.1.4.5. ptx_RUL_BitsExchangeMode

Enables or disables the bits exchange mode. For some use cases the last byte may only contain up to seven bits. This function enables the bits exchange mode so that the PTX2xxR/W knows that it needs to effectively manage the last byte.

Declaration	`ptx_Status_t` **`ptx_RUL_BitsExchangeMode`** (**`ptx_RUL_t`** *rul, **`ptx_RUL_BitsExchangeMode_t`** enable)	
Description	Enables or disables the bits exchange mode.	
Parameters	rul	Pointer to allocated RUL component.
	enable	Enables/Disables the bits exchange mode.
Return Value	Status, indicating whether the operation was successful.	

3.1.4.6. ptx_RUL_BitsExchange

Exchanges a raw RF bitstream. The last byte may be fragmented, according to the parameters, and only those bits are then transmitted.

Declaration	`ptx_Status_t ptx_RUL_BitsExchange ( ptx_RUL_t *rul, ptx_RUL_BitsExchangeParams_t *txParams, ptx_RUL_BitsExchangeParams_t *rxParams, uint32_t *numBits, uint32_t msTimeout))`	
Description	Exchanges a raw RF bitstream.	
Parameters	rul	Pointer to allocated RUL component.
	txParams	Pointer to Tx parameters.
	rxParams	Pointer to Rx parameters.
	numBits	Total number of bits processed.
	msTimeout	Timeout in milliseconds.
Return Value	Status, indicating whether the operation was successful.	

3.1.4.7. ptx_RUL_T3T_SensFReq

This function sends a T3T SensFReq command to the device.

Declaration	`ptx_Status_t ptx_RUL_T3T_SensFReq ( ptx_RUL_t *rul, uint16_t systemCode, uint8_t requestCode, uint8_t tsn, uint8_t* rx, uint32_t* rxLength);`	
Description	Sends a T3T SensFRequest to the device.	
Parameters	rul	Pointer to allocated RUL component.
	systemCode	System code.
	requestCode	Request code.
	tsn	Time Slot Number.
	rx	Rx Buffer.
	rxLength	Rx Buffer Length.
Return Value	Status, indicating whether the operation was successful.	

3.1.4.8. ptx_RUL_T5T_Isolated_EoF

This function sends an isolated EoF to an T5T.

Declaration	`ptx_Status_t` **`ptx_RUL_T5T_Isolated_EoF`** (**`ptx_RUL_t`** `*rul,` **`uint8_t*`** `rx,` **`uint32_t*`** `rxLength,` **`uint32_t`** `msTimeout`)	
Description	Sends an isolated EoF to a T5T device.	
Parameters	rul	Pointer to allocated RUL component.
	rx	Rx buffer.
	rxLength	Length of the Rx buffer and length of the Rx data.
	msTimeout	Timeout in ms.
Return Value	Status, indicating whether the operation was successful.	

3.1.5. Stack Information

3.1.5.1. ptx_RUL_GetStatusInfo

This function provides information about the current state of the stack. Based on the parameter passed to the function, different states can be queried.

Important: Call this function periodically.

The following parameters are possible:

- **ptx_RUL_StatusType_System** – Provides information about the system. The status should always be OK. If it is other than OK, then an unrecoverable error has occurred.
- **ptx_RUL_StatusType_Discover** – Provides information about the current state of the discovery process. It shows if cards are detected.
- **ptx_RUL_StatusType_Lpcd** – Provides the number of LPCD notifications that are received by the system.
- **ptx_RUL_StatusType_Deactivate** – Provides information about the deactivate process.
- **ptx_RUL_StatusType_EmvRemoval** – Provides information if an EMV collision has occurred. This is only set if the filter limit is reached.
- **ptx_RUL_StatusType_LastRfError** – Provides the last RF error that occurred. The value is reset after it is read.

Declaration	`ptx_Status_t` **`ptx_RUL_GetStatusInfo`** (**`ptx_RUL_t`** `*rul,` **`ptx_RUL_Status_Params_t *`**`statusParams`)	
Description	Retrieves various status information about the current state of the RUL stack.	
Parameters	rul	Pointer to allocated RUL component.
	statusParams	Pointer to data structure containing the parameter and the data.
Return Value	Status, indicating whether the operation was successful.	

3.1.5.2. ptx_RUL_GetInfo

This function provides information about the stack. The infoType parameter specifies the information to be extracted. The result values are provided in the **infoBuf** variable, while the number of bytes valid in **infoBuf** are provided in the **infoBufLen**.

The following information can be retrieved from the stack:

- **ptx_RUL_InfoType_HW_Version** – Provides information about the hardware connected to the stack, the minimum buffer size required is one byte.
- **ptx_RUL_InfoType_Stack_Revision** – Provides information about the stack revision, the minimum buffer size required is three bytes.
- **ptx_RUL_InfoType_FW_Revision** – Provides information about the firmware revision on the PTX2xxR/W. The minimum buffer size required is 2 bytes.
- **ptx_RUL_InfoType_FW_Tool_Revision** – Provides the revision information of the tool used to create the PTX2xxR/W firmware. The minimum buffer size required is 2 bytes.
- **ptx_RUL_InfoType_RF_Config_Version** – Provides the version of the RF config used. The minimum buffer size required is 1 byte.
- **ptx_RUL_InfoType_Chip_Derivate** – Provides the ID of the chip derivate. The minimum buffer size required is 2 bytes.

Declaration	`ptx_Status_t` **`ptx_RUL_GetInfo`** `(` **`ptx_RUL_t`** `*rul,` **`ptx_RUL_InfoTypes_t`** `infoType,` **`uint8_t*`** `infoBuf,` **`uint8_t*`** `infoBufLen`**`)`**	
Description	Provides specific information about the stack and the connected hardware.	
Parameters	rul	Pointer to allocated RUL component.
	infoType	Pointer to data structure containing the parameter and the data.
	infoBuf	Information buffer for the requested type.
	infoBufLen	Information buffer length for the requested type.
Return Value	Status, indicating whether the operation was successful.	

3.1.6. Card Registry

The card registry is an essential part of the RUL. The card registry contains all cards that have been discovered during the discovery process and their activation parameters. The card registry is also used to select/activate a single card in case multiple cards are present. The content of the card registry is managed internally and depends on the requested action at application level (for example, API call "**ptx_RUL_Deactivate(IDLE)**" removes all cards from the registry). It is possible to extend the content of each card added to the card registry by enabling (set to ON) the compile switch PTX_BUILD_EXTEND_CARD_REGISTER during build, which will provide a more detailed information about the activation parameters of each card technology type, if it is disabled (set to OFF) the card register will contain the raw response bytes for each card technology type included in the register.

3.1.6.1. ptx_RUL_AddToCardRegistry

This function adds a card to the card registry. This function is called automatically when a card is detected.

Declaration	`ptx_Status_t` **`ptx_RUL_AddToCardRegistry`** (**`ptx_RUL_t`** `*rul,` **`ptx_RUL_CardParams_t`** `*discoveredCard)`	
Description	Adds a card to the card registry.	
Parameters	rul	Pointer to allocated RUL component.
	discoveredCard	Pointer to the card parameters data structure.
Return Value	Status, indicating whether the operation was successful.	

3.1.6.2. ptx_RUL_ClearCardRegistry

This function clears the card registry. It removes all cards from the registry.

Declaration	`ptx_Status_t` **`ptx_RUL_ClearCardRegistry`** (**`ptx_RUL_t`** `*rul)`	
Description	Clears the card registry by removing all cards.	
Parameters	rul	Pointer to allocated RUL component.
Return Value	Status, indicating whether the operation was successful.	

3.1.6.3. ptx_RUL_DetermineProtocol

This function determines which protocols are available for the provided card parameter and stores them in the card parameters.

Declaration	`ptx_Status_t` **`ptx_RUL_DetermineProtocol`** (**`ptx_RUL_t`** `*rul,` **`ptx_RUL_CardParams_t`** `*card)`	
Description	Determines which protocols are available for the provided card and stores them in the card parameters.	
Parameters	rul	Pointer to allocated RUL component.
	card	Pointer to the card parameters data structure.
Return Value	Status, indicating whether the operation was successful.	

3.1.6.4. ptx_RUL_GetCardRegistry

This function returns the whole content of the card registry.

Declaration	`ptx_Status_t` **`ptx_RUL_GetCardRegistry`** (**`ptx_RUL_t`** `*rul,` **`ptx_RUL_CardRegistry_t`** `**registry)`	
Description	Returns the entire card registry.	
Parameters	rul	Pointer to allocated RUL component.
	registry	Pointer to the card registry data structure.
Return Value	Status, indicating whether the operation was successful.	

3.1.6.5. ptx_RUL_CheckCardPresence

This function checks if a card is in the field, by sending special commands to them, depending on the requested protocol.

The following types can be checked:

- **ptx_RUL_CheckPresType_NFCDEP** – Sends an Attention Command if the active protocol is NFC-DEP.
- **ptx_RUL_CheckPresType_ISODEP** – Sends an Attention Command if the active protocol is ISO-DEP (T4T).
- **ptx_RUL_CheckPresType_ISODEP_Empty** – Sends an empty frame if the active protocol is ISO-DEP (T4T).
- **ptx_RUL_CheckPresType_T2T** – Sends a read command, if the active protocol is T2T.
- **ptx_RUL_CheckPresType_T3T** – Sends a presence check command, if the active protocol is T3T.
- **ptx_RUL_CheckPresType_T5T** – Sends a presence check command, if the active protocol is T5T.

Declaration	`ptx_Status_t` **`ptx_RUL_CheckCardPresence`** `(` **`ptx_RUL_t`** `*rul,` **`ptx_RUL_CardPresType_t`** `type)`	
Description	Performs a presence check for the given card type by sending a special command to the card.	
Parameters	rul	Pointer to allocated RUL component.
	type	Type of presence check to be performed.
Return Value	Status, indicating whether the operation was successful.	

3.1.7. Card Activation and Deactivation

If multiple cards have been discovered during DISCOVERY, or a single card supports multiple protocols, or a card has been previously deactivated, it is needed to activate (select) the card to exchange the data with it. On the other hand, if it is needed to deactivate the card (put it to sleep), or if it is needed to restart discovery process or simply to stop any RF activity, deactivation procedure must be started. RUL API provides the functions for Activation and Deactivation.

3.1.7.1. ptx_RUL_SelectCard

This function starts a card (protocol) activation procedure after the card has been successfully added to the Card Registry but is not in active state (see section 4.1).

Declaration	`ptx_Status_t` **`ptx_RUL_SelectCard`** `(` **`ptx_RUL_t`** `*rul,` **`uint32_t`** `cardRegistryId` **`ptx_RUL_CardProtocol_t`** `protocol);`	
Description	Starts the card activation procedure after a card has been added to the card registry but is not in active state.	
Parameters	rul	Pointer to allocated RUL component
	cardRegistryId	Card registry ID to identify the card
	protocol	Specific card protocol
Return Value	Status, indicating whether the operation was successful.	

3.1.7.2. Ptx_RUL_GetCardParameters

This function gets the activation parameters of an activated target. This function shall be called once a card has been activated.

Declaration	`ptx_Status_t` **`ptx_RUL_GetCardParameters`**`(` **`ptx_RUL_t`** `*rul,` **`ptx_RUL_CardParams_t`** `**cardParams);`	
Description	Retrieves the activation parameters of an activated target.	
Parameters	rul	Pointer to allocated RUL component
	cardParams	Card/activation parameters of the activated target
Return Value	Status, indicating whether the operation was successful.	

3.1.7.3. ptx_RUL_Deactivate

This function is used to stop or re-start card discovery, or to perform protocol specific card de-activation. Depending on the use-case, corresponding state transition will take place and it might be necessary to invoke ptx_RUL_InitiateDiscovery to start discovery all over again, as described in section 4.1.

Declaration	`ptx_Status_t` **`ptx_RUL_Deactivate`** `(` **`ptx_RUL_t`** `*rul,` **`ptx_RUL_DeactivateParams_t`** `*params);`	
Description	Perform RF deactivation: stop/restart discovery or start a card protocol specific deactivation.	
Parameters	rul	Pointer to allocated RUL component
	params	Pointer to deactivation parameters
Return Value	Status, indicating whether the operation was successful.	

3.1.7.3.1. ptx_RUL_DeactivateParams_t Parameters

Name	Data Type	Description
Type	ptx_RUL_DeactivateType_t	Deactivation type that will be performed: ▪ ptx_RUL_DeactivateType_Generic – Generic (no protocol specific) ▪ ptx_RUL_DeactivateType_Specific – Protocol Specific Protocol specific deactivation is typically used for card protocol-specific deactivation when RUL is in one of the ACTIVE polling states, to put a card into sleep. Generic is used in all other cases.
State	ptx_RUL_DeactivateState_t	RUL state after deactivation: ▪ ptx_RUL_DeactivateState_Idle – IDLE, no RF activity ▪ ptx_RUL_DeactivateState_Poll – WAIT_FOR_ACTIVATION, Poll mode ▪ ptx_RUL_DeactivateState_Discovery – DISCOVERY ▪ ptx_RUL_DeactivateState_DiscoveryRfReset – DISCOVERY, RF field is reset ▪ ptx_RUL_DeactivateState_ListenSleep – WAIT_FOR_ACTIVATION, Listen mode ▪ ptx_RUL_DeactivateState_ListenIdle – WAIT_FOR_ACTIVATION, Listen mode

Note: State transitions must be obeyed as defined in section 4.1. Invoking ptx_RUL_Deactivate in a wrong state and with parameters that are not supposed to be used in that state might result in unexpected behavior – the API call might fail or in worse case RF activity might be unexpectedly stopped.

3.1.8. PTX2xxR/W GPIOs

The following section describes the functions to configure, write and read from the PTX2xxR/W GPIO pins. Available are the GPIO pins 1 to 8, usable through the enum ptx_RUL_GPIONbr_t. Note that access to every GPIO pin is not always permitted if it is used otherwise. The following GPIO availability rules apply:

1. If the Analog Testbus (see section 7.12) is active for a specific GPIO pin, it is not possible to use this pin as GPIO; others are available.
2. If the Digital Testbus (see section 7.12) is active for a specific GPIO pin, it is not possible to use this Pin as GPIO; others are available.
3. If the Auxiliary Interface is active the GPIO pins 4 to 8 are not available for a usage as GPIOs.
4. If GPIO pin 1 is configured as wake-up source for the Standby State, it is not available to use as GPIO.
5. If GPIO pin 2 is configured as reference clock input, it is not available to use as GPIO.

If any of these rules are violated, all GPIO API functions return the state **PTX_STATUS_E_GEN_NOT_PERMITTED**. Also, it should be noted that only GPIO pins 1, 4 and 6 can keep their configuration during Standby State. All others lose their state. This also applies for Standby State used in discovery loop. Thus, only these pins can be used during discovery if it uses Standby State.

3.1.8.1. ptx_RUL_ConfigGPIO

This function configures the GPIO pins of the PTX2xxR/W. Depending on the configuration, GPIO pins can be used as additional input and output for the PTX2xxR/W. The pins can be controlled via the GPIO Read and Write functions directly from the RUL API. It is possible to define the direction of the GPIO pin to output or input with **ptx_RUL_GPIODir_t**. Also, it is possible to configure certain attributes of the GPIO pin with flags defined in **ptx_RUL_GPIOFlags_t**. Note that most Flags are only valid for a single direction described as follows:

- **Input**
  - ptx_RUL_GPIOFlags_Internal_Pull_Resistor
  - ptx_RUL_GPIOFlags_InputGlitchFilter
- **Output**
  - Driver Strength:
    - ptx_RUL_GPIOFlags_Weakest_Driver_Strength
    - ptx_RUL_GPIOFlags_Weak_Driver_Strength
    - ptx_RUL_GPIOFlags_Strong_Driver_Strength
    - ptx_RUL_GPIOFlags_Strongest_Driver_Strength
- **Both**
  - ptx_RUL_GPIOFlags_None

Note: For the Driver Strength Flag, only one of the four versions must be chosen. Not all GPIO pins can use the Driver Strength or the Input Glitch Filter. These pins with an extended feature set are GPIO Pins 4, 6, 7, 8. Regarding the pull-resistor, it differs as to which pin has a pull-up and pull-down resistor. The only pin with a pull-down resistor is GPIO pin 5; all others have a pull-up.

Declaration	`ptx_Status_t` **`ptx_RUL_ConfigGPIO`** `(` **`ptx_RUL_t`** `*rul,` **`ptx_RUL_GPIONbr_t`** `gpioNbr,` **`ptx_RUL_GPIODir_t`** `gpioDir,` **`ptx_RUL_GPIOFlags_t`** `gpioFlags))`	
Description	Configure a GPIO of the PTX2xxR/W to be input or output.	
Parameters	rul	Pointer to allocated RUL component.
	gpioNbr	GPIO Number to be configured.
	gpioDir	GPIO direction.
	gpioFlags	Configuration flags.
Return Value	Status, indicating whether the operation was successful.	

3.1.8.2. ptx_RUL_WriteGPIO

This function writes to the specified GPIO of the PTX2xxR/W. It is possible to either write a **ptx_RUL_GPIOValue_High** or **ptx_RUL_GPIOValue_Low** value on the pin. Note that it is only possible to use this function if the GPIO pin is configured as an output.

Declaration	`ptx_Status_t` **`ptx_RUL_WriteGPIO`** `(` **`ptx_RUL_t`** `*rul,` **`ptx_RUL_GPIONbr_t`** `gpioNbr,` **`ptx_RUL_GPIOValue_t`** `value)`	
Description	Writes a high or low value to the GPIO, if it was previously configured as an output.	
Parameters	rul	Pointer to allocated RUL component.
	gpioNbr	GPIO Number to be configured.
	value	Value to write.
Return Value	Status, indicating whether the operation was successful.	

3.1.8.3. ptx_RUL_ReadGPIO

This function reads the current value from the configured port from the PTX2xxR/W. It is possible to either read a **ptx_RUL_GPIOValue_High** or **ptx_RUL_GPIOValue_Low** value from the pin. When it is not connected, the pull-resistor defines the value read if activated. If the pin is not connected, and the pull-resistor is deactivated, the value is random and there will be leakage.

Declaration	`ptx_Status_t` **`ptx_RUL_ReadGPIO`** `(` **`ptx_RUL_t`** `*rul,` **`ptx_RUL_GPIONbr_t`** `gpioNbr,` **`ptx_RUL_GPIOValue_t`** `*value)`	
Description	Reads the current value of the GPIO, if the GPIO was previously configured to be an input.	
Parameters	rul	Pointer to allocated RUL component.
	gpioNbr	GPIO Number to be configured.
	value	Value from the pin.
Return Value	Status, indicating whether the operation was successful.	

3.1.9. Extensions

The following sections describe how to handle extensions with RUL. Extensions extend RUL and add additional functionality. Extensions can be registered and removed. For examples of extensions see sections 7.4, 7.11, and 7.13 for extensions.

3.1.9.1. ptx_RUL_RegisterExtensionEndpoints

This function registers an extension endpoint for the use within the RUL. Endpoints are used to further register their own notification handlers with the NSC command processor to receive the data from the PTX2K and operate on the data.

The extension endpoint must point to a data structure containing the following information:

- CompId: The unique component id of the extension. It is mandatory to provide a valid component id.
- Extension: A pointer for the context/internal structure internally used by the extension. It is mandatory to provide an extension context.
- ProcessReset: This callback is executed when a reset is executed. It is not mandatory to use this endpoint but can be beneficial if internal structures need to be reset during a reset.
- ProcessRfDiscover: This callback is executed during a discovery.
- ProcessRfLpcd: This callback is executed when an LPCD event happens.
- ProcessRfActivate: This callback is executed when a card or device is activated.
- ProcessRfDeactivate: This callback is executed when a card or device is deactivated.
- ProcessError: This callback is executed when an error from the PTX2K is received.
- ProcessRfField: This callback is executed when an RF Field is turned on or off. This is only useful if the stack operates in listen mode.
- ProcessWptStop: This callback when WPT Stop is received.
- ProcessDataMsg: This callback is executed when data is received from a card or device.

Declaration	`ptx_Status_t` **`ptx_RUL_RegisterExtensionEndpoints`** `(` **`ptx_RUL_t`** `*rul,` **`ptx_RUL_Extension_Endpoints_t`** `endpoint)`	
Description	Register an add on library and its corresponding endpoints with RUL.	
Parameters	rul	Pointer to allocated RUL component.
	endpoint	Pointer to the endpoint data structure.
Return Value	Status, indicating whether the operation was successful.	

3.1.9.2. ptx_RUL_RemoveExtensionEndpoints

This function removes registered endpoints. They will be deregistered from the RUL and not be operational afterwards. The endpoint to be removed is identified by the unique component id.

Declaration	`ptx_Status_t` **`ptx_RUL_RemoveExtensionEndpoints`** `(` **`ptx_RUL_t`** `*rul,` **`ptx_Comps_t`** `compId)`	
Description	Removes a registered add-on library and its corresponding endpoints from RUL.	
Parameters	rul	Pointer to allocated RUL component.
	compId	The extensions component identifier.
Return Value	Status, indicating whether the operation was successful.	

3.1.10. Utility and Measurement APIs

This section describes utility and measurement APIs for RUL.

3.1.10.1. ptx_RUL_Sleep

This function aloows the stack wait for the specified amount of time in ms. This function can be used to wait for a specified amount of time.

Declaration	`ptx_Status_t` **`ptx_RUL_Sleep`** `(` **`ptx_RUL_t`** `*rul,` **`uint32_t`** `ms)`	
Description	Sleep for a specified duration ins ms.	
Parameters	rul	Pointer to allocated RUL component.
	ms	Sleep duration in milliseconds.
Return Value	Status, indicating whether the operation was successful.	

3.1.10.2. ptx_RUL_WriteDAC

Writes the given value to the Digital to Analog Converter (DAC).

Declaration	`ptx_Status_t` **`ptx_RUL_WriteDAC`** (**`ptx_RUL_t`** `*rul,` **`uint8_t`** `value`**)**	
Description	Writes a value to the Digital to Analog Converter.	
Parameters	rul	Pointer to allocated RUL component.
	value	Digital value to be converted to analog.
Return Value	Status, indicating whether the operation was successful.	

3.1.10.3. ptx_RUL_SetPowerMode

This function set the chip into the desired power mode. The power mode can either be standby or wakeup. While standby is used to get into the standby mode manually by calling this function, wakeup can be used to wakeup from the standby mode and go into full-power mode.

Declaration	`ptx_Status_t` **`ptx_RUL_SetPowerMode`** (**`ptx_RUL_t`** `*rul,` **`ptx_RUL_PowerMode_t`** `powerMode`**)**	
Description	Used to set the power mode of the chip to either standby or wakeup.	
Parameters	rul	Pointer to allocated RUL component.
	powerMode	Desired power mode.
Return Value	Status, indicating whether the operation was successful.	

3.1.10.4. ptx_RUL_RfFieldMeasurement

This function executes the RF field amplitude measurement at the given frequency and is called by the PSL (see section 7.12).

Declaration	`ptx_Status_t` **`ptx_RUL_RfFieldMeasurement`** (**`ptx_RUL_t`** `*rul,` **`uint32_t`** `targetFrequency,` **`ptx_RUL_AdcAveraging_t`** `averaging,` **`uint32_t`** `*rfAmp`**)**	
Description	Executes the RF field amplitude measurement.	
Parameters	rul	Pointer to allocated RUL component.
	targetFrequency	Target frequency for the amplitude measurement
	averaging	Number of measurements to average
	rfAmp	Pointer to the measurement result
Return Value	Status, indicating whether the operation was successful.	

3.1.10.5. ptx_RUL_GetRSSIValue

This function executes the RSSI measurement at 13.56MHz using the given power level.

Declaration	`ptx_Status_t` **`ptx_RUL_GetRSSIValue`** `(` **`ptx_RUL_t`** `*rul,` **`uint8_t`** `cwPowerLevel,` **`ptx_RUL_AdcAveraging_t`** `averaging,` **`uint32_t`** `*result)`	
Description	Executes the RSSI measurement using CW power level 0x58 and 16 samples.	
Parameters	rul	Pointer to allocated RUL component.
	cwPowerLevel	CW multiplier value used for the measurement.
	averaging	The amount of averaged measurements.
	result	Pointer to the measurement result
Return Value	Status, indicating whether the operation was successful.	

4. RUL API States

This section provides an overview about the different states the RUL Stack can operate in. The states implicitly change depending on the called functions from the RUL API and the add-on APIs.

Important: If an API function is called in a state where it is not supposed to be called, a dedicated status code is returned. An overview about all status codes can be found in source file "**ptx_Status.h**".

4.1 State Descriptions

Figure 2 shows a high-level overview of the possible states of the RUL Stack and how they are changed depending on which RUL API function call gets executed. The overview below assumes that all API function calls return successfully. Unless specified otherwise in this document, the RUL Stack remains in the same state in case of an error.

***Important**:* Figure 2 shows only a few possible RUL API and add-on API function calls allowed in a specific state or how individual calls leads to state-transition. It does not cover all possible API calls. Wherever applicable, additional state-transitions are described in the corresponding sections of the RUL and add-on APIs. In addition, the RUL SDK contains various code examples which demonstrate the correct API usage of the RUL SDK.

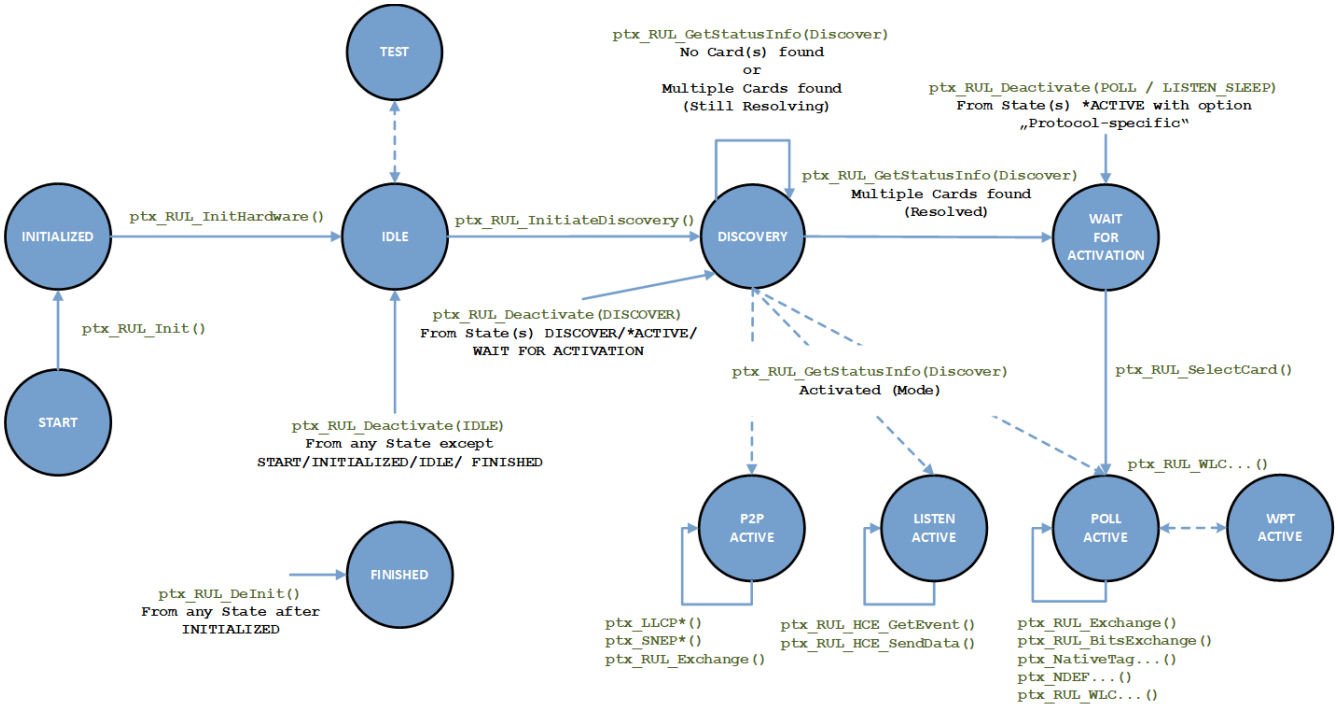

Figure 2. RUL Stack State Overview

- **State: START**

  The RUL Stack must be initialized first via a call to "**ptx_RUL_Init**" before the PTX2xxR/W hardware can be accessed. Besides the actual component initialization, this call also initializes the corresponding default HAL-component used for the PTX2xxR/W including consisting of the corresponding default HAL-device and HAL-driver initialization. It is also recommended to initialize add-on APIs (if used) in this state. Once the RUL Stack is initialized, the system transitions to state **INITIALIZED**.

  ***Important**:* Unless specified otherwise in section 7, add-on APIs can also be initialized in later states or even before the first usage.

- **State: INITIALIZED**

  After the RUL Stack and the corresponding HAL-component are initialized, the PTX2xxR/W hardware must be initialized via a call to "**ptx_RUL_InitHardware**", where it:

  - resets the PTX2xxR/W,
  - downloads the FW image and
  - downloads the default RF configuration.

  Once the PTX2xxR/W is initialized, the system transitions to state **IDLE**.

- **State: IDLE**

  In **IDLE** state, the system is ready to use, and it is up to the application to command the RUL Stack to use various test-related add-on APIs (see below).

  For a typical NFC application, the next step would be to start the RF-discovery procedure, in other words, NFC polling-loop via a call to "**ptx_RUL_InitiateDiscovery**". As described in section 3,the behavior of the RF-discovery procedure is highly configurable and provides various options to control the actual polling loop via a dedicated parameter input structure "**ptx_RUL_DiscoverConfig_t**".

This parameter structure allows the user to:

- Set the actual Polling-scheme: NFC Forum, EMV, ISO or Continuous Wave
- Enable/disable the different Poll-mode/Reader-mode based on NFC technologies A / B / F / V / ACM including various bail-out, polling-interval and device-limit options
- Enable/disable listening in Host-Card-Emulation mode based on NFC technologies A / B / F / ACM
- Activate Stand-by mode and/or Low-Power-Card-Detection (LPCD) during the polling-loop
- Disable automatic activation of higher-level protocols ISO-DEP and/or NFC-DEP
- Enable activation of higher bit-rates for higher-level protocols such as ISO-DEP and NFC-DEP
- Configure the polling loop IDLE-time
- Configure the starting NFC technology for the Poll-/Reader modes
- …

Figure 3 shows the default RF-discovery procedure when all Poll-mode/Reader-mode are enabled. As mentioned above, the individual technologies can be freely enabled/disabled and even the starting order can be configured as needed.

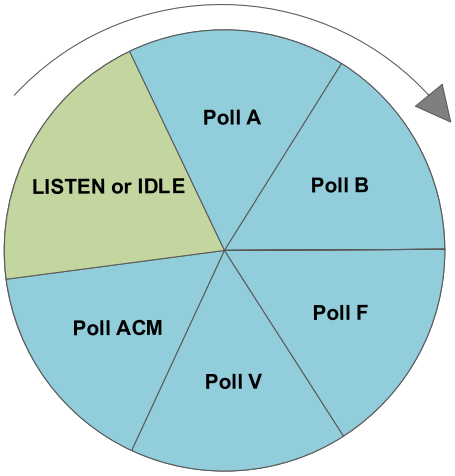

Figure 3. Standard RF-Discovery Procedure Sequence

If, for example, NFC technology B would not be enabled, the sequence would change to A, F, V etc. For NFC technology F, it is possible to select the bit-rate separately during the RF-discovery procedure (either 212 kBit/s or 424 kBit/s).

The parameter “IDLE”-time defines the intervals of the polling-loop cycles, In other words, if IDLE-time gets set to 1 second, the polling-loop would be repeated every second. For battery-powered devices, it is recommended to enable Stand-by mode and/or LPCD during that duration. If any Listen-modes are enabled, the PTX2xxR/W is listening for the presence of an external RF-field during the duration of “IDLE”-time.

***Important**:* The current version of the RUL SDK supports Listen-mode only in Host-Card-Emulation mode. The necessary routing tables are configured internally to forward all incoming NFC traffic automatically to the host / application.

After a successful call to "**ptx_RUL_InitiateDiscovery**", the system transitions to state **DISCOVERY**.

Besides starting the RF-discovery procedure, it is also possible in IDLE-state to:

- Update the RF-configuration via a call to “**ptx_RUL_UpdateHardwareConfiguration**”
- Enter Stand-by mode via a call to “**ptx_RUL_SetPowerMode**”
- Execute various API functions from the add-on APIs: **PSL, RF-TEST, POS-DTE**, **NFC Forum DTA, TRANSPARENT MODE** and **FELICA-DTE**
- Read-out the RSSI-value and use the integrated DAC
- …

- **State: DISCOVER**

  In **DISCOVER** state, the application can retrieve the status of the RF-discovery procedure via a call to "**ptx_RUL_GetStatusInfo(Discover)**".

  The status indicates:

  - If a single card/device was detected and activated in Poller-mode, Reader-more or NFC P2P Initiator mode, the system was activated by a remote device in Listen-mode, Card Emulation mode or P2P Target mode.

    A call to API function "**ptx_RUL_GetCardRegistry**" can be used to retrieve details on the activated card (serial number, information on used RF protocol, etc.) to determine, for example, if a card supports the ISO-DEP protocol.

  - If multiple cards were discovered, the RF-discovery procedure is still ongoing. This state is for information purposes only.

  - If multiple cards were discovered and the RF-discovery procedure is complete.

    A call to "**ptx_RUL_GetCardRegistry**" can be used to retrieve the number of discovered cards including detailed information required for further selection/activation.

  - If a single card was discovered supporting multiple protocols.

    A call to "**ptx_RUL_GetCardRegistry**" can be used to retrieve details on the discovered card required for further selection / activation.

  - If nothing was discovered.

  - If an EMV-collision occurred (only valid in EMV polling mode).

    An ongoing polling operation can be stopped via a call to "**ptx_RUL_Deactivate**".

    If multiple cards were detected or a single card supporting multiple protocols, the system transitions into **WAIT FOR ACTIVATION** state. It is then up to the application to select/activate the desired card from the card registry.

    If the RF-Discovery procedure status is in ACTIVE-state, the actual NFC mode (Poller, Reader, Listen-/Card Emulation, P2P) can be retrieved via a call to "**ptx_RUL_GetStatusInfo(NFC_Mode)**". Depending on the retrieved mode, the system transitions either to **POLL ACTIVE**, **LISTEN ACTIVE** or **P2P ACTIVE** state.

- **State: WAIT FOR ACTIVATION**

  The **WAIT FOR ACTIVATION** state is reached if:

  - multiple cards were previously discovered or
  - a single card supporting multiple RF protocols was discovered or
  - a single card was previously deactivated (in other words, put to sleep).

  The application can use the function " **ptx_RUL_GetCardRegistry** " to gather information about the available cards. To select/activate a specific card or specific RF protocol, a call to " **ptx_RUL_SelectCard**" is required before the system transitions to **POLL ACTIVE** state.

- **State: POLL ACTIVE**

  In this state, the system is ready for actual NFC data exchanges with remote card/Tag devices independent of the previously used Polling-scheme (NFC Forum, EMV, ISO). NFC data exchanges can be triggered via calls to "**ptx_RUL_Exchange**" or "**ptx_RUL_BitsExchange**" (T2T RF protocol only). If needed, the application can also check whether a card is still present or not via a call to "**ptx_RUL_CheckCardPresence**".

  Important: The function "**ptx_RUL_BitsExchange**" can only be used in this state and requires the active card/tag to use the T2T RF protocol. Additionally, the bits-exchange mode must be enabled upon the first usage in this state via a call to "**ptx_RUL_BitsExchangeMode**" and must be disabled after the last usage. The bits-exchange mode is not compatible with standard RF data exchanges via "**ptx_RUL_Exchange**".

  The RF protocols ISO-DEP and NFC-DEP are handled internally by the PTX2xxR/W FW. All other (NFC Forum) protocols like T2T, T3T, and T5T must be handled by the application and add-on APIs on top.

If the API function "**ptx_RUL_Exchange**" gets used in combination with the RF protocols T2T, T3T or T5T, the received data from a card contains one additional byte at the end which represents the status of the data-exchange. If bit 7 of this byte (mask 0x80) is set to1, the received data is invalid (for example, CRC, parity error, etc.). It is up to the application to determine how to handle this scenario. If the higher-level protocols like ISO-DEP and NFC-DEP are used, the received data contains only the payload of the used protocol.

The RUL SDK contains the add-on libraries **Native Tag API**, **NDEF API, RUL WLC** and **Mifare Classic Compatibility Mode (MFCC)** which implement standardized and product-specific command-sets as dedicated API functions to simplify application development.

In addition, the following functions can be used in the **POLL ACTIVE** state:

- **ptx_RUL_T5T_Isolated_EoF**

  This is an optional API function which may be used to send an isolated EoF-packet to a T5T card which may be required for certain commands.

- **ptx_RUL_T3T_SensFReq**

  This is an optional API function which allows to send a T3T SENSF_REQ with given parameters. The data contained in the provided output buffer contains the concatenated responses of each detected card with a prepended length.

▪ **State: WPT ACTIVE**

This state gets entered, once the **RUL WLC** API starts a Wireless Power Transfer (WPT), this can either be done by the user directly via the API function "**ptx_RUL_WLC_StartPowerTransfer**" or through the internal and any externally implemented WLC state machine. In this state, the RF field is kept constantly on for a configured power level to transfer energy to a connected WLC listener. This state can be left by either explicitly stopping the transfer as the user directly with "**ptx_RUL_WLC_StopPowerTransfer**" or by waiting until the WPT stops automatically as configured earlier for various reasons. To check for the current state, call "**ptx_RUL_WLC_GetPowerTransferState**", which gives information if the transmission stopped and for which reason. This state can only be entered from **POLL ACTIVE** and returns to this state once finished.

▪ **State: LISTEN ACTIVE**

When this state gets entered, the system was activated by an external/remote Poller/Reader device. Once entered, the application keeps communicating with the external/remote device until either the external field is turned off or if the system gets deactivated by the application via a call to "**ptx_RUL_Deactivate(IDLE, DISCOVER)**". All incoming NFC data packets and events are internally stored in an event queue which can be read via a call to "**ptx_RUL_HCE_GetEvent**". Possible event types can be:

- External Field On/Off
- Activated in Listen/Card Emulation based on NFC technology A, B or F including RF protocol information.
- Data received

  Once the received data was processed, the response can be sent back using "**ptx_RUL_HCE_SendData**".

When the external field is turned off, the system automatically transitions back to **DISCOVERY** state.

▪ **State: P2P ACTIVE**

This state gets entered when the activated remote device is an NFC P2P device (passive/active). The exact mode (in other words, whether P2P is active in NFC P2P Initiator-mode or Target-mode) can be retrieved via a call to "**ptx_RUL_GetStatusInfo(NFC_Mode)**".

If the remote NFC P2P device supports the LLCP and SNEP protocol, the application can use the API functions from the corresponding **LLCP** and **SNEP** add-on APIs. If the application intends to implement a custom P2P application, the same mechanisms and API function calls applies as described for states **POLL ACTIVE** and **LISTEN ACTIVE**.

- **State: TEST** (Optional)

  The optional **TEST** state is equivalent to the **IDLE** state. Some of the add-on APIs like **RF-TEST**, **NFC Forum DTA**, **POS-DTE**, **RF-TEST**, **TRANSPARENT MODE, PSL**, etc., may require the system to operate in a different mode and to apply different hardware configurations. If applicable, changing modes and configurations are completely managed internally by the "**Open/SetMode/Init (or similar)**" functions of the corresponding add-on APIs. After calling any of these functions, the system is considered to be in **TEST** state and will return to regular **IDLE** state by calling the corresponding "**Close/SetMode/DeInit (or similar)**" function.

  For more information on these add-on APIs, refer to section 7.

- **State: FINISHED**

  If applicable, the RUL Stack may be uninitialized via a call to "**ptx_RUL_DeInit**" to free up all previously allocated/opened resources. This function can be called anytime from state **INITIALIZED** onwards, however it can happen that the PTX2xxR/W hardware remains still in an ACTIVTE-state (for example, has the internal RF field turned on or is executing the RF-discovery procedure). In this case, the application is responsible to switch back to IDLE state via a call to "**ptx_RUL_Deactivate(IDLE)"** or to reset the hardware via a call to "**ptx_RUL_Reset**".

  Important: Unless otherwise specified in the following sections, API function **ptx_RUL_Deactivate(IDLE)** is always to be used to switch the system back to **IDLE** state from states **POLL ACTIVE, LISTEN ACTIVE, P2P ACTIVE** or **WAIT FOR ACTIVATION**.

5. RUL SDK Deliverable

Table 1 shows the package content of the RUL SDK deliverables and in which (sub-)folders the items can be found.

Important*:* The current version of the SDK may already include source files and modules which are not yet officially supported or are only partially implemented.

Table 1. RUL SDK Delivery Content

SDK Folder Location	Feature
SDK_ROOT/SDK_Release_Notes.txt	SDK Release Notes
SDK_ROOT/DOCS/html/	HTML-based API documentation
SDK_ROOT/PROJECTS/PTX2xxR/	CMake-scripts and Renesas e^2 Studio project files for all provided demos/examples and applications
SDK_ROOT/SRC/API/RUL/	RUL Main API
SDK_ROOT/SRC/API/RUL_HCE/	RUL HCE API Extension
SDK_ROOT/SRC/API/RUL_WLC/	RUL WLC-P API Extension
SDK_ROOT/SRC/API/EVK_PERIPHERALS/	EVK Peripherals Add-on Module
SDK_ROOT/SRC/API/FELICA_DTE/	FeliCa-DTE add-on API
SDK_ROOT/SRC/API/NATIVE_TAG/	Native Tag add-on API
SDK_ROOT/SRC/API/NDEF/	NDEF add-on API and support libraries
SDK_ROOT/SRC/API/NFC_FORUM_DTA/	NFC Forum DTA add-on API
SDK_ROOT/SRC/API/POS_DTE/	EMV/POS L1 DTE add-on API
SDK_ROOT/SRC/API/PSL/	Product Support Library add-on API
SDK_ROOT/SRC/API/RF_TEST/	RF-Test add-on API
SDK_ROOT/SRC/API/TRANSPARENT_MODE/	Transparent Mode add-on API

SDK Folder Location	Feature
SDK_ROOT/SRC/COMPS/ptx_Status.h	Generic Status and Error codes
SDK_ROOT/SRC/COMPS/ptx_Components.h	API Component identifiers
SDK_ROOT/SRC/COMPS/HAL/	Generic HAL Device and Driver Interfaces
SDK_ROOT/SRC/COMPS/HAL/DRV/MCU/RA4M2/	Renesas RA4M2 Reference HAL Device and Driver implementation
SDK_ROOT/SRC/COMPS/HAL/DRV/WIN/	Microsoft Windows Reference HAL Device and Driver implementation
SDK_ROOT/SRC/COMPS/HAL/DRV/LIN/	Linux Reference HAL Device and Driver implementation
SDK_ROOT/SRC/COMPS/HIP/	Host Interface Protocol implementation
SDK_ROOT/SRC/COMPS/LOG/	Generic LOG Interface and 3rd party **zf_log**
SDK_ROOT/SRC/COMPS/LOG/CONSOLE/	LOG reference implementation for writing to console
SDK_ROOT/SRC/COMPS/LOG/FILE/	LOG reference implementation for writing to file(s)
SDK_ROOT/SRC/COMPS/LOG/NONE/	Stub-functions in case LOG gets excluded from target build
SDK_ROOT/SRC/COMPS/LOG/SERIAL/	LOG reference implementation for writing to serial-terminal
SDK_ROOT/SRC/COMPS/NSC/	NSC Interface and Command Set library
SDK_ROOT/SRC/COMPS/NSC/UCODE/	PTX2xxR/W Firmware Image
SDK_ROOT/SRC/COMPS/NSC/RUL/ptx_NSC_RfConfigVal.*	Default RF-Config Image
SDK_ROOT/SRC/COMPS/OSAL/	Internal Operating System Abstraction layer used for certain Windows-/Linux HAL reference implementations
SDK_ROOT/SRC/COMPS/UTILS/	Internal Ringbuffer implementation
SDK_ROOT/SRC/EXAMPLES/	SDK code examples/demos and applications
SDK_ROOT/SRC/EXAMPLES/Utils/	Common helper-functions for code examples (for example, parsing command-line arguments, printf, ...)

6. RUL SDK Target System Integration and Porting

6.1 Build System

The stack can be built for the Renesas Platform (RA4M2), Linux and Windows, with CMake and e^2 Studio supported to build the stack. This section describes how to build the stack.

Note: e^2 Studio can only build for the Renesas Platform (RA4M2).

6.1.1. Build Requirements

For building the RUL API and/or the contained code examples, the following tools are required:

- CMake 3.15 or higher (see cmake.org)
- Any C-compiler (depending on target platform) supporting C99 standard
  - When building for the RA4M2 MCU platform, the correct configuration-file depending on the used host interface needs to be set in the corresponding e^2 Studio project.
    - Depending on the used host interface, the available "**configuration_[SPI | I2C | UART].xml**" file needs to be renamed to "configuration.xml".

6.1.2. Configuring and Building the Stack with CMake

After successful installation of the SDK, change to the desired project folder and open the **CMakeLists.txt** file. All configurations are disabled per default and the project cannot be built until the parameters are changed to fit your system. This can either be done in the **CMakeLists.txt** file directly or via the command line when CMake is called.

1. Select the platform the stack should be built for. This can either be MCU, Windows or Linux. For the MCU option, set the option **PTX_BUILD_MCU** to ON; for Windows set the option **PTX_BUILD_WIN** to ON; for Linux set the option **PTX_BUILD_LIN** to ON.

   Important: Ensure that only one of the available options is set. Setting multiple platform options will result in an unusable stack.

2. Select the hardware interface to be used to communicate with the PTX2xxR/W. Choose between SPI, I2C and UART. For SPI set the option, **PTX_ENABLE_SPI** to ON; for I2C, set the option **PTX_ENABLE_I2C** to ON; for UART, set the option **PTX_ENABLE_UART** to ON.

   Important: Ensure that only one of the available hardware interface options is set to ON. Setting multiple hardware interface options to ON will result in an unusable stack.

3. Select the Logging configuration for the stack. In the default configuration, logging is turned off. Three different loggers are available. Logging is available via a serial communication (UART), to the console and to a file. The serial logging is only available if the target platform is MCU, while logging to the console or to a file is only available for Windows and Linux. The logger can be configured by setting the **PTX_LOGGING** variable. It can be set to None, Serial, File and Console.

   Note: Not all logging configurations are suitable for all platforms.

   Note: It is not possible to set Serial for logging and for the utility print module at the same time, as the reference implementation on RA4M2 uses the same UART interface for both.

4. It is optionally possible to change the output type of the utility print module, which can be used as a simple formatted print tool. In default configuration, printing is turned off. For this, the option **PTX_UTILITY_PRINTING** can be either set to use a console or a serial communication interface (UART) as the output, by setting the variable to None, Serial or Console. The serial printing is only available if the target platform is MCU, while logging to the console is only available for Windows and Linux. None represents the configuration when printing is turned off.

   Note: Not all print configurations are suitable for all platforms.

   Note: It is not possible to set Serial for logging and for the utility print module at the same time, as the reference implementation on RA4M2 uses the same UART interface for both.

It is strongly recommended to create a separate build folder in the projects root directory to not have temporary CMake files mixed with your own project files. For this document, a folder named BUILD will be created. Open the build folder and execute CMake. See the code example below for the proper call for Windows and Linux platforms.

- **cmake ..**

  For the MCU platform it is usually required to provide a toolchain file. This can be done by providing the toolchain as parameter to CMake. For the RA4M2 platform, a toolchain file (**armgcc.cmake**) is provided. For the MCU platform, call CMake like shown below

- **cmake .. --toolchain ../MCU/armgcc.cmake**

  Note: To compile for the RA4M2 platform, the FSP source files need to be created first. This can be done via e^2 Studio and is described in the section about building the project with e^2 Studio.

  CMake now configures the projects, checks if all dependencies are fulfilled and all files are available. The next step describes building the executable. This is done with the command depicted below.

- **cmake --build**

  This command starts building the project for your target platform with the configured hardware interfaces and logger.

6.1.2.1. CMake Configuration Switches

This section describes the available switches for CMake. The switches can be used to customize the configuration and build.

6.1.2.1.1. Platform

This section describes the switches and options for the target platform.

Important: Only one switch may be set to ON.

Switch	Value	Description
PTX_BUILD_MCU	ON/OFF	This switch enables a build for MCU as target. For both Eval and Demo board this is a RA4M2 MCU. The default value is set to OFF.
PTX_BUILD_LIN	ON/OFF	This switch enables a build for Linux systems as target for both Eval and Demo boards. The default value is set to OFF.
PTX_BUILD_WIN	ON/OFF	This switch enables a build for Windows systems as target for both Eval and Demo boards. The default value is set to OFF.

6.1.2.1.2. Hardware Interface

This section describes the switches and options for the available hardware interfaces to communicate with the PTX2K.

Switch	Value	Description
PTX_ENABLE_SPI	ON/OFF	Enables SPI as hardware interface to communicate with the PTX2K. The default value is set to OFF. Ensure that PTX2xxR/W is configured properly to communicate via SPI.
PTX_ENABLE_I2C	ON/OFF	Enables I2C as hardware interface to communicate with the PTX2K. The default value is set to OFF. Ensure that PTX2xxR/W is configured properly to communicate via I^2C.
PTX_ENABLE_UART	ON/OFF	Enables UART as hardware interface to communicate with the PTX2K. The default value is set to OFF. Ensure that PTX2xxR/W is configured properly to communicate via UART.

6.1.2.1.3. Logging

This section describes the switches and options for software logging.

Switch	Value	Description
PTX_LOGGING	None/Serial/Console/File	This switch enables the logging component for the internal logging. The options allowed to be set are dependent on the selected platform. ▪ None – Disables logging, does not depend on a specific platform ▪ Serial – Enables logging via additional UART. It is only available if the PTX_BUILD_MCU is set and is not available for Linux or Windows builds. This cannot be set to Serial if PTX_UTILITY_PRINTING is also set to Serial as it is using the same UART interface. ▪ Console – Enables logging to the command line. It is only available if either PTX_BUILD_LIN or PTX_BUILD_WIN is set and is not available for MCU builds. ▪ File – Enables logging to a file. It is only available if either PTX_BUILD_LIN or PTX_BUILD_WIN is set and is not available for MCU builds. The default value is None which disables logging.

6.1.2.1.4. Printing

This section describes the switches and options for the optional simplified utility software printing module.

Switch	Value	Description
PTX_UTILITY_PRINTING	None/Serial/Console	This switch enables the utility print component for a simplified printing alternative to the logging module. The options allowed to be set are dependent on the selected platform. ▪ None – Disables printing, does not depend on a specific platform ▪ Serial – Enables printing via additional UART. It is only available if the PTX_BUILD_MCU is set and is not available for Linux or Windows builds. This cannot be set to Serial if PTX_LOGGING is also set to Serial as it is using the same UART interface. ▪ Console – Enables printing to the command line. It is only available if either PTX_BUILD_LIN or PTX_BUILD_WIN is set and is not available for MCU builds. The default value is None which disables printing.

6.1.3. Configuring and Building the Stack with Renesas e² Studio

After successful installation of the SDK, open e² Studio and go to **File → Open Projects from File System** and choose the SDK directory. All valid e² Studio projects in the SDK are listed. Choose and import the desired project(s).

With e² Studio, it is only possible to build for the MCU Platform and not all configurations of the stack are available. First, Windows and Linux Builds are not available. Second, in case of the Logging configuration, only the two options of none and serial communication (UART) logging are available. For the hardware interface, all options are available. Be aware that the utility print module is deactivated by default for the predefined e² Studio targets but may be added with the explanation from the previous chapter.

To choose between all configurations, there are designated build configurations available. Their names are in the Form "**Debug-SPI-NONE**". The first part of the string defines if it is a Release or Debug build; the second shows the used hardware configuration (SPI, I2C and UART); the last part of the name describes which logging option is active (NONE or SERIAL). With these predefined configurations, no further settings are required to build the stack.

For the RA4M2 Platform there are certain auto-generated files missing which are required for the CMake build for this platform. These can either be generated by opening the **configuration.xml** file and pressing “Generate Project Content” or by simply building the project once in e^2 Studio as this will also trigger the generation of these files.

6.2 Adding Add-On Libraries to the Build System

This section describes how to add and remove Add-On libraries to individual projects. A description of the available addon libraries can be found in section 7 of this document.

Following, we describe how to add the POS DTE add-on.

For the POS DTE, as for all add-ons, identify which files are required to add them to your build system.

The required files can be found in the **SRC/API/POS_DTE** folder. For some add-ons like the NDEF and NativeTag, all files from the subfolders need to be added.

If you plan to extend an existing example with an add-on library, add the path to the file and their filename to the SOURCES variable and the path to the include files to the HEADER_DIR variable.

If you plan to extend an existing e^2 Studio project with an add-on library, open the folder **src/API** in e^2 Studio, right-click and select **New > Folder**. The new folder should be given the name of the add-on. Next, right-click on the new folder and select import. Select the import files option and select the proper API folder.

Now open the project settings and go to the **C/C++ Build > Setting**s entry. Select the Tool Settings tab. Under **GNU ARM Cross C Compiler**, select **Includes** and add the directory to the API in the Include Paths by pressing the add button.

Select Workspace and select the newly added folder. It is suggested to click on the “Add Subdirectories” checkbox to add all subdirectories as well (if there are any).

If not planning to extend existing examples, then add the files according to your build system.

6.2.1. Additional Compile Switches for Add-On Libraries

Add-On libraries can be further configured with compile switches. Currently this is only true for the NDEF add-on library. This section describes the switches for the NDEF add-on library. For a complete description of the NDEF add-on library, refer to section 7. The following table describes the available switches for each add-on library.

Add-on Library	Switch	Description
NDEF	PTX_DISABLE_NDEF_T2T	Disables the compilation of the NDEF T2T features.
NDEF	PTX_DISABLE_NDEF_T3T	Disables the compilation of the NDEF T3T features.
NDEF	PTX_DISABLE_NDEF_T4T	Disables the compilation of the NDEF T4T features.
NDEF	PTX_DISABLE_NDEF_T5T	Disables the compilation of the NDEF T5T features.

These switches affect the compilation of the stack and must be set in the build environment. In **cmake**, you can pass them during the first cmake run as *-D<Switch Name>* where <Switch Name> should be replaced with the switch you want to set. In e^2 Studio, they can be set via a right-click on the *Project* > *Preference*s. Go to the **C/C++ Build Environment > Settings** entry. In the Tool Settings tab, set the *Preprocessor* options under **GNU ARM Cross C Compiler > Preprocessor**.

6.3 Hardware Abstraction Layer (HAL)

6.3.1. Introduction

The HAL layer is the lowest layer of the system. It consists of several sub-modules that all work together for the HAL. Figure 4 provides an overview of the different layers that together form the HAL. The HAL itself is split into two layers. The topmost layer is the HAL Interface, which contains interfaces for the stack to communicate with the PTX2xxR/W. The inner layer is the driver layer. The driver layer implements the low-level communication between the stack and the PTX2xxR/W. The driver does not know about any high-level communication protocols; it just sends data with its designated functions.

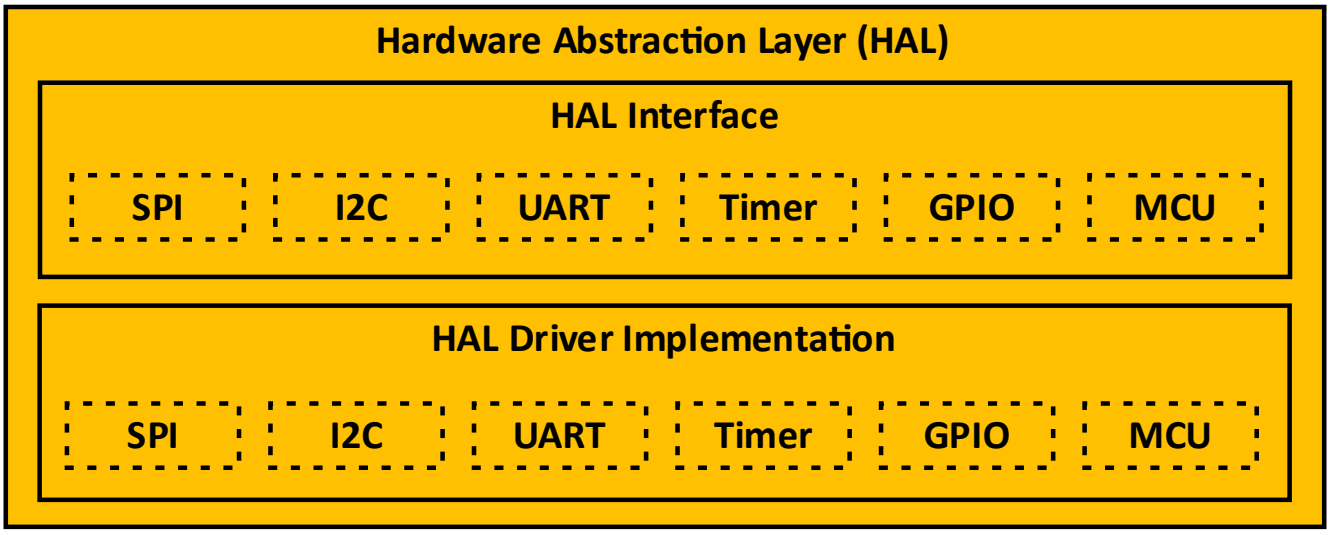

Figure 4. HAL Overview

When porting the HAL to a new device, it is only required to port the driver. The upper layer (HAL Interface) must not change as they function as an API. If changes are made, then they must be made in the stack as well and this is beyond the scope of this document.

All buffers passed to the HAL are passed as pointers. That way, the stack takes care of the proper size of the buffer and the HAL only reads the internals of the buffer.

The design of the PTX2xxR/W stack allows you to easily port the HAL Driver to a new platform implementation. A new platform could be a new operating system, or a different MCU (the latter is most likely). This document describes how the HAL driver can be ported to a new target platform and the behaviours of the functions.

The HAL driver is the interface between the stack and external environment. The driver implements access for interfacing the PTX2xxR/W, optional access to the peripherals of the RRQ35230EB, the available GPIO pins, and timers as well as some optional debugging functionality.

Driver implementations are found in their dedicated directories (for example, DRV/MCU/RA4M2 for the Renesas RA MCU driver). To start porting the HAL Driver, choose the driver that is closest to your target platform. In many cases for a new MCU, the Renesas driver is chosen as a basis. Copy all the files from that folder into a new folder. The new folder should have the name of the MCU for a better separation of the drive. Next, change the **CMakeLists.txt** or your e^2 Studio project, so that they can find the new driver folder and content.

Next, implement the driver. The file names suggest which interface they are associated with. A name SPI in the interface will use SPI, UART will use UART, and so on. Do not change the function names as they are fixed and function as the interface for the device. Instead, use them as a template. The next sections will detail and explain the functions and their roles.

6.3.2. Hardware Port Configuration

HAL Drivers can be implemented for more than one hardware configuration internally. This is useful if the same code should be reused for multiple hardware blocks in parallel. For this purpose, Port Configurations are used. These are statically defined internally used C arrays, which define the available hardware configurations. How this configuration looks depends on the specific driver implementation. The RUL SDK includes several port configurations. For the communication interfaces, there always must exist one configuration, which is used to interact with the PTX2xxR/W NFC hardware. The RUL SDK also contains other configurations, which are used, for example, for controlling peripherals for the PTX2xxR/W NFC hardware on the RRQ35230EB Board. Another use case for a different port configuration is to set a specific SPI HAL driver configuration to Mode-0 for the CS-pin, while another may require Mode-1 for the CS-pin, but still using the same underlying HAL Driver Code. This additional abstraction allows to combine different HAL configurations and settings by keeping the implementation effort and potentially memory requirements low.

It is not required to implement this feature, but if it is desired to have multiple independent instances of the RUL, this is a helpful feature. The Port Configuration is defined in all reference implementations with a specific type named for example **ptx_HAL_DRV_Interface_SPI_PortConfig_t** in case of the SPI Driver Implementation**.** It holds all information required to select a specific hardware (for example, specific SPI interface). For the Renesas RA4M2-MCU, the SPI reference implementation should look like this:

```
/**
 * \brief Port Configuration
 *
 * This structure contains all variables required to operate the SPI driver from the Renesas RA4M2 MCU.
 *
 * \note Integrators must initialise this structure according to their needs! A working example can be found in the
 * corresponding c file.
 */
typedef struct ptx_HAL_DRV_Interface_SPI_PortConfig
{
    ptx_HAL_DRV_Interface_Port_Type_t     Type;                /**< Implementation Type of the hardware interface port. */
    uint8_t                               ID;                  /**< ID of the hardware interface port implementation. */

    const spi_instance_t                  *Instance;           /**< SPI instance, generated by RENESAS Config Generator. */
    bsp_io_port_pin_t                     Mosi;                /**< SPI MOSI Pin, configured by RENESAS Config. */
    bsp_io_port_pin_t                     Clock;               /**< SPI Clock Pin, configured by RENESAS Config. */
    bsp_io_port_pin_t                     Nss;                 /**< SPI CS/NSS Pin, configured by RENESAS Config. */
    bsp_io_port_pin_t                     Irq;                 /**< SPI IRQ Pin, configured by RENESAS Config. */
    uint8_t                               Channel;             /**< SPI Channel, configured by RENESAS Config. */
    volatile uint8_t                      TransferState;       /**< Transfer State, required to determine if data is still in buffer.s */
    volatile int32_t                      TransferWaitCnt;     /**< Transfer Wait Counter, acts as a build in timeout to cancel transfers. */
    const external_irq_instance_t         *ExternalIrqInstance;/**< Global Interrupt Control Unit, configured by RENESAS Config. */
} ptx_HAL_DRV_Interface_SPI_PortConfig_t;
```

Figure 5. HAL Driver RA MCU SPI Reference Implementation Port Configuration Type

As shown, it is possible to set specific pins for the SPI protocol as well as the RA4M2 specific types to safe the MCU SPI instance. With these it is possible to configure specific hardware ports for all required specific use cases. There are two fields used to distinguish and identify different configurations in case of communication interfaces. These are the fields Type and ID. The first one must be always of the type **ptx_HAL_DRV_Interface_Port_Type_t** and it stores an Enum for the different use cases of the configurations. For the reference implementations there are three different types defined. The “Default” type will mostly be used to configure the main interface to the PTX2xxR/W. The “Debug” and “Evk” types are a special configurations for the RA4M2-MCU Driver Implementation. They are used to distinguish between the main “Default” configuration and secondary interfaces. It is not required to use the Type field, but it simplifies the identification and initialization of the right configuration instead of using only the ID field. This one only is of the type **uint8_t** and can be used to add more configurations of the same type. With this, it is for example possible to add two “Default” configurations with id 0 and 1 to control two separate PTX2xxR/W chips in parallel. In general, it is required to add these two fields to the initialization parameters as it is necessary for the HAL Interface, but if the feature of multiple Port Configurations is not needed, these can be ignored.

The actual static array of the RA4M2 reference implementation for SPI is defined as follows:

```
static ptx_HAL_DRV_Interface_SPI_PortConfig_t gDrvSpiPortConfigs[PTX_HAL_DRV_INTERFACE_SPI_NUMBER_PORTS] =
{
#ifndef PTX_ENABLE_BOARD_EVK
    {
        .Type                   = HAL_DRV_Interface_Port_Type_Default,
        .ID                     = 0,
        .Instance               = &ptx_pmod2_spi,
        .Mosi                   = BSP_IO_PORT_04_PIN_11,
        .Clock                  = BSP_IO_PORT_04_PIN_12,
        .Nss                    = BSP_IO_PORT_04_PIN_13,
        .Irq                    = BSP_IO_PORT_04_PIN_14,

        .Channel                = PTX_HAL_DRV_INTERFACE_SPI_DEFAULT_CHANNEL,
        .TransferState          = PTX_HAL_DRV_INTERFACE_SPI_RESET_VALUE,
        .TransferWaitCnt        = 0u,
        .ExternalIrqInstance    = &ptx_pmod2_irq,
    },
#else
    {
        .Type                   = HAL_DRV_Interface_Port_Type_Evk,
        .ID                     = 0,
        .Instance               = &ptx_pmod2_spi,
        .Mosi                   = BSP_IO_PORT_01_PIN_01,
        .Clock                  = BSP_IO_PORT_01_PIN_02,
        .Nss                    = BSP_IO_PORT_01_PIN_03,
        .Irq                    = BSP_IO_PORT_00_PIN_15,

        .Channel                = PTX_HAL_DRV_INTERFACE_SPI_EVK_CHANNEL,
        .TransferState          = PTX_HAL_DRV_INTERFACE_SPI_RESET_VALUE,
        .TransferWaitCnt        = 0u,
        .ExternalIrqInstance    = &ptx_pmod2_irq,
    }
#endif
};
```

Figure 6. HAL Driver RA MCU Reference Implementation Port Configuration

Here are two configurations implemented. The first one is of Type “Default” and the ID “0”. It uses different pins as the second one, which is of type “Evk” with ID “0”. The first one is used for the Custom e2 Studio Project included in the SDK and the second one for the RRQ35230EB board. Both are necessary to communicate with the PTX2xxR/W chips. Once such a Hardware Port Configuration is implemented, it must be used during initialization to route the correct configuration to the current instance according to the initialization parameters.

Apart from the communication interfaces it is also possible to implement Port Configurations for the other Driver modules in a similar way. For Timers it is for example possible to add multiple hardware timer configurations in a MCU to give the RUL and application access to multiple independent timers in parallel.

6.3.3. Required HAL Driver API

In this section, all required HAL Driver API functions and other types and definitions are described for each logical component.

6.3.3.1. Init, Open, Close, DeInit

Init, Open, Close, and DeInit functions exist for all driver implementation components. While Init and DeInit initialize and respectively de-initialize the memory used for the component, open and close functions open and close access to the logical component (for example, SPI Communication Interface).

6.3.3.2. HAL Communication SPI Driver API

The following sub-sections describe which functions need to be implemented, their roles for SPI, and their expected outcome. Additionally, the context **ptx_HAL_DRV_Interface_SPI_t** must be defined, which contains all required data for the context data of the driver and the initialization parameters **ptx_HAL_DRV_Interface_SPI_InitParams_t**, which hold the configuration of the driver for the Init function. How these types need to be designed can be obtained from the other reference implementations.

6.3.3.2.1. ptx_HAL_DRV_Interface_SPI_Tx

This function implements the transmit functionality. It should send out the data it received. If successful, the status SUCCESS is returned. If not, there is a return of an error code to the type of error that has occurred.

In some rare cases, it can be necessary to leave this function empty.

6.3.3.2.2. ptx_HAL_DRV_Interface_SPI_Rx

This function implements the receive functionality. It receives data and writes the data it received into the provided buffer. If successful, the status SUCCESS is returned. If not, there is a return of an error code to the type of error that has occurred.

In some rare cases, it can be necessary to leave this function empty.

6.3.3.2.3. ptx_HAL_DRV_Interface_SPI_SetNss

This function implements the setting of the NSS pin. This is a simple GPIO pin in the reference implementation. If the GPIO pin is set successfully, the function return SUCCESS. If not, there is a return of an error code to the type of error that has occurred.

In some rare cases, it can be necessary to leave this function empty.

6.3.3.2.4. ptx_HAL_DRV_Interface_SPI_ResetNss

This function implements the resetting of the NSS pin. This is a simple GPIO pin in the reference implementation. If the GPIO pin is set successfully, the function return SUCCESS. If not, there is a return of an error code to the type of error that has occurred.

In some rare cases, it can be necessary to leave this function empty.

6.3.3.2.5. ptx_HAL_DRV_Interface_SPI_RxIrqLevel

This function implements the reception of the current interrupt pin level. It is used to determine if the PTX2xxR/W requests some communication with the stack and has data available to be received. The function returns SUCCESS on successful completion. It further returns the current level of the pin as a parameter that needs to be provided to the function.

6.3.3.2.6. ptx_HAL_DRV_Interface_SPI_SetParameter

This function implements the possibility to change communication interface parameters after initialization.

6.3.3.2.7. ptx_HAL_DRV_Interface_SPI_GetParameter

This function implements the possibility to get the current value of communication interface parameters after initialization.

6.3.3.3. HAL Communication UART Driver API

The following sub-sections describe which functions need to be implemented, their roles for UART, and their expected outcome. Additionally, the context **ptx_HAL_DRV_Interface_UART_t** must be defined, which contains all required data for the context data of the driver and the initialization parameters **ptx_HAL_DRV_Interface_UART_InitParams_t**, which hold the configuration of the driver for the Init function. How these types need to be designed can be obtained from the other reference implementations.

6.3.3.3.1. ptx_HAL_DRV_Interface_UART_Tx

This function implements the transmit functionality. It sends out the data it received. If successful, the status SUCCESS is returned. If not, there is a return of an error code to the type of error that has occurred.

In some rare cases, it can be necessary to leave this function empty.

6.3.3.3.2. ptx_HAL_DRV_Interface_UART_Rx

This function implements the receive functionality. It receives data and writes the data it received into the provided buffer. If successful, the status SUCCESS is returned. If not, there is a return of an error code to the type of error that has occurred.

In some rare cases, it can be necessary to leave this function empty.

6.3.3.3.3. ptx_HAL_DRV_Interface_UART_RxIrqLevel

This function implements the reception of the current interrupt pin level. It is used to determine if the PTX2xxR/W requests some communication with the stack and has data available to be received. The function returns SUCCESS on successful completion. It further returns the current level of the pin as a parameter that needs to be provided to the function.

6.3.3.3.4. ptx_HAL_DRV_Interface_UART_SetParameter

This function implements the possibility to change communication interface parameters after initialization.

6.3.3.3.5. ptx_HAL_DRV_Interface_UART_GetParameter

This function implements the possibility to get the current value of communication interface parameters after initialization.

6.3.3.4. HAL Communication I²C Driver API

The following sub-sections describe which functions need to be implemented, their roles for I2C, and their expected outcome. Additionally, the context **ptx_HAL_DRV_Interface_I2C_t** must be defined, which contains all required data for the context data of the driver and the initialization parameters **ptx_HAL_DRV_Interface_I2C_InitParams_t**, which hold the configuration of the driver for the Init function. How these types need to be designed can be obtained from the other reference implementations.

6.3.3.4.1. ptx_HAL_DRV_Interface_I2C_Tx

This function implements the transmit functionality. It should send out the data it received. If successful, the status SUCCESS is returned. If not, there is a return of an error code to the type of error that has occurred.

In some rare cases, it can be necessary to leave this function empty.

6.3.3.4.2. ptx_HAL_DRV_Interface_I2C_Rx

This function implements the receive functionality. It should receive data and writes the data it received into the provided buffer. If successful, the status SUCCESS is returned. If not, there is a return of an error code to the type of error that has occurred.

In some rare cases, it can be necessary to leave this function empty.

6.3.3.4.3. ptx_HAL_DRV_Interface_I2C_RxIrqLevel

This function implements the reception of the current interrupt pin level. It is used to determine if the PTX2xxR/W requests some communication with the stack and has data available to be received. The function returns SUCCESS on successful completion. It further returns the current level of the pin as a parameter that needs to be provided to the function.

6.3.3.4.4. ptx_HAL_DRV_Interface_I2C_SetParameter

This function implements the possibility to change communication interface parameters after initialization.

6.3.3.4.5. ptx_HAL_DRV_Interface_I2C_GetParameter

This function implements the possibility to get the current value of communication interface parameters after initialization.

6.3.3.5. HAL Timer Driver API

The following sub-sections describe which functions need to be implemented, their roles for the timer, and their expected outcome. Additionally, the context **ptx_HAL_DRV_Tmr_t** must be defined, which contains all required data for the context data of the driver and the initialization parameters **ptx_HAL_DRV_Tmr_InitParams_t**, which hold the configuration of the driver for the Init function. How these types need to be designed can be obtained from the other reference implementations.

6.3.3.5.1. ptx_HAL_DRV_Tmr_Start

This function starts the timer as single-shot timer. The time it runs must be configured beforehand. This is usually done by the upper layers. The function returns immediately as it does not check if the timer has been received. The user needs to call the **IsElapsed** function to check if the timer has already expired. The function returns SUCCESS if the timer is activated properly. If not, it should return an error code according to the error.

6.3.3.5.2. ptx_HAL_DRV_Tmr_Stop

This function stops the timer that has been previously started. It returns immediately and does not check if the timer has elapsed. The user needs to call the **IsElapsed** function to check if the timer has already expired. The function returns SUCCESS if the timer is activated properly. If not, it should return an error according to the error.

6.3.3.5.3. ptx_HAL_DRV_Tmr_IsElapsed

This function checks if the timer is already elapsed. It receives a parameter to write to if the timer has relapsed. The function returns SUCCES if the state of the timer is read properly. If not, it should return an error according to the error.

6.3.3.5.4. ptx_HAL_DRV_Tmr_RegExecCb

This function allows registering a callback for the timers ISR. The function must meet the function definition of the callbacks prototype. If a timer interrupt is received, the callback will be executed. The function returns SUCCESS if the callback is successfully registered or an error according to what failed.

6.3.3.5.5. ptx_HAL_DRV_Tmr_Cfg

This function configures the timer. It receives an array of configuration options. The array can have the period the timer runs, as well as its time unit. Further, a duty cycle and duty cycle pin can be configured. The function then iterates over the array and sets the config parameters accordingly. The function returns SUCCESS if all parameters were set successfully or an error according to what failed.

6.3.3.5.6. ptx_HAL_DRV_Tmr_Sleep

This function puts the system to sleep for a period of microseconds. It configures the timer and starts and stops it once it is elapsed. The function returns SUCCESS if the timer elapses and no errors occur during the configuration. Otherwise, the proper error code is returned.

6.3.3.6. HAL GPIO Driver API

The following sub-sections describe which functions need to be implemented, their roles for GPIO, and their expected outcome. Additionally, the context **ptx_HAL_DRV_GPIO_t** must be defined, which contains all required data for the context data of the driver and the initialization parameters **ptx_HAL_DRV_GPIO_InitParams_t**, which hold the configuration of the driver for the Init function. How these types need to be designed can be obtained from the other reference implementations.

6.3.3.6.1. ptx_HAL_DRV_GPIO_ReadPin

This function reads data from a single pin and provides the logic level as 0 for LOW and 1 for HIGH as output to the function. The function returns SUCCESS if the pin is read without errors or an error code describing what failed. The function should use the GPIO device and read the pin.

6.3.3.6.2. ptx_HAL_DRV_GPIO_WritePin

This function writes to a single pin. Logic levels are 0 for LOW and 1 for HIGH. The function returns SUCCESS if the pin was written without errors or an error code describing what failed. The function should use the GPIO device and write the pin.

6.3.3.6.3. ptx_HAL_DRV_GPIO_ReadPort

This function reads all pins from a single port. Logic levels are 0 for LOW and 1 for HIGH as output to the function. The function returns SUCCESS if the port was read without errors or an error code describing what failed. The function should use the GPIO device and read the port.

6.3.3.6.4. ptx_HAL_DRV_GPIO_WritePort

This function writes all pins from a single port. Logic levels are 0 for LOW and 1 for HIGH as input to the function. The function returns SUCCESS if the port was read without errors or an error code describing what failed. The function should use the GPIO device and write the port.

6.3.3.6.5. ptx_HAL_DRV_GPIO_SetCfgPin

This function configures the GPIO pin. It receives an array of configuration options. The pin can be configured as an input/output pin, high impedance, etc. The function then iterates over the array and sets the config parameters accordingly. The function returns SUCCESS if all parameters were set successfully or an error according to what failed.

6.3.3.7. HAL MCU Driver Functions

The following sub-sections describe which functions need to be implemented, their roles for the MCU, and their expected outcome.

6.3.3.7.1. ptx_HAL_DRV_MCU_EnableIrq

This function enables IRQs globally on the MCU. It returns SUCCESS if all interrupts are enabled.

If you are using an operating system like Linux or Windows, this function will be empty and just return SUCCESS.

6.3.3.7.2. ptx_HAL_DRV_MCU_DisableIrq

This function disables IRQs globally on the MCU. It returns SUCCESS if all interrupts are disabled.

If you are using an operating system like Linux or Windows, this function will be empty and just return SUCCESS.

6.3.3.7.3. ptx_HAL_DRV_MCU_WaitForIrq

This function waits for IRQs. It sets the MCU into a deep-sleep-mode and wakes it up if an interrupt is detected. It returns SUCCESS if all interrupts are disabled.

If you are using an operating system like Linux or Windows, this function will be empty and just return SUCCESS.

6.4 Logging Mechanism (LOG)

6.4.1. Introduction

The RUL SDK implements a light-weight logging component called “**LOG**”, which can be used for:

- Debugging
- Catching certain events like unexpected errors
- Tracing the program execution flow
- Getting the contents of data / memory blocks at a specific time at runtime.

The provided LOG component is dependent on the third-party library “**zf_log**” which is publicly available on GitHub under the MIT license. The necessary source files for the “**zf_log**”-component are contained in the RUL SDK package.

6.4.2. LOG Architecture

The LOG API is a core component of the RUL SDK but can be excluded if needed (see below).

Figure 7 shows the structure of the LOG component within the RUL SDK.

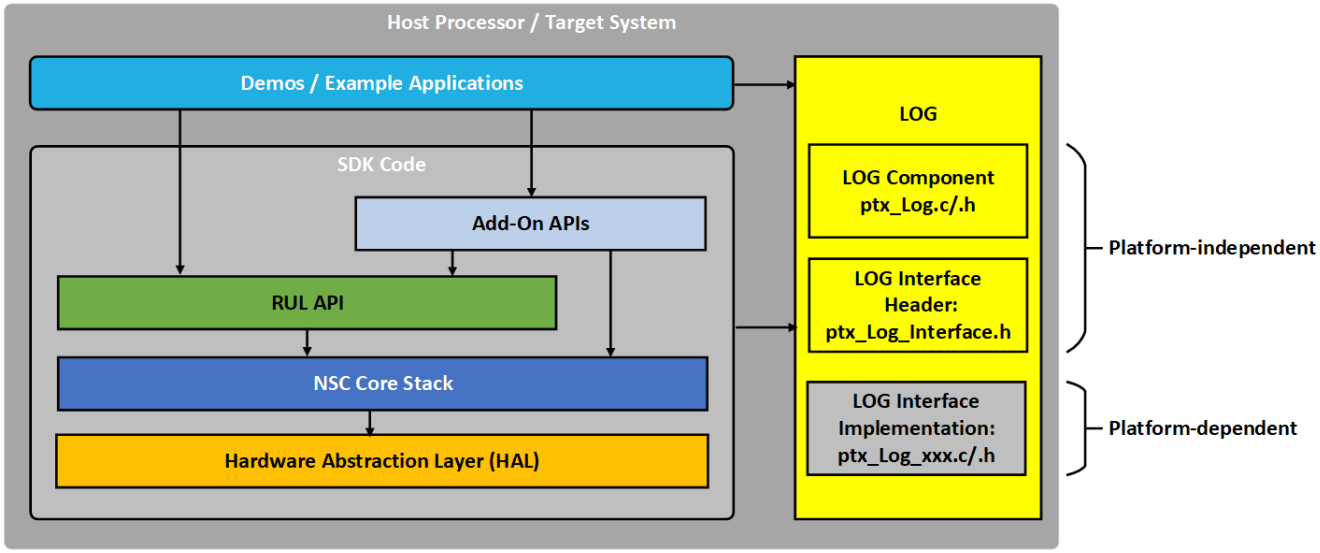

Figure 7. LOG Component Architecture

The LOG component itself gets internally initialized during the call to **ptx_RUL_Init**. After initialization, the LOG entries can be written from anywhere within the code base wherever using components have access to a LOG component reference.

The LOG component itself consists of a platform-independent and platform-dependent section which is structured as described below:

- **Platform-independent**
  - **Core LOG Component**

    The core LOG component contains API functions for initialization and de-initialization of the internal modules and exposes various macros to write actual LOG entries (see below).
  - **LOG Interface Header**

    The LOG Interface Header defines three API functions which must be implemented by a solid Interface implementation located in the platform-dependent section.

    The three API functions to be implemented are:
    - **ptx_Log_Interface_Open**

      Called by Core Log component to perform the custom initialization.

- **ptx_Log_Interface_Close**
  Called by Core Log component to perform the custom de-initialization.
- **ptx_Log_Interface_WriteCB**
  Called by the macros exposed via the Core Log component to write actual LOG entries.

- **Platform-dependent**

  This part contains the concrete LOG implementation which implements at least the three API functions defined in the LOG Interface Header.

  As previously described, LOG entries can be written using the macros exposed via **ptx_Log.h**. The macros also support different logging-levels as shown below:

  - **PTX_LOG_I**
    Writes a given string as LOG entry using logging-level "**INFO**".
  - **PTX_LOG_V**
    Writes a given string as LOG entry using logging-level "**VERBOSE**".
  - **PTX_LOG_D**
    Writes a given string as LOG entry using logging-level "**DEBUG**".
  - **PTX_LOG_W**
    Writes a given string as LOG entry using logging-level "**WARNING**".
  - **PTX_LOG_E**
    Writes a given string as LOG entry using logging-level "**ERROR**".
  - **PTX_LOG_F**
    Writes a given string as LOG entry using logging-level "**FATAL**".
  - **PTX_LOG_I_MEM**
    Writes a given data / memory block and a string as LOG entry using logging-level "**INFO**".
  - **PTX_LOG_V_MEM**
    Writes a given data / memory block and a string as LOG entry using logging-level "**VERBOSE**".
  - **PTX_LOG_D_MEM**
    Writes a given data / memory block and a string as LOG entry using logging-level "**DEBUG**".

All macros based on the provided reference implementations (see below) produce the following output format:

"Logging-Level Character" | "Function-Name"@"SourceFile":"Line-Number" "LOG-Message"

Example: For an INFO LOG message in line 8 with text "This is an Info-Message" called via macro "**PTX_LOG_I**" from function "**main()**" in file "**example.c**", it would appear as follows:

```
I main.example.c:8 This is an Info-Message
```

The output message is highly customable by:

- Changing the preprocessor definition "**LOG_PUT_MASK**" in "**ptx_Log.c**"
- Adapting the implementation of function "**ptx_Log_MsgTrx**" in "**ptx_Log.c**"
- Customizing function "**ptx_Log_Interface_WriteCB"** in the platform-dependent part (for example, adding LOG entry counter, etc.).

Note: The LOG component can be excluded from a target-build. Excluding the LOG component also removes all macro-usages and thus can save certain code space.

6.4.3. Reference LOG Implementations

The RUL SDK already contains a collection of various LOG reference implementations which can be used out of the box. These reference implementations can be used as an example for further custom implementations or can be adapted if needed.

6.4.3.1. Console Reference LOG Implementation

This reference implementation can be found in the SDK folder **LOG\CONSOLE** and writes LOG entries directly onto the console of the system using standard "**printf**" operation. This implementation is suited for Desktop OS platforms such as Windows and Linux.

6.4.3.2. File Reference LOG Implementation

This reference implementation can be found in the SDK folder **LOG\FILE** and writes LOG entries directly into a text-file of the system.

The implementation supports two writing modes "**cached**" and "**direct**" which can be selected at compile-time via preprocessor define "**PTX_LOG_FILE_DIRECT_WRITE_MODE**".

- "**Cached**"-mode writes every LOG entry first into an internal cache and gets finally written into the text file when "**ptx_Log_DeInit**" gets called. The size of the cache as well as the number of characters per LOG entry is configurable in file "**ptx_Log_File.h**" by adapting the pre-processor defines:

  PTX_LOG_FILE_NR_LOG_CACHE_ENTRIES and **PTX_LOG_FILE_ENTRY_MAX_LENGTH**.

- "**Direct**"-mode writes every LOG entry immediately into the text file. There is also no limitation in terms of maximum LOG entries (in other words, cache size or the number of characters per line).

  Important: "Direct"-mode can be also seen as "Real-time Logging" as each entry gets directly written into (in other words, appended) the text file and can therefore significantly decrease system performance. It depends on the use-case / scenario of the target-integration which writing mode to use.

This implementation is suited for Desktop OS platforms such as Windows and Linux.

6.4.3.3. Serial / UART Reference LOG Implementation

This reference implementation can be found in the SDK folder **LOG\SERIAL** and writes LOG entries directly via the serial port / UART of the reference Renesas RA4M2 MCU (UART 0).

Note: This implementation is suited only for the Renesas RA4M2 MCU.

6.5 Integration Notes and Hints

6.5.1. Endianness

Parameters with multiple bytes use the Most Significant Byte (MSB) first format. The RUL SDK is delivered by default for systems operating on little-endianness (for example, commonly used by ARM-Cortex-based systems). Users who implement the RUL-SDK on a system using big-endianness must change the macro PTX_HIP_BSWAP16(x)"within **SRC/COMPS/HIP/ptx_HIP_Int.h**

From:

```
#define PTX_HIP_BSWAP16(x) x = __builtin_bswap16(x)
```

To:

```
#define PTX_HIP_BSWAP16(x) x = (x)
```

6.5.2. FTDI DLL-Path

The HAL reference implementation for Windows contained in the SDK is based on the FTDI FT232H USB-converter IC (https://ftdichip.com/) and implements a simple library to interface FTDI HW-modules via the FTDI DLL “ftd2xx64.dll”. The implementation of the simple library itself can be found in file “ptx_HAL_Ftd2xx.c”.

The implementation uses a hard-coded path to the FTDI DLL (see line 63 in file “ptx_HAL_Ftd2xx.c” below) which needs to be adapted in case the reference implementation gets used by a final application.

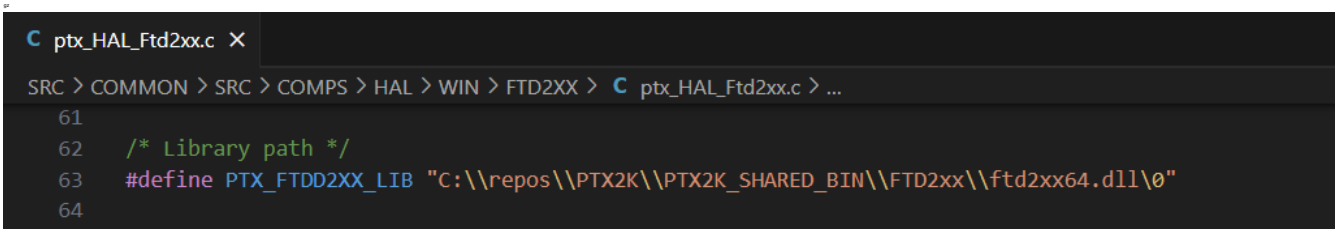

Figure 8. FTDI-DLL Path

6.5.3. FTDI Hardware Pin Connections

The HAL reference implementation for Windows contained in the SDK is based on the FTDI FT232H USB-converter IC (https://ftdichip.com/). To connect PTX2K to the FTDI IC, connect the HIF pins according to the selected interface type. In case the reference implementation is used, the SEN pin is connected to pin 25 / ACBUS1 of the FTDI IC. The interrupt on HIF5 is connected to pin 21 / ACBUS0.

Due to hardware constraints of the used FTDI IC, the pins are different for the Windows UART reference implementation. The SEN pin is connected to pin 15 / ADBUS2 and the IRQ on HIF5 is connected to pin 16 / ADBUS3.

6.5.4. FTDI Pull-Down Resistors for Standby Mode

The HAL reference implementation for Windows contained in the SDK is based on the FTDI FT232H USB-converter IC (https://ftdichip.com/).

If the final application is also using the same FTDI IC, pin 25 / ACBUS1 needs to be pulled-down to have the correct behavior for waking up by HIF in Standby Mode, as the FTDI does not have the capabilities to pull the pin low itself.

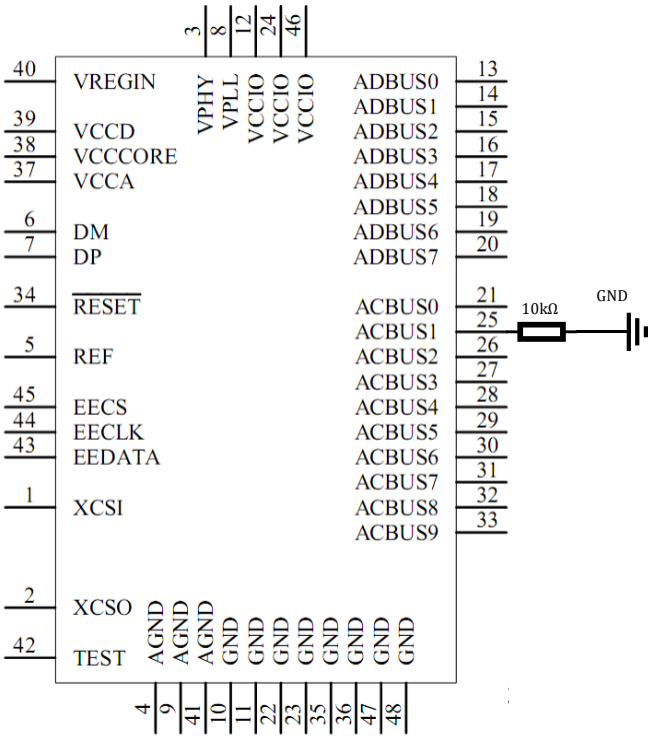

Figure 9. FTDI Pull-Down Resistor for Standby Mode

6.5.4.1. UART Reference Implementation

The used FT232H chip has limitations regarding the use of GPIO pins while in USB-serial converter mode. Therefore, the UART reference implementation using this IC needs pin 16 / ADBUS3 to be pulled down to correctly read the IRQ pin.

6.5.5. Raspberry Pi 4 Hardware Pin Configuration

The HAL reference implementation for Linux contained in the SDK is based on the Raspberry Pi 4. To connect the PTX2K to the Raspberry Pi, connect the HIF pins according to the selected interface type. In case the reference implementation is used, the SEN pin is connected to pin 40 (GPIO 21). The interrupt on HIF5 is connected to pin 35 (GPIO 19).

6.5.6. Raspberry Pi 4 UART Remapping

The HAL reference implementation for Linux contained in the SDK is based on the Raspberry Pi 4. To ensure the correct operation, it is necessary to have at least a one-bit delay between HIP commands. The default UART converter on the Raspberry Pi 4 does not support this setting. Therefore, applying the following setup commands might be necessary. Please note that this will disable the Bluetooth functionality of the Raspberry Pi.

First, enable the UART port on the Raspberry Pi by opening the configuration menu.

```
1  $ sudo raspi-config
```

Select “Interface Options” > “Serial Port” and select “No” for the login shell and “Yes” for the serial port.

The following commands will remap the UART converter of the Bluetooth module to the GPIO pins. This is necessary to get the required functionality.

```
1 $ sudo systemctl disable hciuart
2 $ sudo systemctl disable Bluetooth
3 $ sudo systemctl stop hciuart
4 $ sudo systemctl stop Bluetooth
```

Next add the following line into either one of these files “/boot/config.txt” or “/boot/firmware/config.txt”.

```
1 dtoverlay=disable-bt
```

Lastly, reboot the Raspberry PI.

6.5.7. HAL Driver Default Communication Speeds

The provided HAL driver reference implementations for the various communication interfaces such as SPI, I2C, UART etc. use default communication speeds based on C-preprocessor constants defined in the corresponding reference driver Header-file(s).
Examples:

- PTX_HAL_DRV_INTERFACE_SPI_DEFAULT_SPEED // Default Value: 1MHz
- PTX_HAL_DRV_INTERFACE_I2C_SPEED_DEFAULT // Default Value: 100kHz
- PTX_HAL_DRV_INTERFACE_UART_DEFAULT_BAUDRATE // Default Value: 115.2kBaud

To change the default communication speed, the default values can be changed accordingly at Compile-time (see constants above) or at Runtime via a call to **ptx_HAL_Interface_SetParameter**.

Some toolchains also offer the possibility to change the default communication speed for a communication interface in a graphical manner (toolchain-specific). In such cases a newly implemented / ported HAL Driver component may only use rely on these values and replace the usage of constants in the code.

6.5.8. RRQ35230EB Board

To use the RRQ35230EB board, it is required to control additional hardware components on the PCB. For this purpose, is the EVK_Peripherals API available, it grants a set of functions and types to control the available components on the board. Available are DC/DC converter, power sensor and temperature sensor control implementations. These are required for the RRQ35230EB board but can also be used as reference implementations for a custom design to add other peripheral hardware components, which need to be controllable in the application. By setting the compile switch PTX_ENABLE_BOARD_EVK it is possible to activate the support for the RRQ35230EB board in general. Be aware, that this will not only activate the support for the EVK_Peripherals API but also switch the HAL Driver Port Config to the correct implementation of the RA4M2 Pins.

Attention: Depending on the various versions of the RRQ35230EB board, the I^2C address for the EVK peripheral component needs to be adapted accordingly as shown below.

- r250305: MP8862(0x69)
- r250520: MP8862(0x69)
- A4: MP8859(0x60)

The I^2C address for the RRQ35230EB board is defined via **PTX_EVK_PERIPHERALS_DCDC_I2C_ADDRESS** in file located at **SRC/API/EVK_PERIPHERALS/ptx_EVK_Peripherals_DCDC.h**.

6.5.9. Reference SDK Module Flash / RAM Consumption

Table 2 provides an overview of the Flash and RAM consumption of the various SW components of the RUL SDK when compiled with **ARM-GCC 13.2.rel1** and optimization level “RELEASE” for the Renesas RA4M2 MCU.

Table 2. RUL SDK Module Flash/RAM Consumption (ARM-GCC 13.2.rel1, RELEASE)

Module	Description	Flash [kB]	Data [kB]	BSS [kB]
FELICA_DTE	FeliCa DTE Add-On Library	1,63	0,00	0,00
HAL	Hardware Abstraction Layer Top-Level Library (without reference implementations)	3,02	0,00	0,13
HIP	Host Interface Protocol Library	1,63	0,00	0,00
HIP_FRAME	Host Interface Protocol Frame Library Encoder / Decoder Module	0,80	0,00	0,27
LLCP	Logical Link Control Protocol Add-on Library	9,66	0,00	0,00
LLCP_FRAME	Logical Link Control Protocol Add-on Library Frame Encoder / Decoder Module	3,38	0,00	0,25
LOG	Logger Top-Level Library (without reference implementations)	2,21	0,02	0,06
MFCC	Mifare Classic Compatibility Mode Add-on Library	2,66	0,00	0,00
NDEF	General NDEF Type Tag Operation Add-on Library	0,66	0,00	0,00
NDEF_RECORD	NDEF Record / Message Helper Library	4,85	0,00	0,00
NDEF_T2T	T2T NDEF Type Tag Operation Add-on Library	6,19	0,00	0,00
NDEF_T3T	T3T NDEF Type Tag Operation Add-on Library	2,57	0,00	0,00
NDEF_T4T	T4T NDEF Type Tag Operation Add-on Library	1,86	0,00	0,00
NDEF_T5T	T5T NDEF Type Tag Operation Add-on Library	3,31	0,00	0,00

Module	Description	Flash [kB]	Data [kB]	BSS [kB]
NFCForumDTA	NFC Forum DTA Top-Level Add-on Library	1,83	0,00	0,00
NFCForumLlcpProcessing	NFC Forum DTA LLCP Processing Module	1,67	0,00	0,00
NFCForumSnepProcessing	NFC Forum DTA SNEP Processing Module	0,89	0,00	0,00
NFCForumTagGeneric	NFC Forum DTA Generic Tag Processing Module	2,13	0,00	0,00
NFCForumTagProcessing	NFC Forum DTA Specific Tag Processing Module	3,74	0,00	0,00
NSC	Renesas “NFC Soft Controller” Library	18,18	0,69	0,25
POS_DTE	EMV / POS DTE Add-on Library	4,02	0,00	0,26
PSL	Product Support Add-on Library	6,81	0,00	0,00
RF_TEST	RF-Test Add-on Library	0,18	0,00	0,00
RUL	Reader Universal Library API	9,34	0,00	0,00
RUL_HCE	Reader Universal Library API – HCE Extension	1,40	0,00	0,00
EVK_PERIPHERALS	Reader Universal Library API – EVK Peripherals Extension	2,79	0,00	0,00
RUL_WLC	Reader Universal Library API – WLC Extension	5,67	0,00	0,00
SNEP_CLIENT	Simple NDEF Exchange Protocol Add-on Library / Client	1,78	0,00	0,00
SNEP_SERVER	Simple NDEF Exchange Protocol Add-on Library / Server	1,69	0,00	0,00
T2T	T2T Native Tag Add-on API	0,42	0,00	0,00
T3T	T3T Native Tag Add-on API	0,80	0,00	0,00
T4T	T4T Native Tag Add-on API	1,63	0,00	0,00
T5T	T5T Native Tag Add-on API	1,71	0,00	0,00
TRANSPARENT_MODE	Transparent Mode Add-on API	1,14	0,00	0,00
UCODE	NSC Firmware Image	22,38	0,02	0,00

***Important**:* The values listed in the Table 2 are reference values and are depending on the target system integration, build/environment properties, such as:

- Target platform / MCU architecture
- Compilers/compiler versions and settings (for example, optimization level)
- Included/excluded SDK modules (for example, optional modules)
- SDK configuration (for example, maximum number of supported cards)
- Tested use case/scenario

These properties can have a large impact on the output and the numbers can vary significantly.

7. Add-On Libraries/APIs

In addition to the RUL API, there are additional support libraries that extend the functionality of the RUL component at the application layer. The use of these libraries is optional and they can be included or excluded from the build as needed.

The current SDK contains the following add-on libraries:

- Native Tag API for NFC Forum Type Tags 2 – 5 (T2T, T3T, T4T, T5T) & MFCC API
- NDEF API for NFC Forum Type Tags 2 – 5 (T2T, T3T, T4T, T5T)
- POS DTE API (POS Device Test Environment)
- Transparent Mode API
- RF Test API
- NFC Forum DTA API
- FeliCa DTE API
- LLCP API
- SNEP API

The following sections contain the descriptions of these libraries.

7.1 Native Tag API

The Native Tag API implements the native command set for each of the NFC Forum Tag Types and also allows the sets to be further extended if required (for example, manufacturer product specific command set).

7.1.1. API Architecture

The Native Tag API module is provided as stand-alone functionality of the RUL SDK. It is located on top of the RUL API module as shown in Figure 10 and internally accesses also the NSC Core Stack module.

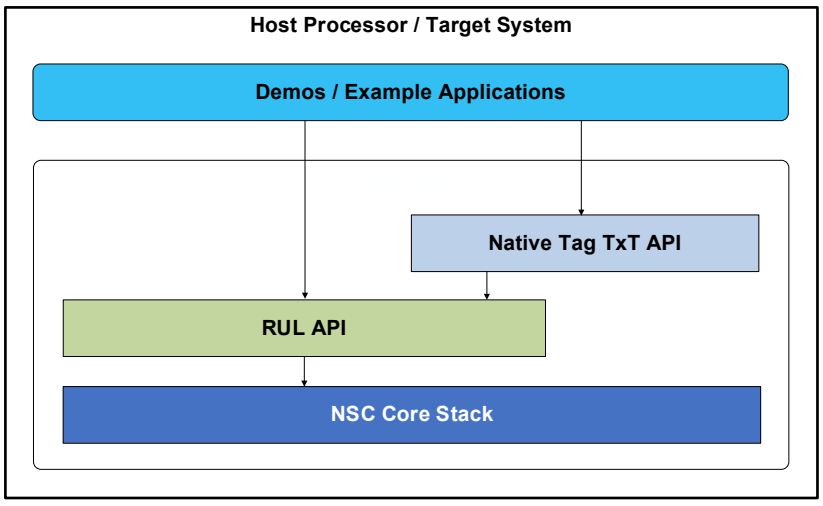

Figure 10. Native Tag API Component Architecture

The Native Tag API is built upon the RUL API and internally uses the **ptx_RUL_Exchange** function to exchange RF data with the corresponding Tag.

The Native Tag APIs all follow the same approach. To initialize any of the components a call to the specific Open function is required and the matching Close function is necessary to free potentially allocated resources once the component is not used anymore. While the component is initialized, all other functions can be used to send Tag specific commands.

Note: While the component initialization can also take place at the beginning (for example, after the stack initialization), the actual Tag specific commands can only be sent in the API state “Data Exchange”. For some protocols that use an ID as part of the command-packet (for example, T3T or T5T), it may be necessary to set the ID (of the active Tag) once when the API state **Data Exchange** is entered. The ID itself can be set via a dedicated function provided by the corresponding API.

7.1.2. Native Tag Type 2 Tag API

The Native Tag API for the Type 2 Tags provides an interface to execute the Native Tag command set for Type 2 Tags as defined in [NFC T2T].

7.1.2.1. Native Tag T2T API

The Native Tag T2T API supports the following commands defined in [NFC T2T]:

- Read
- Write
- Sector Select

7.1.2.1.1. ptx_NativeTag_T2T_Open

The **Open** function initializes the T2T Native Tag component. It requires an initialized instance of a RUL component a buffer which can be used internally and the size of this buffer. These parameters are provided using the **ptx_NativeTag_T2T_InitParams_t** struct.

Declaration	`ptx_Status_t` **`ptx_NativeTag_T2T_Open`** `(` **`ptx_NativeTag_T2T_t`** `*t2t,` **`ptx_NativeTag_T2T_InitParams_t`** `*initParams);`	
Description	Initialization of the Native Tag T2T component.	
Parameters	t2t	Pointer to allocated Native Tag T2T component.
	initParams	Pointer to initialization parameter structure.
Return Value	Status, indicating whether the operation was successful.	

7.1.2.1.2. ptx_NativeTag_T2T_Close

The **Close** function de-initializes the T2T Native Tag component.

Declaration	`ptx_Status_t` **`ptx_NativeTag_T2T_Close`** `(` **`ptx_NativeTag_T2T_t`** `*t2t);`	
Description	De-Initialization of the Native Tag T2T-component.	
Parameters	t2t	Pointer to initialized Native Tag T2T component.
Return Value	Status, indicating whether the operation was successful.	

7.1.2.1.3. ptx_NativeTag_T2T_Read

The **Read** function is used to read a single block from the Type 2 Tag. The function reads the block at the given block number of the current sector into the provided buffer.

Declaration	`ptx_Status_t` **`ptx_NativeTag_T2T_Read`** (**`ptx_NativeTag_T2T_t`** *t2t, **`uint8_t`** blockNr, **`uint8_t`** *rx, **`uint32_t`** *rxLen, **`uint32_t`** msTimeout);	
Description	Read T2T Blocks	
Parameters	t2t	Pointer to initialized Native Tag T2T component.
	blockNr	Block number to be read (within sector).
	rx	Pointer to Rx-buffer where the received data from the card should be stored.
	rxLen	Size of Rx-buffer (in), Length of the received data (out).
	msTimeout	Timeout in ms that the function is going to wait for receiving data from the card.
Return Value	Status, indicating whether the operation was successful.	

7.1.2.1.4. ptx_NativeTag_T2T_Write

The **Write** command is used to write to the Type 2 Tag. The function writes the provided data to the chosen block of the current sector to the tag.

Declaration	`ptx_Status_t` **`ptx_NativeTag_T2T_Write`** (**`ptx_NativeTag_T2T_t`** *t2t, **`uint8_t`** blockNr, **`uint8_t`** *blockData, **`uint8_t`** blockDataLen, **`uint8_t`** *rx, **`uint32_t`** *rxLen, **`uint32_t`** msTimeout);	
Description	Write to the T2T blocks.	
Parameters	t2t	Pointer to initialized Native Tag T2T component.
	blockNr	Block number to be read (within sector).
	blockData	Data to write.
	blockDataLen	Length of data to write.
	rx	Pointer to Rx-buffer where the received data from the card should be stored.
	rxLen	Size of Rx-buffer (in), Length of the received data (out).
	msTimeout	Timeout in ms that the function is going to wait for receiving data from the card.
Return Value	Status, indicating whether the operation was successful.	

7.1.2.1.5. ptx_NativeTag_T2T_SectorSelect

The **SectorSelect** command is used to change the current sector of the Type 2 Tag. The default sector is 0, but if another sector needs to be read or written to, this function can be used to change the chosen sector.

Declaration	`ptx_Status_t ptx_NativeTag_T2T_SectorSelect (`
	`  ptx_NativeTag_T2T_t *t2t,`
	`  uint8_t secNr,`
	`  uint8_t *rx,`
	`  uint32_t *rxLen,`
	`  uint32_t msTimeout);`	

Description	Select the T2T sector.	
Parameters	t2t	Pointer to initialized Native Tag T2T component.
	secNr	Sector number to be used (default sector is 0).
	rx	Pointer to Rx-buffer where the received data from the card should be stored.
	rxLen	Size of Rx-buffer (in), Length of the received data (out).
	msTimeout	Timeout in ms that the function is going to wait for receiving data from the card.
Return Value	Status, indicating whether the operation was successful.	

7.1.3. Native Tag Type 3 Tag API

The Native Tag API for the Type 3 Tags provides an interface to execute the Native Tag command set for Type 3 Tags as defined in [NFC T3T]

7.1.3.1. Native Tag T3T API

The Native Tag T3T API supports the following commands defined in [NFC T3T]:

- SENSF_REQ
- Check
- Update

7.1.3.1.1. ptx_NativeTag_T3T_Open

The **Open** function initializes the T3T Native Tag component. It requires an initialized instance of a RUL component a buffer which can be used internally the size of this buffer, the ID and length of the ID of the Tag if it is known at this point as well as the timing calculation parameters (MRTICheck and MRTIUpdate) if known at this point. These parameters are provided using the **ptx_NativeTag_T3T_InitParams_t** struct.

Note: The Tag ID, length of the ID and the MRTI times can be updated once they are accessible using the **ptx_NativeTag_T3T_SetTagParams** function. They don't have to be set during initialization if they aren't available.

Declaration	`ptx_Status_t` **`ptx_NativeTag_T3T_Open`** `(`
	`  ` **`ptx_NativeTag_T3T_t`** `*t3t,`
	`  ` **`ptx_NativeTag_T3T_InitParams_t`** `*initParams);`	

Description	Initialization of the Native Tag T3T component.	
Parameters	t3t	Pointer to allocated Native Tag T3T component.
	initParams	Pointer to initialization parameter structure.
Return Value	Status, indicating whether the operation was successful.	

7.1.3.1.2. ptx_NativeTag_T3T_Close

The **Close** function de-initializes the T3T Native Tag component.

Declaration	`ptx_Status_t` **`ptx_NativeTag_T3T_Close`** (**`ptx_NativeTag_T3T_t`** `*t3t);`	
Description	De-Initialization of the Native Tag T3T component.	
Parameters	t3t	Pointer to initialized Native Tag T3T component.
Return Value	Status, indicating whether the operation was successful.	

7.1.3.1.3. ptx_NativeTag_T3T_SENSF_REQ

The **SENSF_REQ** function performs a SENSF_REQ as described in [NFC Digital]. The response (SENSF_RES) acquired by this function can be used to extract the ID and timing parameters (MRTI-Check and MRTI-Update) of the tag.

Note: On successful execution, the ID of the tag can be found in the response from byte 2 to 9 and the timing parameters can be found in bytes 15 (MRTI – Check) 16 (MRTI – Update).

Declaration	`ptx_Status_t` **`ptx_NativeTag_T3T_SENSF_REQ`** (**`ptx_NativeTag_T3T_t`** `*t3t,` **`uint16_t`** `sc,` **`uint8_t`** `rc,` **`uint8_t`** `tsn,` **`uint8_t`** `*rx,` **`uint32_t`** `*rxLen);`	
Description	T3T SENSF_REQ Command	
Parameters	t3t	Pointer to initialized Native Tag T3T component.
	sc	System code of the NFC forum device to be polled for.
	rc	Request code for additional information in the SENSF_RES.
	tsn	Time slot number, used for collision resolution.
	rx	Pointer to Rx-buffer where the received data from the card should be stored.
	rxLen	Size of Rx-buffer (in), Length of the received data (out).
Return Value	Status, indicating whether the operation was successful.	

7.1.3.1.4. ptx_NativeTag_T3T_Check

The **Check** function performs a read operation on the specified block as described in [NFC T3T]. The function reads the blocks given by **blockInfo** of the service defined by **serviceInfo**.

Note: Both the **blockInfo** and the **serviceInfo** are provided to the function with the specific structures. One Block/Service is always defined with two bytes. Two bytes of data must be provided for every block/service selected.

Declaration	`ptx_Status_t` **`ptx_NativeTag_T3T_Check`** (**`ptx_NativeTag_T3T_t`** *t3t, **`ptx_NativeTag_T3T_Services_t`** serviceInfo, **`ptx_NativeTag_T3T_Blocks_t`** blockInfo, **`uint8_t`** *rx, **`uint32_t`** *rxLen, **`uint32_t`** msTimeout)	
Description	T3T Check Command.	
Parameters	t3t	Pointer to initialized Native Tag T3T component.
	serviceInfo	Information about Services to be checked.
	blockInfo	Information about Blocks to be checked.
	rx	Pointer to Rx-buffer where the received data from the card should be stored.
	rxLen	Size of Rx-buffer (in), Length of the received data (out).
	msTimeout	Timeout in ms that the function is going to wait for receiving data from the card.
Return Value	Status, indicating whether the operation was successful.	

7.1.3.1.5. ptx_NativeTag_T3T_Update

The **Update** function performs a write operation on the specified block as described in [NFC T3T]. The function writes the provided data to the blocks given by **blockInfo** of the service defined by **serviceInfo**.

Note: Both the **blockInfo** and the **serviceInfo** are provided to the function with the specific structures. One Block/Service is always defined with two bytes. Two bytes of data must be provided for every block/service selected.

Declaration	`ptx_Status_t` **`ptx_NativeTag_T3T_Update`** (**`ptx_NativeTag_T3T_t`** *t3t, **`ptx_NativeTag_T3T_Services_t`** serviceInfo, **`ptx_NativeTag_T3T_Blocks_t`** blockInfo, **`uint8_t`** *blockData, **`uint8_t`** blockDataLen, **`uint8_t`** *rx, **`uint32_t`** *rxLen, **`uint32_t`** msTimeout)	
Description	T3T Check Command.	
Parameters	t3t	Pointer to initialized Native Tag T3T component.
	serviceInfo	Information about Services to be updated.
	blockInfo	Information about Blocks to be updated.
	blockData	Data to be written to the blocks.
	blockDataLen	Length of the data to be written to the blocks.
	rx	Pointer to Rx-buffer where the received data from the card should be stored.
	rxLen	Size of Rx-buffer (in), Length of the received data (out).
	msTimeout	Timeout in ms that the function is going to wait for receiving data from the card.
Return Value	Status, indicating whether the operation was successful.	

7.1.3.1.6. ptx_NativeTag_T3T_SetTagParams

The **SetTagParams** function is used to update the Tag parameters to be used internally. Specifically, the following settings can be updated: The ID of the tag to be communicated with, the length of the respective ID, the maximum number of blocks that can be used per read or write operation and the timing parameters (MRTI – Check and MRTI – Update) of the tag.

Declaration	`ptx_Status_t` **`ptx_NativeTag_T3T_SetTagParams`** (**`ptx_NativeTag_T3T_t`** *t3t, **`uint8_t`** *NFCID2, **`uint8_t`** NFCID2Len, **`uint8_t`** nbr, **`uint8_t`** nbw, **`ptx_NativeTag_T3T_MRTI_t`** mrtiInfo);	
Description	T3T Check Command.	
Parameters	t3t	Pointer to initialized Native Tag T3T component.
	nbr	Number of blocks per read operation.
	nbw	Number of blocks per write operation.
	NFCID2	ID to address the T3T tag.
	NFCID2Len	Length of ID.
	mrtiInfo	Information about MRTI Times of the T3T.
Return Value	Status, indicating whether the operation was successful.	

7.1.4. Native Tag Type 4 Tag API

The Native Tag API for the Type 4 Tags provides an interface to execute the Native Tag command set for Type 4 Tags as defined in [NFC T4T].

7.1.4.1. Native Tag T4T API

The Native Tag T4T API supports the following commands defined in [NFC T4T]:

- Select Application
- Select File
- Select
- Read for Mapping Version 2.0
- Read for Mapping Version 3.0
- Write for Mapping Version 2.0
- Write for Mapping Version 3.0

7.1.4.1.1. ptx_NativeTag_T4T_Open

The **Open** function initializes the T4T Native Tag component. It requires an initialized instance of a RUL component a buffer which can be used internally and the size of this buffer. These parameters are provided using the **ptx_NativeTag_T4T_InitParams_t** struct.

Declaration	`ptx_Status_t` **`ptx_NativeTag_T4T_Open`** `(` **`ptx_NativeTag_T4T_t`** `*t4t,` **`ptx_NativeTag_T4T_InitParams_t`** `*initParams);`	
Description	Initialization of the Native Tag T4T-component.	
Parameters	T4t	Pointer to allocated Native Tag T4T component.
	initParams	Pointer to initialization parameter structure.
Return Value	Status, indicating whether the operation was successful.	

7.1.4.1.2. ptx_NativeTag_T4T_Close

The **Close** function de-initializes the T4T Native Tag component.

Declaration	`ptx_Status_t` **`ptx_NativeTag_T4T_Close`** `(` **`ptx_NativeTag_T4T_t`** `*t4t);`	
Description	De-Initialization of the Native Tag T4T component.	
Parameters	t4t	Pointer to initialized Native Tag T4T component.
Return Value	Status, indicating whether the operation was successful.	

7.1.4.1.3. ptx_NativeTag_T4T_Select_NDEF_Application

The select application function selects the NDEF Tag application of the Type 4 Tag as described in [NFC T4T].

Declaration	`ptx_Status_t` **`ptx_NativeTag_T4T_Select_NDEF_Application`** `(` **`ptx_NativeTag_T4T_t`** `*t4t,` **`uint8_t`** `*rx,` **`uint32_t`** `*rxLen,` **`uint32_t`** `msTimeout);`	
Description	Select NDEF Application for Native Tag T4T.	
Parameters	t4t	Pointer to initialized Native Tag T4T component.
	rx	Pointer to Rx-buffer where the received data from the card should be stored.
	rxLen	Size of Rx-buffer (in), Length of the received data (out).
	msTimeout	Timeout in ms that the function is going to wait for receiving data from the card.
Return Value	Status, indicating whether the operation was successful.	

7.1.4.1.4. ptx_NativeTag_T4T_Select_File

The select file function selects the NDEF File of the Type 4 Tag as described in [NFC T4T].

Declaration	`ptx_Status_t` **`ptx_NativeTag_T4T_Select_File`** (**`ptx_NativeTag_T4T_t`** *t4t, **`uint8_t`** *data, **`uint8_t`** *rx, **`uint32_t`** *rxLen, **`uint32_t`** msTimeout);	
Description	Select Application for Native Tag T4T	
Parameters	t4t	Pointer to initialized Native Tag T4T component.
	data	Pointer to data field.
	rx	Pointer to Rx-buffer where the received data from the card should be stored.
	rxLen	Size of Rx-buffer (in), Length of the received data (out).
	msTimeout	Timeout in ms that the function is going to wait for receiving data from the card.
Return Value	Status, indicating whether the operation was successful.	

7.1.4.1.5. ptx_NativeTag_T4T_Select

The select function is used to select other applications or files of the Type 4 Tag as described in [NFC T4T].

Declaration	`ptx_Status_t` **`ptx_NativeTag_T4T_Select`** (**`ptx_NativeTag_T4T_t`** *t4t, **`uint8_t`** paramByte1, **`uint8_t`** paramByte2, **`uint8_t`** *data, **`uint8_t`** nbrDataBytes, **`uint8_t`** expectedResponseLen, **`uint8_t`** *rx, **`uint32_t`** *rxLen, **`uint32_t`** msTimeout);	
Description	Select command for Native Tag T4T	
Parameters	t4t	Pointer to initialized Native Tag T4T component.
	paramByte1	P1 of command APDU.
	paramByte2	P2 of command APDU.
	data	Pointer to data field.
	nbrDataBytes	Number of data bytes.
	expectedResponseLen	Number of response bytes expected.
	rx	Pointer to Rx-buffer where the received data from the card should be stored.
	rxLen	Size of Rx-buffer (in), Length of the received data (out).
	msTimeout	Timeout in ms that the function is going to wait for receiving data from the card.
Return Value	Status, indicating whether the operation was successful.	

7.1.4.1.6. ptx_NativeTag_T4T_ReadBinary

The **ReadBinary** function performs the READ_BINARY command as described in [NFC T4T]. It is used to read data from a file with mapping version 2.0.

Declaration	`ptx_Status_t` **`ptx_NativeTag_T4T_ReadBinary`** (**`ptx_NativeTag_T4T_t`** *t4t, **`uint16_t`** offset, **`uint8_t`** nbrExpectedBytes, **`uint8_t`** *rx, **`uint32_t`** *rxLen, **`uint32_t`** msTimeout);	
Description	Select Application for Native Tag T4T	
Parameters	t4t	Pointer to initialized Native Tag T4T component.
	offset	Data offset.
	nbrExpectedBytes	Number of expected bytes.
	rx	Pointer to Rx-buffer where the received data from the card should be stored.
	rxLen	Size of Rx-buffer (in), Length of the received data (out).
	msTimeout	Timeout in ms that the function is going to wait for receiving data from the card.
Return Value	Status, indicating whether the operation was successful.	

7.1.4.1.7. ptx_NativeTag_T4T_ReadBinaryODO

The **ReadBinaryODO** function performs the READ_BINARY command using the Offset Data Object feature as described in [NFC T4T]. It is used to read data from a file with mapping version 3.0.

Declaration	`ptx_Status_t` **`ptx_NativeTag_T4T_ReadBinaryODO`** (**`ptx_NativeTag_T4T_t`** *t4t, **`uint16_t`** offset, **`uint8_t`** nbrExpectedBytes, **`uint8_t`** *rx, **`uint32_t`** *rxLen, **`uint32_t`** msTimeout);	
Description	Select Application for Native Tag T4T	
Parameters	t4t	Pointer to initialized Native Tag T4T component.
	offset	Data offset.
	nbrExpectedBytes	Number of expected bytes.
	rx	Pointer to Rx-buffer where the received data from the card should be stored.
	rxLen	Size of Rx-buffer (in), Length of the received data (out).
	msTimeout	Timeout in ms that the function is going to wait for receiving data from the card.
Return Value	Status, indicating whether the operation was successful.	

7.1.4.1.8. ptx_NativeTag_T4T_UpdateBinary

The **UpdateBinary** function performs the UPDATE_BINARY command as described in [NFC T4T]. It is used to write data to a file with mapping version 2.0.

Declaration	`ptx_Status_t` **`ptx_NativeTag_T4T_ UpdateBinary`** (**`ptx_NativeTag_T4T_t`** `*t4t,` **`uint16_t`** `offset,` **`uint8_t`** `*data,` **`uint8_t`** `nbrDataBytes,` **`uint8_t`** `*rx,` **`uint32_t`** `*rxLen,` **`uint32_t`** `msTimeout);`	
Description	Select Application for Native Tag T4T	
Parameters	t4t	Pointer to initialized Native Tag T4T component.
	offset	Data offset.
	data	Pointer to the data field.
	nbrDataBytes	Number of data bytes.
	rx	Pointer to Rx-buffer where the received data from the card should be stored.
	rxLen	Size of Rx-buffer (in), Length of the received data (out).
	msTimeout	Timeout in ms that the function is going to wait for receiving data from the card.
Return Value	Status, indicating whether the operation was successful.	

7.1.4.1.9. ptx_NativeTag_T4T_UpdateBinaryODO

The **UpdateBinaryODO** function performs the UPDATE_BINARY command using the Offset Data Object feature as described in [NFC T4T]. It is used to write data to a file with mapping version 3.0.

Declaration	`ptx_Status_t` **`ptx_NativeTag_T4T_ UpdateBinary`** (**`ptx_NativeTag_T4T_t`** `*t4t,` **`uint16_t`** `offset,` **`uint8_t`** `*data,` **`uint8_t`** `nbrDataBytes,` **`uint8_t`** `*rx,` **`uint32_t`** `*rxLen,` **`uint32_t`** `msTimeout);`	
Description	Select Application for Native Tag T4T	
Parameters	t4t	Pointer to initialized Native Tag T4T component.
	offset	Data offset.
	data	Pointer to the data field.
	nbrDataBytes	Number of data bytes.
	rx	Pointer to Rx-buffer where the received data from the card should be stored.
	rxLen	Size of Rx-buffer (in), Length of the received data (out).
	msTimeout	Timeout in ms that the function is going to wait for receiving data from the card.
Return Value	Status, indicating whether the operation was successful.	

7.1.5. Native Tag Type 5 Tag API

The Native Tag API for the Type 5 Tags provides an interface to execute the Native Tag command set for Type 5 Tags as defined in [NFC T5T]

7.1.5.1. Native Tag T4T API

The Native Tag T5T API supports the following commands defined in [NFC T5T]:

- Read Single Block
- Write Single Block
- Lock Single Block
- Read Multiple Block
- Extended Read Single Block
- Extended Write Single Block
- Extended Lock Single Block
- Extended Read Multiple Block
- Select
- SLPV_REQ

7.1.5.1.1. ptx_NativeTag_T5T_Open

The **Open** function initializes the T5T Native Tag component. It requires an initialized instance of a RUL component a buffer which can be used internally the size of this buffer, the ID and length of the ID of the Tag if it is known and the information if the tag has been selected and can be accessed using the selected mode. These parameters are provided using the **ptx_NativeTag_T5T_InitParams_t** struct.

Note: The Tag ID and the length of the ID can be updated once they are accessible using the **ptx_NativeTag_T5T_SetTagParams** function. They do not have to be set during initialization if they aren't available.

Declaration	`ptx_Status_t` **`ptx_NativeTag_T5T_Open`** `(` **`ptx_NativeTag_T5T_t`** `*t5t,` **`ptx_NativeTag_T5T_InitParams_t`** `*initParams);`	
Description	Initialization of the Native Tag T5T component.	
Parameters	t5t	Pointer to allocated Native Tag T5T component.
	initParams	Pointer to initialization parameter structure.
Return Value	Status, indicating whether the operation was successful.	

7.1.5.1.2. ptx_NativeTag_T5T_Close

The **Close** function de-initializes the T5T Native Tag component.

Declaration	`ptx_Status_t` **`ptx_NativeTag_T5T_Close`** `(` **`ptx_NativeTag_T5T_t`** `*t5t);`	
Description	De-Initialization of the Native Tag T5T component.	
Parameters	t5t	Pointer to initialized Native Tag T5T component.
Return Value	Status, indicating whether the operation was successful.	

7.1.5.1.3. ptx_NativeTag_T5T_ReadSingleBlock

This function performs the READ_SINGLE_BLOCK_REQ command defined in [NFC T5T]. It is used to read a single block of the Type 5 Tag.

Note: The option flag is an additional parameter which is used to request the Block Security status byte in the response. This is the behavior for all Read-alike commands according to [NFC T5T].

Declaration	`ptx_Status_t` **`ptx_NativeTag_T5T_ReadSingleBlock`** (**`ptx_NativeTag_T5T_t`** *t5t, **`uint8_t`** optionFlag, **`uint8_t`** blockNr, **`uint8_t`** *rx, **`uint32_t`** *rxLen, **`uint32_t`** msTimeout);	
Description	Performs a READ_SINGLE_BLOCK_REQ command.	
Parameters	t5t	Pointer to initialized Native Tag T5T component.
	optionFlag	Option-flag. ("Request Block Security Status" for Read-alike commands).
	blockNr	Number of blocks to read.
	rx	Pointer to Rx-buffer where the received data from the card should be stored.
	rxLen	Size of Rx-buffer (in), Length of the received data (out).
	msTimeout	Timeout in ms that the function is going to wait for receiving data from the card.
Return Value	Status, indicating whether the operation was successful.	

7.1.5.1.4. ptx_NativeTag_T5T_WriteSingleBlock

This function performs the WRITE_SINGLE_BLOCK_REQ command defined in [NFC T5T]. It is used to write a single block of the Type 5 Tag.

Note: The option flag is an additional parameter which is used to specify the use of Special Frames for Write-alike commands according to [NFC T5T] and [NFC Digital].

Declaration	`ptx_Status_t` **`ptx_NativeTag_T5T_WriteSingleBlock`** `(` **`ptx_NativeTag_T5T_t`** `*t5t,` **`uint8_t`** `optionFlag,` **`uint8_t`** `blockNr,` **`uint8_t`** `*blockData,` **`uint8_t`** `blockDataLen,` **`uint8_t`** `*rx,` **`uint32_t`** `*rxLen,` **`uint32_t`** `msTimeout);`	
Description	Performs a WRITE_SINGLE_BLOCK_REQ command.	
Parameters	t5t	Pointer to initialized Native Tag T5T component.
	optionFlag	Option-flag. ("Use special Framing" for Write-alike commands).
	blockNr	Number of blocks to write.
	blockData	Data to write.
	blockDataLen	Length of data to write.
	rx	Pointer to Rx-buffer where the received data from the card should be stored.
	rxLen	Size of Rx-buffer (in), Length of the received data (out).
	msTimeout	Timeout in ms that the function is going to wait for receiving data from the card.
Return Value	Status, indicating whether the operation was successful.	

7.1.5.1.5. ptx_NativeTag_T5T_LockSingleBlock

This function performs the LOCK_SINGLE_BLOCK_REQ command defined in [NFC T5T]. It is used to lock a single block of the Type 5 Tag.

Note: The option flag is an additional parameter which is used to specify the use of Special Frames for Write-alike commands according to [NFC T5T] and [NFC Digital].

Declaration	`ptx_Status_t` **`ptx_NativeTag_T5T_LockSingleBlock`** `(`
	`  ` **`ptx_NativeTag_T5T_t`** `*t5t,`
	`  ` **`uint8_t`** `optionFlag,`
	`  ` **`uint8_t`** `blockNr,`
	`  ` **`uint8_t`** `*rx,`
	`  ` **`uint32_t`** `*rxLen,`
	`  ` **`uint32_t`** `msTimeout);`	

Description	Performs a LOCK_SINGLE_BLOCK_REQ command.	
Parameters	t5t	Pointer to initialized Native Tag T5T component.
	optionFlag	Option-flag. ("Use special Framing" for Write-alike commands).
	blockNr	Number of blocks to lock.
	rx	Pointer to Rx-buffer where the received data from the card should be stored.
	rxLen	Size of Rx-buffer (in), Length of the received data (out).
	msTimeout	Timeout in ms that the function is going to wait for receiving data from the card.
Return Value	Status, indicating whether the operation was successful.	

7.1.5.1.6. ptx_NativeTag_T5T_ReadMultipleBlock

This function performs the READ_MULTIPLE_BLOCK_REQ Command defined in [NFC T5T]. It is used to read multiple blocks of the Type 5 Tag.

Note: The option flag is an additional parameter which is used to request the Block Security status byte in the response. This is the behavior for all Read-Alike commands according to [NFC T5T].

Declaration	`ptx_Status_t` **`ptx_NativeTag_T5T_ReadMultipleBlock`** `(` **`ptx_NativeTag_T5T_t`** `*t5t,` **`uint8_t`** `optionFlag,` **`uint8_t`** `blockNr,` **`uint8_t`** `nrBlocks,` **`uint8_t`** `*rx,` **`uint32_t`** `*rxLen,` **`uint32_t`** `msTimeout);`	
Description	Performs a READ_MUTLIPLE_BLOCK_REQ command.	
Parameters	t5t	Pointer to initialized Native Tag T5T component.
	optionFlag	Option-flag. ("Request Block Security Status" for Read-alike commands).
	blockNr	Number of start-block.
	nrBlocks	Number of blocks to read.
	rx	Pointer to Rx-buffer where the received data from the card should be stored.
	rxLen	Size of Rx-buffer (in), Length of the received data (out).
	msTimeout	Timeout in ms that the function is going to wait for receiving data from the card.
Return Value	Status, indicating whether the operation was successful.	

7.1.5.1.7. ptx_NativeTag_T5T_ExtReadSingleBlock

This function performs the EXTENDED_READ_SINGLE_BLOCK_REQ command defined in [NFC T5T]. It is used to read a single block of the Type 5 Tag with Extended Memory.

Note: The option flag is an additional parameter which is used to request the Block Security status byte in the response. This is the behavior for all Read-alike commands according to [NFC T5T].

Declaration	`ptx_Status_t ptx_NativeTag_T5T_ExtReadSingleBlock (`
	`  ptx_NativeTag_T5T_t *t5t,`
	`  uint8_t optionFlag,`
	`  uint16_t blockNr,`
	`  uint8_t *rx,`
	`  uint32_t *rxLen,`
	`  uint32_t msTimeout);`	

Description	Performs an EXTENDED_READ_SINGLE_BLOCK_REQ command.	
Parameters	t5t	Pointer to initialized Native Tag T5T component.
	optionFlag	Option-flag. ("Request Block Security Status" for Read-alike commands).
	blockNr	Number of start-block.
	rx	Pointer to Rx-buffer where the received data from the card should be stored.
	rxLen	Size of Rx-buffer (in), Length of the received data (out).
	msTimeout	Timeout in ms that the function is going to wait for receiving data from the card.
Return Value	Status, indicating whether the operation was successful.	

7.1.5.1.8. ptx_NativeTag_T5T_ExtWriteSingleBlock

This function performs the EXTENDED_WRITE_SINGLE_BLOCK_REQ command defined in [NFC T5T]. It is used to write a single block of the Type 5 Tag with Extended Memory.

Note: The option flag is an additional parameter which is used to specify the use of Special Frames for Write-alike commands according to [NFC T5T] and [NFC Digital].

Declaration	`ptx_Status_t` **`ptx_NativeTag_T5T_ExtWriteSingleBlock`** `(` **`ptx_NativeTag_T5T_t`** `*t5t,` **`uint8_t`** `optionFlag,` **`uint8_t`** `blockNr,` **`uint8_t`** `*blockData,` **`uint8_t`** `blockDataLen,` **`uint8_t`** `*rx,` **`uint32_t`** `*rxLen,` **`uint32_t`** `msTimeout);`	
Description	Performs a EXTENDED_WRITE_SINGLE_BLOCK_REQ command.	
Parameters	t5t	Pointer to initialized Native Tag T5T component.
	optionFlag	Option-flag. ("Use special Framing" for Write-alike commands).
	blockNr	Number of blocks to write.
	blockData	Data to write.
	blockDataLen	Length of data to write.
	rx	Pointer to Rx-buffer where the received data from the card should be stored.
	rxLen	Size of Rx-buffer (in), Length of the received data (out).
	msTimeout	Timeout in ms that the function is going to wait for receiving data from the card.
Return Value	Status, indicating whether the operation was successful.	

7.1.5.1.9. ptx_NativeTag_T5T_ExtLockSingleBlock

This function performs the EXTENDED_LOCK_SINGLE_BLOCK_REQ command defined in [NFC T5T]. It is used to lock a single block of the Type 5 Tag with Extended Memory.

Note: The option flag is an additional parameter which is used to specify the use of Special Frames for Write-alike commands according to [NFC T5T] and [NFC Digital].

Declaration	`ptx_Status_t` **`ptx_NativeTag_T5T_ExtLockSingleBlock`** `(` **`ptx_NativeTag_T5T_t`** `*t5t,` **`uint8_t`** `optionFlag,` **`uint8_t`** `blockNr,` **`uint8_t`** `*rx,` **`uint32_t`** `*rxLen,` **`uint32_t`** `msTimeout);`	
Description	Performs a EXTENDED_LOCK_SINGLE_BLOCK_REQ command.	
Parameters	t5t	Pointer to initialized Native Tag T5T component
	optionFlag	Option-flag. ("Use special Framing" for Write-Alike commands).
	blockNr	Number of blocks to lock.
	rx	Pointer to Rx-buffer where the received data from the card should be stored.
	rxLen	Size of Rx-buffer (in), Length of the received data (out).
	msTimeout	Timeout in ms that the function is going to wait for receiving data from the card.
Return Value	Status, indicating whether the operation was successful.	

7.1.5.1.10. ptx_NativeTag_T5T_ExtReadMultipleBlock

This function performs the EXTENDED_READ_MULTIPLE_BLOCK_REQ command defined in [NFC T5T]. It is used to read multiple blocks of the Type 5 Tag with Extended Memory.

Note: The option flag is an additional parameter which is used to request the Block Security status byte in the response. This is the behavior for all Read-alike commands according to [NFC T5T].

Declaration	`ptx_Status_t` **`ptx_NativeTag_T5T_ExtReadMultipleBlock`** `(` **`ptx_NativeTag_T5T_t`** `*t5t,` **`uint8_t`** `optionFlag,` **`uint8_t`** `blockNr,` **`uint8_t`** `nrBlocks,` **`uint8_t`** `*rx,` **`uint32_t`** `*rxLen,` **`uint32_t`** `msTimeout);`	
Description	Performs a EXTENDED_READ_MUTLIPLE_BLOCK_REQ command.	
Parameters	t5t	Pointer to initialized Native Tag T5T component.
	optionFlag	Option-flag. (“Request Block Security Status” for Read-alike commands).
	blockNr	Number of start-block.
	nrBlocks	Number of blocks to read.
	rx	Pointer to Rx-buffer where the received data from the card should be stored.
	rxLen	Size of Rx-buffer (in), Length of the received data (out).
	msTimeout	Timeout in ms that the function is going to wait for receiving data from the card.
Return Value	Status, indicating whether the operation was successful.	

7.1.5.1.11. ptx_NativeTag_T5T_Select

This function performs the SELECT_REQ command defined in [NFC T5T]. It is used to select one Type 5 Tag. After the Tag has been selected using this function, it is necessary to call **ptx_NativeTag_T5T_SetUID** if the addressed mode should be used.

Note: The option flag is an additional parameter which is used to request the Block Security status byte in the response. This is the behavior for all Read-alike commands according to [NFC T5T].

Declaration	`ptx_Status_t ptx_NativeTag_T5T_Select ( ptx_NativeTag_T5T_t *t5t, uint8_t optionFlag, uint8_t *uid, uint8_t uidLen, uint8_t *rx, uint32_t *rxLen, uint32_t msTimeout);`	
Description	Performs a SELECT_REQ command.	
Parameters	t5t	Pointer to initialized Native Tag T5T component.
	optionFlag	Option-flag. (“Request Block Security Status” for Read-alike commands).
	uid	Pointer to UID of the given Tag.
	uidLen	Length of UID.
	rx	Pointer to Rx-buffer where the received data from the card should be stored.
	rxLen	Size of Rx-buffer (in), Length of the received data (out).
	msTimeout	Timeout in ms that the function is going to wait for receiving data from the card.
Return Value	Status, indicating whether the operation was successful.	

7.1.5.1.12. ptx_NativeTag_T5T_Sleep

This function performs the SLPV_REQ command defined in [NFC T5T]. It is used to put a Type 5 Tag to sleep.

Note: The option flag is an additional parameter which is used to request the Block Security status byte in the response. This is the behavior for all Read-alike commands according to [NFC T5T].

Declaration	`ptx_Status_t` **`ptx_NativeTag_T5T_Sleep`** `(` **`ptx_NativeTag_T5T_t`** `*t5t,` **`uint8_t`** `optionFlag,` **`uint8_t`** `*uid,` **`uint8_t`** `uidLen,` **`uint8_t`** `*rx,` **`uint32_t`** `*rxLen,` **`uint32_t`** `msTimeout);`	
Description	Performs a SLPV_REQ command.	
Parameters	t5t	Pointer to initialized Native Tag T5T component.
	optionFlag	Option-flag. ("Request Block Security Status" for Read-alike commands).
	uid	Pointer to UID of the given Tag.
	uidLen	Length of UID.
	rx	Pointer to Rx-buffer where the received data from the card should be stored.
	rxLen	Size of Rx-buffer (in), Length of the received data (out).
	msTimeout	Timeout in ms that the function is going to wait for receiving data from the card.
Return Value	Status, indicating whether the operation was successful.	

7.1.5.1.13. ptx_NativeTag_T5T_SetUID

This function sets the UID of the T5T Component. It can be used after a Select command to use the addressed mode or to update the Tag parameters of the component.

Declaration	`ptx_Status_t` **`ptx_NativeTag_T5T_SetUID`** `(` **`ptx_NativeTag_T5T_t`** `*t5t,` **`uint8_t`** `*uid,` **`uint8_t`** `uidLen);`	
Description	Sets / Updates the UID to be used.	
Parameters	t5t	Pointer to initialized Native Tag T5T component.
	uid	Pointer to UID of the given Tag.
	uidLen	Length of UID.
Return Value	Status, indicating whether the operation was successful.	

7.1.6. Native Tag API States

The Native Tag APIs are intended to be used as a straightforward way to send the different Tag related commands as described in the individual NFC Forum Tag Type specification documents ([NFC T2T], [NFC T3T], [NFC T4T], [NFC T5T]).

The Native Tag APIs do not check if the command is allowed to be sent at the time of the request. Thus, the API state diagram only consists of two total states.

7.1.6.1. State Overview

Figure 11 describes the state diagrams of all Native Tag Type APIs.

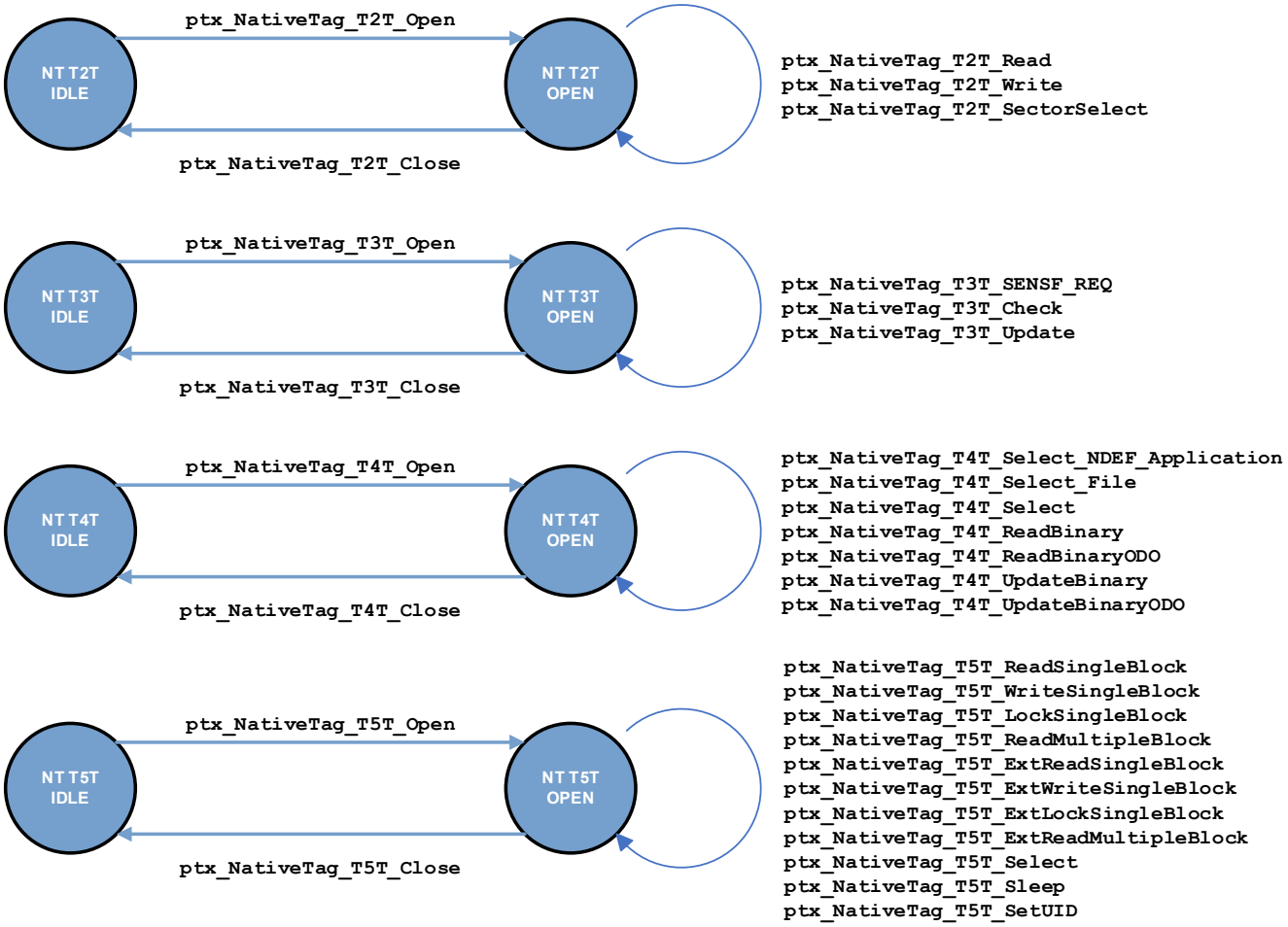

Figure 11. Native Tag Type API States

7.1.6.2. State Description

Each of the Native Tag APIs consist of two states in total. An Idle state before initialization and an open state after the appropriate open function was called.

- **NT TXT IDLE**: Idle state of the Native Tag Type X Tag API. The API must be initialized using the "**ptx_NativeTag_TXT_Open**" function before being able to use the other API functions.
- **NT TXT OPEN:** The Native Tag Type X Tag API has been initialized and the different API commands can be used. To de-initialize the Native Tag Type X API again, the user must call the appropriate "**ptx_NativeTag_TXT_Close**" function.

7.1.7. Implementation Guidelines/Suggestions

The Native Tag APIs are meant to abstract and simplify the usage of the different Tag related commands described in the individual NFC Forum Tag Type specification documents ([NFC T2T], [NFC T3T], [NFC T4T], [NFC T5T]). The Native Tag API functions do not check for the order in which the commands are executed and therefore it is expected that the user is aware of expected procedures to communicate with the specific Tag.

If the user intends to read or write NDEF messages to the diverse types of tags, the NDEF Add-on Library as described in section 7.3.

Note: The SDK contains a code example named "**ptx_Example_MFCC**" which relies on the user-specific implementation of the "Crypto-1" cipher module. Since the MFCC module / SDK only contains empty i.e. stub functions for the "Crypto-1" cipher module, it is up to the user to implement the empty / stub functions first.

Without the implementation, the code example is still compile-able, but the MFCC module will report errors during any memory operation on a MFC card that requires prior authentication due to the missing concrete implementation of the "Crypto-1" cipher.

Dependent on the user-specific implementation of the "Crypto-1" cipher module, a user-specific context / handle needs to be passed via the initialization parameter "UserContextCrypto1" (see "ptx_NativeTag_MFCC_InitParams_t") to MFCC initialization function "ptx_NativeTag_MFCC_Open". Prior to the call to the MFCC initialization function, the user needs to create / allocate the user-specific context / handle at application level.

NDEF API should be used. It contains all the necessary functionality to read and write NDEF Messages to the different supported NFC Forum Tag Types

Special considerations for the **Native Tag Type 2 Tag API**: As described in [NFC T2T], the first 3 blocks (Blocks 0, 1, and 2) contain the internal bytes Internal_0 to Internal_9. These bytes are **READ-ONLY**. Similarly, the last two bytes of the third block (Block 2) are static-lock bytes. These bytes can either have the values 00h, or FFh. If these are set to FFh, the first 52 bytes of the tag have been made **READ-ONLY** and will lock the tag. If these are set to 00h, memory can be written to the Type 2 Tag.

It is possible to have more of these locking bytes in what is known as Lock TLVs, as described in [NFC T2T].

The third block of a Type 2 Tag contains the Capability Container (CC). It contains the information for reading and writing an NDEF Message. For more information, refer to [NFC T2T].

Special considerations for the **Native Tag Type 3 Tag API:** The memory layout of Type 3 Tags consists of blocks of 16 bytes each, thus, any read operation will return the complete block of 16 bytes and every write command must be provided with the full 16 bytes as well.

Special considerations for the **Native Tag Type 4 Tag API**. The read and update binary commands, both standard and the ones using Offset Data Object (ODO) are implemented according to the NFC Forum Tag Type specification of the T4T ([NFC T4T]). These commands expect the Tag to be in the NDEF Application mode in order to execute read or update commands. The NDEF Application can be selected using the "**ptx_NativeTag_T4T_Select_NDEF_Application**" function.

With the NDEF application selected, the CC File can be selected by calling "**ptx_NativeTag_T4T_Select_File"** with the data field set to E103h**.** The contents of the CC File are read only and are defined in [NFC T4T]. The CC file mostly contains information about the capabilities of the Tag and the size of the NDEF File of the tag.

In order to read or update the contents of the NDEF file of the tag, the "**ptx_NativeTag_T4T_Select_File"** function must be called with the data field set to the NDEF file identifier which can be read from the CC file as defined in [NFC T4T]. After the NDEFfile has been selected, the user can read and write data to the NDEF file using the provided Read and Update binary functions.

Special considerations for the **Native Tag Type 5 Tag API**: As defined in [NFC T5T], the Type 5 Tag commands can be sent in Addressed and Non-Addressed mode. The addressed mode is used if the UID of the Type 5 Tag to be addressed is provided either using the Initialization parameters of the open function or alternatively by using the "**ptx_NativeTag_T5T_SetUID"** function.

Additionally, there are two methods to use the selected mode. The first method is to configure the Native Tag Type 5 Tag API accordingly using the initialization parameters for the Open function. By providing the UID, the UID length and setting the **isSelected** flag to true, the device will be communicated within the selected mode. Alternatively, the selected mode can be activated by calling the "**ptx_NativeTag_T5T_Select"** function with the parameters of the device to select.

7.2 MIFARE Classic® Compatibility (MFCC)

The MFCC API module provides simple access to memory operations on MIFARE Classic cards and can be used together with the RUL API. The MFCC API module is a stand-alone add-on to the Native Tag API.

7.2.1. API Architecture

The MFCC API module is a stand-alone add-on library of the Native Tag APIs and belongs logically to it. It is located on top of the RUL API module as shown below in Figure 12 and internally accesses also the NSC Core Stack module.

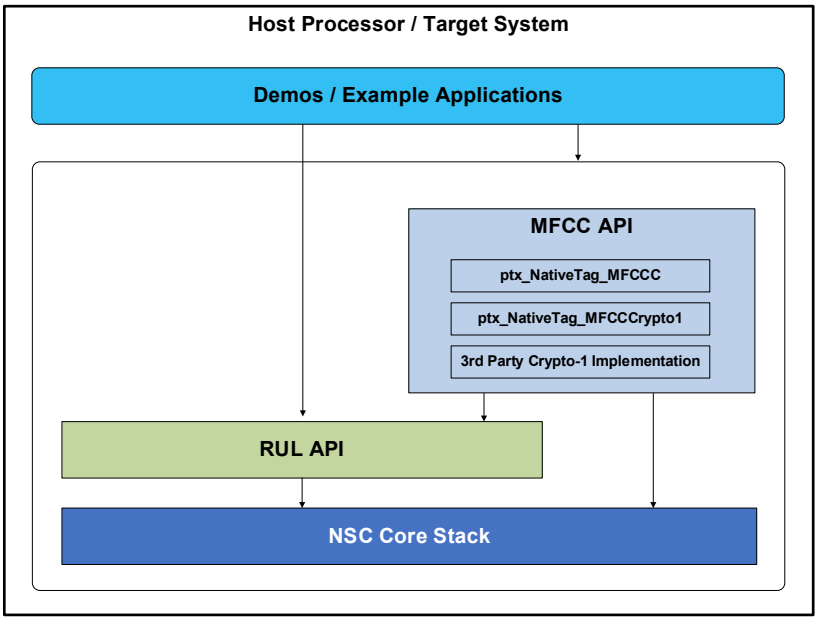

Figure 12. MFCC API Component Architecture

The MFCC API is comprised of three sub-components:

7.2.1.1. ptx_NativeTag_MFCC

Implements the actual API for the user application and contains:

- The MIFARE Classic command-set (see [MFCC 2017]):
  - Authenticate A/B
  - Read
  - Write
  - Increment
  - Decrement
  - Transfer
  - Restore
  - Halt
- Support functions
- Generic functions to open/close the component

Depending on the MIFARE Classic cards configuration, all memory operations require a prior authentication procedure on the intended memory block/sector. The authentication procedure itself is based on the proprietary “Crypto-1” cipher.

The MFCC module does not implement the proprietary “Crypto-1” cipher on its own but contains an abstraction layer for it which must be implemented by the user (see 7.2.1.2).

7.2.1.2. ptx_NativeTag_MFCCCrypto1

Implements an abstraction layer for the Crypto-1 cipher algorithm. The Crypto-1 functions are to be implemented in this layer (see 7.2.4.1). The RUL SDK contains empty sub functions.

7.2.1.3. 3rd Crypto-1 Implementation

Must contain a solid implementation of the proprietary Crypto-1 algorithm (see 7.2.4.1).

7.2.2. API Description

7.2.2.1. ptx_NativeTag_MFCC_Open

Declaration	`ptx_Status_t` **`ptx_NativeTag_MFCC_Open`** `(` **`ptx_NativeTag_MFCC_t`** `*mfcc,` **`ptx_NativeTag_MFCC_InitParams_t`** `*initParams);`	
Description	Initializes the MFCC component.	
Parameters	mfcc	Pointer to allocated MFCC component.
	initParams	Pointer to initialization parameter structure.
Return Value	Status, indicating whether the operation was successful.	

7.2.2.2. ptx_NativeTag_MFCC_Close

Declaration	`ptx_Status_t` **`ptx_NativeTag_MFCC_Close`** `(` **`ptx_NativeTag_MFCC_t`** `*mfcc);`	
Description	Component De-Initialization.	
Parameters	mfcc	Pointer to initialized MFCC component.
Return Value	Status, indicating whether the operation was successful.	

7.2.2.3. ptx_NativeTag_MFCC_SetCryptoParams

Declaration	`ptx_Status_t` **`ptx_NativeTag_MFCC_SetCryptoParams`** `(` **`ptx_NativeTag_MFCC_t`** `*mfcc,` **`ptx_NativeTag_MFCCCrypto1_t`** `*crypto1Params);`	
Description	Set Crypto1 Callbacks.	
Parameters	mfcc	Pointer to initialized MFCC component
	crypto1Params	Pointer to structure containing the callback
Return Value	Status, indicating whether the operation was successful.	

7.2.2.4. ptx_NativeTag_MFCC_Enable

Declaration	`ptx_Status_t` **`ptx_NativeTag_MFC_Enable`** (**`ptx_NativeTag_MFCC_t`** `*mfcc,` **`ptx_NativeTag_MFCC_Mode_t`** `enableMode);`	
Description	Enables or disables the Mifare Compatibility mode. ***Important***: The MFCC mode needs to be enabled prior to the first call of any MIFARE Classic command and must be disabled afterwards. The MFCC mode enables internally the Type A bitstream mode which is only usable if the T2T RF-protocol is active. In case the RF discovery procedure is (re-)started, the MFCC mode is by default disabled.	
Parameters	mfcc	Pointer to initialized MFCC component.
	enableMode	Enable or Disable flag.
Return Value	Status, indicating whether the operation was successful.	

7.2.2.5. ptx_NativeTag_MFCC_Authenticate

Declaration	`ptx_Status_t` **`ptx_NativeTag_MFCC_Authenticate`** (**`ptx_NativeTag_MFCC_t`** `*mfcc,` **`ptx_NativeTag_MFCC_KeyType_t`** `keyType,` **`uint8_t`** `address,` **`uint8_t`** `*key,` **`uint8_t`** `keyLen,` **`uint8_t`** `*uid,` **`uint8_t`** `uidLen);`	
Description	Performs a Mifare AUTHENTICATE command with Key-A or -B. ***Important***: The authentication procedure requires 4 bytes of a cards UID. Which parts of the UID are taken from double- or triple-size UID cards needs to be checked within the datasheet of the corresponding card.	
Parameters	mfcc	Pointer to initialized MFCC component.
	keyType	Key type A or B.
	address	Number of block/sector.
	key	Pointer to key.
	keyLen	Length of key (must be 6 Bytes).
	uid	Pointer to UID of card.
	uidLen	Length of UID (must be 4 Bytes).
Return Value	Status, indicating whether the operation was successful.	

7.2.2.6. ptx_NativeTag_MFCC_Read

Declaration	`ptx_Status_t` **`ptx_NativeTag_MFCC_Read`** `(`**`ptx_NativeTag_MFCC_t`** `*mfcc,` **`uint8_t`** `address,` **`uint8_t`** `*rx,` **`uint32_t`** `*rxLen);`	
Description	Performs a MIFARE Classic Read (block) command.	
Parameters	mfcc	Pointer to initialized MFCC component.
	address	Number of block/sector.
	rx	Pointer to buffer holding received block data.
	rxLen	**In**: Size of reception buffer **Out**: Length of received data
Return Value	Status, indicating whether the operation was successful.	

7.2.2.7. ptx_NativeTag_MFCC_Write

Declaration	`ptx_Status_t` **`ptx_NativeTag_MFCC_Write`** `(`**`ptx_NativeTag_MFCC_t`** `*mfcc,` **`uint8_t`** `address,` **`uint8_t`** `*blockData,` **`uint8_t`** `blockDataLen);`	
Description	Performs a MIFARE Classic Write (block) command.	
Parameters	mfcc	Pointer to initialized MFCC component.
	address	Number of block/sector.
	blockData	Pointer to buffer holding block data to write.
	blockDataLen	Length of data to write (must be 16 bytes).
Return Value	Status, indicating whether the operation was successful.	

7.2.2.8. ptx_NativeTag_MFCC_Increment

Declaration	`ptx_Status_t` **`ptx_NativeTag_MFCC_Increment`** `(`**`ptx_NativeTag_MFCC_t`** `*mfcc,` **`uint8_t`** `address,` **`uint32_t`** `value);`	
Description	Performs a MIFARE Classic Increment (value) command. ***Important***: This command requires a block to be formatted as Value block.	
Parameters	mfcc	Pointer to initialized MFCC component.
	address	Number of block/sector.
	value	Value operand.
Return Value	Status, indicating whether the operation was successful.	

7.2.2.9. ptx_NativeTag_MFCC_Decrement

Declaration	`ptx_Status_t` **`ptx_NativeTag_MFCC_Decrement`** **`(ptx_NativeTag_MFCC_t`** `*mfcc,` **`uint8_t`** `address,` **`uint32_t`** `value);`	
Description	Performs a MIFARE Classic Decrement (value) command. ***Important***: This command requires a block to be formatted as Value block.	
Parameters	mfcc	Pointer to initialized MFCC component.
	address	Number of block/sector.
	value	4-Byte Operand/Value.
Return Value	Status, indicating whether the operation was successful.	

7.2.2.10. ptx_NativeTag_MFCC_Transfer

Declaration	`ptx_Status_t` **`ptx_NativeTag_MFCC_Transfer`** **`(ptx_NativeTag_MFCC_t`** `*mfcc,` **`uint8_t`** `address);`	
Description	Performs a MIFARE Classic Restore (value) command. ***Important***: This command requires a block to be formatted as Value block.	
Parameters	mfcc	Pointer to initialized MFCC component.
	address	Number of block/sector.
Return Value	Status, indicating whether the operation was successful.	

7.2.2.11. ptx_NativeTag_MFCC_Restore

Declaration	`ptx_Status_t` **`ptx_NativeTag_MFCC_Restore`** **`(ptx_NativeTag_MFCC_t`** `*mfcc,` **`uint8_t`** `address,` **`uint32_t`** `value);`	
Description	Performs a MIFARE Classic Restore (value) command. ***Important***: This command requires a block to be formatted as Value block.	
Parameters	mfcc	Pointer to initialized MFCC component.
	address	Number of block/sector.
	value	4-Byte Operand/Value.
Return Value	Status, indicating whether the operation was successful.	

7.2.2.12. ptx_NativeTag_MFCC_GetFormattedValueBlock

Declaration	`ptx_Status_t` **`ptx_NativeTag_MFCC_GetFormattedValueBlock`** `(`**`ptx_NativeTag_MFCC_t`** `*mfcc,` **`uint8_t`** `address,` **`uint32_t`** `value,` **`uint8_t`** `*buffer,` **`uint8_t`** `bufferSize);`	
Description	Formats a given buffer as MIFARE Classic Value block.	
Parameters	mfcc	Pointer to initialized MFCC component.
	address	Number of block/sector.
	value	4-Byte Operand/Value.
	buffer	Pointer to buffer to be formatted.
	bufferSize	Size of buffer (must be at least 16 Bytes).
Return Value	Status, indicating whether the operation was successful.	

7.2.2.13. ptx_NativeTag_MFCC_Halt

Declaration	`ptx_Status_t` **`ptx_NativeTag_MFCC_Halt`** `(`**`ptx_NativeTag_MFCC_t`** `*mfcc);`	
Description	Performs a ISO/IEC 14443-A / MIFARE Classic Halt-A command.	
Parameters	mfcc	Pointer to initialized MFCC component.
Return Value	Status, indicating whether the operation was successful.	

7.2.2.14. ptx_NativeTag_MFCC_BitsExchange

Declaration	`ptx_Status_t` **`ptx_NativeTag_MFCC_BitsExchange`** `(`**`ptx_NativeTag_MFCC_t`** `*mfcc,` **`uint32_t`** `*rxLength,` **`uint32_t`** `*numTotBits);`	
Description	Raw bitstream exchange.	
Parameters	mfcc	Pointer to initialized MFCC component.
	rxLength	Pointer to response length.
	numTotBits	Total number of bits of Tx and Rx data.
Return Value	Status, indicating whether the operation was successful.	

7.2.2.15. ptx_NativeTag_MFCC_ResetState

Declaration	`ptx_Status_t` **`ptx_NativeTag_MFCC_ResetState`** **`ptx_NativeTag_MFCC_t`** `*mfcc);`	
Description	Resets the internal component state and Crypto1 component without restarting the entire API.	
Parameters	mfcc	Pointer to initialized MFCC component.
Return Value	Status, indicating whether the operation was successful.	

7.2.3. MFCC API States

7.2.3.1. State Overview

Figure 13 shows the state diagram of the MFCC API.

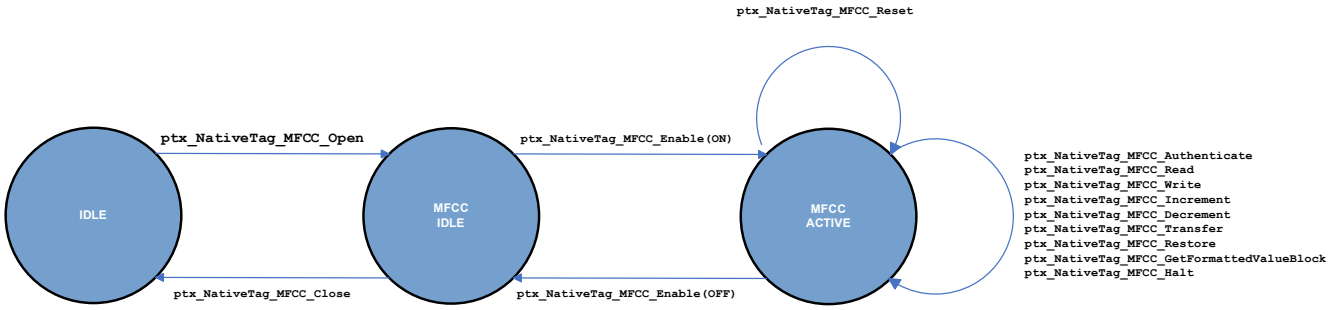

Figure 13. MFCC API States

7.2.3.2. State Description

- **IDLE**: The default state of the system after Stack and HW initialization. By calling "**ptx_NativeTag_MFCC_Open**", the system moves to MFCC IDLE state.
- **MFCC IDLE**: Intermediate state where the MFCC API is initialized, and the 3rd party Crypto1 component was opened. MFCC can now be activated on the system by calling "**ptx_NativeTag_MFCC_Enable(ON)**" moving to the next state.
- **MFCC ACTIVE**: In this state the system is now able to communicate with MIFARE Classic cards. The RUL API should be used to detect and verify the card type. Optionally in this state, "**ptx_NativeTag_MFCC_Reset**" can be called to reset the Crypto1 component without having to disable the MFCC component itself.

For almost all MIFARE Classic commands, a prior authentication of the desired block is necessary via a call to "**ptx_NativeTag_MFCC_Authenticate**" and must be called on the desired block with the correct card parameters. If successful, the rest of the MIFARE commands (WRITE, READ, INCREMENT, DECREMENT, TRANSFER, RESTORE) can be called on the authenticated block. Additionally, the helper function "**ptx_NativeTag_MFCC_GetFormattedValueBlock**" can be used to convert a given value into the MIFARE Classic Value-Block format.

The HALT command (see "**ptx_NativeTag_MFCC_Halt**") puts the MIFARE Classic card to sleep-state. To reuse the card, it needs to be activated again (for example, via "**ptx_RUL_InitiateDiscovery(DISCOVER)**".

7.2.4. Implementation Guidelines/Suggestions

7.2.4.1. Third-Party Crypto-1 Implementation

The required “Crypto-1” cipher implementation can be implemented in many ways. As an example, the publicly available 3rd party software Implementation “**Crapto-1”** may be used for the MFCC API integration. It is part of the following projects hosted on GitHub (among many other sources):

- MIFARE Classic Offline Cracker "MFOC"
- MIFARE Classic Universal ToolKit "MFCUK"

From these, three files are required:

- crapto1.h
- crapto1.c
- crypto1.c

However, since dynamic memory allocation is used, a few modifications are needed. These changes/modifications are shown in the following images – originals on the left and modified on the right:

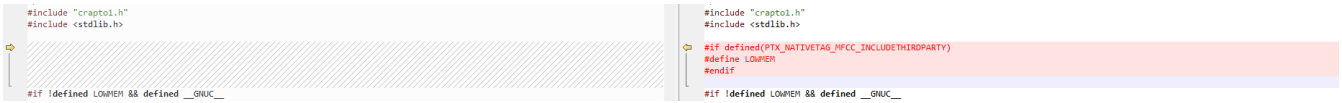

Figure 14. crapto1.c Changes for Memory Allocation Purposes

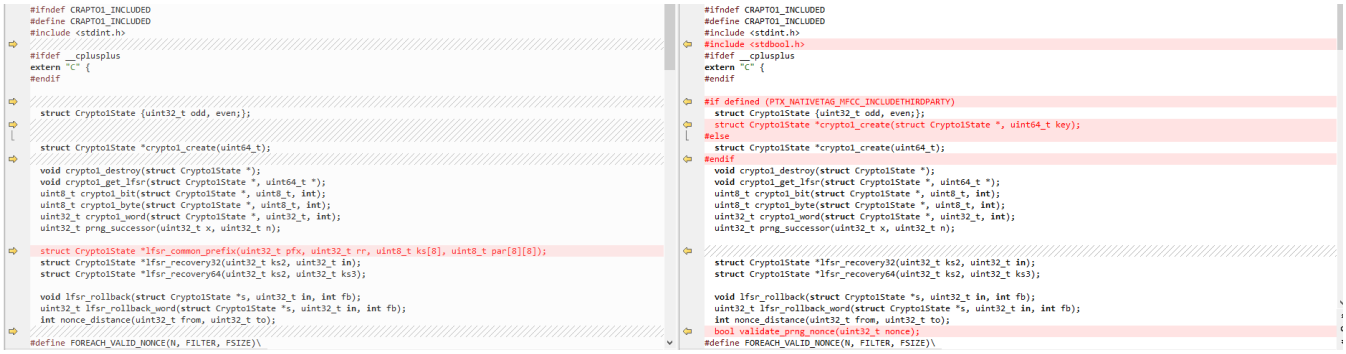

Figure 15. crapto1.h Changes for Memory Allocation Purposes and Definition for Custom Validation Helper Function

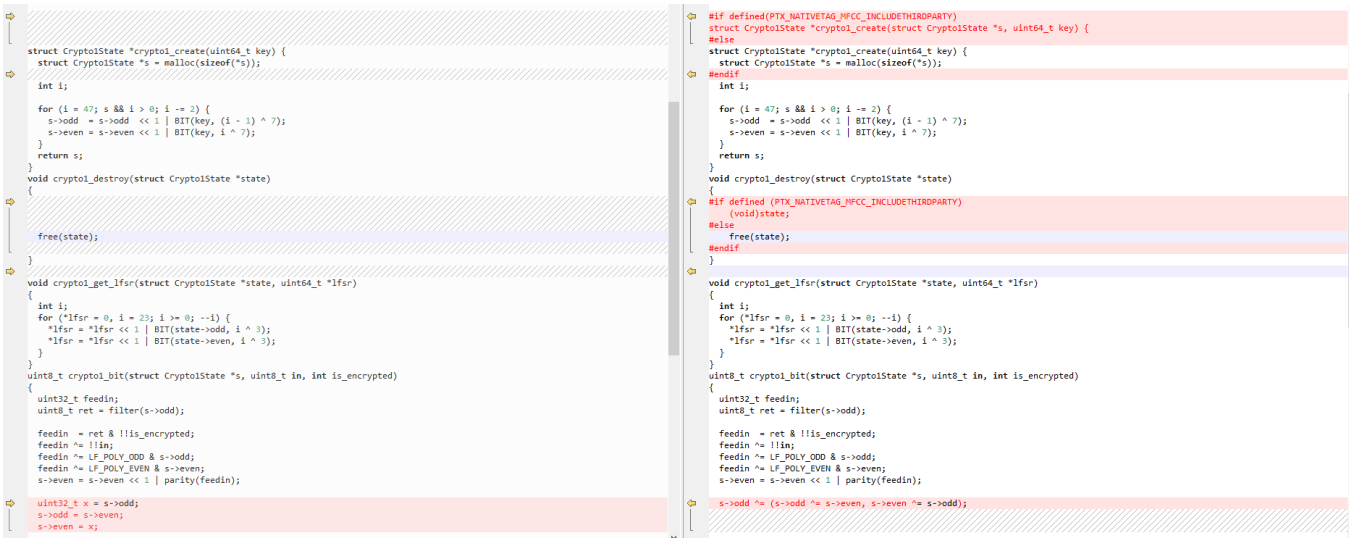

Figure 16. crypto1.c Changes for Memory Allocation Purposes and Performance Improvement

Figure 17. crapto1.c Changes for Performance Purposes

Furthermore, a few changes were done to crapto1.c for performance and usability reasons, originals on the left and modified on the right:

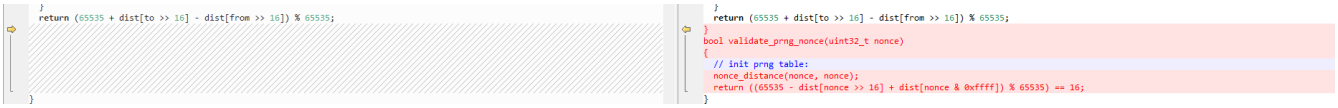

Figure 18. crapto1.c Changes for Custom Validation Helper Function

Figure 19. crapto1.c Changes for Performance Purposes

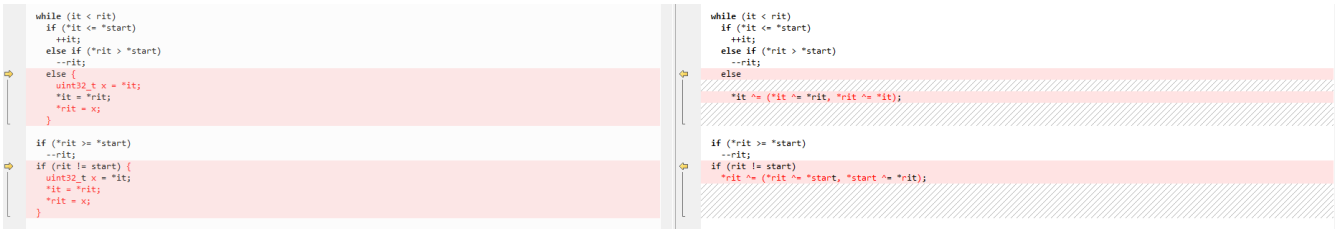

Figure 20. crapto1.c Changes for Performance Purposes

Note: The SDK contains a code example named “ptx_Example_MFCC” which relies on the user-specific implementation of the “Crypto-1” cipher module. Since the MFCC module / SDK only contains empty i.e. stub functions for the “Crypto-1” cipher module, it is up to the user to implement the empty / stub functions first.

Without the implementation, the code example is still compile-able, but the MFCC module will report errors during any memory operation on a MFC card that requires prior authentication due to the missing concrete implementation of the “Crypto-1” cipher.

Dependent on the user-specific implementation of the “Crypto-1” cipher module, a user-specific context / handle needs to be passed via the initialization parameter “UserContextCrypto1” (see “**ptx_NativeTag_MFCC_InitParams_t**”) to MFCC initialization function “**ptx_NativeTag_MFCC_Open**”. Prior to the call to the MFCC initialization function, the user needs to create / allocate the user-specific context / handle at application level.

7.3 NDEF API

The NDEF API implements the NDEF operations for each of the NFC Forum Tag Types which consists of the following set of functions:

- FORMAT (*)
- CHECK
- READ
- WRITE
- LOCK

Note (*): While the operations CHECK, READ, WRITE and LOCK are defined by the NFC Forum, the operation FORMAT is out of scope of the specifications and often depends on parameters and/or sequences, such as:

- Tag Memory Size
- Timing parameters
- Certain file systems (may also use cryptographic algorithms for creation)
- Certain format of memory blocks

Some other parameters/sequences are proprietary/manufacturer specific. Although prepared in the NDEF API(s), the final implementation must be handled by the integrator (if required).

The NDEF API consists of three larger components:

- The Tag specific NDEF operations
- The generic NDEF operation layer
- And the NDEF record creator/parser

7.3.1. API Architecture

The RF Test API module is provided as a stand-alone functionality of the RUL SDK. It is located on top of the RUL API module (see Figure 21) and also internally accesses the NSC Core Stack module.

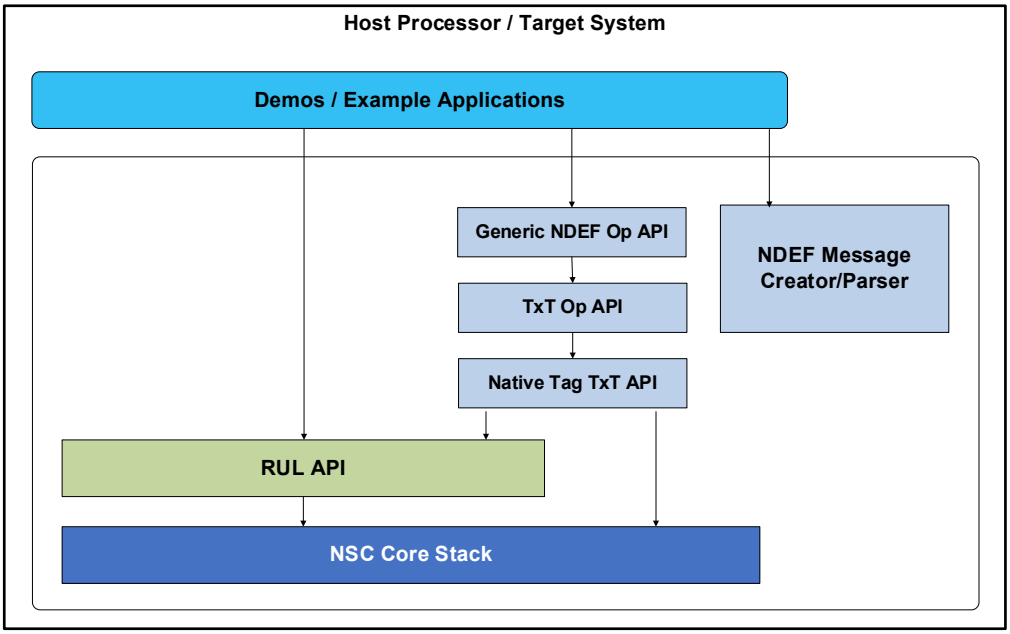

Figure 21. NDEF API Component Architecture

The NDEF API is meant to be a simplified interface for the Native Tag API. It builds upon the Native Tag API and will not function with the Native Tag disabled/excluded from the build. However, if only one Tag Type is to be supported, the other Tags can be excluded. For example, if only Tag Type 3 is used, the others (T2T, T4T, and T5T) can be excluded. The generic NDEF operation layer requires all of the subcomponents to be included.

7.3.2. Tag-Specific NDEF Operations

Each of the individual Tag-operation APIs follow the same approach where an initial call to **ptx_NDEF_TxT_Open** is required to initialize the component, and a final call to **ptx_NDEF_TxT_Close** is required to free potentially allocated resources.

Additionally, an initial NDEF check operation must be performed before any other command can be executed, otherwise the other commands will return with errors.

The Tag-specific NDEF operations provide a simplified interface to the Native Tag API to access NDEF operations. It is necessary to include the respective Native Tag API component if the Tag-specific NDEF operations APIs are to be used.

To perform read, write or lock operations, the user must first execute a check command for the internal component to get all the necessary information about the tag.

7.3.2.1. NDEF T2T Operations

7.3.2.1.1. ptx_NDEF_T2T_Open

The **Open** function initializes the T2T Component. It uses the respective Native Tag Component internally and requires the same initialization parameters as the Native Tag Component, as well as some additional ones.

It takes a pointer to an initialized instance of the RUL component, a Tx Buffer, a work buffer, and an Rx Buffer with their respective sizes. The Tx and Rx buffers are used for the internal data exchanges of the T2T NDEF component while the work buffer is used to store intermediate information between these exchanges. The recommended size for all these three buffers is 256 bytes. These parameters are all provided using the **ptx_NDEF_T2T_InitParams_t** struct.

Declaration	`ptx_Status_t` **`ptx_NDEF_T2T_Open`** `(` **`ptx_NDEF_T2T_t`** `*ndefT2T,` **`ptx_NDEF_T2T_InitParams_t`** `*initParams);`	
Description	NDEF T2T operations component initialization	
Parameters	ndefT2T	Pointer to component.
	initParams	Pointer to initialization parameter structure.
Return Value	Status, indicating whether the operation was successful.	

7.3.2.1.2. ptx_NDEF_T2T_Close

The **Close** function de-initializes the T2T NDEF Operations component.

Declaration	`ptx_Status_t` **`ptx_NDEF_T2T_Close`** `(` **`ptx_NDEF_T2T_t`** `*ndefT2T);`	
Description	NDEF T2T operations component de-initialization	
Parameters	ndefT2T	Pointer to component.
Return Value	Status, indicating whether the operation was successful.	

7.3.2.1.3. ptx_NDEF_T2T_CheckMessage

The **CheckMessage** command performs initial read operations to determine the information about the NDEF message stored on the Tag. It records info such as the message size and if the tag is locked. This function must be called before calling the Read or Write message functions.

Declaration	`ptx_Status_t` **`ptx_NDEF_T2T_CheckMessage`** (**`ptx_NDEF_T2T_t`** *ndefT2T);	
Description	Check for NDEF message	
Parameters	ndefT2T	Pointer to component.
Return Value	Status, indicating whether the operation was successful.	

7.3.2.1.4. ptx_NDEF_T2T_ReadMessage

The **ReadMessage** command reads the tag content. The check message function must be executed before a read message command can be performed. The read NDEF message is written to the provided Rx buffer.

Declaration	`ptx_Status_t` **`ptx_NDEF_T2T_ReadMessage`** (**`ptx_NDEF_T2T_t`** *ndefT2T, **`uint8_t`** *msgBuffer, **`uint32_t`** *msgLen);	
Description	Read NDEF message	
Parameters	ndefT2T	Pointer to component.
	msgBuffer	Pointer to buffer holding the read NDEF message.
	msgLen	Size of the buffer on input, Length of the read NDEF message on output
Return Value	Status, indicating whether the operation was successful.	

7.3.2.1.5. ptx_NDEF_T2T_WriteMessage

The **WriteMessage** command writes the provided NDEF message to the Tag. If an NDEF TLV exists, it will be overwritten. This function can only be executed if a check message command has been executed prior.

Declaration	`ptx_Status_t` **`ptx_NDEF_T2T_WriteMessage`** (**`ptx_NDEF_T2T_t`** *ndefT2T, **`uint8_t`** *msgBuffer, **`uint32_t`** msgLen);	
Description	Write an NDEF message. Overwrites existing NDEF TLV if one exists.	
Parameters	ndefT2T	Pointer to component.
	msgBuffer	Pointer to buffer holding the NDEF message to write. If the msgBuffer is NULL, an empty NDEF message is written.
	msgLen	Size of NDEF-message If the length is 0, an empty NDEF message is written.
Return Value	Status, indicating whether the operation was successful.	

7.3.2.1.6. ptx_NDEF_T2T_FormatTag

The **FormatTag** function is used to reset the Tag. However, the format process is dependent on multiple factors such as tag memory size and timings. The format function must be implemented by the integrator if it is required. The available function stub will currently return an Error.

Declaration	`ptx_Status_t` **`ptx_NDEF_T2T_FormatTag`** (**`ptx_NDEF_T2T_t`** `*ndefT2T);`	
Description	Format NDEF Tag	
Parameters	ndefT2T	Pointer to component.
Return Value	Status, indicating whether the operation was successful.	

7.3.2.1.7. ptx_NDEF_T2T_LockTag

The **LockTag** function transforms the Tag into read only, meaning that the currently stored NDEF Message will become permanent and can no longer be edited. **This is a permanent operation.**

Declaration	`ptx_Status_t` **`ptx_NDEF_T2T_LockTag`** (**`ptx_NDEF_T2T_t`** `*ndefT2T);`	
Description	Lock NDEF Tag	
Parameters	ndefT2T	Pointer to component.
Return Value	Status, indicating whether the operation was successful.	

7.3.2.2. NDEF T3T Operations

7.3.2.2.1. ptx_NDEF_T3T_Open

The **Open** function initializes the T3T Component. It uses the respective Native Tag Component internally and requires the same initialization parameters as the Native Tag Component as well as some additional ones.

It takes a pointer to an initialized instance of the RUL component, a Tx Buffer, and an Rx Buffer with their respective sizes. The recommended size for these buffers is 256 bytes. Additionally, it requires the ID of the Tag, the ID's length and the timing parameters used to calculate the timeout values. These parameters are all provided using the **ptx_NDEF_T3T_InitParams_t** struct.

The ID of the Tag, it's length as well as the timing parameters can be extracted from the **SensFRes** according to [NFC DIGITAL].

Declaration	`ptx_Status_t` **`ptx_NDEF_T3T_Open`** (**`ptx_NDEF_T3T_t`** `*ndefT3T,` **`ptx_NDEF_T3T_InitParams_t`** `*initParams);`	
Description	NDEF T3T operations component initialization.	
Parameters	ndefT3T	Pointer to component.
	initParams	Pointer to initialization parameter structure.
Return Value	Status, indicating whether the operation was successful.	

7.3.2.2.2. ptx_NDEF_T3T_Close

The **Close** function de-initializes the T3T NDEF operations component.

Declaration	`ptx_Status_t` **`ptx_NDEF_T3T_Close`** (**`ptx_NDEF_T3T_t`** `*ndefT3T);`	
Description	NDEF T3T operations component de-initialization.	
Parameters	ndefT3T	Pointer to an existing instance of the T3T OP component.
Return Value	Status, indicating whether the operation was successful.	

7.3.2.2.3. ptx_NDEF_T3T_CheckMessage

The **CheckMessage** command performs initial read operations to determine the information about the NDEF message stored on the Tag. It records info such as the message size and if the tag is locked. This function must be called before calling the Read or Write message functions.

Declaration	`ptx_Status_t` **`ptx_NDEF_T3T_CheckMessage`** (**`ptx_NDEF_T3T_t`** `*ndefT3T);`	
Description	Check for NDEF message.	
Parameters	ndefT3T	Pointer to an existing instance of the T3T OP component.
Return Value	Status, indicating whether the operation was successful.	

7.3.2.2.4. ptx_NDEF_T3T_ReadMessage

The **ReadMessage** command reads the tag content. The check message function must be executed before a read message command can be performed. The read NDEF message is written to the provided Rx Buffer.

Declaration	`ptx_Status_t` **`ptx_NDEF_T3T_ReadMessage`** (**`ptx_NDEF_T3T_t`** `*ndefT3T,` **`uint8_t`** `*msgBuffer,` **`uint32_t`** `*msgLen);`	
Description	Read NDEF message	
Parameters	ndefT3T	Pointer to an existing instance of the T3T OP component.
	msgBuffer	Pointer to buffer holding the read NDEF message.
	msgLen	Size of the buffer on input, Length of the read NDEF-message on output.
Return Value	Status, indicating whether the operation was successful.	

7.3.2.2.5. ptx_NDEF_T3T_WriteMessage

The **WriteMessage** command writes the provided NDEF message to the Tag. If an NDEF TLV exists, it will be overwritten. This function can only be executed if a check message command has been previously executed.

Declaration	`ptx_Status_t` **`ptx_NDEF_T3T_WriteMessage`** (**`ptx_NDEF_T3T_t`** `*ndefT3T,` **`uint8_t`** `*msgBuffer,` **`uint32_t`** `msgLen);`	
Description	Write an NDEF message. Overwrites existing NDEF TLV if one exists.	
Parameters	ndefT3T	Pointer to an existing instance of the T3T OP component.
	msgBuffer	Pointer to buffer holding the NDEF message to write. If the msgBuffer is NULL, an empty NDEF message is written.
	msgLen	Size of NDEF-message If the length is 0, an empty NDEF message is written.
Return Value	Status, indicating whether the operation was successful.	

7.3.2.2.6. ptx_NDEF_T3T_FormatTag

The **FormatTag** function is used to reset the Tag. However, the format process is dependent on multiple factors such as tag memory size and timings. The format function must be implemented by the integrator if it is required. The available function stub will currently return an Error.

Declaration	`ptx_Status_t` **`ptx_NDEF_T3T_FormatTag`** (**`ptx_NDEF_T3T_t`** `*ndefT3T);`	
Description	Format NDEF Tag	
Parameters	ndefT3T	Pointer to an existing instance of the T3T OP component.
Return Value	Status, indicating whether the operation was successful.	

7.3.2.2.7. ptx_NDEF_T3T_LockTag

The **LockTag** function transforms the Tag into read only, meaning that the currently stored NDEF message will become permanent and can no longer be edited. **This is a permanent operation**.

Declaration	`ptx_Status_t` **`ptx_NDEF_T3T_LockTag`** (**`ptx_NDEF_T3T_t`** `*ndefT3T);`	
Description	Lock NDEF Tag	
Parameters	ndefT3T	Pointer to an existing instance of the T3T OP component.
Return Value	Status, indicating whether the operation was successful.	

7.3.2.3. NDEF T4T Operations

7.3.2.3.1. ptx_NDEF_T4T_Open

The **Open** function initializes the T4T component. It uses the respective Native Tag Component internally and requires the same initialization parameters as the Native Tag component, as well as some additional ones.

It takes a pointer to an initialized instance of the RUL component, a Tx Buffer, and an Rx Buffer with their respective sizes. These buffers are used for internal exchanges. The recommended size for these buffers is 256 bytes. These parameters are all provided using the **ptx_NDEF_T4T_InitParams_t** struct.

Declaration	`ptx_Status_t` **`ptx_NDEF_T4T_Open`** `(` **`ptx_NDEF_T4T_t`** `*ndefT4T,` **`ptx_NDEF_T4T_InitParams_t`** `*initParams);`	
Description	NDEF T4T operations component initialization.	
Parameters	ndefT4T	Pointer to component.
	initParams	Pointer to initialization parameter structure.
Return Value	Status, indicating whether the operation was successful.	

7.3.2.3.2. ptx_NDEF_T4T_Close

The **Close** function de-initializes the T4T NDEF Operations component.

Declaration	`ptx_Status_t` **`ptx_NDEF_T4T_Close`** `(` **`ptx_NDEF_T4T_t`** `*ndefT4T);`	
Description	NDEF T4T operations component de-initialization.	
Parameters	ndefT4T	Pointer to component.
Return Value	Status, indicating whether the operation was successful.	

7.3.2.3.3. ptx_NDEF_T4T_CheckMessage

The **CheckMessage** command performs initial read operations to determine the information about the NDEF message stored on the Tag. It records info such as the message size and if the tag is locked. This function must be called before calling the Read or Write message functions.

Declaration	`ptx_Status_t` **`ptx_NDEF_T4T_CheckMessage`** `(` **`ptx_NDEF_T4T_t`** `*ndefT4T);`	
Description	Check for NDEF message.	
Parameters	ndefT4T	Pointer to component.
Return Value	Status, indicating whether the operation was successful.	

7.3.2.3.4. ptx_NDEF_T4T_ReadMessage

The **ReadMessage** command reads the tag content. The CheckMessage function must be executed before a read message command can be performed. The read NDEF message is written to the provided Rx Buffer.

Declaration	`ptx_Status_t` **`ptx_NDEF_T4T_ReadMessage`** (**`ptx_NDEF_T4T_t`** *ndefT4T, **`uint8_t`** *msgBuffer, **`uint32_t`** *msgLen);	
Description	Read NDEF message.	
Parameters	ndefT4T	Pointer to component.
	msgBuffer	Pointer to buffer holding the read NDEF message.
	msgLen	Size of the buffer on input, Length of the read NDEF message on output
Return Value	Status, indicating whether the operation was successful.	

7.3.2.3.5. ptx_NDEF_T4T_WriteMessage

The **WriteMessage** command writes the provided NDEF message to the Tag. If an NDEF TLV exists, it will be overwritten. This function can only be executed if a check message command has been executed prior.

Declaration	`ptx_Status_t` **`ptx_NDEF_T4T_WriteMessage`** (**`ptx_NDEF_T4T_t`** *ndefT4T, **`uint8_t`** *msgBuffer, **`uint32_t`** msgLen);	
Description	Write an NDEF message. Overwrites existing NDEF TLV if one exists.	
Parameters	ndefT4T	Pointer to component.
	msgBuffer	Pointer to buffer holding the NDEF message to write. If the msgBuffer is NULL, an empty NDEF message is written.
	msgLen	Size of NDEF-message If the length is 0, an empty NDEF message is written.
Return Value	Status, indicating whether the operation was successful.	

7.3.2.3.6. ptx_NDEF_T4T_FormatTag

The **FormatTag** function is used to reset the Tag. However, the format process is dependent on multiple factors such as tag memory size and timings. The format function must be implemented by the integrator if it is required. The available function stub will currently return an Error.

Declaration	`ptx_Status_t` **`ptx_NDEF_T4T_FormatTag`** (**`ptx_NDEF_T4T_t`** *ndefT4T);	
Description	Format NDEF Tag	
Parameters	ndefT4T	Pointer to component.
Return Value	Status, indicating whether the operation was successful.	

7.3.2.3.7. ptx_NDEF_T4T_LockTag

The **LockTag** function transforms the Tag into read only, meaning that the currently stored NDEF message will become permanent and can no longer be edited. **This is a permanent operation**.

Declaration	`ptx_Status_t` **`ptx_NDEF_T4T_LockTag`** (**`ptx_NDEF_T4T_t`** `*ndefT4T);`	
Description	Lock NDEF Tag	
Parameters	ndefT4T	Pointer to component.
Return Value	Status, indicating whether the operation was successful.	

7.3.2.4. NDEF T5T Operations

7.3.2.4.1. ptx_NDEF_T5T_Open

The **Open** function initializes the T5T component. It uses the respective Native Tag component internally and requires the same initialization parameters as the Native Tag component, as well as some additional ones.

It takes a pointer to an initialized instance of the RUL component, a Tx Buffer, a work Buffer, and an Rx Buffer with their respective sizes. The Tx and Rx buffers are used for the internal data exchanges of the T5T NDEF component while the work buffer is used to store intermediate information between these exchanges. The recommended size for all these three buffers is 256 bytes. Additionally, it requires the ID of the Tag, the ID's length and if the Tag is in SELECTED mode. The **isAddressed** parameter can be used to change between the addressed and non-addressed modes. These parameters are all provided using the **ptx_NDEF_T5T_InitParams_t** struct.

Declaration	`ptx_Status_t` **`ptx_NDEF_T5T_Open`** (**`ptx_NDEF_T5T_t`** `*ndefT5T,` **`ptx_NDEF_T5T_InitParams_t`** `*initParams);`	
Description	NDEF T5T operations component initialization.	
Parameters	ndefT5T	Pointer to component.
	initParams	Pointer to initialization parameter structure.
Return Value	Status, indicating whether the operation was successful.	

7.3.2.4.2. ptx_NDEF_T5T_Close

The **Close** function de-initializes the T5T NDEF operations component.

Declaration	`ptx_Status_t` **`ptx_NDEF_T5T_Close`** (**`ptx_NDEF_T5T_t`** `*ndefT5T);`	
Description	NDEF T5T operations component de-initialization.	
Parameters	ndefT5T	Pointer to component.
Return Value	Status, indicating whether the operation was successful.	

7.3.2.4.3. ptx_NDEF_T5T_CheckMessage

The **CheckMessage** command performs initial read operations to determine the information about the NDEF message stored on the Tag. It records info such as the message size and if the tag is locked. This function must be called before calling the Read or Write message functions.

Declaration	`ptx_Status_t` **`ptx_NDEF_T5T_CheckMessage`** (**`ptx_NDEF_T5T_t`** *ndefT5T);	
Description	Check for NDEF message.	
Parameters	ndefT5T	Pointer to component.
Return Value	Status, indicating whether the operation was successful.	

7.3.2.4.4. ptx_NDEF_T5T_ReadMessage

The **ReadMessage** command reads the tag content. The **CheckMessage** function must be executed before a read message command can be performed. The read NDEF message is written to the provided Rx Buffer.

Declaration	`ptx_Status_t` **`ptx_NDEF_T5T_ReadMessage`** (**`ptx_NDEF_T5T_t`** *ndefT5T, **`uint8_t`** *msgBuffer, **`uint32_t`** *msgLen);	
Description	Read NDEF message.	
Parameters	ndefT5T	Pointer to component.
	msgBuffer	Pointer to buffer holding the read NDEF message.
	msgLen	Size of the buffer on input, Length of the read NDEF message on output.
Return Value	Status, indicating whether the operation was successful.	

7.3.2.4.5. ptx_NDEF_T5T_WriteMessage

The **WriteMessage** command writes the provided NDEF message to the Tag. If an NDEF TLV exists, it will be overwritten. This function can only be executed if a check message command has been executed prior.

Declaration	`ptx_Status_t` **`ptx_NDEF_T5T_WriteMessage`** (**`ptx_NDEF_T5T_t`** *ndefT5T, **`uint8_t`** *msgBuffer, **`uint32_t`** msgLen);	
Description	Write an NDEF message. Overwrites existing NDEF TLV if one exists.	
Parameters	ndefT5T	Pointer to component.
	msgBuffer	Pointer to buffer holding the NDEF message to write. If the msgBuffer is NULL, an empty NDEF message is written.
	msgLen	Size of NDEF-message If the length is 0, an empty NDEF message is written.
Return Value	Status, indicating whether the operation was successful.	

7.3.2.4.6. ptx_NDEF_T5T_FormatTag

The **FormatTag** function is used to reset the Tag. However, the format process is dependent on multiple factors such as tag memory size and timings. The format function must be implemented by the integrator if it is required. The available function stub will currently return an Error.

Declaration	`ptx_Status_t` **`ptx_NDEF_T5T_FormatTag`** `(` **`ptx_NDEF_T5T_t`** `*ndefT5T);`	
Description	Format NDEF Tag	
Parameters	ndefT5T	Pointer to component
Return Value	Status, indicating whether the operation was successful.	

7.3.2.4.7. ptx_NDEF_T5T_LockTag

The **LockTag** function transforms the Tag into read only, meaning that the currently stored NDEF message will become permanent and can no longer be edited. **This is a permanent operation**.

Declaration	`ptx_Status_t` **`ptx_NDEF_T5T_LockTag`** `(` **`ptx_NDEF_T5T_t`** `*ndefT5T);`	
Description	Lock NDEF Tag	
Parameters	ndefT5T	Pointer to component.
Return Value	Status, indicating whether the operation was successful.	

7.3.3. Generic NDEF Operations

All the individual Tag Operation APIs can be accessed through the generic NDEF API. It follows the same approach as the individual NDEF Operations APIs, meaning it requires an initial call to **ptx_NDEF_Open** to initialize the component and a final call to **ptx_NDEF_Close** to free potentially allocated resources.

Additionally, an initial NDEF Check operation must be performed before any other command can be executed, otherwise other commands will return with errors.

The generic NDEF operations API provides an interface with which the commands of all the underlying Tag specific NDEF operation APIs can be accessed. The Tag APIs used are determined with the initialization parameters of the open function.

After the initialization, the generic NDEF operations API can be used to read and write or lock the selected Tag independent of the Tag Type.

The generic NDEF operations API can only be used if all the individual NDEF APIs are also enabled, meaning that if any of the individual Tag Type NDEF Operations APIs are disabled, the generic NDEF operations API is automatically excluded as well.

7.3.3.1. ptx_NDEF_Open

The **Open** function is used to initialize the generic NDEF Operations API. Internally, it initializes the Tag specific Tag operations API determined by the card protocol type of the current active card of the RUL card registry.

The generic NDEF Operations API component requires an initialized RUL instance and three internal buffers to function: one Rx Buffer, one Tx Buffer and a Work buffer with their respective sizes. The Tx and Rx buffers are used for the internal data exchanges of the Tag specific NDEF components while the work buffer is used to store intermediate information between these exchanges. The recommended size for all these three buffers is 256 bytes. These initialization parameters are provided using the **ptx_NDEF_InitParams_t** struct.

The Tag Type specific initialization parameters are handled internally. If the active Tag is of type T5T, the t5t communication mode can be chosen with the t5t mode field of the initialization parameter struct.

Declaration	`ptx_Status_t` **`ptx_NDEF_Open`** `(` **`ptx_NDEF_t`** `*ndef,` **`ptx_NDEF_InitParams_t`** `*initParams);`	
Description	Generic NDEF operations component initialization.	
Parameters	ndef	Pointer to component.
	initParams	Pointer to initialization parameter structure.
Return Value	Status, indicating whether the operation was successful.	

7.3.3.2. ptx_NDEF_Close

De-Initilization of the Generic NDEF operations component and the internally used Tag specific component.

Declaration	`ptx_Status_t` **`ptx_NDEF_Close`** `(` **`ptx_NDEF_t`** `*ndef);`	
Description	Generic NDEF operations component de-Initialization.	
Parameters	ndef	Pointer to component.
Return Value	Status, indicating whether the operation was successful.	

7.3.3.3. ptx_NDEF_CheckMessage

The **CheckMessage** command performs initial read operations to determine the information about the NDEF message stored on the Tag. It records info such as the message size and if the tag is locked. This function must be called before calling the Read or Write message functions.

Declaration	`ptx_Status_t` **`ptx_NDEF_CheckMessage`** `(` **`ptx_NDEF_t`** `*ndef);`	
Description	Generic NDEF operations component de-Initialization.	
Parameters	ndef	Pointer to component.
Return Value	Status, indicating whether the operation was successful.	

7.3.3.4. ptx_NDEF_ReadMessage

The **ReadMessage** command reads the tag content. The **CheckMessage** function must be executed before a read message command can be performed. The read NDEF message is written to the provided Rx Buffer.

Declaration	`ptx_Status_t` **`ptx_NDEF_ReadMessage`** (**`ptx_NDEF_t`** *ndef, **`uint8_t`** *msgBuffer, **`uint32_t`** *msgLen);	
Description	Read NDEF message.	
Parameters	ndef	Pointer to component.
	msgBuffer	Pointer to buffer holding the read NDEF message.
	msgLen	Size of the buffer on input, Length of the read NDEF message on output.
Return Value	Status, indicating whether the operation was successful.	

7.3.3.5. ptx_NDEF_WriteMessage

The **WriteMessage** command writes the provided NDEF message to the Tag. If an NDEF TLV exists, it will be overwritten. This function can only be executed if a check message command has been executed prior.

Declaration	`ptx_Status_t` **`ptx_NDEF_WriteMessage`** (**`ptx_NDEF_t`** *ndef, **`uint8_t`** *msgBuffer, **`uint32_t`** msgLen);	
Description	Write an NDEF message. Overwrites existing NDEF TLV if one exists.	
Parameters	ndef	Pointer to component.
	msgBuffer	Pointer to buffer holding the NDEF message to write. If the msgBuffer is NULL, an empty NDEF message is written.
	msgLen	Size of NDEF message If the length is 0, an empty NDEF message is written.
Return Value	Status, indicating whether the operation was successful.	

7.3.3.6. ptx_NDEF_FormatTag

The **FormatTag** function is used to reset the Tag. However, the format process is dependent on multiple factors such as tag memory size and timings. The format function must be implemented by the integrator if it is required. The available function stub will currently return an Error.

Declaration	`ptx_Status_t` **`ptx_NDEF_FormatTag`** (**`ptx_NDEF_t`** *ndef);	
Description	Format NDEF Tag	
Parameters	ndef	Pointer to component.
Return Value	Status, indicating whether the operation was successful.	

7.3.3.7. ptx_NDEF_LockTag

The **LockTag** function transforms the Tag into read only, meaning that the currently stored NDEF message will become permanent and can no longer be edited. **This is a permanent operation**.

Declaration	`ptx_Status_t` **`ptx_NDEF_LockTag`** `(` **`ptx_NDEF_`**`t *ndef);`	
Description	Lock NDEF Tag	
Parameters	ndef	Pointer to component.
Return Value	Status, indicating whether the operation was successful.	

7.3.4. NDEF Message Creator/Parser

The NDEF Message Creator and Parser API implements helper functions to create and parse NDEF data structures.

The main goal of this API is to simplify the way to interact with the NDEF APIs, in other words, to simplify how the message read from the Tag is processed as well as simplify the way an NDEF message is written.

On a high-level view there are two main functionalities of the NDEF Message API:

- Parsing of a received NDEF message into its NDEF records and
- Creating and NDEF message out of NDEF records

NDEF records are the units for carrying a payload within an NDEF message and each payload is described by its own set of parameters. More Information on the different NDEF Records can be found in [NFC NDEF].

The NDEF Message APIs are split up into the NDEF message handling parts and handlers for the specific NDEF Records.

7.3.4.1. NDEF Message API

The NDEF Message API is used to split up an NDEF message into its individual NDEF records or to do the inverse, creating an NDEF message based on the provided NDEF records.

Additionally, it provides a function to find a specific NDEF record of a specific type in an array of NDEF Records.

7.3.4.1.1. ptx_NDEFMessage_Parse

This function is used to parse a received NDEF message into its different NDEF record subcomponents for easier use. The parsed NDEF records are written into an existing array of NDEF records.

Declaration	`ptx_Status_t` **`ptx_NDEFMessage_Parse`** (**`uint8_t`** *srcBuffer, **`uint32_t`** bufferLen, **`ptx_NDEFRecord_t`** *recordBuffer, **`uint32_t`** *recordLen, **`uint8_t`** ignoreMBME);	
Description	Parses an NDEF message from a buffer, into its individual NDEF records.	
Parameters	srcBuffer	Pointer to the buffer from where to parse the NDEF message.
	bufferLen	Length of the buffer that needs to be parsed.
	recordBuffer	Buffer of NDEF records to store the parsed data.
	recordLen	Length of the NDEF record buffer.
	ignoreMBME	Ignore message begin and message end flags during parsing.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.1.2. ptx_NDEFMessage_Create

This function is used to create an NDEF message out of the provided NDEF records. The resulting NDEF message is written into the destination buffer which can subsequently be passed to an NDEF write function to update the NDEF message on the tag.

Declaration	`ptx_Status_t` **`ptx_NDEFMessage_Create`** (**`ptx_NDEFRecord_t`** *recordBuffer, **`uint32_t`** recordSize, **`uint8_t`** *dstBuffer, **`uint32_t`** *bufferLen);	
Description	Creates an NDEF message from individual NDEF records and stores it in a buffer.	
Parameters	recordBuffer	Pointer to an existing (and initialized) NDEF record buffer array.
	recordSize	Size of the NDEF record buffer.
	dstBuffer	Pointer to an existing buffer to where the data gets written.
	bufferLen	Pointer to an integer describing the available '**dstBuffer**' length (in). Actual written length (out).
Return Value	Status, indicating whether the operation was successful.	

7.3.4.1.3. ptx_NDEFMessage_FilterRecordsByType

This function provides the ability to filter an array of NDEF records to find all the instances of a specific NDEF record type. It provides the indices of all the type matching NDEF records.

Declaration	`ptx_Status_t` **`ptx_NDEFMessage_FilterRecordsByType`** `(`
	`  ` **`ptx_NDEFRecord_t`** `*recordBuffer,`
	`  ` **`uint32_t`** `*recordsLen,`
	`  ` **`char*`** `type,`
	`  ` **`uint8_t`** `typeLen,`
	`  ` **`uint32_t`** `*foundRecords);`	

Description	Filters a given array of NDEF records by its record type.	
Parameters	recordBuffer	Pointer to an existing NDEF record buffer array.
	recordsLen	Number of records to filter.
	type	Pointer to the record type to search for.
	typeLen	Length of the given record type.
	foundRecords	Pointer to an existing array, to store the IDs of the found/filtered records.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.2. Supported NDEF Record Types

The following NDEF record types are supported and are described in the following sub-sections:

- Device Information records
- Text records
- URI records
- WLC Capability records
- WLC Control records
- WLC Foreign Object Detection method records
- WLC Information records
- WLC Status and Info records

More detailed information on each of the records can be found in [NFC NDEF], [NFC DI], [NFC TEXT], [NFC URI] and [NFC WLC].

The NDEF record-specific APIs simplify the use of the NDEF records for the user by providing setters and getters for the specific information in the records.

The available functionalities for the different records are described in the following sub-sections.

7.3.4.3. Device Information Records

These records contain fundamental device information that can be used across different carrier types or service types.

7.3.4.3.1. ptx_NDEFRecord_Di_Init

This function initializes an NDEF Device Information record.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_Di_Init`** `(` **`ptx_NDEFRecord_DI_t`** `*container);`	
Description	Sets the device information container to zero.	
Parameters	container	Pointer to the device information record.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.3.2. ptx_NDEFRecord_Di_Create

This function creates an NDEF record container from a given Device Information record.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_Di_Create`** `(` **`ptx_NDEFRecord_t`** `*record,` **`ptx_NDEFRecord_DI_t`** `*container,` **`uint8_t`** `*payloadBuffer);`	
Description	Creates an NDEF record container from the given **ptx_NDEFRecord_DI_t** struct.	
Parameters	record	Pointer to an uninitialized NDEF record structure.
	container	Pointer to the initialized **ptx_NDEFRecord_DI_t** container.
	payloadBuffer	Pointer to a payload buffer for storing the data of dynamic length.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.3.3. ptx_NDEFRecord_Di_Parse

This function parses a given NDEF record as a Device Information record and stores it in the provided Device Information container.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_Di_Parse`** `(` **`ptx_NDEFRecord_t`** `*srcRecord,` **`ptx_NDEFRecord_DI_t`** `*dst);`	
Description	Parses a given NDEF record as Device Information record and stores it in the **ptx_NDEFRecord_DI_t** container.	
Parameters	srcRecord	Pointer to the NDEF record to be parsed.
	dst	Pointer to the device information record for storing the parsed data.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.3.4. ptx_NDEFRecord_Di_SetManufacturerName

This function sets the mandatory manufacturer name field within the device information container.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_Di_SetManufacturerName`** `(` **`ptx_NDEFRecord_DI_t`** `*container,` **`char`** `*manufacturerName);`	
Description	Sets the mandatory manufacturer name field within the device information container and updates its TLV.	
Parameters	container	Pointer to the Device Information container to be updated.
	manufacturerName	Pointer to the manufacturer character array (must be null terminated).
Return Value	Status, indicating whether the operation was successful.	

7.3.4.3.5. ptx_NDEFRecord_Di_SetModelName

This function sets the mandatory model name field within the device information container.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_Di_SetModelName`** `(` **`ptx_NDEFRecord_DI_t`** `*container,` **`char`** `*modelName);`	
Description	Sets the mandatory model name field within the device information container and updates its TLV.	
Parameters	container	Pointer to the Device Information container to be updated.
	modelName	Pointer to the manufacturer character array (must be null terminated).
Return Value	Status, indicating whether the operation was successful.	

7.3.4.3.6. ptx_NDEFRecord_Di_SetDeviceUniqueName

This function sets the optional device unique name field within the device information container.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_Di_SetDeviceUniqueName`** `(` **`ptx_NDEFRecord_DI_t`** `*container,` **`char`** `* deviceUniqueName);`	
Description	Sets the optional device unique name field within the device information container and updates its TLV.	
Parameters	container	Pointer to the Device Information container to be updated.
	deviceUniqueName	Pointer to the manufacturer character array (must be null terminated).
Return Value	Status, indicating whether the operation was successful.	

7.3.4.3.7. ptx_NDEFRecord_Di_SetUUID

This function sets the optional UUID field within the device information container.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_Di_SetUUID`** (**`ptx_NDEFRecord_DI_t`** *container, **`uint8_t`** uuid[16]);	
Description	Sets the optional UUID field within the device information container and updates its TLV.	
Parameters	container	Pointer to the Device Information container to be updated.
	uuid	Pointer to the 16 byte UUID.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.3.8. ptx_NDEFRecord_Di_SetFirmwareVersion

This function sets the optional firmware version name within the device information container.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_Di_SetFirmwareVersion`** (**`ptx_NDEFRecord_DI_t`** *record, **`char`** *firmwareVersion);	
Description	Sets the optional firmware version name within the device information container and updates its TLV.	
Parameters	container	Pointer to the Device Information container to be updated.
	firmwareVersion	Pointer to the unique name character array (must be null terminated).
Return Value	Status, indicating whether the operation was successful.	

7.3.4.4. Text Records

These records contain plain text data.

7.3.4.4.1. ptx_NDEFRecord_Text_Create

This function creates an NDEF record struct for a Text record with the given plain text and language code.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_Text_Create`** (**`ptx_NDEFRecord_t`** *record, **`uint8_t`** *workBuffer, **`uint32_t`** workBufferLen, **`char`** *languageCode, **`char`** *text);	
Description	Creates an NDEF text record from specified plain text and language code.	
Parameters	record	Pointer to an uninitialized NDEF record structure.
	workBuffer	Pointer to a buffer which will be used to hold the data of the NDEF record.
	workBufferLen	Size of the provided **workBuffer**.
	languageCode	Language code (must be null terminated).
	text	Plain text (must be null terminated).
Return Value	Status, indicating whether the operation was successful.	

7.3.4.5. URI Records

These records contain URIs.

7.3.4.5.1. ptx_NDEFRecord_Uri_Create

This function creates an NDEF record struct from a specified URI and URI record identifier.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_Uri_Create`** `(` **`ptx_NDEFRecord_t`** `*record,` **`uint8_t`** `*workBuffer,` **`uint32_t`** `workBufferLen,` **`ptx_NDEFRecord_Uri_Identifier_t`** `idf,` **`char`** `*uri);`	
Description	Creates an NDEF URI record from a specified URI and URI Record identifier.	
Parameters	record	Pointer to an uninitialized NDEF record structure.
	workBuffer	Pointer to a buffer which will be used to hold the data of the NDEF record.
	workBufferLen	Size of the provided **workBuffer**.
	idf	URI record identifier option.
	uri	Pointer to the character array describing the URI (must be null terminated).
Return Value	Status, indicating whether the operation was successful.	

7.3.4.6. WLC Capability Records

These records contain WLC Capability information, more information can be found in [NFC WLC]

7.3.4.6.1. ptx_NDEFRecord_WLCCAP_Init

This function initializes the WLC capability container with zeros.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCAP_Init`** `(` **`ptx_NDEFRecord_WLCCAP_t`** `*container);`	
Description	Initializes the WLCCAP container with zeros.	
Parameters	container	Pointer to the uninitialized **ptx_WLCCAP_t** container.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.6.2. ptx_NDEFRecord_WLCCAP_Create

This function creates an NDEF record container from the given **ptx_WLCCAP_t** struct.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCAP_Create`**`(` **`ptx_NDEFRecord_t`** `*record,` **`ptx_NDEFRecord_WLCCAP_t`** `*container);`	
Description	Creates an NDEF record container from the given **ptx_WLCCAP_t** struct.	
Parameters	record	Pointer to an uninitialized NDEF record structure.
	container	Pointer to the initialized WLCCAP container.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.6.3. ptx_NDEFRecord_WLCCAP_Parse

This function extracts the NDEF record payload into the **ptx_WLCCAP_t** container.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCAP_Parse`** `(` **`ptx_NDEFRecord_t`** `*record,` **`ptx_NDEFRecord_WLCCAP_t`** `*container);`	
Description	Extracts the NDEF record payload into the **ptx_WLCCAP_t** container.	
Parameters	record	Pointer to an uninitialized NDEF record structure.
	container	Pointer to the initialized WLCCAP container.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.6.4. ptx_NDEFRecord_WLCCAP_SetProtVersion

This function is used to set the WLCCAP protocol version. The version is split into major and minor version numbers.

Declaration	`ptx_Status_t` **`ptx_NDEFREcord_WLCCAP_SetProtVersion`** `(` **`ptx_NDEFRecord_WLCCAP_t`** `*container,` **`uint8_t`** `majorVersion,` **`uint8_t`** `minorVersion);`	
Description	Setter for the WLCCAP protocol version, split into major and minor version numbers.	
Parameters	container	Pointer to the WLCCAP container.
	majorVersion	Major version number.
	minorVersion	Minor version number.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.6.5. ptx_NDEFRecord_WLCCAP_GetProtVersion

This function is used to get the WLCCAP protocol version. The version is split into major and minor version numbers.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCAP_GetProtVersion`** `(` **`ptx_NDEFRecord_WLCCAP_t`** `*container,` **`uint8_t`** `*majorVersion,` **`uint8_t`** `*minorVersion);`	
Description	Getter for the WLCCAP protocol version, split into major and minor version numbers.	
Parameters	container	Pointer to the WLCCAP container.
	majorVersion	Valid pointer to store major version number.
	minorVersion	Valid pointer to store minor version number.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.6.6. ptx_NDEFRecord_WLCCAP_SetMode

This function is used to set the WLCCAP mode. The following modes are available as defined in [NFC WLC]. These are mapped to the **ptx_NDEFRecord_WLCMode_t** as follows:

- **Static**: WLCMode_Static
- **Negotiated**: WLCMode_Negotiated
- **Battery Full**: WLCMode_BattFull
- **RFU**: WLCMode_RFU

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCAP_SetMode`** `(` **`ptx_NDEFRecord_WLCCAP_t`** `*container,` **`ptx_NDEFRecord_WLCMode_t`** `mode);`	
Description	Setter for the WLCCAP mode.	
Parameters	container	Pointer to the WLCCAP container.
	mode	Mode of operation.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.6.7. ptx_NDEFRecord_WLCCAP_GetMode

This function is used to get the WLCCAP mode. The following modes are available as defined in [NFC WLC]. These are mapped to the **ptx_NDEFRecord_WLCMode_t** as follows:

- **Static**: WLCMode_Static
- **Negotiated**: WLCMode_Negotiated
- **Battery Full**: WLCMode_BattFull
- **RFU**: WLCMode_RFU

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCAP_GetMode`** `(` **`ptx_NDEFRecord_WLCCAP_t`** `*container,` **`ptx_NDEFRecord_WLCMode_t`** `*mode);`	
Description	Setter for the WLCCAP mode.	
Parameters	container	Pointer to the WLCCAP container.
	mode	Valid pointer to mode of operation.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.6.8. ptx_NDEFRecord_WLCCAP_SetNWtRetries

This function is used to set the maximum number of repetitions of CAP_WT in WLCCAP.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCAP_SetNWtRetries`** `(` **`ptx_NDEFRecord_WLCCAP_t`** `*container,` **`uint8_t`** `retries);`	
Description	Setter for the maximum number of repetitions of CAP_WT in WLCCAP.	
Parameters	container	Pointer to the WLCCAP container.
	retries	Maximum number of repetitions.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.6.9. ptx_NDEFRecord_WLCCAP_GetNWtRetries

This function is used to get the maximum number of repetitions of CAP_WT in WLCCAP.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCAP_GetNWtRetries`** `(` **`ptx_NDEFRecord_WLCCAP_t`** `*container,` **`uint8_t`** `*retries);`	
Description	Getter for the maximum number of repetitions of CAP_WT in WLCCAP.	
Parameters	container	Pointer to the WLCCAP container.
	retries	Valid pointer to store retries value.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.6.10. ptx_NDEFREcord_WLCCAP_SetNegWait

This function is used to set the NEGO_WAIT flag in WLCCAP.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCAP_SetNegWait`** (**`ptx_NDEFREcord_WLCCAP_t`** *container, **`uint8_t`** negoWait);	
Description	Setter for the NEGO_WAIT flag in WLCCAP.	
Parameters	container	Pointer to the WLCCAP container.
	negoWait	Enable/Disable NEGO_WAIT.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.6.11. ptx_NDEFRecord_WLCCAP_GetNegWait

This function is used to get the NEGO_WAIT flag in WLCCAP.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCAP_GetNegWait`** (**`ptx_NDEFRecord_WLCCAP_t`** *container, **`uint8_t`** *negoWait);	
Description	Getter for the NEGO_WAIT flag in WLCCAP.	
Parameters	container	Pointer to the WLCCAP container.
	negoWait	Valid pointer to store NEGO_WAIT flag.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.6.12. ptx_NDEFRecord_WLCCAP_SetRdConf

This function is used to set the read confirmation flag in WLCCAP.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCAP_SetRdConf`** (**`ptx_NDEFRecord_WLCCAP_t`** *container, **`uint8_t`** rdConf);	
Description	Setter for the read confirmation flag in WLCCAP.	
Parameters	container	Pointer to the WLCCAP container.
	rdConf	Enable/Disable read confirmation flag.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.6.13. ptx_NDEFRecord_WLCCAP_GetRdConf

This function is used to get the read confirmation flag in WLCCAP.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCAP_GetRdConf`** (**`ptx_NDEFRecord_WLCCAP_t`** *container, **`uint8_t`** *rdConf);	
Description	Getter for the read confirmation flag in WLCCAP.	
Parameters	container	Pointer to the WLCCAP container.
	rdConf	Valid Pointer to store the read confirmation flag.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.6.14. ptx_NDEFRecord_WLCCAP_SetCapWt

This function is used to set the CAP_WT integer value.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCAP_SetCapWt`** (**`ptx_NDEFRecord_WLCCAP_t`** *container, **`uint8_t`** capWt);	
Description	Setter for the CAP_WT integer value.	
Parameters	container	Pointer to the WLCCAP container.
	capWt	CAP_WT integer value.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.6.15. ptx_NDEFREcord_WLCCAP_GetCapWt

This function is used to get the CAP_WT integer value.

Declaration	`ptx_Status_t` **`ptx_NDEFREcord_WLCCAP_GetCapWt`** (**`ptx_NDEFRecord_WLCCAP_t`** *container, **`uint8_t`** *capWt);	
Description	Getter for the CAP_WT integer value.	
Parameters	container	Pointer to the WLCCAP container.
	capWt	Valid pointer to store the CAP_WT value.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.6.16. ptx_NDEFREcord_WLCCAP_GetCapWtMillis

This function is used to get the CAP_WT value in milliseconds.

Declaration	`ptx_Status_t` **`ptx_NDEFREcord_WLCCAP_GetCapWtMillis`** (**`ptx_NDEFRecord_WLCCAP_t`** *container, **`uint32_t`** *capWtMillis);	
Description	Getter for the CAP_WT value in milliseconds.	
Parameters	container	Pointer to the WLCCAP container.
	capWtMillis	Valid pointer to store the CAP_WT value in milliseconds.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.6.17. ptx_NDEFRecord_WLCCAP_SetRdWtInt

This function is used to set the response waiting time.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCAP_SetRdWtInt`** (**`ptx_NDEFRecord_WLCCAP_t`** *container, **`uint8_t`** rdWT);	
Description	Setter for the response waiting time.	
Parameters	container	Pointer to the WLCCAP container.
	rdWT	RD_WT integer value to be set.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.6.18. ptx_NDEFRecord_WLCCAP_GetRdWtInt

This function is used to get the response waiting time.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCAP_GetRdWtInt`** (**`ptx_NDEFRecordWLCCAP_t`** `*container,` **`uint8_t`** `*rdWT);`	
Description	Getter for the response waiting time.	
Parameters	container	Pointer to the WLCCAP container.
	rdWT	Valid pointer to store the RD_WT integer.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.6.19. ptx_NDEFRecord_WLCCAP_SetWrToInt

This function is used to set the NDEF write timeout integer.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCAP_SetWrToInt`** (**`ptx_NDEFRecord_WLCCAP_t`** `*container,` **`uint8_t`** `timeout);`	
Description	Setter for the NDEF write timeout integer.	
Parameters	container	Pointer to the WLCCAP container.
	timeout	WR_TO integer.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.6.20. ptx_NDEFRecord_WLCCAP_GetWrToInt

This function is used to get the NDEF write timeout integer.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCAP_GetWrToInt`** (**`ptx_NDEFRecord_WLCCAP_t`** `*container,` **`uint8_t`** `*timeout);`	
Description	Getter for the NDEF write timeout integer.	
Parameters	container	Pointer to the WLCCAP container.
	timeout	Valid pointer to store the WR_TO integer.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.6.21. ptx_NDEFRecord_WLCCAP_SetWrWtInt

This function is used to set the NDEF write waiting time integer.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCAP_SetWrWtInt`** (**`ptx_NDEFRecord_WLCCAP_t`** `*container,` **`uint8_t`** `waitingTime);`	
Description	Setter for the NDEF write waiting time integer.	
Parameters	container	Pointer to the WLCCAP container.
	waitingTime	Write waiting time integer value to be set.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.6.22. ptx_NDEFRecord_WLCCAP_GetWrWtInt

This function is used to get the NDEF write waiting time integer.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCAP_GetWrWtInt`** `(` **`ptx_NDEFRecord_WLCCAP_t`** `*container,` **`uint8_t`** `*waitingTime);`	
Description	Getter for the NDEF write waiting time integer.	
Parameters	container	Pointer to the WLCCAP container.
	waitingTime	Valid pointer to store the write waiting time integer.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.7. WLC Listen Control Records

These records contain status information from the Wireless Charging Listener Device and are used to request and configure the next Wireless Power Transfer phase. More information can be found in [NFC WLC]

7.3.4.7.1. ptx_NDEFRecord_WLCCTL_Init

This function initializes the WLCCTL container with zeros.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCTL_Init`** `(` **`ptx_NDEFRecord_WLCCTL_t`** `*container);`	
Description	Initializes the WLCCTL container with zeros.	
Parameters	container	Pointer to the uninitialized WLCCTL container.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.7.2. ptx_NDEFRecord_WLCCTL_Create

This function creates an NDEF record container from the given **ptx_NDEFRecord_WLCCTL_t** struct.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCTL_Create`**`(` **`ptx_NDEFRecord_t`** `*record,` **`ptx_NDEFRecord_WLCCTL_t`** `*container);`	
Description	Create an NDEF record container from the given ptx_WLCCTL_t struct.	
Parameters	record	Pointer to an uninitialized NDEF record structure.
	container	Pointer to the initialized WLCCTL container.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.7.3. ptx_NDEFRecord_WLCCTL_Parse

This function extracts the NDEF record payload into the **ptx_NDEFRecord_WLCCTL_t** container.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCTL_Parse`**`(` **`ptx_NDEFRecord_t`** `*record,` **`ptx_NDEFRecord_WLCCTL_t`** `*container);`	
Description	Extracts the NDEF record payload into the **ptx_WLCCTL_t** container.	
Parameters	record	Pointer to an NDEF record structure.
	container	Pointer to the uninitialized WLCCTL container.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.7.4. ptx_NDEFRecord_WLCCTL_GetLength

This function is used to get the WLCCTL length of the WLCCTL record. ERROR_INFO is optional, thus the length can differ between WLCCTL records.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCTL_GetLength`**`(` **`ptx_NDEFRecord_WLCCTL_t`** `*container,` **`uint8_t`** `*len);`	
Description	Getter for the WLCCTL length of the WLCCTL record.	
Parameters	container	Pointer to the WLCCTL container.
	len	Valid pointer to store the container length.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.7.5. ptx_NDEFRecord_WLCCTL_SetErrorFlag

This function is used to set the WLCCTL error flag within STATUS_INFO field.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCTL_SetErrorFlag`**`(` **`ptx_NDEFRecord_WLCCTL_t`** `*container,` **`uint8_t`** `error);`	
Description	Setter for the WLCCTL error flag within STATUS_INFO field.	
Parameters	container	Pointer to the WLCCTL container.
	error	Set/clear error flag.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.7.6. ptx_NDEFRecord_WLCCTL_GetErrorFlag

This function is used to get the WLCCTL error flag within the STATUS_INFO field.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCTL_GetErrorFlag`**`(` **`ptx_NDEFRecord_WLCCTL_t`** `*container,` **`uint8_t`** `*error);`	
Description	Getter for the WLCCTL error flag within the STATUS_INFO field.	
Parameters	container	Pointer to the WLCCTL container.
	error	Valid pointer to store the error flag.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.7.7. ptx_NDEFRecord_WLCCTL_SetBatteryStatus

This function is used to set the WLCCTL battery status information.

The battery status can have the following states defined as defined in [NFC WLC]. These can be accessed using the **ptx_NDEFRecord_WLCCTL_BatteryStatus_t** enum:

- No information provided: **NoInformation**
- WLC-L is using WPT to charge a battery: **BatteryCharging**
- WLC-L is not using WPT to charge a battery: **NoBatteryCharging**

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCTL_SetBatteryStatus`**`(` **`ptx_NDEFRecord_WLCCTL_t`** `*container,` **`ptx_NDEFRecord_WLCCTL_BatteryStatus_t`** `batteryStatus);`	
Description	Setter for the WLCCTL battery status information.	
Parameters	container	Pointer to the WLCCTL container.
	battStatus	Current battery status.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.7.8. ptx_NDEFRecord_WLCCTL_GetBatteryStatus

This function is used to get the WLCCTL battery status field.

The battery status can have the following states defined as defined in [NFC WLC]. These can be accessed using the **ptx_NDEFRecord_WLCCTL_BatteryStatus_t** enum:

- No information provided: **NoInformation**
- WLC-L is using WPT to charge a battery: **BatteryCharging**
- WLC-L is not using WPT to charge a battery: **NoBatteryCharging**

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCTL_GetBatteryStatus`**`(` **`ptx_NDEFRecord_WLCCTL_t`** `*container,` **`ptx_NDEFRecord_WLCCTL_BatteryStatus_t`** `*batteryStatus);`	
Description	Getter for the WLCCTL battery status field.	
Parameters	container	Pointer to the WLCCTL container.
	battStatus	Valid pointer to store the battery status.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.7.9. ptx_NDEFRecord_WLCCTL_SetCounter

This function is used to set the WLCCTL counter.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCTL_SetCounter`**`(` **`ptx_NDEFRecord_WLCCTL_t`** `*container,` **`uint8_t`** `counter);`	
Description	Setter for the WLCCTL counter.	
Parameters	container	Pointer to the WLCCTL container.
	counter	Counter value (3 Bit).
Return Value	Status, indicating whether the operation was successful.	

7.3.4.7.10. ptx_NDEFRecord_WLCCTL_GetCounter

This function is used to get the WLCCTL counter field. If the counter changes between readouts, this indicates to the poller that the listener updated its WLCCTL record.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCTL_GetCounter`**`(` **`ptx_NDEFRecord_WLCCTL_t`** `*container,` **`uint8_t`** `*counter);`	
Description	Getter for the WLCCTL counter field. If the counter changes between readouts, this indicates to the poller that the listener updated its WLCCTL record.	
Parameters	container	Pointer to the WLCCTL container.
	counter	Valid pointer to store the counter value.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.7.11. ptx_NDEFRecord_WLCCTL_SetWptReq

This function is used to set the WLCCTL WPT duration integer.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCTL_SetWptReq`**`(` **`ptx_NDEFRecord_WLCCTL_t`** `*container,` **`uint8_t`** `wptRequest);`	
Description	Setter for the WLCCTL WPT duration integer.	
Parameters	container	Pointer to the WLCCTL container.
	duration	WPT duration integer.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.7.12. ptx_NDEFRecord_WLCCTL_GetWptReq

This function is used to get the WLCCTL WPT request flag.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCTL_GetWptReq`**`(` **`ptx_NDEFRecord_WLCCTL_t`** `*container,` **`uint8_t`** `*wptRequest);`	
Description	Getter for the WLCCTL WPT request flag.	
Parameters	container	Pointer to the WLCCTL container.
	wptRequest	Valid pointer to store the WPT request flag.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.7.13. ptx_NDEFRecord_WLCCTL_SetWptDuration

This function is used to set the WLCCTL WPT duration integer.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCTL_SetWptDuration`**`(` **`ptx_NDEFRecord_WLCCTL_t`** `*container,` **`uint8_t`** `duration);`	
Description	Setter for the WLCCTL WPT duration integer.	
Parameters	container	Pointer to the WLCCTL container.
	duration	WPT duration integer.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.7.14. ptx_NDEFRecord_WLCCTL_GetWptDuration

This function is used to get the WLCCTL WPT duration integer.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCTL_GetWptDuration`**`(` **`ptx_NDEFRecord_WLCCTL_t`** `*container,` **`uint8_t`** `*duration);`	
Description	Getter for the WLCCTL WPT duration integer.	
Parameters	container	Pointer to the WLCCTL container.
	duration	Valid pointer to store the WPT duration integer
Return Value	Status, indicating whether the operation was successful.	

7.3.4.7.15. ptx_NDEFRecord_WLCCTL_GetWptDurationMillis

This function is used to get the WLCCTL WPT duration time in milliseconds.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCTL_GetWptDurationMillis`**`(` **`ptx_NDEFRecord_WLCCTL_t`** `*container,` **`uint32_t`** `*wptDurationMs);`	
Description	Getter for the WLCCTL WPT duration time in milliseconds.	
Parameters	container	Pointer to the WLCCTL container.
	wptDurationMs	Valid pointer to store the WPT duration in milliseconds.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.7.16. ptx_NDEFRecord_WLCCTL_SetWptInfoReq

This function is used to set the WLCCTL INFO_REQ flag

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCTL_SetWptInfoReq`**`(` **`ptx_NDEFRecord_WLCCTL_t`** `*container,` **`uint8_t`** `request);`	
Description	Setter for the WLCCTL INFO_REQ flag.	
Parameters	container	Pointer to the WLCCTL container.
	request	Set/clear request flag.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.7.17. ptx_NDEFRecord_WLCCTL_GetWptInfoReq

This function is used to get the WLCCTL INFO_REQ flag.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCTL_GetWptInfoReq`**`(` **`ptx_NDEFRecord_WLCCTL_t`** `*container,` **`uint8_t`** `*request);`	
Description	Getter for the WLCCTL INFO_REQ flag.	
Parameters	container	Pointer to the WLCCTL container.
	request	Valid pointer to store the INFO_REQ flag.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.7.18. ptx_NDEFRecord_WLCCTL_SetPowerAdjReq

This function is used to set the WLCCTL power adjust level.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCTL_SetPowerAdjReq`**`(` **`ptx_NDEFRecord_WLCCTL_t`** `*container,` **`int8_t`** `adjustLevel);`	
Description	Setter for the WLCCTL power adjust level.	
Parameters	container	Pointer to the WLCCTL container.
	adjustLevel	Power adjust level.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.7.19. ptx_NDEFRecord_WLCCTL_GetPowerAdjReq

This function is used to get the WLCCTL power adjust request of the listener.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCTL_GetPowerAdjReq`**`(` **`ptx_NDEFRecord_WLCCTL_t`** `*container,` **`int8_t`** `*adjustLevel);`	
Description	Getter for the WLCCTL power adjust request of the listener.	
Parameters	container	Pointer to the WLCCTL container.
	adjustLevel	Valid pointer to store the power adjust integer.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.7.20. ptx_NDEFRecord_WLCCTL_SetBatteryLevel

This function is used to set the WLCCTL battery level.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCTL_SetBatteryLevel`**`(` **`ptx_NDEFRecord_WLCCTL_t`** `*container,` **`uint8_t`** `batteryLevel);`	
Description	Setter for the WLCCTL battery level.	
Parameters	container	Pointer to the WLCCTL container.
	battLevel	Battery level in percent.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.7.21. ptx_NDEFRecord_WLCCTL_GetBatteryLevel

This function is used to get the WLCCTL battery level.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCTL_GetBatteryLevel`**`(` **`ptx_NDEFRecord_WLCCTL_t`** `*container,` **`uint8_t`** `*batteryLevel);`	
Description	Getter for the WLCCTL battery level.	
Parameters	container	Pointer to the WLCCTL container.
	battLevel	Valid pointer to store the battery level in percent.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.7.22. ptx_NDEFRecord_WLCCTL_SetDvrFlag

This function is used to set the WLCCTL DVR support value.

DVR support can have the following states defined in [NFC WLC]. These are mapped to the **ptx_NDEFRecord_WLCCTL_DvrSupport_t** enum as follows:

- DVR Information not Supported: **DVRSupport_No**
- DVR Information supported but not available: **DVRSupport_Yes_NA**
- DVR Information available: **DVRSupport_Yes**

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCTL_SetDvrFlag`**`(` **`ptx_NDEFRecord_WLCCTL_t`** `*container,` **`ptx_NDEFRecord_WLCCTL_DvrSupport_t`** `support);`	
Description	Setter for the WLCCTL DVR support value.	
Parameters	container	Pointer to the WLCCTL container.
	support	DVR support value.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.7.23. ptx_NDEFRecord_WLCCTL_GetDvrFlag

This function is used to get the WLCCTL DVR_INFO support.

DVR support can have the following states defined in [NFC WLC]. These are mapped to the **ptx_NDEFRecord_WLCCTL_DvrSupport_t** enum as follows:

- DVR Information not Supported: **DVRSupport_No**
- DVR Information supported but not available: **DVRSupport_Yes_NA**
- DVR Information available: **DVRSupport_Yes**

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCTL_GetDvrFlag`**`(` **`ptx_NDEFRecord_WLCCTL_t`** `*container,` **`ptx_NDEFRecord_WLCCTL_DvrSupport_t`** `*support);`	
Description	Getter for the WLCCTL DVR_INFO support.	
Parameters	container	Pointer to the WLCCTL container.
	support	Valid pointer to store the support information.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.7.24. ptx_NDEFRecord_WLCCTL_SetDvrValue

This function is used to set the WLCCTL DVR integer.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCTL_SetDvrValue`**`(` **`ptx_NDEFRecord_WLCCTL_t`** `*container,` **`uint8_t`** `currDvr);`	
Description	Setter for the WLCCTL DVR integer.	
Parameters	container	Pointer to the WLCCTL container.
	currDvr	Current DVR integer.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.7.25. ptx_NDEFRecord_WLCCTL_GetDvrValue

This function is used to get the WLCCTL DVR integer.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCTL_GetDvrValue`**`(` **`ptx_NDEFRecord_WLCCTL_t`** `*container,` **`uint8_t`** `*currDvr);`	
Description	Getter for the WLCCTL DVR integer.	
Parameters	Container	Pointer to the WLCCTL container.
	currDvr	Valid pointer to store the DVR integer.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.7.26. ptx_NDEFRecord_WLCCTL_GetHoldOffWt

This function is used to get the WLCCTL HOLD_OFF_WT integer.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCTL_GetHoldOffWt`**`(` **`ptx_NDEFRecord_WLCCTL_t`** `*container,` **`uint8_t`** `*holdOffWt);`	
Description	Getter for the WLCCTL HOLD_OFF_WT integer.	
Parameters	container	Pointer to the WLCCTL container.
	holdOffWt	Valid pointer to store HOLD_OFF_WT integer.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.7.27. ptx_NDEFRecord_WLCCTL_GetHoldOffWtMillis

This function is used to get the WLCCTL HOLD_OFF_WT in milliseconds.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCTL_GetHoldOffWtMillis`**`(` **`ptx_NDEFRecord_WLCCTL_t`** `*container,` **`uint32_t`** `*holdOffWtMillis);`	
Description	Getter for the WLCCTL HOLD_OFF_WT in milliseconds.	
Parameters	container	Pointer to the WLCCTL container.
	holdOffWtMillis	Valid pointer to store HOLD_OFF_WT in milliseconds.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.7.28. ptx_NDEFRecord_WLCCTL_SetHoldOffWt

This function is used to set the WLCCTL HOLD_OFF_WT integer.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCTL_SetHoldOffWt`**`(` **`ptx_NDEFRecord_WLCCTL_t`** `*container,` **`uint8_t`** `holdOffWt);`	
Description	Setter for the WLCCTL HOLD_OFF_WT integer.	
Parameters	container	Pointer to the WLCCTL container.
	holdOffWt	HOLD_OFF_WT integer.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.7.29. ptx_NDEFRecord_WLCCTL_SetErrorInfoOvTemp

This function is used to set the WLCCTL over temperature error.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCTL_SetErrorInfoOvTemp`**`(` **`ptx_NDEFRecord_WLCCTL_t`** `*container,` **`uint8_t`** `ovTemp);`	
Description	Setter for the WLCCTL over temperature error.	
Parameters	container	Pointer to the WLCCTL container.
	ovTemp	Set/Clear temperature error flag.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.7.30. ptx_NDEFRecord_WLCCTL_GetErrorInfoOvTemp

This function is used to get the WLCCTL over temperature error.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCTL_GetErrorInfoOvTemp`**`(` **`ptx_NDEFRecord_WLCCTL_t`** `*container,` **`uint8_t`** `*ovTemp);`	
Description	Getter for the WLCCTL over temperature error.	
Parameters	container	Pointer to the WLCCTL container.
	ovTemp	Valid pointer to store flag, if an error has occured.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.7.31. ptx_NDEFRecord_WLCCTL_SetErrorInfoProtocol

This function is used to set the WLCCTL protocol error.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCTL_SetErrorInfoProtocol`**`(` **`ptx_NDEFRecord_WLCCTL_t`** `*container,` **`uint8_t`** `protError);`	
Description	Setter for the WLCCTL protocol error.	
Parameters	container	Pointer to the WLCCTL container.
	protError	Set/Clear protocol error flag.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.7.32. ptx_NDEFRecord_WLCCTL_GetErrorInfoProtocol

This function is used to get the WLCCTL protocol error.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCTL_GetErrorInfoProtocol`**`(` **`ptx_NDEFRecord_WLCCTL_t`** `*container,` **`uint8_t`** `*protError);`	
Description	Getter for the WLCCTL protocol error.	
Parameters	container	Pointer to the WLCCTL container.
	protError	Valid pointer to store flag, if an error has occured.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.7.33. ptx_NDEFRecord_WLCCTL_GetErrorInfoProprietary

This function is used to get the proprietary PTX error codes.

The following PTX error codes are defined using the **ptx_NDEFRecord_WLCCTL_Proprietary_ErrorInfo_t** enum:

- Error OK (No error): **ErrorInfo_OK**
- IC Temperature error: **ErrorInfo_ICTemp**
- Battery not connected error: **ErrorInfo_BatNotConnected**
- Battery temperature error: **ErrorInfo_BatTemp**
- Trickle Charge mode timeout error: **ErrorInfo_TcmTimeout**
- Constant current mode timeout error: **ErrorInfo_CcmTimeout**
- Constant voltage mode timeout error: **ErrorInfo_CvmTimeout**
- Protocol error: **ErrorInfo_ProtocolError**

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCCTL_GetErrorInfoProprietary`**`(` **`ptx_NDEFRecord_WLCCTL_t`** `*container,` **`ptx_NDEFRecord_WLCCTL_Proprietary_ErrorInfo_t`** `*err);`	
Description	Getter for the proprietary PTX error codes.	
Parameters	container	Pointer to the WLCCTL container.
	err	Valid pointer to store the PTX error code.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.8. WLC Foreign Object Detection method records

These records allow the identification of the Foreign Object Detection method. It can also include additional, method-specific data required for the Foreign Object Detection method execution. More information can be found in [NFC WLC].

7.3.4.8.1. ptx_NDEFRecord_WLCFOD_Init

This function initializes the WLCFOD container with zeros.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCFOD_Init`** `(` **`ptx_NDEFRecord_WLCFOD_t`** `*container);`	
Description	Initializes the WLCFOD container with zeros.	
Parameters	container	Pointer to the uninitialized WLCFOD container.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.8.2. ptx_NDEFRecord_WLCFOD_Create

This function creates an NDEF record container from the given **ptx_WLCFOD_t** struct.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCFOD_Create`**`(` **`ptx_NDEFRecord_t`** `*record,` **`ptx_NDEFRecord_WLCFOD_t`** `*container);`	
Description	Create an NDEF record container from the given **ptx_WLCFOD_t** struct.	
Parameters	record	Pointer to an uninitialized NDEF record structure.
	container	Pointer to the initialized capability container.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.8.3. ptx_NDEFRecord_WLCCTL_Parse

This function extracts the NDEF record payload into the **ptx_WLCFOD_t** container.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCFOD_Parse`**`(` **`ptx_NDEFRecord_t`** `*record,` **`ptx_NDEFRecord_WLCFOD_t`** `*container);`	
Description	Extracts the NDEF record payload into the **ptx_WLCFOD_t** container.	
Parameters	record	Pointer to an NDEF record structure.
	container	Pointer to the uninitialized WLCFOD container.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.8.4. ptx_NDEFRecord_WLCFOD_SetCompanyId

This function is used to set the WLCFOD company ID.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCFOD_SetCompanyId`**`(` **`ptx_NDEFRecord_WLCFOD_t`** `*container,` **`uint8_t`** `companyId);`	
Description	Setter for the WLCFOD company ID.	
Parameters	container	Pointer to the WLCFOD container.
	companyId	Company ID.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.8.5. ptx_NDEFRecord_WLCFOD_GetCompanyId

This function is used to get the WLCFOD company ID.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCFOD_GetCompanyId`**`(` **`ptx_NDEFRecord_WLCFOD_t`** `*container,` **`uint8_t`** `*companyId);`	
Description	Getter for the WLCFOD company ID.	
Parameters	container	Pointer to the WLCFOD container.
	companyId	Valid pointer to store the Company ID.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.8.6. ptx_NDEFRecord_WLCFOD_SetMethodId

This function is used to set the WLCFOD method ID.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCFOD_SetMethodId`**`(` **`ptx_NDEFRecord_WLCFOD_t`** `*container,` **`uint8_t`** `methodId);`	
Description	Setter for the WLCFOD method ID.	
Parameters	container	Pointer to the WLCFOD container.
	methodId	Used method ID for FOD.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.8.7. ptx_NDEFRecord_WLCFOD_GetMethodId

This function is used to get the WLCFOD method ID.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCFOD_GetMethodId`**`(` **`ptx_NDEFRecord_WLCFOD_t`** `*container,` **`uint8_t`** `*methodId);`	
Description	Getter for the WLCFOD method ID.	
Parameters	container	Pointer to the WLCFOD container.
	methodId	Valid pointer to store the used method ID for FOD.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.9. WLC Poll Information records

These records contain information on the power status of the WLC Poller. This message is transmitted by the Poller to the WLC Listener and impacts the parameter settings for the next WPT phase. More information can be found in [NFC WLC].

7.3.4.9.1. ptx_NDEFRecord_WLCINF_Init

This function initializes the WLCINF container with zeros.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCINF_Init`** `(` **`ptx_NDEFRecord_WLCINF_t`** `*container);`	
Description	Initializes the WLCINF container with zeros.	
Parameters	container	Pointer to the uninitialized **ptx_WLCINF_t** container.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.9.2. ptx_NDEFRecord_WLCINF_Create

This function creates an NDEF record container from the given **ptx_NDEFRecord_WLCINF_t** struct.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCINF_Create`**`(` **`ptx_NDEFRecord_t`** `*record,` **`ptx_NDEFRecord_WLCINF_t`** `*container);`	
Description	Creates an NDEF record container from the given **ptx_WLCINF_t** struct.	
Parameters	record	Pointer to an uninitialized NDEF record structure.
	container	Pointer to the initialized WLCINF container.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.9.3. ptx_NDEFRecord_WLCINF_Parse

This function extracts the NDEF record payload into the **ptx_NDEFRecord_WLCINF_t** container.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCINF_Parse`**`(` **`ptx_NDEFRecord_t`** `*record,` **`ptx_NDEFRecord_WLCINF_t`** `*container);`	
Description	Extracts the NDEF record payload into the **ptx_WLCINF_t** container.	
Parameters	record	Pointer to an NDEF record structure.
	container	Pointer to the uninitialized WLCINF container.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.9.4. ptx_NDEFRecord_WLCINF_SetPTX

This function is used to set the WLCINF poller transmit power.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCINF_SetPTX`**`(` **`ptx_NDEFRecord_WLCINF_t`** `*container,` **`uint8_t`** `percent);`	
Description	Setter for the WLCINF poller transmit power.	
Parameters	container	Pointer to the WLCINF container.
	percent	Current poller transmit power in percent.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.9.5. ptx_NDEFRecord_WLCINF_GetPTX

This function is used to get the WLCINF poller transmit power.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCINF_GetPTX`**`(` **`ptx_NDEFRecord_WLCINF_t`** `*container,` **`uint8_t`** `*percent);`	
Description	Getter for the WLCINF poller transmit power.	
Parameters	container	Pointer to the WLCINF container.
	percent	Valid pointer to store the current poller transmit power in percent.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.9.6. ptx_NDEFRecord_WLCINF_SetWptStopSupport

This function is used to set the WLCINF WPT stop support.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCINF_SetWptStopSupport`**`(` **`ptx_NDEFRecord_WLCINF_t`** `*container,` **`uint8_t`** `wptStopSupport);`	
Description	Setter for the WLCINF WPT stop support.	
Parameters	container	Pointer to the WLCINF container.
	wptStopSupport	Enable/Disable WPT stop support.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.9.7. ptx_NDEFRecord_WLCINF_GetWptStopSupport

This function is used to get the WLCINF stop support.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCINF_GetWptStopSupport`**`(` **`ptx_NDEFRecord_WLCINF_t`** `*container,` **`uint8_t`** `*wptStopSupport);`	
Description	Getter for WLCINF stop support.	
Parameters	container	Pointer to the WLCINF container.
	wptStopSupport	Valid pointer to store the stop support flag.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.9.8. ptx_NDEFRecord_WLCINF_SetPowerClass

This function is used to set the WLCINF power class.

The power classes are defined in [NFC WLC] and are mapped to the respective fields in the **ptx_NDEFRecord_WLCINF_PowerClass_t** enum:

- Power Class 0: 250mW: **PowerClass0**
- Power Class 1: 500mW: **PowerClass1**
- Power Class 2: 750mW: **PowerClass2**
- Power Class 3: 1000mW: **PowerClass3**

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCINF_SetPowerClass`**`(` **`ptx_NDEFRecord_WLCINF_t`** `*container,` **`ptx_NDEFRecord_WLCINF_PowerClass_t`** `powerClass);`	
Description	Setter for the WLCINF power class.	
Parameters	container	Pointer to the WLCINF container.
	powerClass	Power class to be used for upcoming WPTs.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.9.9. ptx_NDEFRecord_WLCINF_GetPowerClass

This function is used to get the WLCINF power class.

The power classes are defined in [NFC WLC] and are mapped to the respective fields in the ptx_WLCINF_PowerClass_t enum:

- Power Class 0: 250mW: **PowerClass0**
- Power Class 1: 500mW: **PowerClass1**
- Power Class 2: 750mW: **PowerClass2**
- Power Class 3: 1000mW: **PowerClass3**

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCINF_GetPowerClass`**`(` **`ptx_NDEFRecord_WLCINF_t`** `*container,` **`ptx_NDEFRecord_WLCINF_PowerClass_t`** `*powerClass);`	
Description	Getter for WLCINF power class.	
Parameters	container	Pointer to the WLCINF container.
	powerClass	Valid pointer to store current power class into.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.9.10. ptx_NDEFRecord_WLCINF_SetTotPowerSteps

This function is used to set the WLCINF total power steps.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCINF_SetTotPowerSteps`**`(` **`ptx_NDEFRecord_WLCINF_t`** `*container,` **`uint8_t`** `step);`	
Description	Setter for the WLCINF total power steps.	
Parameters	container	Pointer to the WLCINF container.
	step	Amount of power steps supported by the poller.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.9.11. ptx_NDEFRecord_WLCINF_GetTotPowerSteps

This function is used to get the WLCINF total amount of power steps.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCINF_GetTotPowerSteps`**`(` **`ptx_NDEFRecord_WLCINF_t`** `*container,` **`uint8_t`** `*step);`	
Description	Getter for WLCINF total amount of power steps.	
Parameters	container	Pointer to the WLCINF container.
	step	Valid pointer to store the amount of supported power steps.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.9.12. ptx_NDEFRecord_WLCINF_SetCurPowerStep

This function is used to set the WLCINF currently used power step.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCINF_SetCurPowerStep`**`(` **`ptx_NDEFRecord_WLCINF_t`** `*container,` **`uint8_t`** `step);`	
Description	Setter for the WLCINF currently used power step.	
Parameters	container	Pointer to the WLCINF container.
	step	Current power step in use.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.9.13. ptx_NDEFRecord_WLCINF_GetCurPowerStep

This function is used to get the WLCINF current power step.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCINF_GetCurPowerStep`**`(` **`ptx_NDEFRecord_WLCINF_t`** `*container,` **`uint8_t`** `*step);`	
Description	Getter for the WLCINF current power step.	
Parameters	container	Pointer to the WLCINF container.
	step	Valid pointer to store the current power step.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.9.14. ptx_NDEFRecord_WLCINF_SetNextMinStepInc

This function is used to set the WLCINF minimum power step increase.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCINF_SetNextMinStepInc`**`(` **`ptx_NDEFRecord_WLCINF_t`** `*container,` **`int8_t`** `stepSize);`	
Description	Setter for the WLCINF minimum power step increase.	
Parameters	container	Pointer to the WLCINF container.
	stepSize	Minimum power step increase.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.9.15. ptx_NDEFRecord_WLCINF_GetNextMinStepInc

This function is used to get the WLCINF minimum step increase.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCINF_GetNextMinStepInc`**`(` **`ptx_NDEFRecord_WLCINF_t`** `*container,` **`int8_t`** `*stepSize);`	
Description	Getter for the WLCINF minimum step increase.	
Parameters	container	Pointer to the WLCINF container.
	stepSize	Minimum power step decrease.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.9.16. ptx_NDEFRecord_WLCINF_SetNextMinStepDec

This function is used to set the WLCINF minimum power step decrease.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCINF_SetNextMinStepDec`**`(` **`ptx_NDEFRecord_WLCINF_t`** `*container,` **`int8_t`** `stepSize);`	
Description	Setter for the WLCINF minimum power step decrease.	
Parameters	container	Pointer to the WLCINF container.
	stepSize	Minimum power step decrease.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.9.17. ptx_NDEFRecord_WLCINF_GetNextMinStepDec

This function is used to get the WLCINF minimum step decrease.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCINF_GetNextMinStepDec`**`(` **`ptx_NDEFRecord_WLCINF_t`** `*container,` **`int8_t`** `*stepSize);`	
Description	Getter for the WLCINF minimum step decrease.	
Parameters	container	Pointer to the WLCINF container.
	stepSize	Valid pointer to store the step size.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.10. WLC Status and Info Records

These records contain information such as battery level, receive power, receive voltage, receive current, battery temperature and WLC Listener temperature. More information can be found in [NFC WLC].

7.3.4.10.1. ptx_NDEFRecord_WLCSTAI_Init

This function initializes the WLCSTAI container with zeros.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCSTAI_Init`** `(` **`ptx_NDEFRecord_WLCSTAI_t`** `*container);`	
Description	Initializes the WLCSTAI container with zeros.	
Parameters	container	Pointer to the uninitialized **ptx_WLCSTAI_t** container.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.10.2. ptx_NDEFRecord_WLCSTAI_Create

This function creates an NDEF record container from the given **ptx_NDEFRecord_WLCSTAI_t** struct.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCSTAI_Create`**`(` **`ptx_NDEFRecord_t`** `*record,` **`ptx_NDEFRecord_WLCSTAI_t`** `*container);`	
Description	Create an NDEF record container from the given **ptx_WLCSTAI_t** struct.	
Parameters	record	Pointer to an uninitialized NDEF record structure.
	container	Pointer to the initialized WLCSTAI container.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.10.3. ptx_NDEFRecord_WLCSTAI_Parse

This function extracts the NDEF record payload into the **ptx_NDEFRecord_WLCSTAI_t** container.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCSTAI_Parse`**`(` **`ptx_NDEFRecord_t`** `*record,` **`ptx_NDEFRecord_WLCSTAI_t`** `*container);`	
Description	Extracts the NDEF record payload into the **ptx_WLCSTAI_t** container.	
Parameters	record	Pointer to an NDEF record structure.
	container	Pointer to the uninitialized WLCSTAI container.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.10.4. ptx_NDEFRecord_WLCSTAI_SetBatteryLevel

This function is used to set the WLCSTAI battery level.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCSTAI_SetBatteryLevel`**`(` **`ptx_NDEFRecord_WLCSTAI_t`** `*container,` **`uint8_t`** `percent);`	
Description	Setter for the WLCSTAI battery level.	
Parameters	container	Pointer to the WLCSTAI container.
	percent	Battery level in percent.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.10.5. ptx_NDEFRecord_WLCSTAI_GetBatteryLevel

This function is used to get the WLCSTAI battery level.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCSTAI_GetBatteryLevel`**`(` **`ptx_NDEFRecord_WLCSTAI_t`** `*container,` **`uint8_t`** `*percent);`	
Description	Getter for the WLCSTAI battery level measured by the listener.	
Parameters	container	Pointer to the WLCSTAI container.
	percent	Valid pointer to store the battery level in percent.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.10.6. ptx_NDEFRecord_WLCSTAI_SetReceivePower

This function is used to set the WLCSTAI power level.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCSTAI_SetReceivePower`**`(` **`ptx_NDEFRecord_WLCSTAI_t`** `*container,` **`uint8_t`** `power);`	
Description	Setter for the WLCSTAI power level.	
Parameters	container	Pointer to the WLCSTAI container.
	power	Power level.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.10.7. ptx_NDEFRecord_WLCSTAI_GetReceivePower

This function is used to get the WLCSTAI received power by the listener.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCSTAI_GetReceivePower`**`(` **`ptx_NDEFRecord_WLCSTAI_t`** `*container,` **`uint8_t`** `*power);`	
Description	Getter for the WLCSTAI received power by the listener.	
Parameters	container	Pointer to the WLCSTAI container.
	power	Valid pointer to store the power.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.10.8. ptx_NDEFRecord_WLCSTAI_SetReceiveVoltage

This function is used to set the WLCSTAI voltage level.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCSTAI_SetReceiveVoltage`**`(` **`ptx_NDEFRecord_WLCSTAI_t`** `*container,` **`uint8_t`** `voltage);`	
Description	Setter for the WLCSTAI voltage level.	
Parameters	container	Pointer to the WLCSTAI container.
	voltage	Voltage level.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.10.9. ptx_NDEFRecord_WLCSTAI_GetReceiveVoltage

This function is used to get the WLCSTAI received voltage by the listener.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCSTAI_GetReceiveVoltage`**`(` **`ptx_NDEFRecord_WLCSTAI_t`** `*container,` **`uint8_t`** `*voltage);`	
Description	Getter for the WLCSTAI received voltage by the listener.	
Parameters	container	Pointer to the WLCSTAI container.
	voltage	Valid pointer to store the voltage.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.10.10. ptx_NDEFRecord_WLCSTAI_SetReceiveCurrent

This function is used to set the WLCSTAI current.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCSTAI_SetReceiveCurrent`**`(` **`ptx_NDEFRecord_WLCSTAI_t`** `*container,` **`uint8_t`** `current);`	
Description	Setter for the WLCSTAI current.	
Parameters	container	Pointer to the WLCSTAI container.
	current	Current value.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.10.11. ptx_NDEFRecord_WLCSTAI_GetReceiveCurrent

This function is used to get the WLCSTAI received current by the listener.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCSTAI_GetReceiveCurrent`**`(` **`ptx_NDEFRecord_WLCSTAI_t`** `*container,` **`uint8_t`** `*current);`	
Description	Getter for the WLCSTAI received current by the listener.	
Parameters	container	Pointer to the WLCSTAI container.
	current	Valid pointer to store the current.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.10.12. ptx_NDEFRecord_WLCSTAI_SetTemperatureBat

This function is used to set the WLCSTAI battery temperature.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCSTAI_SetTemperatureBat`**`(` **`ptx_NDEFRecord_WLCSTAI_t`** `*container,` **`int8_t`** `celsius);`	
Description	Setter for the WLCSTAI battery temperature.	
Parameters	container	Pointer to the WLCSTAI container.
	current	Temperature in celsius.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.10.13. ptx_NDEFRecord_WLCSTAI_GetTemperatureBat

This function is used to get the WLCSTAI battery temperature.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCSTAI_GetTemperatureBat`**`(` **`ptx_NDEFRecord_WLCSTAI_t`** `*container,` **`int8_t`** `*celsius);`	
Description	Getter for the WLCSTAI battery temperature.	
Parameters	container	Pointer to the WLCSTAI container.
	celsius	Valid pointer to store the temperature in degrees celsius.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.10.14. ptx_NDEFRecord_WLCSTAI_SetTemperatureWlcl

This function is used to set the WLCSTAI listener temperature.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCSTAI_SetTemperatureWlcl`**`(` **`ptx_NDEFRecord_WLCSTAI_t`** `*container,` **`int8_t`** `celsius);`	
Description	Setter for the WLCSTAI listener temperature.	
Parameters	container	Pointer to the WLCSTAI container.
	current	Temperature in celsius.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.10.15. ptx_NDEFRecord_WLCSTAI_GetTemperatureWlcl

This function is used to get the WLCSTAI listener temperature.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCSTAI_GetTemperatureWlcl`**`(` **`ptx_NDEFRecord_WLCSTAI_t`** `*container,` **`int8_t`** `*celsius);`	
Description	Getter for the WLCSTAI listener temperature.	
Parameters	container	Pointer to the WLCSTAI container.
	celsius	Valid pointer to store the temperature in degrees celsius.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.10.16. ptx_NDEFRecord_WLCSTAI_SetBatteryVoltage

This function is used to set the WLCSTAI battery voltage.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCSTAI_SetBatteryVoltage`**`(` **`ptx_NDEFRecord_WLCSTAI_t`** `*container,` **`uint8_t`** `voltage);`	
Description	Setter for the WLCSTAI battery voltage.	
Parameters	container	Pointer to the WLCSTAI container.
	voltage	Battery voltage.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.10.17. ptx_NDEFRecord_WLCSTAI_GetBatteryVoltage

This function is used to get the WLCSTAI battery voltage.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCSTAI_GetBatteryVoltage`**`(` **`ptx_NDEFRecord_WLCSTAI_t`** `*container,` **`uint8_t`** `*voltage);`	
Description	Getter for the WLCSTAI battery voltage.	
Parameters	container	Pointer to the WLCSTAI container.
	voltage	Valid pointer to store the battery voltage.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.10.18. ptx_NDEFRecord_WLCSTAI_SetBatteryCurrent

This function is used to set the WLCSTAI battery current.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCSTAI_SetBatteryCurrent`**`(` **`ptx_NDEFRecord_WLCSTAI_t`** `*container,` **`uint8_t`** `current);`	
Description	Setter for the WLCSTAI battery current.	
Parameters	container	Pointer to the WLCSTAI container.
	current	Battery charge current.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.10.19. ptx_NDEFRecord_WLCSTAI_GetBatteryCurrent

This function is used to get the WLCSTAI battery current.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_WLCSTAI_GetBatteryCurrent`**`(` **`ptx_NDEFRecord_WLCSTAI_t`** `*container,` **`uint8_t`** `*current);`	
Description	Getter for the WLCSTAI battery current.	
Parameters	container	Pointer to the WLCSTAI container.
	current	Valid pointer to store the battery charge current.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.11. Generic NDEF Record API

The generic NDEF Record API provides functions to initialize and create NDEF records as well as transform serialized data to NDEF records.

7.3.4.11.1. ptx_NDEFRecord_Init

This function initializes an NDEF record container with zeros.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_Init`**`(` **`ptx_NDEFRecord_t`** `*record);`	
Description	Initializes NDEF record container.	
Parameters	record	Pointer to the NDEF record container to be initialized.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.11.2. ptx_NDEFRecord_Create

This function creates an NDEF record container from the given parameters.

The create function takes the following parameters: Type Name Format field, defined in [NFC NDEF] is mapped using the **ptx_NDEFRecord_TNF_t** enum, and maps them as follows:

- Empty: **TNF_EMPTY**
- NFC Forum Well-known Type: **TNF_WELL_KNOWN_TYPE**
- Media-type as defined in: **TNF_MEDIA_TYPE**
- Absolute URI as defined in: **TNF_ABSOLUTE_URI**
- NFC Forum External Type: **TNF_EXTERNAL_TYPE**
- Unknown: **TNF_UNKNOWN**
- Unchanged: **TNF_UNCHANGED**

Additionally, it requires the type, the payload and the individual lengths of these buffers.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_Create`** `(` **`ptx_NDEFRecord_t`** `*record,` **`ptx_NDEFRecord_TNF_t`** `tnf,` **`uint8_t`** `*type,` **`uint8_t`** `typeLen,` **`uint8_t`** `*id,` **`uint8_t`** `idLen,` **`uint8_t`** `*payload,` **`uint32_t`** `payloadLen);`	
Description	Creates an NDEF record container from the given parameters.	
Parameters	record	Pointer to an uninitialized NDEF record structure.
	tnf	Type Name Format field.
	type	Pointer to the NDEF record type.
	typeLen	Record type length.
	id	Pointer to the NDEF Record ID.
	idLen	Record ID length.
	payload	Pointer to the payload data.
	payloadLen	Payload data length.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.11.3. ptx_NDEFRecord_CreateEmptyRecord

This function prepares the **ptx_NDEFRecord_t** struct to create an empty NDEF record.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_CreateEmptyRecord`** `(` **`ptx_NDEFRecord_t`** `*record);`	
Description	Prepares the **ptx_NDEFRecord_t** struct to create an empty NDEF record.	
Parameters	record	Pointer to the NDEF record container to be initialized as empty record.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.11.4. ptx_NDEFRecord_Write

This function writes a **ptx_NDEFRecord_t** struct to a given buffer.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_Write`**`(`
	`ptx_NDEFRecord_t *ndefRecord,`
	`uint8_t *dstBuffer,`
	`uint32_t dstBufferSize,`
	`uint32_t *dataWritten);`	

Description	Writes a **ptx_NDEFRecord_t** struct to a given buffer.	
Parameters	ndefRecord	Initialized NDEF record.
	dstBuffer	Pointer to the buffer, which will contain the written data.
	dstBufferSize	Length of the provided buffer.
	dataWritten	Valid pointer to an integer, that will contain the actual amount of written data.
Return Value	Status, indicating whether the operation was successful.	

7.3.4.11.5. ptx_NDEFRecord_Parse

This function parses an NDEF record from a given buffer.

Declaration	`ptx_Status_t` **`ptx_NDEFRecord_Parse`**`(`
	`uint8_t *srcBuffer,`
	`uint32_t bufferLen,`
	`ptx_NDEFRecord_t *record);`	

Description	Parses a NDEF record from a given buffer.	
Parameters	srcBuffer	Buffer to parse the data from.
	bufferLen	Length of the provided buffer.
	record	Pointer to an **ptx_NDEFRecord_t** struct which will be initialized with the parsed data.
Return Value	Status, indicating whether the operation was successful.	

7.3.5. NDEF API States

7.3.5.1. State Overview

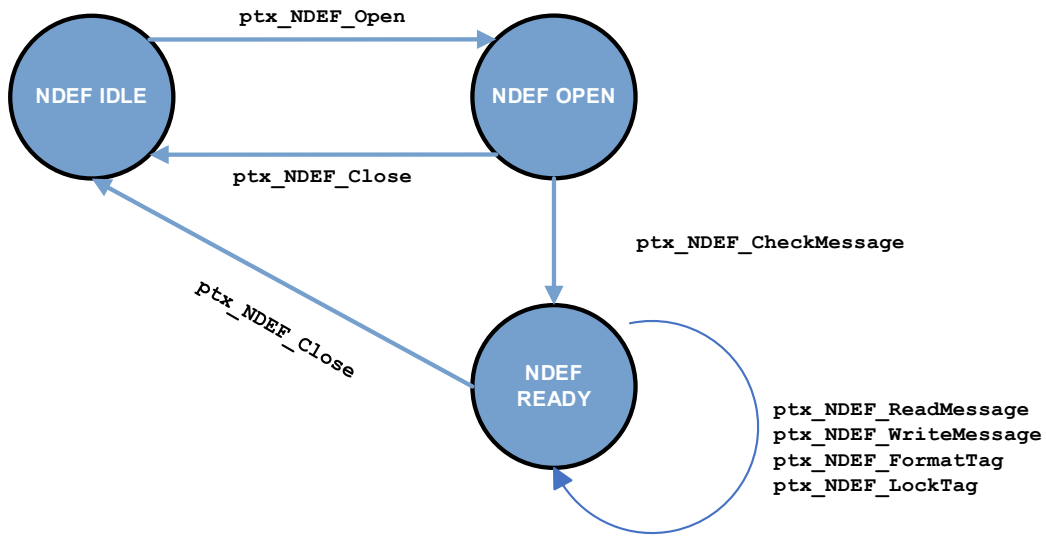

Figure 22. NDEF API States

Figure 22 describes the State diagram of the NDEF API. This State Diagram represents the structure for all the individual Tag Type APIs as well as the generic NDEF API.

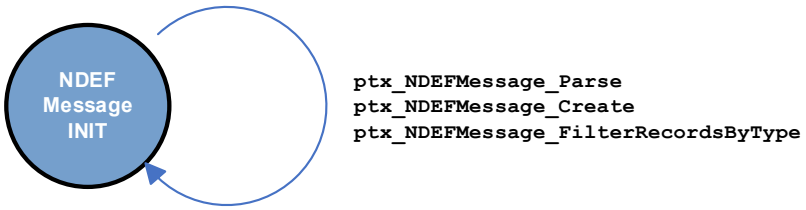

Figure 23. Message API States

Figure 23 describes the State diagram of the NDEF Message API.

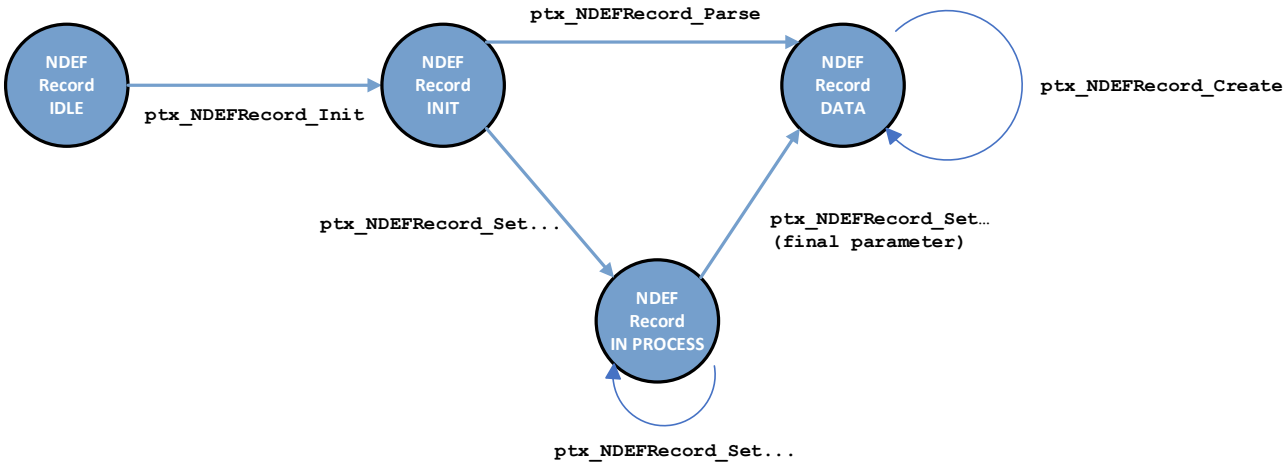

Figure 24. NDEF Record API States

Figure 24 describes the States of the NDEF Record APIs. This applies to all the supported NDEF Record Types.

7.3.5.2. State Description

The Different NDEF Tag Type APIs as well as the Generic NDEF API have the following states:

- **NDEF IDLE:** The Idle state of the NDEF API. To initialize the NDEF API, the appropriate Open function must be called. In order to return to this state, the Close function can be called at any time.
- **NDEF INIT:** The Initialized state of the NDEF API. In this state everything is ready to communicate with a Tag. In order to ensure that an NDEF message exists on the Tag, the appropriate Check function must be called.
- **NDEF READ:** In this state we can read, write and lock the NDEF message on the currently active Tag.

The NDEF message API only has a single state:

- **NDEF Message INIT:** All of the available NDEF Message API calls can be used in this state.

The NDEF Record API is very similar between each of the supported NDEF Record types. The different supported Record Types share the same API States:

- **NDEF Record IDLE:** Call the appropriate Init function to initialize the corresponding NDEF Record API.
- **NDEF Record INIT:** The appropriate NDEF Record API has been initialized and is ready for use. The NDEF Record Parse function can be used to fill the information of the Record using a preexisting buffer and can be used in combination with the received data buffer from an NDEF read message operation. Alternatively, the individual Set Parameter functions of the NDEF Record APIs can be used to manually create a record with the required information
- **NDEF Record IN PROCESS:** This state is reached when the first individual setting has been set until all the other settings have been filled out using the set parameter functions of the appropriate NDEF Record API.
- **NDEF Record DATA:** Once all of the fields have either been automatically or individually filled, this state is reached. Once this state is reached, the user can create the NDEF message containing the complete and correctly formatted NDEF record by calling the Create function.

7.3.6. Implementation Guidelines/Suggestions

- A reference usage example of the NFC Forum DTA API can be found under the "**ptx_Example_NFCForumNDEF**" folder in the reference examples.
- The NDEF, NDEF Message and NDEF Record APIs are supposed to be used in connection with one another.
  - To read the current contents of a Tag, it is suggested that you first discover the card, then use the NDEF APIs check function and following the successful execution of the Check command, that the user reads the NDEF message using the Read Message command of the NDEF API.
  - With the complete NDEF message, it is possible to use the NDEF Message API in combination with the NDEF Record APIs to extract the individual NDEF records from the received NDEF message using the Parse method of the NDEF Message API.
  - In the opposite case where the user wishes to write an NDEF message, it is suggested that the user creates the individual NDEF records using the NDEF Records API and combine them into one NDEF message with the NDEF Message Create function.
  - The created NDEF message can then be written to the Tag using the Write Message command of the NDEF API.
  - Be aware that there is a special case for MIFARE DESFire EV1 cards. For these type of cards it is not possible to call Check Message more than once after it was activated and the NDEF application was selected on the card. The card needs to be deactivated first, by removing the field before proceeding.
- The NDEF API requires three buffers to function correctly. These are passed to the component during initialization and are used to store the commands and responses that are sent and received from the tag. The standard size for these buffers is 256 bytes.

7.4 Logical Link Control Protocol (LLCP) API

The Logical Link Control Protocol (LLCP) is a communication protocol operating at the data link layer of the NFC protocol stack. It serves as a standardized protocol for establishing logical connections between NFC peer-to-peer enabled devices. One of LLCPs key features is the ability to handle multiple logical connection across a single physical NFC interface allowing concurrent communication sessions, making it possible for multiple applications to use the NFC connection simultaneously. LLCP also incorporates flow control mechanisms to manage data transmission rates and to prevent buffering.

7.4.1. API Architecture

The LLCP module registers itself as an add-on library to the RUL component, which in turn forwards vital information back to LLCP via callbacks, notifying the LLCP component. For example, new communications peers in the RF environment or also to receive data from the NFC peer, or to get notified when the RF-Field of the LLCP initiator is lost. All outgoing data transmission of LLCP are done via the **ptx_RUL_Exchange()** API.

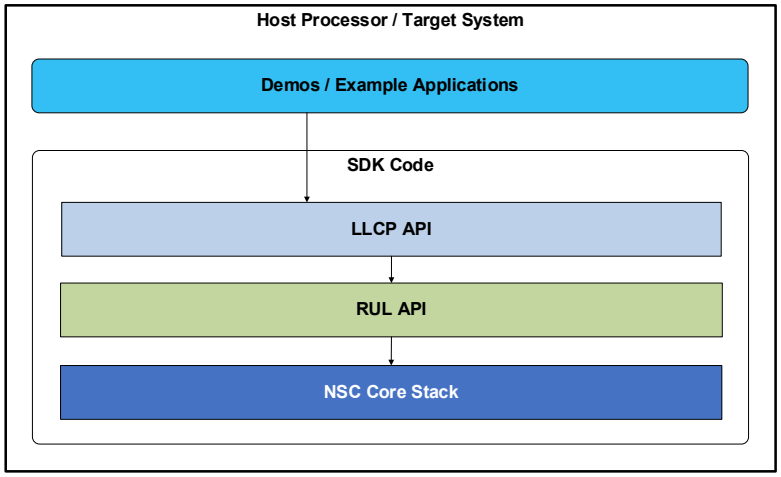

Figure 25. LLCP Module API Component Architecture

7.4.2. API Description

7.4.2.1. ptx_RUL_LLCP_Init()

Declaration	`ptx_Status_t ptx_RUL_LLCP_Init ( ptx_RUL_LLCP_t *ctx, ptx_RUL_LLCP_InitParameters_t *initParameters );`	
Description	This function initializes the LLCP Addon API, sets up all internal data structures and registers the required callbacks to RUL.	
Parameters	ctx	Pointer to the allocated LLCP component.
	initParameters	Pointer to the initialization parameters.
Return Value	Status, indicating whether the operation was successful.	

7.4.2.2. ptx_RUL_LLCP_DeInit()

Declaration	`ptx_Status_t ptx_RUL_LLCP_Init ( ptx_RUL_LLCP_t *ctx );`	
Description	This function deinitializes the LLCP Addon API, clears the internal data structures and deregisters all callbacks.	
Parameters	ctx	Pointer to the allocated LLCP component.
Return Value	Status, indicating whether the operation was successful.	

7.4.2.3. ptx_RUL_LLCP_SetCustomGeneralBytes()

Declaration	`ptx_Status_t ptx_RUL_LLCP_SetCustomGeneralBytes ( ptx_RUL_LLCP_t *ctx, uint8_t *generalBytes, uint8_t generalBytesLen );`	
Description	This function allows the user to set custom general bytes, which are being used during the device activation.	
Parameters	ctx	Pointer to the allocated LLCP component.
	generalBytes	Pointer to a buffer containing the general bytes.
	generalBytesLen	Length of the general bytes to be set.
Return Value	Status, indicating whether the operation was successful.	

7.4.2.4. ptx_RUL_LLCP_SetApplicationTimeout()

Declaration	`ptx_Status_t ptx_RUL_LLCP_SetApplicationTimeout ( ptx_RUL_LLCP_t *ctx, uint32_t applicationTimeout );`	
Description	Sets the application timeout in milliseconds, hence defining the maximum amount of time an LLCP API function blocks the execution.	
Parameters	ctx	Pointer to the allocated LLCP component.
	applicationTimeout	Application timeout in milliseconds
Return Value	Status, indicating whether the operation was successful.	

7.4.2.5. ptx_RUL_LLCP_GenerateDefaultGeneralBytes()

Declaration	`ptx_Status_t` **`ptx_RUL_LLCP_GenerateDefaultGeneralBytes`** `(`
	`ptx_RUL_LLCP_t` `*ctx,`
	`uint8_t` `*generalBytes,`
	`uint8_t` `*generalBytesLen`
	`);`	

Description	This function writes the default general bytes to a buffer provided by the user.	
Parameters	ctx	Pointer to the allocated LLCP component.
	generalBytes	Pointer to a buffer for storing the general bytes.
	generalBytesLen	Size of the buffer.
Return Value	Status, indicating whether the operation was successful.	

7.4.2.6. ptx_RUL_LLCP_StartDefaultDiscovery()

Declaration	`ptx_Status_t` **`ptx_RUL_LLCP_StartDefaultDiscovery`** `(`
	`ptx_RUL_LLCP_t` `*ctx`
	`);`	

Description	Starts the default RF-Discovery in NFC-DEP Listen & Poll mode either with the provided or the default general bytes.	
Parameters	ctx	Pointer to the allocated LLCP component.
Return Value	Status, indicating whether the operation was successful.	

7.4.2.7. ptx_RUL_LLCP_WaitForActivation()

Declaration	`ptx_Status_t` **`ptx_RUL_LLCP_WaitForActivation`** `(`
	`ptx_RUL_LLCP_t` `*ctx,`
	`uint8_t` `*activated,`
	`uint32_t` `timeout,`
	`);`	

Description	Waits until a successful LLCP connection has been established (determined be the LLCP Magic Bytes). This API works both for Target and Listen mode.	
Parameters	ctx	Pointer to the allocated LLCP component.
	activated	Pointer to store the information whether the device has been activated.
	timeout	Timeout in milliseconds determining the maximum waiting time.
Return Value	Status, indicating whether the operation was successful.	

7.4.2.8. ptx_RUL_LLCP_WaitForConnection()

Declaration	`ptx_Status_t` **`ptx_RUL_LLCP_WaitForConnection`** `(`
	`  `**`ptx_RUL_LLCP_t`** `*ctx,`
	`  `**`ptx_RUL_LLCP_Link_t`** `*link,`
	`);`	

Description	Waits until any connection has been made by the remote peer to the specified link.	
Parameters	ctx	Pointer to the allocated LLCP component.
	link	Pointer to the link of interest.
Return Value	Status, indicating whether the operation was successful.	

7.4.2.9. ptx_RUL_LLCP_WaitForAnyConnection()

Declaration	`ptx_Status_t` **`ptx_RUL_LLCP_WaitForAnyConnection`** `(`
	`  `**`ptx_RUL_LLCP_t`** `*ctx,`
	`  `**`uint32_t`** `timeout`
	`);`	

Description	Waits until any connection has been made by the remote peer. This API is mainly needed for the DTA and will likely not be used in a real-world scenario.	
Parameters	ctx	Pointer to the allocated LLCP component.
	timeout	Timeout determining the maximum waiting time
Return Value	Status, indicating whether the operation was successful.	

7.4.2.10. ptx_RUL_LLCP_SendInfo()

Declaration	`ptx_Status_t` **`ptx_RUL_LLCP_SendInfo`** `(`
	`  `**`ptx_RUL_LLCP_t`** `*ctx,`
	`  `**`ptx_RUL_LLCP_Link_t`** `*link,`
	`  `**`uint8_t`** `*payload,`
	`  `**`uint8_t`** `*payloadLength,`
	`  `**`uint8_t`** `waitForAck`
	`);`	

Description	Transmits an Information PDU across the specifier LLCP Link.	
Parameters	ctx	Pointer to the initialized LLCP component.
	link	Pointer to the link to be used for the transmission.
	payload	Pointer to the payload buffer to be transmitted.
	payloadLength	Length of the payload to be transmitted.
	waitForAck	If set to 0, the Information PDU will be transmitted, and the function will return immediately. Otherwise, the function will block until the communication Peer has acknowledged the message.
Return Value	Status, indicating whether the operation was successful.	

7.4.2.11. ptx_RUL_LLCP_SendUInfo()

Declaration	`ptx_Status_t ptx_RUL_LLCP_SendUInfo ( ptx_RUL_LLCP_t *ctx, ptx_RUL_LLCP_Link_t *link, uint8_t *payload, uint8_t *payloadLength );`	
Description	Transmits an Unnumbered Information PDU across the specifier LLCP Link.	
Parameters	ctx	Pointer to the initialized LLCP component.
	link	Pointer to the link to be used for the transmission.
	payload	Pointer to the payload buffer to be transmitted.
	payloadLength	Length of the payload to be transmitted.
Return Value	Status, indicating whether the operation was successful.	

7.4.2.12. ptx_RUL_LLCP_DiscoverService()

Declaration	`ptx_Status_t ptx_RUL_LLCP_DiscoverService ( ptx_RUL_LLCP_t *ctx, ptx_RUL_LLCP_ServiceName_t *serviceName, uint8_t *dSap );`	
Description	Sends a Service Discovery request to the communication Peer and returns the destination SAP address linked to the specified service name.	
Parameters	ctx	Pointer to the initialized LLCP component.
	serviceName	Pointer to the service name.
	dSap	Pointer to store the received destination SAP address.
Return Value	Status, indicating whether the operation was successful.	

7.4.2.13. ptx_RUL_LLCP_ReceiveFromLink()

Declaration	`ptx_Status_t ptx_RUL_LLCP_ReceiveFromLink ( ptx_RUL_LLCP_t *ctx, ptx_RUL_LLCP_Link_t *link );`	
Description	This function blocks until a PDU has been received by the provided link or the application timeout has been reached.	
Parameters	ctx	Pointer to the initialized LLCP component.
	link	Pointer to the link to be used for the transmission.
	payload	Pointer to the payload buffer to be transmitted.
	payloadLength	Length of the payload to be transmitted.
Return Value	Status, indicating whether the operation was successful.	

7.4.2.14. ptx_RUL_LLCP_ConnectSap()

Declaration	`ptx_Status_t ptx_RUL_LLCP_ConnectSap ( ptx_RUL_LLCP_t *ctx, ptx_RUL_LLCP_Link_t *link, uint8_t dsap );`	
Description	This function connects to the remote SAP using its address.	
Parameters	ctx	Pointer to the initialized LLCP component.
	link	Pointer to the allocated link to be used for the connected establishment.
	dSap	The destination SAP address to connect to.
Return Value	Status, indicating whether the operation was successful.	

7.4.2.15. ptx_RUL_LLCP_ConnectSapByName()

Declaration	`ptx_Status_t ptx_RUL_LLCP_ConnectSapByName ( ptx_RUL_LLCP_t *ctx, ptx_RUL_LLCP_Link_t *link, char* name );`	
Description	This function connects to the remote SAP using its name.	
Parameters	ctx	Pointer to the initialized LLCP component.
	link	Pointer to the allocated link to be used for the connected establishment.
	name	The destination SAP name to connect to. The char array must be zero terminated.
Return Value	Status, indicating whether the operation was successful.	

7.4.2.16. ptx_RUL_LLCP_DisconnectSap()

Declaration	`ptx_Status_t ptx_RUL_LLCP_DisconnectSap ( ptx_RUL_LLCP_t *ctx, ptx_RUL_LLCP_Link_t *link );`	
Description	This function disconnects the specified logical link.	
Parameters	ctx	Pointer to the initialized LLCP component.
	link	Pointer to the logical link to be disconnected.
Return Value	Status, indicating whether the operation was successful.	

7.4.2.17. ptx_RUL_LLCP_RegisterSap()

Declaration	`ptx_Status_t ptx_RUL_LLCP_RegisterSap ( ptx_RUL_LLCP_t *ctx, ptx_RUL_LLCP_Link_t **link, ptx_RUL_LLCP_Link_InitParameters_t *params );`	
Description	Registers an SAP locally using the provided initialization parameters. Once registered, the external peer can connect to it.	
Parameters	ctx	Pointer to the initialized LLCP component.
	link	A pointer for storing the link structure.
	params	A pointer to the link's initialization parameters.
Return Value	Status, indicating whether the operation was successful.	

7.4.2.18. ptx_RUL_LLCP_UnregisterSap()

Declaration	`ptx_Status_t ptx_RUL_LLCP_UnregisterSap ( ptx_RUL_LLCP_t *ctx, ptx_RUL_LLCP_Link_t *link);`	
Description	Unregisters a local SAP. The remote peer will not be able to connect to the SAP anymore.	
Parameters	ctx	Pointer to the initialized LLCP component.
	link	Pointer to a Link pointer.
Return Value	Status, indicating whether the operation was successful.	

7.4.2.19. ptx_RUL_LLCP_GetDisconnectReason()

Declaration	`ptx_Status_t ptx_RUL_LLCP_GetDisconnectReason ( ptx_RUL_LLCP_t *ctx, ptx_RUL_LLCP_DisconnectReason_t *reason );`	
Description	Retrieves the last reason why the connection has been terminated.	
Parameters	ctx	Pointer to the initialized LLCP component.
	reason	Pointer to store the information why the connection has been terminated.
Return Value	Status, indicating whether the operation was successful.	

7.4.2.20. ptx_RUL_LLCP_IsLlcpSupported()

Declaration	`ptx_Status_t` **`ptx_RUL_LLCP_IsLlcpSupported`** (**`ptx_RUL_LLCP_t`** `*ctx,` **`ptx_RUL_LLCP_Link_t`** `**link,` **`ptx_RUL_LLCP_Link_InitParameters_t`** `*params);`	
Description	Registers an SAP locally using the provided initialization parameters. Once registered, the external Peer can connect to it.	
Parameters	ctx	Pointer to the initialized LLCP component.
	link	Pointer to a Link pointer.
Return Value	Status, indicating whether the operation was successful.	

7.4.2.21. ptx_RUL_LLCP_IsLinkDown()

Declaration	`ptx_Status_t` **`ptx_RUL_LLCP_IsLinkDown`** (**`ptx_RUL_LLCP_t`** `*ctx,` **`uint8_t`** `*isDown);`	
Description	Checks if the main link is down.	
Parameters	ctx	Pointer to the initialized LLCP component.
	isDown	Pointer to store the information whether the link is down or not.
Return Value	Status, indicating whether the operation was successful.	

7.4.2.22. ptx_RUL_LLCP_Handle()

Declaration	`ptx_Status_t` **`ptx_RUL_LLCP_Handle`** (**`ptx_RUL_LLCP_t`** `*ctx);`	
Description	Handles all LLCP related communications. Must be called periodically.	
Parameters	ctx	Pointer to the initialized LLCP component.
Return Value	Status, indicating whether the operation was successful.	

7.4.3. LLCP API States

7.4.3.1. LLCP State Overview

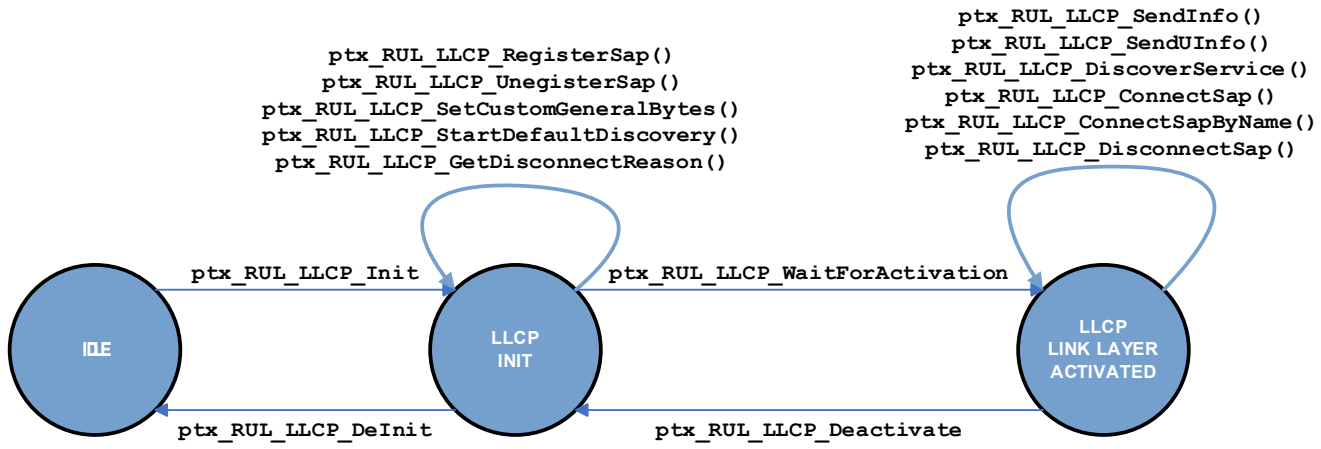

Figure 26. LLCP API State

7.4.3.2. State Description

- **IDLE**: The Idle state is the initial state of the component when it is still uninitialized. To get to the **INIT** state, make a call to **ptx_RUL_LLCP_Init().**
- **INIT**: The component and all internal data structures are initialized. In this state, the RF-Discovery can be started via the API call **ptx_RUL_LLCP_StartDefaultDiscovery()**. In this state SAPs can be registered and deregistered. To recognize whether a successful activation of a counterpart supporting LLCP has occurred, call the API function **ptx_RUL_LLCP_WaitForActivation()**.
- **ACTIVATED**: If the connection is successfully established, the peers can now use the full functionality of LLCP. In this state (Unnumbered) Information PDUs can be sent. Connections to remote SAPs can be established or torn down, as well as service discoveries can be started.

7.4.4. Implementation Guidelines/Suggestions

- LLCP requires permanent RF communication via the RF link. If one of the communication partners has nothing to send, it must send an empty data packet, the so-called Symmetry-PDU within the "link timeout". The LLCP component takes care of this action. However, if long delays are built into the main application, for example, by calling **ptx_RUL_Sleep()**, the LLCP component will most likely not send the Symmetry-PDU in time, causing a communication fail. Sleeps should generally be avoided if they are longer than 10ms. Longer timeouts should be replaced with small sleeps in a loop where **ptx_RUL_LLCP_Handle()** is regularly called to handle all LLCP related communications (see example below).
- LLCP is a component that can take up a lot of RAM in the RUL-Stack if misconfigured. The implementor must weigh up exactly which application requirements have to be fulfilled while considering the available RAM on the chosen HW platform.

  Two fundamental definitions of RAM utilization are described in the following paragraphs:

  - **PTX_LLCP_DEFAULT_RW**

    This define sets the maximum size if the Receive Window (RW), a key part of the flow control mechanism in LLCP, is defined by this setting. It determines the number of consecutive Information (I) frames that can be sent by a peer without waiting for an acknowledgment. This setting, while crucial for managing data flow, has a substantial effect on RAM usage. An LLCP PDU can have a maximum information unit of 253 bytes. With a receive window of 16, this results in a RAM consumption of (253 × 16 × 2) = 8kByte. Buffering occurs in both (Rx and Tx) directions, hence the multiplicator of two.
  - **PTX_LLCP_MAX_LINKS**

    This define specifies the maximum number of supported LLCP connections that can run in parallel. If the definition is set to four, a maximum of four connections to the LLCP peer can be maintained in parallel. Considering this definition together with the default RW size is particularly important. The number of links is added as a multiplicative factor to the calculation for the Receive Window RAM consumption above. For example, having a maximum of four supported links, with an RW size of 16, results in a minimum RAM consumption of (253 × 16 × 2) × 4 = 32kByte.

7.5 Simple NDEF Exchange Protocol (SNEP) API

The Simple NDEF Exchange Protocol (SNEP) facilitates the exchange of NDEF (NFC Data Exchange Format) messages between two NFC Peer-to-Peer enabled devices.

SNEP operates on a client-server model, where one device acts as the client, initiating the request, and the other as the server, responding to the request. It is commonly used for data sharing, such as exchanging contact information, URLs, or small data files between devices. For example, users can share business cards by simply tapping their phones together.

7.5.1. SNEP Operations

The two primary operations within SNEP are the “PUT” and “GET” operations, each serving a distinct function in the communication process.

7.5.1.1. SNEP PUT Operation

The SNEP PUT operation is used to send an NDEF message from one device to another. This process begins when the client device constructs an NDEF message that it wishes to send. The client then transmits this message to the server device using SNEP. Upon receiving the NDEF message, the server processes it according to its intended purpose, which could involve storing the message, acting on its payload, or forwarding it to another service or application. The PUT operation is commonly used for various applications, such as sharing contact information, URLs, or small data files between devices. For instance, when two smartphones are tapped together to exchange business cards, the NDEF message containing the contact information is transmitted using the PUT operation.

7.5.1.2. SNEP GET Operation

The SNEP GET operation allows a device to request an NDEF message from another device. The process begins when the client device sends a request to the server device, indicating its desire to retrieve an NDEF message. The server processes this GET request and retrieves the appropriate NDEF message, which could involve reading the message from memory or generating it dynamically based on the request. The server then transmits the retrieved NDEF message back to the client device, which receives the message and processes its content as needed. The GET operation is particularly useful in scenarios where a device needs to pull specific information from another device. For example, a smartphone might use the GET operation to retrieve promotional content from an NFC-enabled poster or kiosk.

7.5.2. API Architecture

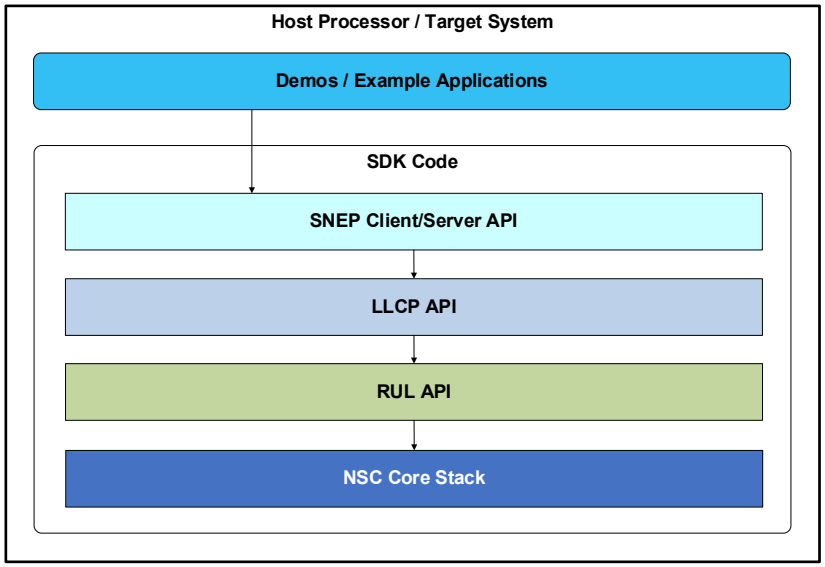

Figure 27. SnepClient Module API Component Architecture

7.5.3. API Description

7.5.3.1. ptx_RUL_SnepClient_Init

Declaration	`ptx_Status_t` **`ptx_RUL_SnepClient_Init`** (**`ptx_RUL_SnepClient_t`** *ctx, **`ptx_RUL_t`** *rul, **`uint8_t`** enableDtaMode);	
Description	This function initializes the SNEP Client Addon API, sets up all internal data structures and registers the required callbacks to RUL.	
Parameters	ctx	Pointer to allocated SnepClient component.
	rul	Pointer to the initialized RUL component.
	enableDtaMode	Flag indicating if the DTA mode should be enabled. Certain compliance test cases require this special operating mode. Set to 0 if not used.
Return Value	Status, indicating whether the operation was successful.	

7.5.3.2. ptx_RUL_SnepClient_DeInit

Declaration	`ptx_Status_t` **`ptx_RUL_SnepClient_DeInit`** (**`ptx_RUL_SnepClient_t`** *ctx);	
Description	This function deinitializes the SNEP Client Addon API, resets all internal data structures, and de-registers all callbacks.	
Parameters	ctx	Pointer to initialized SnepClient component.
Return Value	Status, indicating whether the operation was successful.	

7.5.3.3. ptx_RUL_SnepClient_StartDefaultDiscovery

Declaration	`ptx_Status_t` **`ptx_RUL_SnepClient_StartDefaultDiscovery`** (**`ptx_RUL_SnepClient_t`** *ctx);	
Description	Starts the default RF-discovery required for SNEP.	
Parameters	ctx	Pointer to initialized SnepClient component.
	IdleTime	IdleTime to be used for the RF-discovery.
Return Value	Status, indicating whether the operation was successful.	

7.5.3.4. ptx_RUL_SnepClient_WaitForActivation

Declaration	`ptx_Status_t` **`ptx_RUL_SnepClient_WaitForActivation`** (**`ptx_RUL_SnepClient_t`** *ctx);	
Description	Will wait until a connection to a peer has been established.	
Parameters	ctx	Pointer to initialized SnepClient component.
Return Value	Status, indicating whether the operation was successful.	

7.5.3.5. ptx_RUL_SnepClient_ConnectToServer()

Declaration	`ptx_Status_t` **`ptx_RUL_SnepClient_ConnectToServer`** (**`ptx_RUL_SnepClient_t`** *ctx,);	
Description	This function will start the default discovery for SNEP and will wait until the connection to the server has been established or a timeout has occurred.	
Parameters	Ctx	Pointer to the initialized SnepClient component.
Return Value	Status, indicating whether the operation was successful.	

7.5.3.6. ptx_RUL_SnepClient_PutMessage()

Declaration	`ptx_Status_t` **`ptx_RUL_SnepClient_PutMessage`** (**`ptx_RUL_SnepClient_t`** *ctx, **`uint8_t`** *message, **`uint32_t`** messageLength);	
Description	Initiates a SNEP PUT-Request to the SNEP server. This API will block until the request is finished or until an error has occurred.	
Parameters	ctx	Pointer to the initialized SnepClient component.
	message	Pointer to the buffer storing the NDEF message to be transmitted.
	messageLength	Length of the message to be transmitted.
Return Value	Status, indicating whether the operation was successful.	

7.5.3.7. ptx_RUL_SnepClient_GetMessage()

Declaration	`ptx_Status_t` **`ptx_RUL_SnepClient_GetMessage`** (**`ptx_RUL_SnepClient_t`** *ctx, **`uint8_t`** message, **`uint32_t`** messageLength);	
Description	Initiates a SNEP GET-Request to the SNEP server. This API will block until the request is finished or until an error has occurred.	
Parameters	ctx	Pointer to the initialized SnepClient component.
	message	Pointer to the buffer containing the NDEF request message.
	messageLength	Length of the NDEF request message.
Return Value	Status, indicating whether the operation was successful.	

7.5.3.8. ptx_RUL_SnepClient_Handle()

Declaration	`ptx_Status_t` **`ptx_RUL_SnepClient_Handle`** (**`ptx_RUL_SnepClient_t`** *ctx);	
Description	Handles the complete SNEP protocol communication. This API function must be called periodically.	
Parameters	ctx	Pointer to the initialized SnepClient component.
Return Value	Status, indicating whether the operation was successful.	

7.5.3.9. ptx_RUL_SnepServer_Init

Declaration	`ptx_Status_t` **`ptx_RUL_SnepServer_Init`** (**`ptx_RUL_SnepServer_t`** *ctx, **`ptx_RUL_t`** *rul, **`uint8_t`** enableDtaMode);	
Description	This function initializes the SNEP Server Add-on API, sets up all internal data structures and registers the required callbacks to RUL.	
Parameters	ctx	Pointer to allocated SnepServer component.
	rul	Pointer to the initialized RUL component.
	enableDtaMode	Flag indicating if the DTA mode should be enabled. Certain compliance test cases require this special operating mode. Set to 0 if not used.
Return Value	Status, indicating whether the operation was successful.	

7.5.3.10. ptx_RUL_SnepServer_DeInit

Declaration	`ptx_Status_t` **`ptx_RUL_SnepServer_DeInit`** (**`ptx_RUL_SnepServer_t`** *ctx);	
Description	This function deinitializes the SNEP Server Add-on API, resets all internal data structures, and de-registers the callbacks.	
Parameters	ctx	Pointer to initialized SnepServer component.
Return Value	Status, indicating whether the operation was successful.	

7.5.3.11. ptx_RUL_SnepServer_StartDefaultDiscovery

Declaration	`ptx_Status_t` **`ptx_RUL_SnepClient_StartDefaultDiscovery`** (**`ptx_RUL_SnepServer_t`** *ctx);	
Description	Starts the default RF-discovery required for SNEP.	
Parameters	ctx	Pointer to initialized SnepServer component.
	idleTime	IdleTime to be used for the RF-discovery.
Return Value	Status, indicating whether the operation was successful.	

7.5.3.12. ptx_RUL_SnepServer_WaitForActivation

Declaration	`ptx_Status_t` **`ptx_RUL_SnepClient_WaitForActivation`** (**`ptx_RUL_SnepServer_t`** *ctx);	
Description	Will wait until a connection to a peer has been established.	
Parameters	ctx	Pointer to initialized SnepServer component.
Return Value	Status, indicating whether the operation was successful.	

7.5.3.13. ptx_RUL_SnepServer_WaitForClient()

Declaration	`ptx_Status_t ptx_RUL_SnepServer_WaitForClient ( ptx_RUL_SnepServer_t *ctx, uint32_t timeout );`	
Description	Waits for an incoming connection from a client.	
Parameters	ctx	Pointer to the initialized SnepServer component.
Return Value	Status, indicating whether the operation was successful.	

7.5.3.14. ptx_RUL_SnepServer_PutMessage()

Declaration	`ptx_Status_t ptx_RUL_SnepServer_PutMessage ( ptx_RUL_SnepServer_t *ctx, uint8_t *message, uint32_t messageLength);`	
Description	Writes an NDEF message into the internal data buffer. The message can be retrieved by the SnepClient via a SNEP GET operation.	
Parameters	ctx	Pointer to the initialized SnepServer component.
	message	Pointer to the buffer storing the NDEF message to be transmitted.
	messageLength	Length of the message to be transmitted.
Return Value	Status, indicating whether the operation was successful.	

7.5.3.15. ptx_RUL_SnepServer_GetMessage()

Declaration	`ptx_Status_t ptx_RUL_SnepServer_GetMessage ( ptx_RUL_SnepServer_t *ctx, uint8_t *message, uint32_t *messageLength);`	
Description	Retrieves an NDEF message from the internal data buffer written to the Server via a SNEP PUT operation.	
Parameters	ctx	Pointer to the initialized SnepServer component.
	message	Pointer to a buffer for storing the received NDEF message.
	messageLength	Pointer to a variable storing the size of the buffer.
Return Value	Status, indicating whether the operation was successful.	

7.5.3.16. ptx_RUL_SnepServer_Handle()

Declaration	`ptx_Status_t ptx_RUL_SnepServer_Handle( ptx_RUL_SnepServer_t *ctx);`	
Description	This function must be executed regularly to answer the client's requests.	
Parameters	ctx	Pointer to the initialized SnepServer component.
Return Value	Status, indicating whether the operation was successful.	

7.5.4. SNEP API States

The following sections show the API state for the SNEP Client and the SNEP server.

7.5.4.1. SnepClient State Overview

Figure 28 shows the state sequence of the SnepClient.

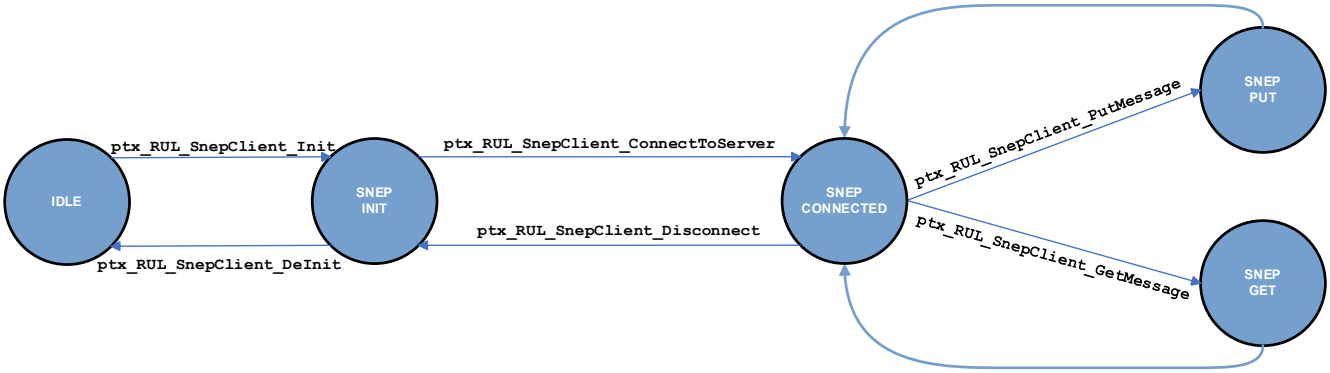

Figure 28. SnepClient API State

7.5.4.2. SnepClient State Description

- **IDLE**: The Idle state is the initial state of the component when it is still uninitialized. To get to the **INIT** state, make the API call **ptx_RUL_SnepClient_Init().**
- **INIT**: The component and all internal data structures are initialized. In this state, the RF discovery can be started via the API call **ptx_RUL_SnepCLient_ConnectToServer()**. It remains in this state until the connection to a server has been established. If this is the case, the **CONNECTED** state is reached.
- **CONNECTED**: If the connection is successfully established, the client can now either enter the **PUT** or **GET** state by calling **ptx_RUL_SnepClient_PutMessage**() or **ptx_RUL_SnepClient_GetMessage()** respectively.
- **PUT**: In this state, the client issues a PUT request to the SNEP server. Once the request has been processed, it will return to the state CONNECTED. Another request can be made, if necessary.
- **GET**: In this state, the client issues a GET request to the SNEP server. Once the request has been processed, it will return to the state CONNECTED. Another request can be made, if necessary.

7.5.4.3. SnepServer State Overview

The state diagram below shows the state sequence of the SnepServer.

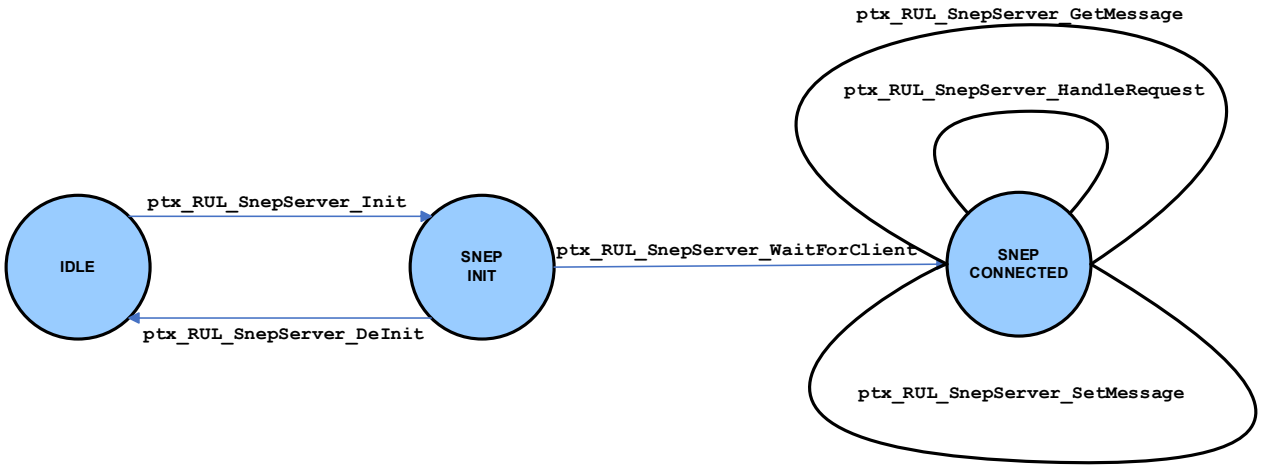

Figure 29. SnepServer API State

7.5.4.4. SnepServer State Description

- **IDLE**: The Idle state is the initial state of the component when it is still uninitialized. To get to the **INIT** state, make the API call **ptx_RUL_SnepServer_Init().**
- **INIT**: The component and all internal data structures are initialized. In this state, the RF-discovery can be started via the API call **ptx_RUL_SnepServer_WaitForClient()**. It remains in this state until the connection with a client has been established. If this is the case, the **CONNECTED** state is reached.
- **CONNECTED**: If the connection is successfully established, the client should repeatedly call **ptx_RUL_SnepServer_Handle()** to get notified about incoming requests.
- **PUT**: In this state, the client issues a PUT request to the SNEP server. Once the request has been processed, it will return to the state CONNECTED. Another request can be made, if necessary.
- **GET**: In this state, the client issues a GET request to the SNEP server. Once the request has been processed, it will return to the state CONNECTED. Another request can be made, if necessary.

7.5.5. Implementation Guidelines/Suggestions

- The Simple NDEF Exchange Protocol (SNEP) is supported on Android devices running versions 4.0 (Ice Cream Sandwich) through 13 (Tiramisu). To use SNEP for peer-to-peer NFC data exchange, you need to enable “Android Beam” on your device.
- All data transfers using SNEP must be encapsulated within NDEF (NFC Data Exchange Format). Use the NDEF API to properly format/encapsulate your data.
- SNEP build on top of LLCP which requires permanent RF communication via the RF link. If one of the communication partners has nothing to send, it must send an empty data packet, the so-called Symmetry-PDU within the "link timeout". The LLCP component takes care of this. However, if long delays are built into the main application, for example, by calling **ptx_RUL_Sleep()**, the LLCP component can most likely not send the Symmetry-PDU in time, causing a communication fail. Sleeps should generally be avoided if they are longer than 10ms. Longer timeouts should be replaced with small sleeps in a loop where **ptx_RUL_LLCP_Handle()** is regularly called to handle all LLCP related communications (see example below).

7.6 Transparent Mode API

The optional Transparent Mode API enables an application to implement standard or proprietary protocols based on low-level RF commands, such as:

- RF-Technology Type A / NFC – A
  - REQA / WUPA, ANTICOLLISION, SELECT, …
- RF-Technology Type F / NFC – F
  - SENSF_REQ, SENSF_RES
- Apple VAS/ECP
- Former NFC Forum Type Tag 1
- …

7.6.1. API Architecture

The Transparent Mode API module is built on top of the NSC Core Stack component. The internal dependency is shown in Figure 30 below.

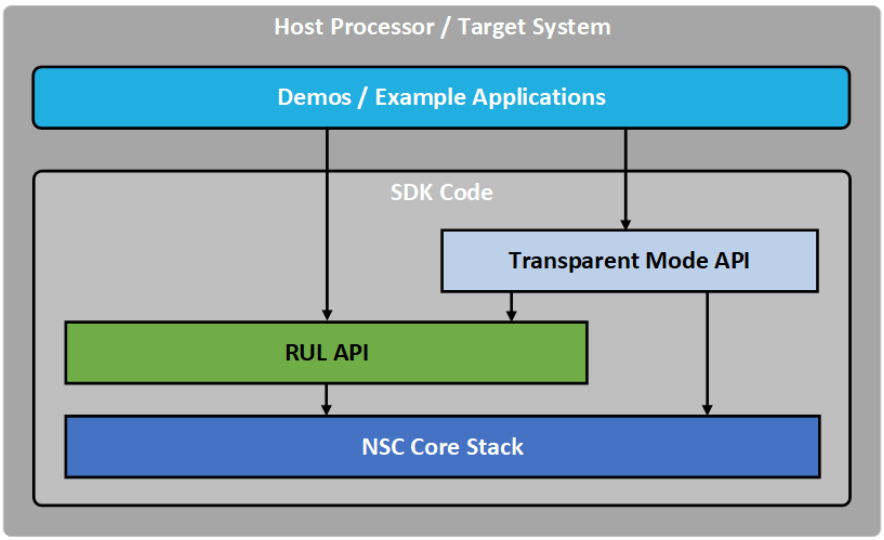

Figure 30. Transparent Mode API Component Architecture

Important: This API works independently of the implemented on-chip standard RF discovery procedure (for example, NFC Forum/ISO). Until explicitly mentioned, this API should not be mixed up with the standard RUL API functions provided by this SDK as described in section 3.

7.6.2. API Description

7.6.2.1. ptx_TransparentMode_Open

The Open function initializes the Transparent Mode component. It requires an initialized instance of a RUL component and a flag that indicates if default RF-Parameter reset values should be used for a deactivated RF field in **RF OFF** state of the Transparent Mode. These parameters are provided using the **ptx_TransparentMode_InitParams_t** struct. This function initializes the Transparent Mode component to the **RF OFF** state.

Declaration	`ptx_Status_t` **`ptx_TransparentMode_Open`** `(` **`ptx_TransparentMode_t`** `*tmComp,` **`ptx_TransparentMode_InitParams_t`** `*initParams);`	
Description	Initialization of the Transparent mode component.	
Parameters	tmComp	Pointer to allocated Transparent mode component.
	initParams	Pointer to initialization parameter structure.
Return Value	Status, indicating whether the operation was successful.	

7.6.2.2. ptx_TransparentMode_Close

The **Close** function de-initializes the Transparent mode component. If not in **RF OFF** State, this function automatically switches to it before de-initializing the component.

Declaration	`ptx_Status_t` **`ptx_TransparentMode_Close`** (**`ptx_TransparentMode_t`** `*tmComp);`	
Description	De-Initialization of the Transparent mode component.	
Parameters	tmComp	Pointer to allocated Transparent mode component.
Return Value	Status, indicating whether the operation was successful.	

7.6.2.3. ptx_TransparentMode_ResetState

This function resets the internal component state without the need to restart the entire API. Settings that were set in the initializer function are kept.

Declaration	`ptx_Status_t` **`ptx_TransparentMode_ResetState`** (**`ptx_TransparentMode_t`** `*tmComp);`	
Description	Resets the internal component state.	
Parameters	tmComp	Pointer to allocated Transparent mode component.
Return Value	Status, indicating whether the operation was successful.	

7.6.2.4. ptx_TransparentMode_SetState

The SetState function enables to switch between certain states of the Transparent mode component. The available states are **RF OFF**, **RF ON** and **LPCD POLL**. The SetState function requires an initialized component and one of the three available states. An application must call this function before any further call to **ptx_TransparentMode_SetRFParameters()**, **ptx_TransparentMode_CheckLpcd()** and/or **ptx_TransparentMode_Exchange()**, a description which function is valid for which state is given in the following sections.

Declaration	`ptx_Status_t` **`ptx_TransparentMode_SetState`** (**`ptx_TransparentMode_t`** `*tmComp,` **`ptx_TransparentMode_State_t`** `state);`	
Description	Switches state of Transparent Mode.	
Parameters	tmComp	Pointer to allocated Transparent mode component.
	state	State of Transparent mode.
Return Value	Status, indicating whether the operation was successful.	

7.6.2.5. ptx_TransparentMode_CheckLpcd

Checks whether a card was detected or not. If a card was detected, it switches Transparent mode automatically to **RF ON** State. This function can only be used in **LPCD POLL** state. Note that the LPCD measurement detects impedance changes in the RF field and not directly a card.

Declaration	`ptx_Status_t` **`ptx_TransparentMode_CheckLpcd`** (**`ptx_TransparentMode_t`** `*tmComp,` **`ptx_TransparentMode_Lpcd_State_t*`** `cardDetected);`	
Description	Checks whether a card was detected or not.	
Parameters	tmComp	Pointer to allocated Transparent mode component.
	cardDetected	Gives information whether a card was detected or not.
Return Value	Status, indicating whether the operation was successful.	

7.6.2.6. ptx_TransparentMode_SetRFParameters

The SetRFParameters function configures the HW for the following call(s) to **ptx_TransparentMode_Exchange()**. The required parameters are provided using the **ptx_TransparentMode_RFParams_t** struct. The following configuration parameters are supported:

- RF Technology: A, B, F, V or BPrime
- Tx and Rx bit-rates: 106 / 212 / 424 / 848 / 26.5 kBit/s
  - Type A and B work with every bit-rate except 26.5 kBit/s
  - Type F only works with 212 and 424 kBit/s
  - Type V only works with 26.5 kBit/s
  - Type BPrime only works with 106 kBit/s
- Flags (Parity, CRC, Rx SOF Only, Skip 1st Byte CRC): Enable / Disable Tx-/Rx-CRC handling, Enable / Disable Tx-/Rx-Parity bit handling (only for RF Technology A), Enable/Disable Rx SOF Only (RF Technology V), Enable/Disable Skip 1st Byte CRC (RF Technology V)
  - Tx-Parity: b0
  - Rx-Parity: b1
  - Tx-CRC: b2
  - Rx-CRC: b3
  - Rx-SOF-Only: b4
  - Skip1st-Byte-CRC: b5
- Rx Bit Offset: Number of bits for the offset of the first received byte (only for RF Technology A). Can be set from 0 to 7, where 0 means that all 8 bits should be sent.
- RES Limit: Maximum number of responses to receive.

This function can only be used in **RF ON** state.

Declaration	`ptx_Status_t` **`ptx_TransparentMode_SetRFParameters`** (**`ptx_TransparentMode_t`** `*tmComp,` **`ptx_TransparentMode_RFParams_t`** `*rfParams);`	
Description	Configures the HW using the provided RF-Parameters.	
Parameters	tmComp	Pointer to allocated Transparent Mode-component.
	rfParams	Pointer to RF-parameters.
Return Value	Status, indicating whether the operation was successful.	

7.6.2.7. ptx_TransparentMode_Exchange

This Exchange function performs the actual data exchange via RF. Same as **ptx_TransparentMode_SetRFParameters(),** accepts the function optional a RF parameter struct to reconfigure the HW before the data exchange. The required parameters for the exchange are provided using the **ptx_TransparentMode_ExchangeParams_t** struct. The struct consists of the following exchange parameters:

- ***TxData**: Buffer containing the data to send
- **TxLength**: Length of the TxData buffer
- **NrTxBits**: Number of bits to be sent for the last byte (only for RF Technology A)
- ***RxData**: Buffer where the data will be received
- **RxLength**: As input, capacity of RXData. As output, actual number of bytes written into RxData
- **NrResidualRxBits**: Number of (residual/valid) bits received in the last byte (only for RF Technology A)
- **TimeoutMS**: Timeout given in [ms]

*The function can only be used in **RF ON** State.

Declaration	`ptx_Status_t` **`ptx_TransparentMode_Exchange`** `(` **`ptx_TransparentMode_t`** `*tmComp,` **`ptx_TransparentMode_RFParams_t`** `*rfParams,` **`ptx_TransparentMode_ExchangeParams_t`** `*exchangeParams);`	
Description	Performs a RF data exchange.	
Parameters	tmComp	Pointer to allocated Transparent mode component.
	rfParams	Pointer to RF parameters.
	exchangeParams	Pointer to Exchange parameters
Return Value	Status, indicating whether the operation was successful.	

7.6.3. Transparent Mode API States

7.6.3.1. TransparentMode State Overview

Figure 31 describes the state diagram of Transparent mode.

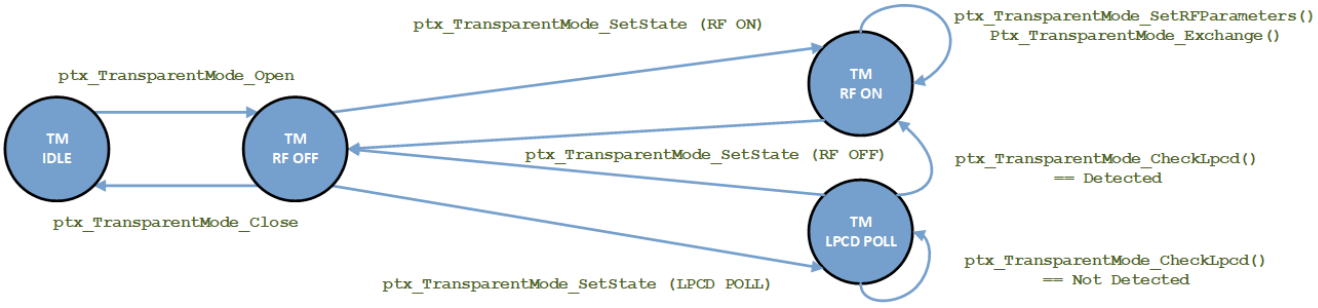

Figure 31. Transparent Mode API States

7.6.3.2. State Description

- **TM IDLE**: Represents the uninitialized state of Transparent Mode. It can be left or entered with **ptx_TransparentMode_Open** and **ptx_TransparentMode_Close.**
- **TM RF OFF**: This disables any transaction in Transparent Mode by turning the RF-Field off. It can be entered by initially calling **ptx_TransparentMode_Open** to initialize Transparent Mode, or by setting explicitly this state with **ptx_TransparentMode_SetState.**

- **TM RF ON**: Enables transactions in Transparent Mode by turning on the RF field. It can be entered either by explicitly switching to this state with **ptx_TransparentMode_SetState**, or if a card was detected and in **LPCD POLL** state and processed with **ptx_TransparentMode_CheckLpcd.**
- **LPCD Poll**: Enables LPCD (Low Power Card Detection). It can be entered by explicitly switching the state with **ptx_TransparentMode_SetState.** The function **ptx_TransparentMode_CheckLpcd** can be used to check for a detection and switches automatically to **RF ON** if a card was detected. Be aware that LPCD calibrates on the current field and then detects impedance changes and not directly cards. Therefore, it is important to ensure that there is no card in the field during calibration as well as performing card activation processing after a detection to check for an actual card.

7.6.4. Implementation Guidelines/Suggestions

- To implement a protocol which always uses the same RF parameters, a single call to "**ptx_TransparentMode_SetRFParameters"** is sufficient. The function "**ptx_TransparentMode_Exchange"** can be used to overwrite the parameters for a single call.
- For RF Technology F, the LEN-byte to a Tx or Rx frame needs to be prepended. For the Tx frame, this must be done manually. If multiple responses are received (for example, SENSF_REQ), it is up to the application to parse the received bytes.
- Performing a SENSF_REQ (RF Technology F) requires re-enabling the HW receiver n-times (in other words, within a given time). This can be achieved with the RES Limit parameter. To send a SENSF_REQ with a TSN value ! = 0, the RES Limit parameter must be set to 0. In this case, the HW receiver gets re-enabled until the timeout parameter ("**ptx_TransparentMode_Exchange"**) expires. For all other cases, RES Limit should be set to 1.
- For some transmitted frames, the field *NrTxBits* needs to be set. For example, WUP-A requires NrTxBits = 7. This is required since WUP-A is a Short Frame.

Examples:

- **Type-A – REQ-A / SENS_REQ**:
  - RF Technology = A
  - Tx/Rx Bit-rate = 106kBit/s
  - Flags = Tx/Rx Parity = Enabled, Tx/Rx CRC = Disabled
  - RES-Limit = 1
  - Timeout = 1ms
  - tx[0] = 0x26
  - txLength = 1
  - NrTxBits = 7
- **Type-A – SELECT / SEL_REQ**:
  - RF Technology = A
  - Tx/Rx Bit-rate = 106kBit/s
  - Flags = Tx/Rx Parity = Enabled, Tx/Rx CRC = Enabled
  - RES-Limit = 1
  - Timeout = 10ms
  - tx[…] = 0x93, 0x70, 4 x UID, 1 x BCC
  - txLength = 7

- **Type-B – REQ-B / SENSB_REQ**:
  - RF Technology = F
  - Tx/Rx Bit-rate = 106kBit/s
  - Flags = Tx/Rx Parity = d.c., Tx/Rx CRC = Enabled
  - RES-Limit = 1
  - Timeout = 10ms
  - tx[…] = 0x05, 0x00, 0x00
  - txLength = 3
- **Type-F – SENSF_REQ (Single Card or TSN = 0)**:
  - RF Technology = F
  - Tx/Rx Bit-rate = 212 or 424 kBit/s
  - Flags = Tx/Rx Parity = d.c., Tx/Rx CRC = Enabled
  - RES-Limit = 1
  - Timeout = 5ms
  - tx[…] = 0x06 0x00 0xFF 0xFF 0x00 0x0F
  - txLength = 6
- **Type-F – SENSF_REQ (Multiple Cards or TSN != 0)**:
  - RF-Technology = F
  - Tx/Rx Bit-rate = 212 or 424 kBit/s
  - Flags = Tx/Rx Parity = d.c., Tx/Rx CRC = Enabled
  - RES-Limit = 0
  - Timeout = 2.4ms + TSN * 1.2 (Note: Timeout is always given in integers)
  - tx[…] = 0x06 0x00 0xFF 0xFF 0x00 0x0F
  - txLength = 6
- **Type-V – Inventory Command**:
  - RF Technology = V
  - Tx/Rx Bit-rate = 26.5kBit/s
  - Flags = Tx/Rx Parity = d.c., Tx/Rx CRC = Enabled
  - RES-Limit = 1
  - Timeout = 10ms
  - tx[…] = 0x26 0x01 0x00
  - txLength = 3
- **Type-B Prime – APGEN**:
  - RF-Technology = BPrime
  - Tx/Rx Bit-rate = 106 kBit/s
  - Flags = Tx-/Rx Parity = d.c., Tx-/Rx-CRC = Enabled
  - RES-Limit = 1
  - Timeout = 10ms
  - tx[…] = 0x01, 0x0B, 0x3F, 0x80
  - txLength = 4

7.7 RF Test API

The Radio Frequency (RF) Test API allows to perform various general RF tests to optimize and verify the emitted RF-field.

All tests can be started through the same method "**ptx_RfTest_RunTest**", which receives a structure containing the parameters needed for the specific test to be executed.

Available are PRBS9/15 and Carrier. For each there is a corresponding struct for input parameters. Description of the parameters can be found in the specific sections below.

- **PRBS9 or PRBS15**

  To transmit a Pseudo Random Binary Sequence (PRBS) on the RF interface in accordance with [ITU.T O.150 1992].

  Input parameters:

  - Technology: RF technology according to enum **ptx_RfTest_TechType_t**
  - Technology: RF technology according to enum **ptx_RfTest_TechType_t**
  - Bit-rate: Supported RF-Bit-rates, see enum **ptx_RfTest_BitRate_t**
  - Enable/disable Start- and End-of-frame for **NFC-B**
  - Enable/disable Start- and Stop-Bits for **NFC-B**
  - Enable/disable parity bits for **NFC-A**

- **Carrier**

  To generate RF carrier only signal with a specific amplitude, which is parsed as an input parameter.

7.7.1. API Architecture

The RF Test API module is provided as stand-alone functionality of the RUL SDK. It is located on top of the RUL API module as shown in Figure 32 below and internally accesses also the NSC Core Stack module.

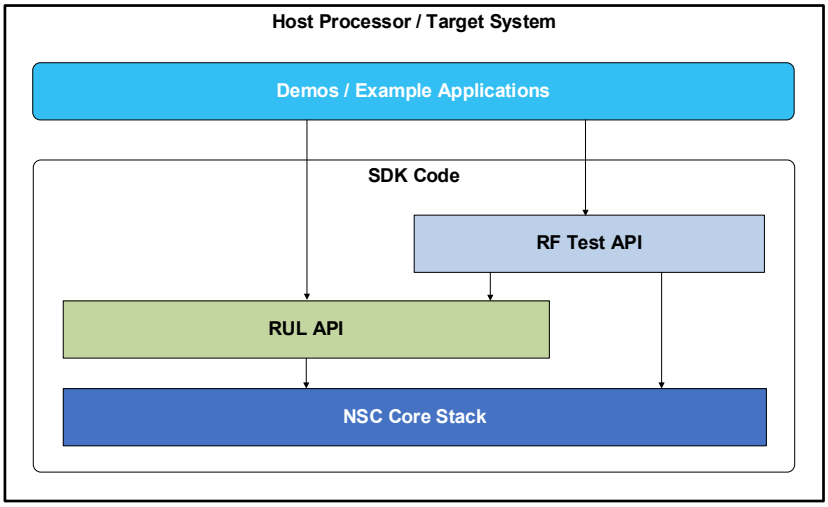

Figure 32. RF Test API Component Architecture

Note: The RF Test API module is optional and can be completely excluded from a target-build by following the instructions described in section 6.1.

7.7.2. API Description

7.7.2.1. ptx_RfTest_Init

Declaration	`ptx_Status_t` **`ptx_RfTest_Init`** (**`ptx_RfTest_t`** `*rfTestComp,` **`ptx_RfTest_InitParams_t`** `*initParams);`	
Description	Initializes the RF-Test component.	
Parameters	rfTestComp	Pointer to allocated RF-Test component
	initParams	Pointer to initialization parameter structure
Return Value	Status, indicating whether the operation was successful.	

7.7.2.2. ptx_RfTest_Deinit

Declaration	`ptx_Status_t` **`ptx_RfTest_Deinit`** (**`ptx_RfTest_t`** `*rfTestComp);`	
Description	Deinitializes the RF-Test component.	
Parameters	rfTestComp	Pointer to existing instance of RF-Test component
Return Value	Status, indicating whether the operation was successful.	

7.7.2.3. ptx_RfTest_RunTest

Declaration	`ptx_Status_t` **`ptx_RfTest_RunTest`** (**`ptx_RfTest_t`** `*rfTestComp,` **`ptx_RfTest_Params_t`** `*testParams);`	
Description	Performs a given Rf-Test using the provided test parameters.	
Parameters	rfTestComp	Pointer to existing instance of RF-Test component
	testParams	Various test parameters for RF-Tests
Return Value	Status, indicating whether the operation was successful.	

7.7.2.4. ptx_RfTest_StopTest

Declaration	`ptx_Status_t` **`ptx_RfTest_StopTest`** (**`ptx_RfTest_t`** `*rfTestComp);`	
Description	Stops an ongoing RF-Test	
Parameters	rfTestComp	Pointer to existing instance of RF-Test component
Return Value	Status, indicating whether the operation was successful.	

7.7.3. RF Test API States

7.7.3.1. RF Test API State Overview

Figure 33 shows the state diagram of the RF Test API.

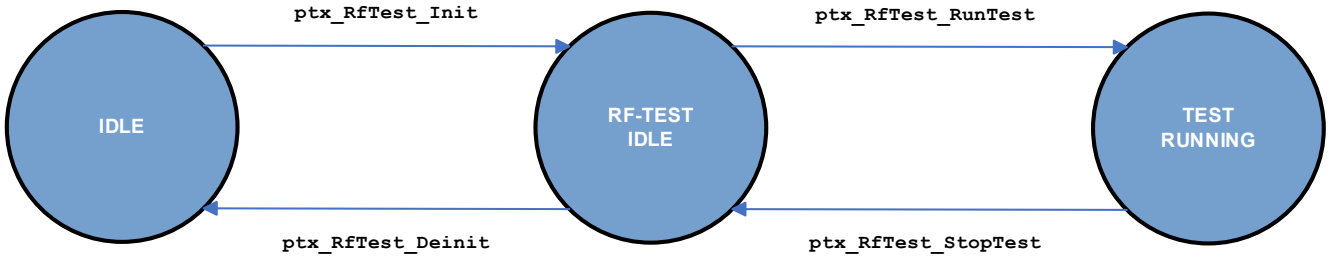

Figure 33. RF Test API states

7.7.3.2. State Description

- **IDLE**: The default state of the system after Stack and HW initialization. Calling "**ptx_RfTestInit**" moves the system to RF-TEST IDLE state.
- **RF-TEST IDLE**: Intermediate state where the RF Test API is initialized and ready to run tests with the "**ptx_RfTest_RunTest**" function.
- **TEST RUNNING**: When a test sequence is started, the system is in a RUNNING state to continuously generate the expected sequence. To get out of this state, call "**ptx_RfTest_StopTest**". The system moves back into RF-TEST IDLE state and another test sequence can be started.

7.7.4. Implementation Guidelines/Suggestions

PRBS sequences are used to measure bit error and allowing assessment of the performance of transmission equipment. The test patterns simulate real traffic as closely as possible while knowing the exact bit sequence. Details on the generation of the bit sequence can be found in the standard [ITU.T O.150 1992].

7.8 POS DTE API

The optional POS DTE API component implements the command-set as defined in [EMV DTE] and may be used for "Type Approval" testing of the EMV L1 stack of the provided RUL SDK.

The implemented command-set consists of the following low and high-level functions:

- Carrier (On/Off)
- Polling
- Reset
- Power-off
- WUPA
- WUPB
- RATS
- ATTRIB
- TRANSAC_A
- TRANSAC_B
- Loopback functionality
- Interoperability Loopback functionality

7.8.1. API Architecture

The POS DTE API module is provided as stand-alone functionality of the RUL SDK. It is located on top of the RUL API module (see Figure 34) and internally accesses the NSC Core Stack module.

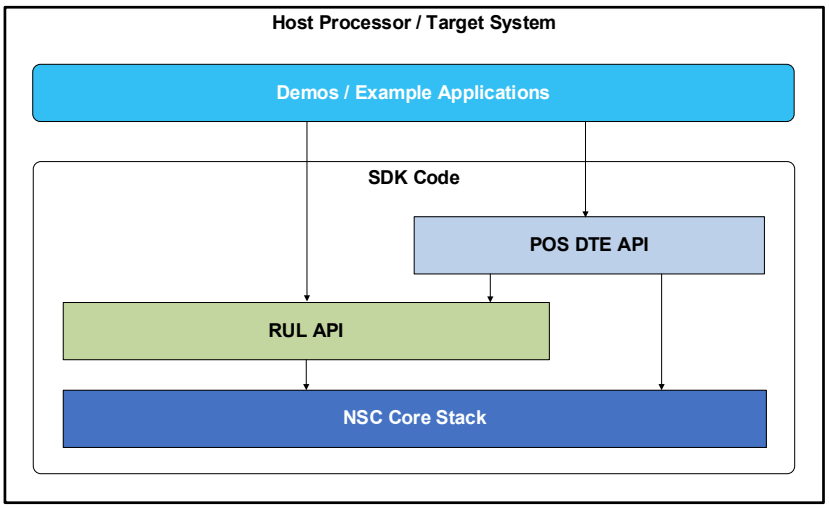

Figure 34. POS DTE API Component Architecture

Note: The POS DTE API module is optional and can be completely excluded from a target-build by following the instructions described in section 6.1.

7.8.2. API Description

7.8.2.1.1. ptx_POS_DTE_Init

This function initializes the POS DTE component. It requires an initialized instance of a RUL component and a LOG component. It allows the selection of the Loopback functionality mode and related callback-functions, as well as some generic parameters via "**ptx_POS_DTE_InitParams_t**" parameter structure.

Declaration	`ptx_Status_t` **`ptx_POS_DTE_Init`** `(` **`ptx_POS_DTE_t`** `*ctx,` **`ptx_POS_DTE_InitParams_t`** `*initParams);`	
Description	Initialization of the POS DTE API component.	
Parameters	ctx	Pointer to allocated POS DTE component structure.
	initParams	Pointer to initialization parameter structure.
Return Value	Status, indicating whether the operation was successful.	

7.8.2.1.2. ptx_POS_DTE_DeInit

This function de-initializes a previously initialized POS DTE component. Calling this API function after using any POS DTE API function (except **ptx_POS_DTE_Init**) is mandatory to reset the chips internal HW-configuration.

Declaration	`ptx_Status_t` **`ptx_POS_DTE_DeInit`** `(` **`ptx_POS_DTE_t`** `*ctx);`	
Description	De-initializes the POS DTE API component.	
Parameters	ctx	Pointer to an initialized POS DTE component structure.
Return Value	Status, indicating whether the operation was successful.	

7.8.2.1.3. ptx_POS_DTE_Carrier

This function turns the internal Carrier RF field on or off.

Declaration	`ptx_Status_t` **`ptx_POS_DTE_Carrier`** (**`ptx_POS_DTE_t`** *ctx, **`ptx_POS_DTE_CarrierState_t`** carrierReq);	
Description	Initialization of the POS DTE API component.	
Parameters	ctx	Pointer to an initialized POS DTE component structure.
	carrierReq	Turns internal Carrier / RF field on or off.
Return Value	Status, indicating whether the operation was successful.	

7.8.2.1.4. ptx_POS_DTE_Polling

This function starts the Polling procedure according to EMV. The application can choose to turn on RF technologies A, B, or both.

Declaration	`ptx_Status_t` **`ptx_POS_DTE_Polling`** (**`ptx_POS_DTE_t`** *ctx, **`ptx_POS_DTE_TargetRFTech_t`** rfTech);	
Description	Initialization of the POS DTE API component.	
Parameters	ctx	Pointer to an initialized POS DTE component structure.
	rfTech	Enables RF technology A, B, or both during the Polling.
Return Value	Status, indicating whether the operation was successful.	

7.8.2.1.5. ptx_POS_DTE_Reset

This function resets the internal Carrier RF field for a given time. The time can be configured during component initialization using initialization parameter "**ResetDelay**".

Declaration	`ptx_Status_t` **`ptx_POS_DTE_Reset`** (**`ptx_POS_DTE_t`** *ctx);	
Description	Initialization of the POS DTE API component.	
Parameters	ctx	Pointer to an initialized POS DTE component structure.
Return Value	Status, indicating whether the operation was successful.	

7.8.2.1.6. ptx_POS_DTE_PowerOff

This function switches off the internal Carrier RF field and is equivalent to calling "**ptx_POS_DTE_Carrier**" with **carrierReq** = Off.

Declaration	`ptx_Status_t` **`ptx_POS_DTE_PowerOff`** (**`ptx_POS_DTE_t`** *ctx);	
Description	Initialization of the POS DTE API component.	
Parameters	ctx	Pointer to an initialized POS DTE component structure.
Return Value	Status, indicating whether the operation was successful.	

7.8.2.1.7. ptx_POS_DTE_WupA

This function sends a **WUPA** command according to ISO 14443-3A (also referred to SENS_REQ in accordance with NFC Forum) via the NFC interface. If the optional output parameters "**rx**" and "**rxLen**" are provided, a card's response is stored in the following format:

2 byte(s) **ATQA** / **SENS_RES**

Declaration	`ptx_Status_t` **`ptx_POS_DTE_WupA`** `(` **`ptx_POS_DTE_t`** `*ctx,` **`uint8_t`** `*rx,` **`uint8_t`** `*rxLen);`	
Description	Sends a WUPA command via NFC interface.	
Parameters	ctx	Pointer to an initialized POS DTE component structure.
	rx	Optional pointer to store card response.
	rxLen	Optional pointer to store the length of the card response.
Return Value	Status, indicating whether the operation was successful.	

7.8.2.1.8. ptx_POS_DTE_RATS

This function performs a complete Type-A activation procedure up to and including the **RATS** command according to ISO 14443-3A / NFC Forum. If the optional output parameters "**rx**" and "**rxLen**" are provided, a card's response is stored in the following format:

2 Byte(s) **ATQA / SENS_RES**

1 Byte(s) **UID / NFCID1 Length m**

m Byte(s) **UID / NFCID1**

1 Byte(s) **SAK / SEL_RES**

1 Byte(s) **ATS / RATS_RES Length n**

n Byte(s) **ATS / RATS_RES**

Declaration	`ptx_Status_t` **`ptx_POS_DTE_RATS`** `(` **`ptx_POS_DTE_t`** `*ctx,` **`uint8_t`** `*rx,` **`uint8_t`** `*rxLen);`	
Description	Performs a full Type-A activation and sends a RATS command via NFC interface.	
Parameters	ctx	Pointer to an initialized POS DTE component structure.
	rx	Optional pointer to store card response.
	rxLen	Optional pointer to store the length of the card response.
Return Value	Status, indicating whether the operation was successful.	

7.8.2.1.9. ptx_POS_DTE_WupB

This function sends a **WUPB** command according to ISO 14443-3B (also referred to SENSB_REQ in accordance with NFC Forum) via the NFC interface. If the optional output parameters “**rx**” and “**rxLen**” are provided, a card’s response is stored in the following format:

12- or 13-byte(s) **ATQB** / **SENSB_RES (**depending on card**)**

Declaration	`ptx_Status_t` **`ptx_POS_DTE_WupB`** `(` **`ptx_POS_DTE_t`** `*ctx,` **`uint8_t`** `*rx,` **`uint8_t`** `*rxLen);`	
Description	Sends a WUPB command via NFC interface.	
Parameters	ctx	Pointer to an initialized POS DTE component structure.
	rx	Optional pointer to store card response.
	rxLen	Optional pointer to store the length of the card response.
Return Value	Status, indicating whether the operation was successful.	

7.8.2.1.10. ptx_POS_DTE_ATTRIB

This function performs a complete Type-B activation procedure up to and including the **ATTRIB**-command according to ISO 14443-3B / NFC Forum. If the optional output parameters “**rx**” and “**rxLen**” are provided, a card’s response is stored in the following format:

1 Byte(s) **ATQB/SENSB_RES Length m**

m Byte(s) **ATQB/SENSB_RES**

1 Byte(s) **ATTRIB_RES Length n**

n Byte(s) **ATTRIB_RES**

Declaration	`ptx_Status_t` **`ptx_POS_DTE_Attrib`** `(` **`ptx_POS_DTE_t`** `*ctx,` **`uint8_t`** `*rx,` **`uint8_t`** `*rxLen);`	
Description	Performs a full Type-B activation and sends a ATTRIB command via NFC interface.	
Parameters	ctx	Pointer to an initialized POS DTE component structure.
	rx	Optional pointer to store card response.
	rxLen	Optional pointer to store the length of the card response.
Return Value	Status, indicating whether the operation was successful.	

7.8.2.1.11. ptx_POS_DTE_TransacA

This function sends a pre-defined Type-A set of commands as a sequence as defined in [EMV DTE]. using a fixed timing interval between the commands. The timing interval is platform-dependent and might need to be adapted.

The function is based on a state-machine implementation and a non-blocking manner. It is supposed to be called in a loop at Application-level. The Application can choose whether the **TRANSAC_A** sequence application is executed endlessly or just for n-times.

Declaration	`ptx_Status_t` **`ptx_POS_DTE_TransacA`** `(` **`ptx_POS_DTE_t`** `*ctx);`	
Description	Performs the TRANSAC_A sequence according to [EMV DTE].	
Parameters	ctx	Pointer to an initialized POS DTE component structure
Return Value	Status, indicating whether the operation was successful.	

7.8.2.1.12. ptx_POS_DTE_GetTransacAState

This function returns the internal state of the **TRANSAC_A** sequence which is used at Application-level to display certain status information.

Declaration	`ptx_Status_t` **`ptx_POS_DTE_GetTransacAState`** `(` **`ptx_POS_DTE_t`** `*ctx,` **`ptx_POS_DTE_TransacA_State_t`** `*state);`	
Description	Returns the internal state of the TRANSAC_A sequence.	
Parameters	ctx	Pointer to an initialized POS DTE component structure.
	state	Ponter to store the internal TRANSAC state.
Return Value	Status, indicating whether the operation was successful.	

7.8.2.1.13. ptx_POS_DTE_TransacB

This function sends a pre-defined Type-B set of commands as a sequence, as defined in [EMV DTE], using a fixed timing interval between the commands. The timing interval is platform-dependent and might need to be adapted.

The function is based on a state-machine implementation and a non-blocking manner. It is supposed to be called in a loop at Application-level. The Application can choose whether the **TRANSAC_B** sequence application is executed endlessly or just for n-times.

Declaration	`ptx_Status_t` **`ptx_POS_DTE_TransacB`** `(` **`ptx_POS_DTE_t`** `*ctx);`	
Description	Performs the TRANSAC_B sequence according to [EMV DTE].	
Parameters	ctx	Pointer to an initialized POS DTE component structure.
Return Value	Status, indicating whether the operation was successful.	

7.8.2.1.14. ptx_POS_DTE_GetTransacBState

This function returns the internal state of the TRANSAC_B sequence which can is at Application-level to display certain status information.

Declaration	`ptx_Status_t` **`ptx_POS_DTE_GetTransacBState`** `(` **`ptx_POS_DTE_t`** `*ctx,` **`ptx_POS_DTE_TransacB_State_t`** `*state);`	
Description	Returns the internal state of the TRANSAC_B sequence.	
Parameters	ctx	Pointer to an initialized POS DTE component structure.
	state	Ponter to store the internal TRANSAC state.
Return Value	Status, indicating whether the operation was successful.	

7.8.2.1.15. ptx_POS_DTE_Transac_Stop

This function stops an ongoing transmission of a TRANSAC_A or TRANSAC_B sequence.

Declaration	`ptx_Status_t` **`ptx_POS_DTE_Transac_Stop`** `(` **`ptx_POS_DTE_t`** `*ctx);`	
Description	Performs the TRANSAC_A sequence according to [EMV DTE].	
Parameters	ctx	Pointer to an initialized POS DTE component structure.
	state	Ponter to store the internal TRANSAC state.
Return Value	Status, indicating whether the operation was successful.	

7.8.2.1.16. ptx_POS_DTE_Loopback

This function performs the Loopback or Interoperability Loopback application, as defined in [EMV DTE], using the given RF technology A, B, or both. The Loopback mode can be selected via component initialization parameter "**lopLoopback**". If the Interoperability Loopback mode is selected, the application can additionally initialize various callback-functions to implement custom behavior on certain events.

The function is based on a state-machine implementation and a non-blocking manner. It is supposed to be called in a loop at Application-level. The Application can choose whether the Loopback application is executed endlessly or just for n-times.

Declaration	`ptx_Status_t` **`ptx_POS_DTE_Loopback`** `(` **`ptx_POS_DTE_t`** `*ctx,` **`ptx_POS_DTE_TargetRFTech_t`** `rfTech);`	
Description	Performs the TRANSAC_B sequence according to [EMV DTE].	
Parameters	ctx	Pointer to an initialized POS DTE component structure.
Return Value	Status, indicating whether the operation was successful.	

7.8.2.1.17. ptx_POS_DTE_GetLoopbackState

This function returns the internal state of the (Interoperability) Loopback application which can be used at Application-level to display certain status information.

Declaration	`ptx_Status_t` **`ptx_POS_DTE_GetLoopbackState`** `(` **`ptx_POS_DTE_t`** `*ctx,` **`ptx_POS_DTE_LoopbackState_t`** `*state);`	
Description	Returns the internal state of the (Interoperability) Loopback application.	
Parameters	ctx	Pointer to an initialized POS DTE component structure.
	state	Ponter to store the internal Loopback-state.
Return Value	Status, indicating whether the operation was successful.	

7.8.2.1.18. ptx_POS_DTE_LoopbackStop

This function stops an ongoing (Interoperability) Loopback application.

Declaration	`ptx_Status_t` **`ptx_POS_DTE_LoopbackStop`** `(` **`ptx_POS_DTE_t`** `*ctx);`	
Description	Stops an ongoing (Interoperability) Loopback application.	
Parameters	ctx	Pointer to an initialized POS DTE component structure.
Return Value	Status, indicating whether the operation was successful.	

7.8.2.1.19. ptx_POS_DTE_ResetState

This function resets the internal component state without the need to restart the entire API. Settings that were set in the initializer function are kept.

Declaration	`ptx_Status_t` **`ptx_POS_DTE_ResetState`** `(` **`ptx_POS_DTE_t`** `*ctx);`	
Description	Resets the internal component state.	
Parameters	ctx	Pointer to an initialized POS DTE component structure.
Return Value	Status, indicating whether the operation was successful.	

7.8.3. POS DTE API States

7.8.3.1. POS DTE State Overview

Figure 35 shows the operating states of the POS DTE API and which API functions can be called in each state.

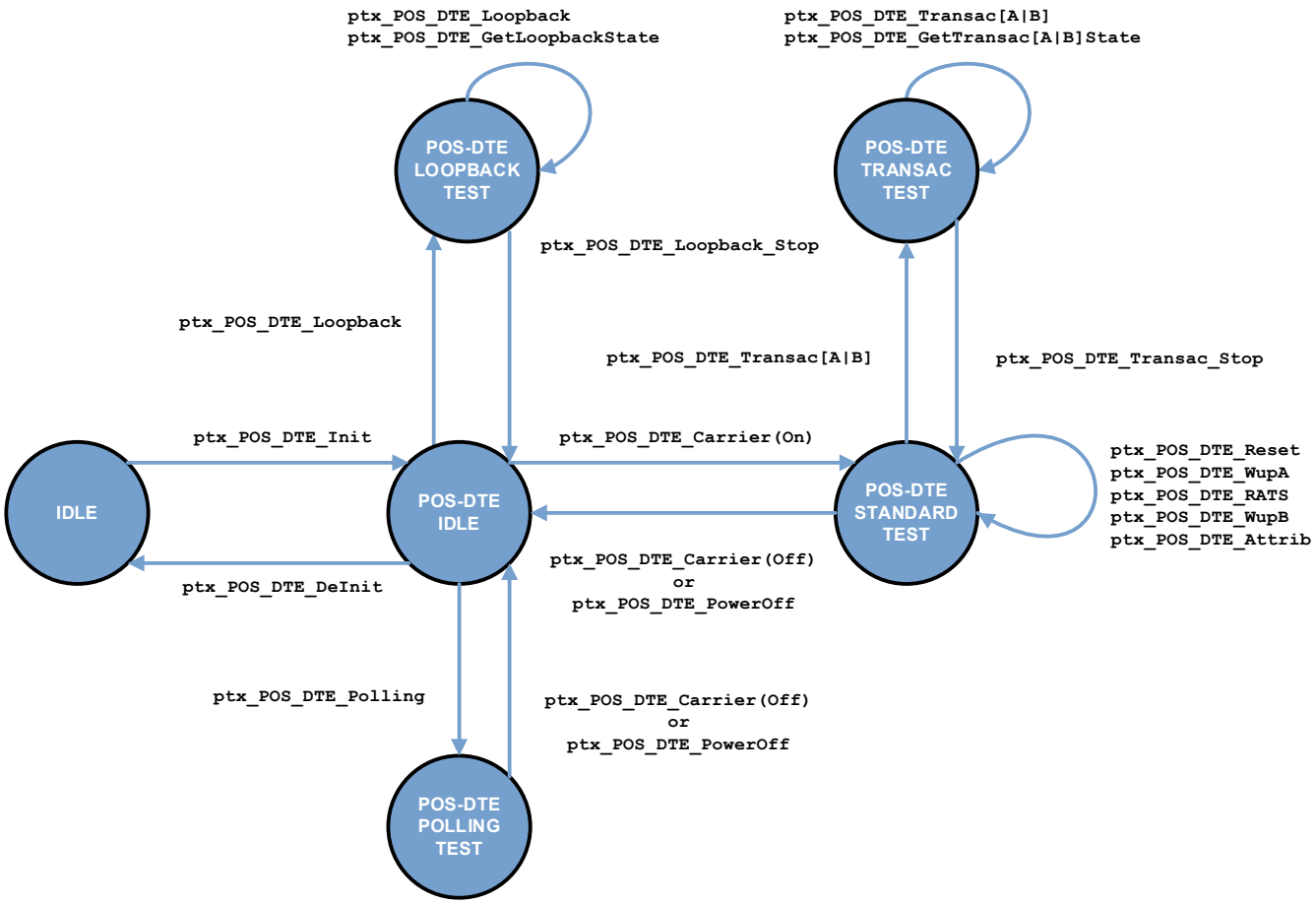

Figure 35. POS DTE API States

7.8.3.2. State Description

- **IDLE**

  The default state of the system after Stack and HW initialization. The system switches to state "**POS-DTE IDLE**" by calling "**ptx_POS_DTE_Init**".

- **POS-DTE IDLE**

  This state is used to switch into a dedicated internal POS DTE state depending on which API function is called. Those internal states can be "**POS-DTE STANDARD / TRANSAC / POLLING / LOOPBACK** or **TEST**". To leave this state and to reset the internal HW configuration, it is required to call "**ptx_POS_DTE_DeInit**" after usage of the POS DTE API functions.

- **POS-DTE STANDARD TEST**

  This state executes the low-level NFC-commands according to [EMV DTE] consisting of:

  - **ptx_POS_DTE_WupA**
  - **ptx_POS_DTE_RATS** (including full prior Type-A activation)
  - **ptx_POS_DTE_WupB**
  - **ptx_POS_DTE_ATTRIB** (including full prior Type-B activation)

To execute any of the low-level NFC commands, the internal carrier/RF field needs to be turned on via "**ptx_POS_DTE_Carrier (On)**", otherwise **PTX_STATUS_E_GEN_INVALID_STATE** gets returned.

Having the carrier/RF field turned on is also a pre-condition to switch into the "**POS-DTE TRANSAC TEST**" state. If needed, the internal carrier / RF field can be reset via "**ptx_POS_DTE_Reset**". To move from this state back to "**POS-DTE IDLE**", either "**ptx_POS_DTE_Carrier (Off)**" or "**ptx_POS_DTE_PowerOff**" must be called.

Note: The application needs to follow the rules of the corresponding contactless standard, otherwise the API functions may not return the desired result. For example, calling "**ptx_POS_DTE_WupA**" sequentially will lead to an error during the second call because the card will switch its internal state. As the card does not expect a 2nd consecutive WUPA request, it resets its internal state and replies again to the next WUPA request. In other words, only every 2nd WUPA request will be successful. This behavior is according to the ISO 14443 and/or NFC Forum standards.

- **POS-DTE TRANSAC TEST**

  This state can only be reached from "**POS-DTE STANDARD TEST**" state and is used to execute either the TRANSAC_A or TRANSAC_B sequence according to [EMV DTE].

  The functions "**ptx_POS_DTE_TransacA**" and "**ptx_POS_DTE_TransacB**" are implemented in a non-blocking manner based on an internal state-machine. This allows an application to call the function in a loop, where every call performs the action defined in the current state before returning. An application can retrieve the current internal state for custom processing by calling the corresponding "**ptx_POS_DTE_GetTransac[A | B]State**" function. To switch back to state "**POS-DTE STANDARD TEST**", the function "**ptx_POS_DTE_Transac_Stop**" must be called.

- **POS-DTE POLLING TEST**

  This state is used to activate the EMV Polling procedure using the RF technologies A, B, or both. To switch back to state "**POS-DTE IDLE**", either "**ptx_POS_DTE_Carrier (Off)**" or "**ptx_POS_DTE_PowerOff**" must be called.

- **POS-DTE LOOPBACK TEST**

  This state is used to execute either the standard or Interoperability Loopback application according to [EMV DTE]. The function "**ptx_POS_DTE_Loopback**" is implemented in a non-blocking manner based on an internal state-machine. This allows an application to call the function in a loop, where every call performs the action defined in the current state before returning. An application can retrieve the current internal state for custom processing by calling the corresponding "**ptx_POS_DTE_GetLoopbackState**" function. To switch back to state "**POS-DTE IDLE**", the function "**ptx_POS_DTE_Loopback_Stop**" must be called.

7.8.4. Implementation Guidelines/Suggestions

- The standard Loopback and the Interoperability Loopback functionality behave identically. The only exception is that that the Interoperability Loopback function supports dedicated user/application notifications through optional callback functions. These callback functions may be needed during "Type Approval" testing to implement either acoustic (for example, buzzer) or visual (for example, LEDs) feedback in case of certain events such as successful/erroneous Loopback transactions or sent/received APDUs.

  The mode can be selected via component initialization parameter "**IopLoopback**".

  The callback functions can be initialized via component initialization parameters "**CbSuccess**", "**CbError**" and "**CbApdu**".

- The component initialization allows to set parameter "**ResetDelay**".

  This parameter is used by API function "**ptx_POS_DTE_Reset**" and implements a delay given in [ms] between the RF field gets turned off and turned on again. Based on the host platform and application on top, the value of "**ResetDelay**" might need to be adapted to match the requirements of the "Type Approval" tests which is usually between 5.1ms and 10ms.

- The component initialization allows to set parameter “**IgnoreRFError**”.

  This parameter is used by the standard and Interoperability Loopback functions to suppress potential RF errors (for example, CRC, Parity, Timeout, …) which may occur for example, if the (test)system is used close to the operating volume. In case “**IgnoreRFError**” is set, both Loopback functions would always return “**PTX_STATUS_SUCCESS**” in case of an RF error.

- The TRANSAC_A and TRANSAC_B sequences are relying on very accurate timings between the actual commands sent via the NFC interface. The typical timing values are 2ms between the commands and 100ms after a completed sequence. The accuracy of the timings is typically platform-dependent and may require adaptations if certain granularity can’t be achieved or latencies are introduced.

  To overcome possible latencies and inaccuracies, the following preprocessor constants in **ptx_POS_DTE.h** can be adapted: **PTX_POS_DTE_MIN_TIME_MS_TRANSAC_END** and **PTX_POS_DTE_MIN_TIME_MS_TRANSAC_CMD**.

7.9 NFC Forum DTA API

The optional NFC Forum DTA API component implements the “Device Test Application”-specification defined by the NFC Forum and can be used for NFC Forum Compliance testing [NFC DTA].

The NFC Forum DTA works with so called Pattern numbers which specifies a specific scenario and the expected behavior of the NFC device.

For more information on Pattern numbers and the specific test-scenarios, refer to the “Device Test Application Specification” document [NFC DTA].

Note: The NFC Forum DTA API only implements the expected behavior based on the Pattern number. It does not implement the actual test-cases defined by the NFC Forum. The test case implementations depend on the tools/environment of the test vendor and is therefore out of scope of this NFC Forum DTA implementation.

Note: The implementation of the NFC Forum DTA is blocking. An optional callback function is supported and can be set after module initialization. This callback can be used to cancel/stop an ongoing execution of the Pattern number based behavior.

7.9.1. API Architecture

The NFC Forum DTA API module is built on top of the RUL API, the RUL HCE API, the LLCP API, the SNEP API, the NDEF APIs, and also, the Native Tag APIs. The internal dependencies are shown in Figure 36.

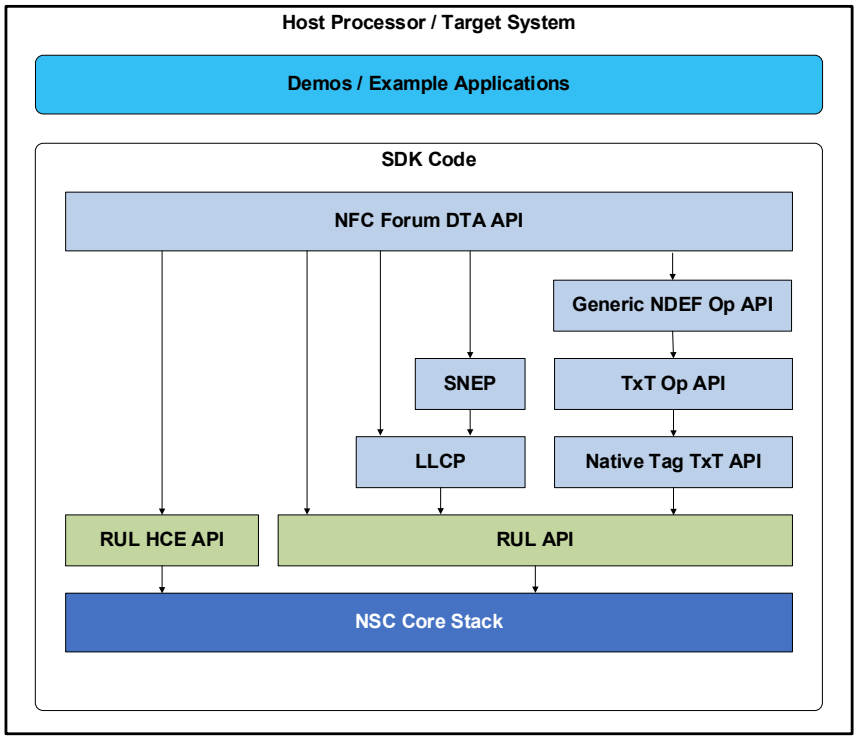

Figure 36. NFC Forum DTA API Component Architecture

7.9.2. API Description

7.9.2.1.1. ptx_NFCForumDTA_Init

This function initializes the NFC Forum DTA component and sets up all internal pointers. The Init function expects the user to provide a previously initialized RUL component as well as two buffers which will be used for the data transfers required for the test cases, one for the reader and another for listen mode. The size of those buffers can be chosen by the user, however a smaller buffer might cause test case failures due to the lack of space for storing the required data. Those buffers must stay defined until the NFC Forum DTA API is no longer used.

Declaration	ptx_Status_t **ptx_NFCForumDTA_Init** (**ptx_NFCForumDTA_t** *dta, **ptx_NFCForumDTA_InitParams_t** *initParams);	
Description	NFC Forum DTA component initialization.	
Parameters	dta	Pointer to NFC Forum DTA component.
	initParams	Pointer to initialization parameter structure.
Return Value	Status, indicating whether the operation was successful.	

7.9.2.1.2. ptx_NFCForumDTA_DeInit

This function de-initializes a previously initialized NFC Forum DTA component.

Declaration	ptx_Status_t **ptx_NFCForumDTA_DeInit** (**ptx_NFCForumDTA_t** *dta);	
Description	NFC Forum DTA component de-initialization.	
Parameters	dta	Pointer to NFC Forum DTA component.
Return Value	Status, indicating whether the operation was successful.	

7.9.2.1.3. ptx_NFCForumDTA_StartDTARequest

This function starts the execution of an NFC Forum DTA execution with the provided test configuration. The test configuration contains information about how the DTA should behave. This test configuration is used to pass the pattern number, the discovery timeout, the device class to be tested for, as well as the usage of DDPC.

The pattern number specifies which scenario of the NFC Forum DTA to execute and is defined by the test case. The device class decides the capabilities to be tested. If the universal device class is selected, host card emulation and active communication mode are activated if the NFC Forum DTA scenario of the pattern number indicates it. If the reader mode is selected, the host card emulation mode and the active communication mode will be disabled.

Declaration	ptx_Status_t **ptx_NFCForumDTA_StartDTARequest** (**ptx_NFCForumDTA_t** *dta, **ptx_NFCForumDTA_Test_Config_t** *testCfg);	
Description	Starts an NFC Forum DTA execution using the current configuration	
Parameters	dta	Pointer to NFC Forum DTA component.
	testCfg	Pointer to test parameter structure.
Return Value	Status, indicating whether the operation was successful.	

7.9.2.1.4. ptx_NFCForumDTA_RepeatTest

This function starts the execution of an NFC Forum DTA execution with the provided test configuration a specified number of times or continuously. The endless execution can be exited by setting **quitEndlessExecution** to 1 using the application callback.

Declaration	`ptx_Status_t` **`ptx_NFCForumDTA_RepeatTest`** (**`ptx_NFCForumDTA_t`** *dta, **`ptx_NFCForumDTA_Test_Config_t`** *testCfg, **`uint32_t`** repetitions, **`uint8_t`** endless);	
Description	Repeats NFC Forum DTA request for repetitions.	
Parameters	dta	Pointer to NFC Forum DTA component.
	testCfg	Pointer to test parameter structure.
	repetitions	Number of times the test should be repeated.
	endless	Enables endless execution of the DTA request.
Return Value	Status, indicating whether the operation was successful.	

7.9.2.1.5. ptx_NFCForumDTA_RegisterApplicationCallback

This function Registers the additional checks callback. This callback can be used to pre-emptively stop the execution of an NFC Forum DTA request by setting the internal DTA parameter **quitDTA** to 1.

Declaration	`ptx_Status_t` **`ptx_NFCForumDTA_RegisterApplicationCallback`** (**`ptx_NFCForumDTA_t`** *dta, **`ptx_NFCForumDTA_ApplicationCallback_t`** cb);	
Description	Registers the additional checks callback	
Parameters	dta	Pointer to NFC Forum DTA component.
	cb	Function pointer to callback to register.
Return Value	Status, indicating whether the operation was successful.	

7.9.2.1.6. ptx_NFCForumDTA_GetVersion

This function returns the version number of the currently implemented NFC Forum DTA version.

Declaration	`ptx_Status_t` **`ptx_NFCForumDTA_GetVersion`** (**`uint8_t`** *version);	
Description	Version of the NFC Forum DTA implementation.	
Parameters	version	Version of the NFC Forum DTA implementation.
Return Value	Status, indicating whether the operation was successful.	

7.9.3. NFC Forum API States

7.9.3.1. NFC Forum DTA API State Overview

Figure 37 describes the state diagram of the NFC Forum DTA API.

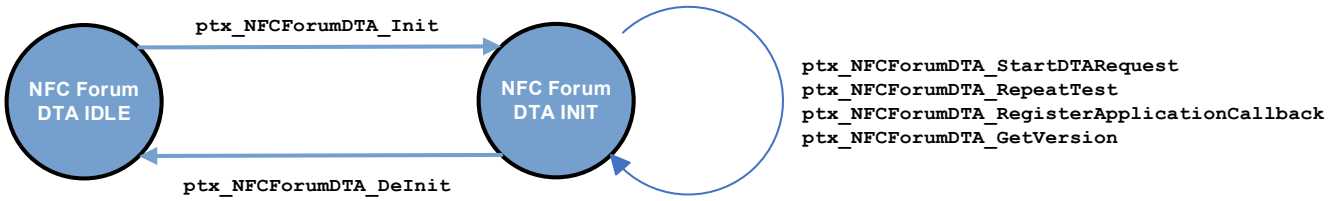

Figure 37. NFC Forum DTA API States

7.9.3.2. State Description

The NFC Forum DTA API only consists of two states:

- **NFC Forum DTA IDLE**: Idle state of the NFC Forum DTA API, the API must be initialized using the "**ptx_NFCForumDTA_Init**" function before any of the other API functions can be used.
- **NFC Forum DTA INIT**: The initialized state of the NFC Forum DTA API. All of the remaining API functions such as "**ptx_NFCForumDTA_StartDTARequest**" or "**ptx_NFCForumDTA_RepeatTest**" can be called from this state. In order to de-initialize the NFC Forum DTA API again and to return to the IDLE state, the "**ptx_NFCForumDTA_DeInit**" function must be called.

7.9.4. Implementation Guidelines/Suggestions

- A reference usage example of the NFC Forum DTA API can be found under the "**ptx_Example_NFCForumDta**" folder in the reference examples.
- The NFC Forum DTA API is designed to be used to pass the compliance tests defined by the NFC Forum.
  - To execute these compliance tests, additional test hardware is required to act as the counterpart for the different test cases. These counterparts will provide the specific Pattern Number required to pass the specific testcases.
  - This Pattern Number should then be provided to the NFC Forum DTA API as the "**PATTERN_NUMBER**" parameter of the test configuration for either the "**ptx_NFCForumDTA_StartDTARequest**" or "**ptx_NFCForumDTA_RepeatTest**" functions.
  - With the provided pattern number and device type, the NFC Forum DTA API executes the test procedure according to [NFC DTA].
  - Unlike other disciplines, the SNEP DTA does not define standardized Pattern Numbers due to NFC Forum's assumption of a GUI-based environment for triggering and displaying Text RTD content. However, since embedded systems often lack GUI capabilities, we have defined custom pattern numbers to facilitate efficient testing.

    The following table outlines the supported SNEP Pattern Numbers:

Pattern Number	Role	Operation Description
0x1301	SNEP Client	PUT operation using a normal NDEF record.
0x1302	SNEP Client	PUT operation using a long NDEF record.
0x1303	SNEP Client	GET operation.
0x1300	SNEP Server	Operates in normal server mode.
0x1340	SNEP Server	DTA mode enabled, supports GET request.

In all cases, the bitmask **0x0010** can be used to enable high bitrate testing (Type F - 424 kBit/s), e.g. 0x1350 will cause the device to operate as SNEP Server in DTA mode with 424 kBit/s enabled.

- As there are various certification releases which have differences in the Test sequences, the NFC DTA is currently only designed to be working with a single Certification Release. The currently supported Certification Release is CR13. The currently implemented DTA Version can be verified by using the "**ptx_NFCForumDTA_GetVersion**" function.
- If the user wishes to prematurely stop the execution of a test case, or wishes to stop the execution of multiple test executions started with the repeatTest function, the application callback should be used.
  - The application callback is called in every instance where the DTA could potentially stay for a longer period such as a loopback application.
  - If the user wishes to terminate the current test execution using the application callback, set the "**quitDTA**" flag of the NFC Forum DTA API component to any non-zero value.
  - If the user decides to prematurely stop the execution of a test case, the state is not defined and therefore it is suggested that the user should call the RUL deactivate function to return to a known state.

7.10 FeliCa DTE API

The optional FeliCa DTE API component allows for the execution of FeliCa-related tests to pass the FeliCa M-class certification [FELICA PERF] and/or to run the test-sequences defined in the Reader/Writer Digital Protocol Requirements specification [FELICA PROT].

7.10.1. API Architecture

The FeliCa DTE API module is built on top of the RUL API and accesses functionality from the NSC Core Stack component. The internal dependency is shown in Figure 38.

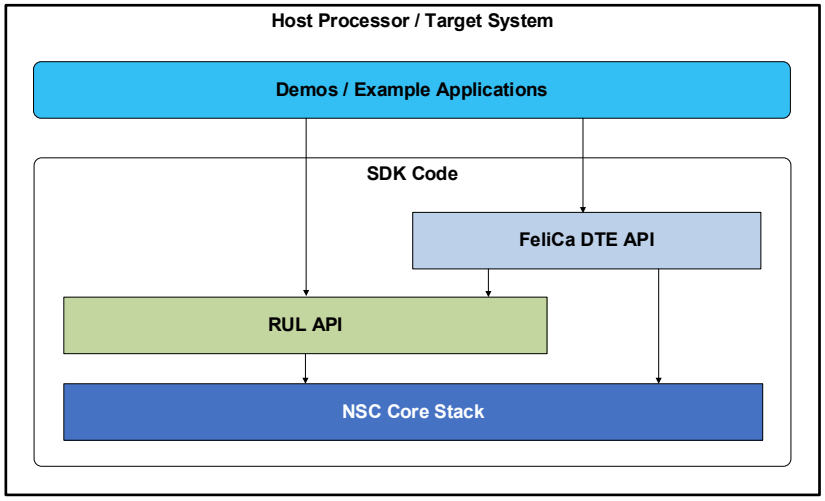

Figure 38. FeliCa DTE API Component Architecture

7.10.2. API Description

7.10.2.1. ptx_FeliCaDTE_Open

The **Open** function is used to initialize the FeliCa-DTE component.

The FeliCa-DTE requires an initialized RUL component as the initialization parameter. Additionally, the EnableDDPC flag can be provided. When enabled, an RF-field reset and a DDPC cycle is performed after each performed test.

Declaration	`ptx_Status_t` **`ptx_FeliCaDTE_Open`** `(` **`ptx_FeliCaDTE_t`** `*feliCaDTE,` **`ptx_FeliCaDTE_InitParams_t`** `*initParams);`	
Description	Initializes the FeliCa-DTE component.	
Parameters	feliCaDTE	Pointer to an instance of the FeliCa-DTE component.
	initParams	Pointer to initialization parameters.
Return Value	Status, indicating whether the operation was successful.	

7.10.2.2. ptx_FeliCaDTE_Close

The **Close** function is used to de-initialize the FeliCa-DTE component.

Declaration	`ptx_Status_t` **`ptx_FeliCaDTE_Close`** `(` **`ptx_FeliCaDTE_t`** `*feliCaDTE);`	
Description	Deinitializes the FeliCa-DTE component.	
Parameters	feliCaDTE	Pointer to an instance of the FeliCa-DTE component.
Return Value	Status, indicating whether the operation was successful.	

7.10.2.3. ptx_FeliCaDTE_EnableMode

The **Enable** function is used to enable and disable the FeliCa mode and to configure the HW for the specific test parameters provided.

The enable parameter defines if the FeliCa is turned on or off. The mode is turned off if the enable value equals 0 and turned on otherwise.

The provided test parameters provide information as to how the HW should be configured. It contains the “ID” parameter which specifies if the test to be executed is a FeliCa performance or protocol test case. The “Params” parameter of the test parameters contains the specific test settings for the execution of the performance or protocol tests. The test parameters can also be given a progress callback. This callback can be used to track the current number of executed testcases or to prematurely exit the test execution based on an external event.

Both the performance and protocol tests require a results buffer, the length of this buffer and a timeout value to be used for any single test run in milliseconds. The size of the result buffer must be greater or equal to the number of tests to be performed for the performance tests and greater than 256 bytes for protocol tests. The timeout must be set to a value greater than 0.

The performance tests require the number of tests to be executed and the desired bit-rate, in addition to the results buffer, the length of the results buffer and the timeout.

The protocol tests on the other hand require the sub-test identifier (The Reader / Writer Digital Protocol sequence ID) and the Result buffer, its size, and the timeout. The “**T_OffGuardTime**” parameter specifies the guard time between Field-Off and Field-On. The default value is set to 30ms, but this value can be modified by the user. The “**T_OnGuardTime**” parameter specifies the guard time between Field-On and the first command. The default value for this guard time is 25ms, but this value can be modified by the user as well.

If the chosen sub-test identifier is selected as a generic test, the buffer containing the generic application command and its respective length must also be provided using the protocol test parameters.

Declaration	`ptx_Status_t` **`ptx_FeliCaDTE_EnableMode`** (**`ptx_FeliCaDTE_t`** `*feliCaDTE,` **`uint8_t`** `enable,` **`ptx_FeliCaDTE_TestParams_t`** `*testParams);`	
Description	Enables/Disables the FeliCa-DTE mode in the system and configures the test parameters.	
Parameters	feliCaDTE	Pointer to an existing instance of the FeliCa-DTE component.
	enable	Enables (!= 0) or Disables (0) the FeliCa-DTE mode.
	testParams	Various test parameters for FeliCa compliance tests.
Return Value	Status, indicating whether the operation was successful.	

7.10.2.4. ptx_FeliCaDTE_RunTest

This function is used to execute the FeliCa performance or protocol test with the provided parameters.

Declaration	`ptx_Status_t` **`ptx_FeliCaDTE_RunTest`** (**`ptx_FeliCaDTE_t`** `*feliCaDTE,` **`ptx_FeliCaDTE_TestParams_t`** `*testParams);`	
Description	Performs a FeliCa compliance test with given test parameters.	
Parameters	feliCaDTE	Pointer to an existing instance of the FeliCa-DTE component.
	testParams	Various test parameters for FeliCa compliance tests.
Return Value	Status, indicating whether the operation was successful.	

7.10.3. FeliCa DTE API States

7.10.3.1. State Overview

Figure 39 shows which FeliCa DTE API commands can be used in which state.

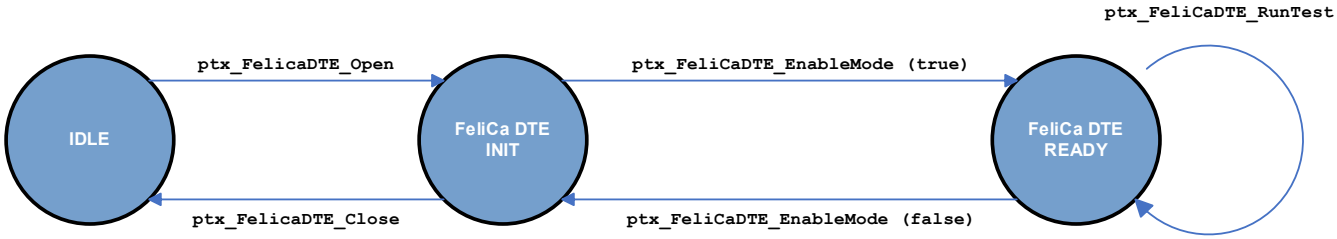

Figure 39. FeliCa DTE API States

7.10.3.2. State Description

7.10.3.2.1. IDLE State

The IDLE state is the default state of the system after Stack and HW initialization. By calling "**ptx_FeliCaDTE_Open**", the system switches to state "**FeliCa DTE INIT**".

7.10.3.2.2. FeliCa DTE INIT State

This state is an intermediary state. It has not made any hardware configuration changes. Thus, the FeliCa DTE can be left in this state while the application performs other tasks in the meantime.

To get into a state where FeliCa performance and protocol tests can be executed, the FeliCa DTE must be brought to the "**FeliCa DTE READY**" state by calling "**ptx_FeliCaDTE_EnableMode**" with the parameter enable set to "true".

To leave this state and return to the **IDLE** state, call "**ptx_FeliCaDTE_Close".**

7.10.3.2.3. FeliCa DTE READY State

This state is reached by enabling the FeliCa mode with "**ptx_FeliCaDTE_EnableMode".** The user can execute both FeliCa performance and protocol tests in this state by calling "**ptx_FeliCaDTE_RunTest".**

To exit this mode and revert the hardware settings made previously by enabling the FeliCa mode, call the "**ptx_FeliCaDTE_EnableMode**" function with the enable parameter set to "false". This transfers the device back into the "**FeliCa DTE INIT**" state.

7.10.4. Implementation Guidelines/Suggestions

As described above, a progress callback can be parsed to the test parameters. To use this functionality, the user needs to define a method according to the defined callback header as shown in Listing 1.

Listing 1. FeliCaDTE Progress Callback Header Definition

```
typedef void (*ptx_FeliCaDTE_ProgressFn_t)(ptx_FeliCaDTE_TestProgressParams_t
*progressParams);
```

For example, to allow an application to cancel running tests and get the number of executed test cases, the callback could be implemented as shown in Listing 2.

Listing 2. FeliCaDTE Progress Callback Example Implementation

```
static void FeliCaDTE_ProgressCb(ptx_FeliCaDTE_TestProgressParams_t* params)
{
    if (NULL != params)
    {
        params->ExitProcessing = gProgress.ExitProcessing;

        gProgress.NrTestsProcessed = params->NrTestsProcessed;
    }
}
```

Where "**gProgress**" represents a global variable of type "**ptx_FeliCaDTE_TestProgressParams_t**" on the application layer. When this function handle is parsed with the test parameters, the API then updates the "**gProgress.NrTestsProcessed**" member in the global variable, for the user being able to track the progress. Through setting the "**gProgress.ExitProcessing**" member the user can exit the test execution prematurely if required.

7.11 HCE API

The Host Card Emulation (HCE) API implements a type agnostic host card emulation framework and also allows a further expansion for tags if required (for example, Type 3 Tag, Type 4 Tag).

The optional Host Card Emulation API component implements basis for host card emulation with the PTX2K.

The Host Card Emulation API uses an internal message queue to communicate the events to the user. The user must take care to implement the actions in regard to the required tag type that is emulated.

7.11.1. API Architecture

The HCE API module is built on top of the RUL API. The internal dependencies are shown in Figure 40.

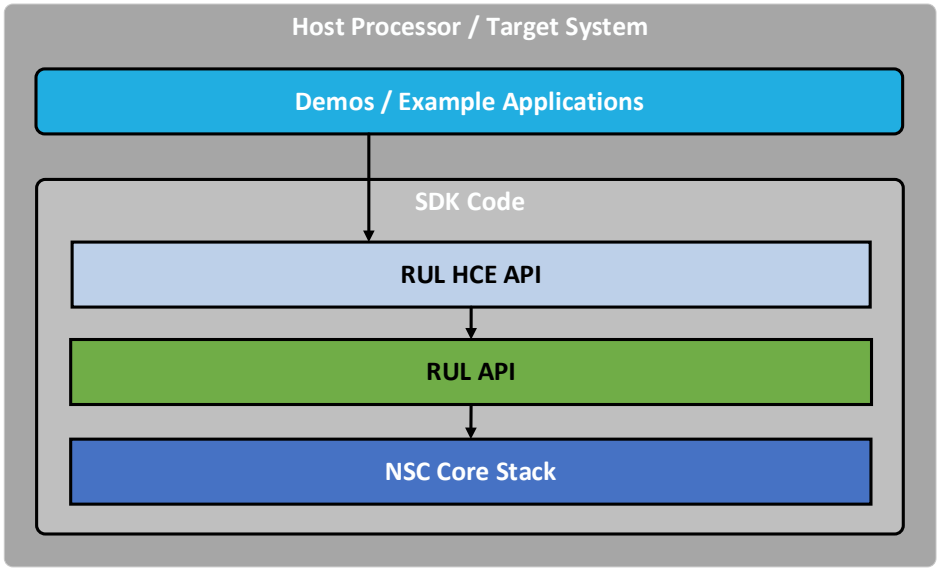

Figure 40. HCE API Component Architecture

7.11.2. API Description

7.11.2.1. ptx_RUL_HCE_Init

Declaration	`ptx_Status_t` **`ptx_RUL_HCE_Init`** `(`
	`ptx_RUL_HCE_t` `*ctx,`
	`ptx_RUL_HCE_InitParameters_t` `*initParameters`
	`);`	

Description	This function initializes the HCE Addon API, sets up all internal data structures and registers the required callbacks to RUL.	
Parameters	ctx	Pointer to the allocated HCE component.
	initParameters	Pointer to the initialization parameters.
Return Value	Status, indicating whether the operation was successful.	

7.11.2.2. ptx_RUL_HCE_DeInit

Declaration	`ptx_Status_t` **`ptx_RUL_HCE_DeInit`** `(`
	`ptx_RUL_HCE_t` `*ctx`
	`);`	

Description	RUL HCE component deinitialization.	
Parameters	ctx	Pointer to the allocated HCE component.
Return Value	Status, indicating whether the operation was successful.	

7.11.2.3. ptx_RUL_HCE_ResetState

Declaration	`ptx_Status_t ptx_RUL_HCE_ResetState ( ptx_RUL_HCE_t *ctx );`	
Description	Internal reset of the components state.	
Parameters	ctx	Pointer to the allocated HCE component.
Return Value	Status, indicating whether the operation was successful.	

7.11.2.4. ptx_RUL_HCE_GetEvent

Declaration	`ptx_Status_t ptx_RUL_HCE_GetEvent ( ptx_RUL_HCE_t *ctx, ptx_RUL_HCE_EventRecord_t **event );`	
Description	This function allows the user to request the latest event notification data received from the PTX card emulation device.	
Parameters	ctx	Pointer to the allocated HCE component.
	event	Pointer to the event record supplied by the AP into with event details can be entered.
Return Value	Status, indicating whether the operation was successful.	

7.11.2.5. ptx_RUL_HCE_SendData

Declaration	`ptx_Status_t ptx_RUL_HCE_SendData ( ptx_RUL_HCE_t *ctx, uint8_t *tx, uint32_t txLen );`	
Description	This function allows the user to send data to a nearby card reader.	
Parameters	ctx	Pointer to the allocated HCE component.
	tx	Pointer to a user buffer containing the ‘raw’ RF data message to be sent.
	txLen	Number of bytes contained in tx.
Return Value	Status, indicating whether the operation was successful.	

7.11.2.6. ptx_RUL_HCE_ReserveEventRecord

Declaration	`ptx_Status_t ptx_RUL_HCE_ReserveEventRecord ( ptx_RUL_HCE_t *ctx, ptx_RUL_HCE_EventRecord_t **eventRecord, uint16_t *eventRecordIndex );`	
Description	This function reserves an event record at the tail of the event queue.	
Parameters	ctx	Pointer to the allocated HCE component.
	eventRecord	Reference to the event record at the end of the queue.
	eventRecordIndex	Index of the reserved eventRecord.
Return Value	Status, indicating whether the operation was successful.	

7.11.2.7. ptx_RUL_HCE_AddNewEventPending

Declaration	`ptx_Status_t` **`ptx_RUL_HCE_AddNewEventPending`** (
	`ptx_RUL_HCE_t *ctx,`
	`uint16_t recordIndex`
	);	

Description	This function adds a new pending event record at the end of the queue.	
Parameters	ctx	Pointer to the allocated HCE component.
	recordIndex	Index of the added event.
Return Value	Status, indicating whether the operation was successful.	

7.11.3. HCE API States

7.11.3.1. HCE API State Overview

Figure 41 describes the state diagram of the HCE API.

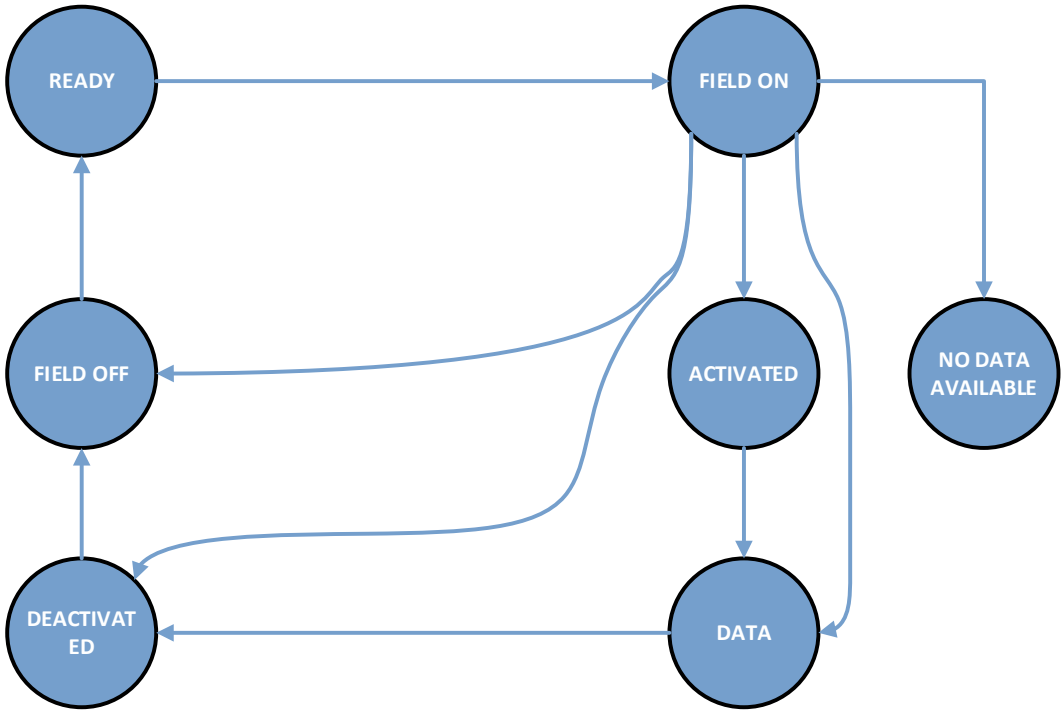

Figure 41. HCE API States

7.11.3.2. State Description

The HCE API only consists of the states:

- **READY**: Idle state of the HCE API, the API must be initialized using the "**ptx_RUL_HCE_Init**" function before any of the other API functions can be used. In th is state the listen mode is active, and the device is ready to be used
- **FIELD ON**: The device detected an external field. The user needs to call the "**ptx_RUL_HCE_GetEvent**" function to identify the current API state.
- **ACTIVATED**: The device is activated and is ready to transmit data. The user needs to call the "**ptx_RUL_HCE_GetEvent**" function to identify the current API state.

- **DATA**: The user can exchange data with the reader. The user needs to call the "**ptx_RUL_HCE_GetEvent**" function to identify the current API state. Data can be exchanged by calling the "**ptx_RUL_HCE_SendData**" function.
- **DEACTIVATED**: The device is deactivated. The user needs to call the "**ptx_RUL_HCE_GetEvent**" function to identify the current API state
- **FIELD OFF**: No external field is detected while it was on previously. The user needs to call the "**ptx_RUL_HCE_GetEvent**" function to identify the current API state.
- **NO DATA AVAILABLE**: No current event is in the message queue. The user needs to call the "**ptx_RUL_HCE_GetEvent**" function to identify the current API state

7.11.4. Implementation Guidelines/Suggestions

- A reference usage example of the HCE API to emulate an T3T can be found under the "**ptx_Example_HCET3T**" folder in the reference examples.
- A reference usage example of the HCE API to emulate an T4T can be found under the "**ptx_Example_HCET4T**" folder in the reference examples.
- The HCE API uses an internal message queue which must be read out by the user using "ptx_RUL_HCE_GetEvent". The user is responsible to handle the events in the message queue according to the state machine in section 7.11.3.1.
- Field On/Off events can fill the message queue if it is not periodically read out. This can happen if the PTX2K is operated at the edge of the field.

7.12 PSL API

The Product Support Library (PSL) is designed to assist customers in seamlessly integrating the PTX2xxR/W into the final target application. It provides a toolset of tailored APIs to identify and debug potential performance issues during the design-in phase of the product development, thus streamlining the integration process and ensuring that any performance bottlenecks are detected at an early stage.

The following sub-sections give an overview of each available API.

7.12.1. API Architecture

The PSL API module is provided as stand-alone functionality of the RUL SDK. It is located on top of the RUL API module (see Figure 42) and internally accesses the NSC Core Stack module.

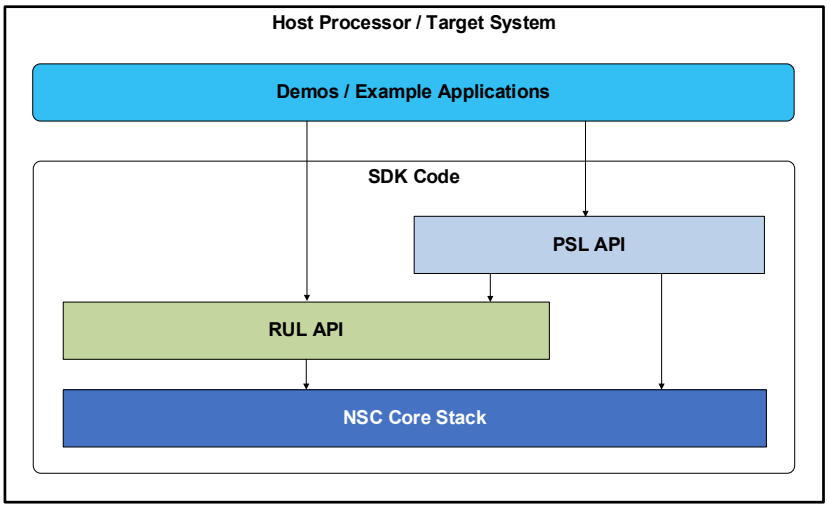

Figure 42. PSL API Component Architecture

Note: The PSL API module is optional and can be completely excluded from a target-build by following the instructions described in section 6.1.

7.12.2. API Description

7.12.2.1. ptx_PSL_Init

The function must be called before any other APIs of the PSL component. It prepares the PSL library for use by configuring internal settings and initializing the contact data structure required for the library to operate. Without invoking the **ptx_PSL_Init()** API, the other PSL APIs will return an error code.

Declaration	`ptx_Status_t` **`ptx_PSL_Init`** `(` **`ptx_PSL_t`** `*psl,` **`ptx_PSL_InitParams_t`** `*initParams);`	
Description	Initializes the PSL component.	
Parameters	psl	Pointer to an uninitialized PSL context.
	initParams	Various initialization parameters for the PSL component such as the **ptx_RUL_t** context and the (optional) **ptx_Log_t** context.
Return Value	Status, indicating whether the operation was successful.	

7.12.2.2. ptx_PSL_QMeasurement

The **ptx_PSL_QMeasurement()** API determines the most critical parameters of the attached antenna system: the Q-factor, resonance frequency, and the lower and upper 3dB corner frequencies. The system autonomously measures and characterizes the antenna without requiring any external measurement equipment; thus, the PTX2xxR/W will emit an RF field at various frequency steps within a defined specified span. At each measurement point, the evoked amplitude response from the attached antenna system is measured, gathering the necessary data to calculate the antenna system's performance characteristics.

Declaration	`ptx_Status_t` **`ptx_PSL_QMeasurement`**`(` **`ptx_PSL_t`** `*psl,` **`ptx_PSL_QMeas_Parameters_t`** `*measParams,` **`ptx_PSL_QMeas_Result_t`** `*measResult);`	
Description	Runs the Q-Measurement with the defined measurement parameters and returns the measurement result.	
Parameters	psl	Pointer to an initialized PSL context.
	measParams	Pointer to the measurement parameters containing the frequency span, frequency step and the ADC averaging configuration.
	measResult	Pointer to the struct for storing the measurement results.
Return Value	Status, indicating whether the operation was successful.	

7.12.2.3. ptx_PSL_RssiMeasurement

The Received Signal Strength Indicator (RSSI) signifies the signal strength picked up by the receiver of the PTX2xxR/W. It can be used to estimate the amplitude of the RF field currently available on the RF pins. The value is derived from the differential configuration of the receive chain and depends on utilizing both the RXp- and RXn-pins. It is used during the design-in phase to better understand the impact of an object/tag/coil entering the RF field of the PTX2xxR/W.

Declaration	`ptx_Status_t` **`ptx_PSL_RssiMeasurement`**`(`
	`  ptx_PSL_t *psl,`
	`  uint8_t cwPowerLevel,`
	`  ptx_RUL_AdcAveraging_t averaging,`
	`  uint32_t *result);`	

Description	Runs the RSSI measurement at 13.56MHz using the specified power level and ADC averaging configuration.	
Parameters	psl	Pointer to an initialized PSL context.
	cwPowerLevel	The power level to be used for the measurement. A range of 0–177 is allowed.
	averaging	The averaging configuration for the ADC.
	result	Pointer to store the measurement result.
Return Value	Status, indicating whether the operation was successful.	

7.12.2.4. ptx_PSL_SigDetMeasurement

Declaration	`ptx_Status_t` **`ptx_PSL_SigDetMeasurement`**`(`
	`  ptx_PSL_t *psl,`
	`  ptx_PSL_SigDet_InParams_t *inParams,`
	`  uint32_t *sigDetValue`
	`);`	

Description	Measures the system inherent noise seen by the receiver to determine the signal detector threshold necessary to prevent unintended activation of the digital receiver. The measurement is data rate and type depending.	
Parameters	psl	Pointer to an initialized PSL context.
	inParams	Pointer to the measurement parameters.
	sigDetValue	Pointer to store the measurement result.
Return Value	Status, indicating whether the operation was successful.	

7.12.2.5. ptx_PSL_SetDdpcParameters

The Digital Dynamic Power Control (DPPC) measurement functionality dynamically adjusts the emitted RF field strength in response to the presence or absence of objects within the RF field. This mechanism optimizes system performance by ensuring efficient power consumption and meeting the requirements of various standards such as EMVCo.

The primary objective of the DDPC measurement is to achieve a more homogeneous RF field within a specified operating volume. To accomplish this, the IC automatically alternates between two distinct output power levels during polling, maintaining the selected level throughout the communication cycle. The judgment of the appropriate power level is influenced by the presence of an external coil within the emitted RF field. In the absence of an external coil, the device defaults to a higher power level. Conversely, if an external coil sufficiently impacts the field (as when transmitter and receiver antennas converge), the device selects a lower power level.

The API is used to temporarily configure the parameters for the DDPC, including the power levels to be used, the thresholds used by the system to determine when to switch to another power level, and the averaging factor for the internal ADC. Once an acceptable configuration has been found by empirical evaluation, it can be added to the RF configuration of the PTX2xxR/W so that it is applied upon each system bootup.

Declaration	`ptx_Status_t` **`ptx_PSL_SetDdpcParameters`**`(` **`ptx_PSL_t`** `*psl,` **`ptx_PSL_Ddpc_Params_t`** `*params);`	
Description	Sets the DDPC parameters temporarily. The function is used to test	
Parameters	psl	Pointer to an initialized PSL context.
	cwPowerLevel	The power level to be used for the measurement. A range of 0–177 is allowed.
	averaging	The averaging configuration for the ADC.
	result	Pointer to store the measurement result.
Return Value	Status, indicating whether the operation was successful.	

7.12.2.6. ptx_PSL_RunDdpcEvaluation

The DDPC evaluation API performs two RSSI measurements at the two power levels specified by the user (High Power and Low Power). The DDPC evaluation API should be executed twice, once in an unloaded field (in other words, without a card in the immediate vicinity) and the second time with the card in the field at the exact position where the switching process is to take place. The two measurements result in four measured RSSI values which can be used as a basis for setting suitable thresholds for the DDPC via **SetDdpcParameters()**.

Declaration	`ptx_Status_t` **`ptx_PSL_RunDdpcEvaluation`**`(` **`ptx_PSL_t`** `*psl,` **`ptx_PSL_Ddpc_Measurement_Params_t`** `*measParams,` **`ptx_PSL_Ddpc_Results_t`** `*rssiResults);`	
Description	Evaluates the RSSI using the given power levels and the averaging configuration of the ADC.	
Parameters	psl	Pointer to an initialized PSL context.
	measParams	Pointer to the specified measurement parameters.
	rssiResults	Pointer to store the measurement results.
Return Value	Status, indicating whether the operation was successful.	

7.12.2.7. ptx_PSL_RunDdpcTest

The API runs the DDPC measurement procedure with a configurable measurement interval for testing purposes. The system will use the DDPC parameters defined in the RF configuration or, if previously set, the parameters provided via the **ptx_PSL_SetDdpcParameters()** API. Upon API execution, the PTX2xxR/W activates the RF field with a constant sine wave. If correctly configured, it automatically adjusts the field strength based on the distance between its antenna and the tag's antenna. Reducing the distance will eventually trigger a switch to the lower power level, while increasing the distance triggers the switch to the higher one.

Declaration	`ptx_Status_t` **`ptx_PSL_RunDdpcTest`**`(` **`ptx_PSL_t`** `*psl,` **`uint8_t`** `measIntervalMs);`	
Description	Continuously runs the DDPC test procedure with a defined period.	
Parameters	psl	Pointer to an initialized PSL context.
	measIntervalMs	Measurement interval in milliseconds.
Return Value	Status, indicating whether the operation was successful.	

7.12.2.8. ptx_PSL_GetDdpcStatus

This function is executed while the DDPC measurement procedure is in progress (triggered via **ptx_PSL_RunDdpcTest()**). The user receives feedback on the current RSSI value and the power level in use. These values change depending on any tags or metal objects moved within the RF field.

Declaration	`ptx_Status_t` **`ptx_PSL_GetDdpcStatus`**`(` **`ptx_PSL_t`** `*psl,` **`ptx_PSL_Ddpc_Status_t`** `*status);`	
Description	Continuously runs the DDPC test procedure with a defined period.	
Parameters	psl	Pointer to an initialized PSL context.
	status	Pointer to store the DDPC measurement status.
Return Value	Status, indicating whether the operation was successful.	

7.12.2.9. ptx_PSL_StopDdpcTest

Executing this API terminates the DDPC measurement procedure. The system switches off the RF field and transitions into the RF_IDLE state.

Declaration	`ptx_Status_t` **`ptx_PSL_StopDdpcTest`**`(` **`ptx_PSL_t`** `*psl);`	
Description	Stops the DDPC test procedure.	
Parameters	psl	Pointer to an initialized PSL context.
Return Value	Status, indicating whether the operation was successful.	

7.12.2.10. ptx_PSL_AnaMuxSelectSignal

This API selects internal analog test signals and multiplexes them to one of the Atst0–Atst3 pins. The Atst0–Atst3 pins correspond to the pins DIG4, DIG6, DIG7, and DIG8, respectively. Additionally, output buffers can be enabled on Atst0 and Atst1 to buffer the signals accordingly. Note that once a specific Atst pin is selected to output one of the internal analog signals, it will lose its digital configuration.

Declaration	`ptx_Status_t` **`ptx_PSL_AnaMuxSelectSignal`**`(` **`ptx_PSL_t`** `*psl,` **`ptx_PSL_AnaTestPin_t`** `testPinNr,` **`ptx_PSL_AnaMuxTestSignals_t`** `signal);`	
Description	Deinitializes the PSL component.	
Parameters	psl	Pointer to an initialized PSL context.
	testPinNr	Test pin number (Atst0–Atst3).
	signal	Test signal to be muxed to the specified pin.
Return Value	Status, indicating whether the operation was successful.	

7.12.2.11. ptx_PSL_DigMuxConfigure

This API provides the initial configuration to the digital test multiplexers. Internally, the digital test multiplexer consists of test signal vectors which are divided into two sub-vectors, each 8-bit wide. These 8-bit vectors can be swapped between TestBus0 (TB0) and TB1. The signal vectors can be accessed via GPIOs 1–8. The bits within the test busses can be swapped.

For TB0, there are four possibilities:

- Direct connected through { [7:0] }
- Swapping the nibbles { [3:0], [7:4] }
- Tuple-wise swapping per nibble { [5:4], [7:6], [1:0], [3:2] }
- Inverse tuple-wise swapping { [1:0], [3:2], [5:4], [7:6] }

On TB1, only 5 bits can be switched to the pads, but the switching choices are the same as those for TB0. Each debug signal can additionally be replaced by the system clock.

Declaration	`ptx_Status_t` **`ptx_PSL_DigMuxConfigure(`** **`ptx_PSL_t`** `*psl,` **`uint8_t`** `swapTestMux,` **`ptx_PSL_TbSwapType_t`** `swapTB1,` **`ptx_PSL_TbSwapType_t`** `swapTB0,` **`uint8_t`** `sysClkPinNbr);`	
Description	Configure the digital multiplexer with the provided configuration to analyze internal digital test signals.	
Parameters	psl	Pointer to an initialized PSL context.
	swapTestMux	Enables swapping for complete TB0, TB1.
	swapTB1	Defines the bit swapping configuration within TB1.
	swapTB0	Defines the bit swapping configuration within TB0.
	sysClkPinNbr	Defines the pin number for the system clock output.
Return Value	Status, indicating whether the operation was successful.	

7.12.2.12. ptx_PSL_DigMuxSelectSignal

This function selects the actual test vector to be muxed out via DIG1–DIG8.

Declaration	`ptx_Status_t` **`ptx_PSL_DigMuxSelectSignal`**`(` **`ptx_PSL_t`** `*psl,` **`ptx_PSL_TB_t`** `signal);`	
Description	Select a test vector signal.	
Parameters	psl	Pointer to an initialized PSL context.
	signal	Test vector to be muxed out.
Return Value	Status, indicating whether the operation was successful.	

7.12.2.13. ptx_PSL_DigMuxReset

The function resets the digital multiplexer and the configuration of the GPIOs back to their initial reset state.

Declaration	`ptx_Status_t` **`ptx_PSL_DigMuxReset`**`(` **`ptx_PSL_t`** `*psl);`	
Description	Resets the digital multiplexer.	
Parameters	psl	Pointer to an initialized PSL context.
Return Value	Status, indicating whether the operation was successful.	

7.12.2.14. ptx_PSL_DeInit

The function must be called to de-initialize the PSL component.

Declaration	`ptx_Status_t` **`ptx_PSL_DeInit`**`(` **`ptx_PSL_t`** `*psl);`	
Description	Deinitializes the PSL component.	
Parameters	psl	Pointer to an initialized PSL context.
Return Value	Status, indicating whether the operation was successful.	

7.12.3. PSL API States

7.12.3.1. PSL State Overview

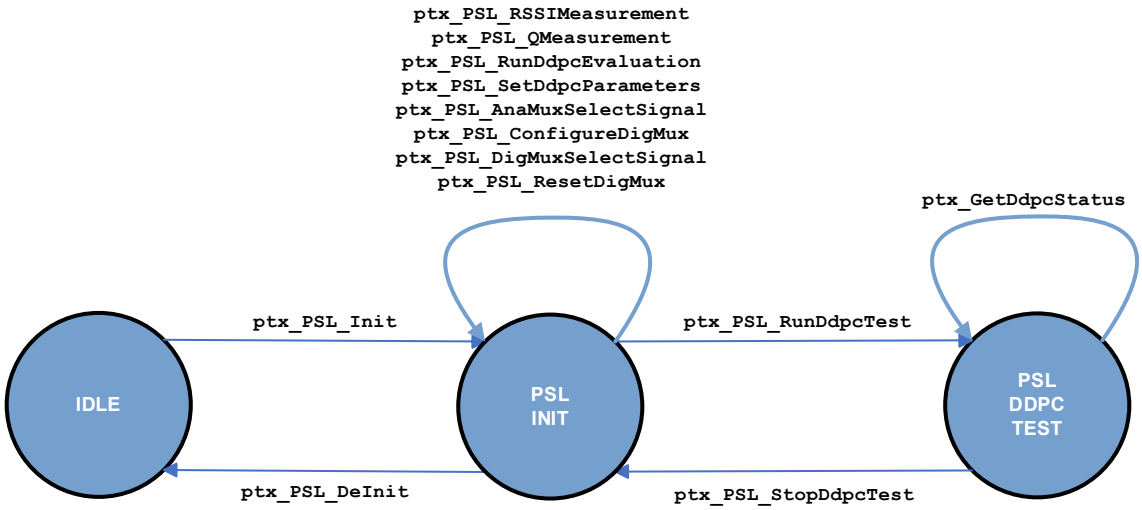

Figure 43. PSL API States

7.12.3.2. State Description

- **IDLE**: The Idle state is the initial state of the component when it is still uninitialized. To get to the **INIT** state, one has to call **ptx_PSL_Init().**
- **INIT**: The component and all internal data structures are initialized. In this state, almost all available API functions can be called except for the APIs related to the DDPC test routines.
- **DDPC TEST**: By calling **ptx_PSL_RunDdpcTest()** the system enters the “DDPC Test” state. In this state. The PTX2K will apply an RF field while permanently measuring the RSSI. Based on the values set through **ptx_PSL_SetDdpcParameters()** the user should see an increase/decrease in the applied field strength by varying the distance between an RFID Tag and the PTX2K’s antenna. The state can be left by calling **ptx_PSL_StopDdpcTest().**

7.12.4. Implementation Guidelines/Suggestions

- Special care must be taken with the TestMux APIs **ptx_PSL_ConfigureDigMux(), ptx_PSL_DigMuxSelectSignal() and ptx_PSL_AnaMuxSelectSignal()** as the GPIO pins 1–8 of the PTX2K are reused. The PSL overrides any settings previously made by the GPIOS, for example, via **ptx_RUL_ConfigGPIO()**.

  Ensure that in a final System/Product, these pins are not connected to other system components when calling the APIs. This may lead to unexpected behavior.
- In case of the **ptx_PSL_AnaMuxSelectSignal()** please consider the mapping of the TestSignal to the GPIOS:
  - ATST0 = GPIO4
  - ATST1 = GPIO6
  - ATST2 = GPIO7
  - ATST3 = GPIO8

7.13 Wireless Charging API

The Wireless Charging (WLC) API adds the NFC Forum Wireless Charging Protocol functionality for the WLC Poller device to the PTX230W. It can be used to perform the compliant protocol via an internal state machine, or it is possible to create a user defined state machine or protocol with the provided API. It acts as an essential API module which uses other Add-On Libraries to perform WLC operations with the PTX230W.

The following sub-sections give an overview of each available API.

Attention: Usage of the Wireless Charging Protocol Poller functionality requires an initial calibration of the on-chip FW during design-in phase to prevent potential TX-ESD events. For more details on the calibration, please refer to sections 3.1.2.1 ptx_RUL_CalibrateTxEsdPreventionThreshold and 9.4 Tx-ESD Prevention Calibration.

7.13.1. API Architecture

The WLC API module is provided as an essential module of the RUL SDK when using it for the purpose of wireless charging. It is located on top of the RUL API module (see Figure 44) and internally accesses the NSC Core Stack module. Additionally, it also uses the Add-On Libraries for NDEF and Native Tag to be able to detect any tags in the field and identify them as WLC Listeners. They are also required to perform the actual wireless charging protocol.

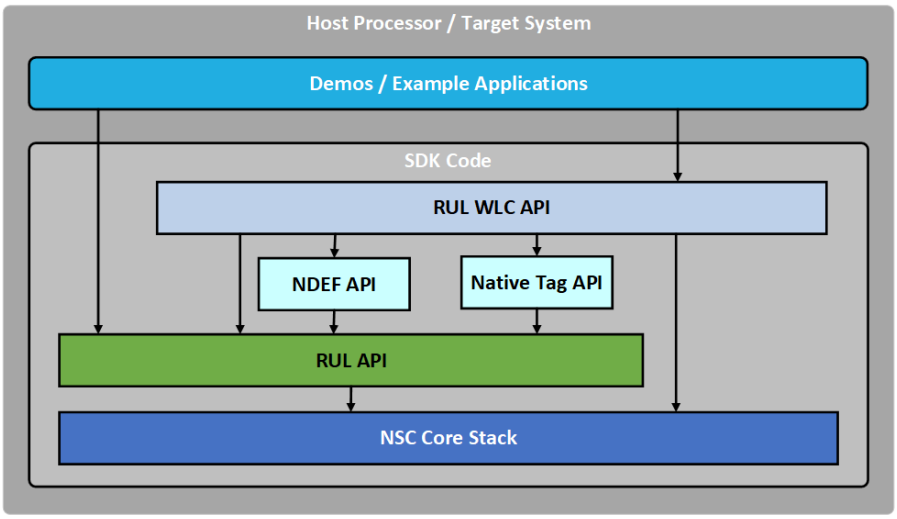

Figure 44. WLC API Component Architecture

7.13.2. General WLC API

The general WLC API functions are either used in all applications for initializing or deinitializing the WLC component, or to give direct access to the power transfer of the PTX230W. These functions can directly start and stop a power transfer as well as get the current state of it. These are used internally for the implemented NFC Forum compliant state machine but can also be used to realize user specific proprietary WLC protocols.

7.13.2.1. ptx_RUL_WLC_Init

Declaration	`ptx_Status_t ptx_RUL_WLC_Init(`
	`  ptx_RUL_WLC_t *ctx,`
	`  ptx_RUL_WLC_InitParams_t *initParam);`	

Description	Initializes the software for any operation with the WLC module. This function must be called before any other API functions of this component.	
Parameters	ctx	Pointer to an uninitialized WLC context.
	initParam	Pointer to init parameters to configure software for proper operation.
Return Value	Status, indicating whether the operation was successful.	

7.13.2.2. ptx_RUL_WLC_DeInit

Declaration	`ptx_Status_t ptx_RUL_WLC_DeInit(` `  ptx_RUL_WLC_t *ctx);`	
Description	Deinitializes the software to deactivate any operation with the WLC module. This function must be called at last for this component.	
Parameters	ctx	Pointer to an initialized WLC context.
Return Value	Status, indicating whether the operation was successful.	

7.13.2.3. ptx_RUL_WLC_GetListenerType

Declaration	`ptx_Status_t ptx_RUL_WLC_GetListenerType(`
	`  ptx_RUL_WLC_t *ctx,`
	`  ptx_RUL_WLC_ListenerType_t *type);`	

Description	Retrieves the type of the listener device (either “Generic” or “Proprietary”). This function needs to be used to verify that the Proprietary WLC API is applicable to the currently activated listener.	
Parameters	ctx	Pointer to an initialized WLC context.
	type	Pointer to store the information whether the tag is a Generic or Proprietary Listener.
Return Value	Status, indicating whether the operation was successful.	

7.13.2.4. ptx_RUL_WLC_StartPowerTransfer

Declaration	`ptx_Status_t ptx_RUL_WLC_StartPowerTransfer(`
	`  ptx_RUL_WLC_t *ctx,`
	`  ptx_RUL_WLC_WptOptions_t *wptOptions);`	

Description	Starts a Wireless Power Transfer (WPT) Cycle with the given options.	
Parameters	ctx	Pointer to an initialized WLC context.
	wptOptions	Pointer to the options, containing all required settings for the WPT cycle.
Return Value	Status, indicating whether the operation was successful.	

7.13.2.5. ptx_RUL_WLC_StopPowerTransfer

Declaration	`ptx_Status_t ptx_RUL_WLC_StopPowerTransfer(`
	`  ptx_RUL_WLC_t *ctx,`
	`  uint8_t postPowerLevel);`	

Description	Stops an ongoing Wireless Power Transfer (WPT) Cycle and sets the given power level for the next communication.	
Parameters	ctx	Pointer to an initialized WLC context.
	postPowerLevel	Power level to proceed with in communication, once the WPT cycle has been stopped/cancelled.
Return Value	Status, indicating whether the operation was successful.	

7.13.2.6. ptx_RUL_WLC_GetPowerTransferState

Declaration	`ptx_Status_t ptx_RUL_WLC_GetPowerTransferState(`
	`  ptx_RUL_WLC_t *ctx,`
	`  ptx_RUL_WLC_WptState_t *state);`	

Description	Retrieves information about the state of the currently ongoing or stopped WPT cycle.	
Parameters	ctx	Pointer to an initialized WLC context.
	state	Pointer for storing the WPT state information.
Return Value	Status, indicating whether the operation was successful.	

7.13.2.7. ptx_RUL_WLC_RunPollerStateMachine

This will start the internally implemented NFC Forum WLC-P compliant state machine. Be aware that this function blocks until the state machine terminates for two different scenarios by setting **ptx_RUL_WLC_t.StateMachineActive** to zero.

1. It will terminate by itself due to an error, if the corresponding notification is not handled properly at application level. This must be done by processing the error and setting the status/systemInfo in the argument of the callback to PTX_STATUS_SUCCESS/ptx_RUL_StatusInfoSystem_OK. Otherwise, will the state machine terminate by setting **ptx_RUL_WLC_t.StateMachineActive** to zero.
2. It is possible to terminate the state machine in any other notification (see **ptx_RUL_WLC_Notification_t**). To do this, the variable **ptx_RUL_WLC_t.StateMachineActive** needs to be set to zero.

Declaration	`ptx_Status_t ptx_RUL_WLC_RunPollerStateMachine(` `  ptx_RUL_WLC_t *ctx);`	
Description	Run the internal BLOCKING NFC Forum WLC-P compliant State Machine. This function shall only be called when **`ptx_RUL_WLC_t.StateMachineActive`** is set to zero (No currently active State Machine).	
Parameters	ctx	Pointer to an initialized WLC context.
Return Value	Status, indicating whether the operation was successful.	

7.13.3. State Machine WLC API

The different state machine functions of the WLC API represent the functionality for each state in the NFC Forum compliant WLC Poller state machine. These functions are used in the internal state machine to realize the NFC Forum compliant WLC Poller behavior. But it is also possible to use these functions to create a user specific state machine to, for example, use the proprietary API functions when connected to a compatible Renesas WLC Listener Device.

7.13.3.1. ptx_RUL_WLC_StateNFCLE_StartPollListener

Declaration	`ptx_Status_t ptx_RUL_WLC_StateNFCLE_StartPollListener(` `  ptx_RUL_WLC_t *ctx);`	
Description	Starts the polling sequence for listeners in the RF field. This function shall only be called when **ptx_RUL_WLC_t.State** is State_NFCLE_StartPollListener.	
Parameters	ctx	Pointer to an initialized WLC context.
Return Value	Status, indicating whether the operation was successful.	

7.13.3.2. ptx_RUL_WLC_StateNFCLE_PollListener

Declaration	`ptx_Status_t ptx_RUL_WLC_StateNFCLE_PollListener(` `ptx_RUL_WLC_t *ctx);`	
Description	Checks for detected listeners in the RF field through LPCD/Discovery. This function shall only be called when **ptx_RUL_WLC_t.State** is State_NFCLE_PollListener.	
Parameters	ctx	Pointer to an initialized WLC context.
Return Value	Status, indicating whether the operation was successful.	

7.13.3.3. ptx_RUL_WLC_StateNFCLE_ActivateListener

Declaration	`ptx_Status_t ptx_RUL_WLC_StateNFCLE_ActivateListener(` `ptx_RUL_WLC_t *ctx);`	
Description	In case multiple Listeners were detected/discovered, this function activates one Listener after another, until a WLC compatible device was found. This function shall only be called when **ptx_RUL_WLC_t.State** is State_NFCLE_ActivateListener.	
Parameters	ctx	Pointer to an initialized WLC context.
Return Value	Status, indicating whether the operation was successful.	

7.13.3.4. ptx_RUL_WLC_StateWCCA_ReadCapabilityRecord

Declaration	`ptx_Status_t ptx_RUL_WLC_StateWCCA_ReadCapabilityRecord(` `ptx_RUL_WLC_t *ctx);`	
Description	Reads and parses the WLC Capability record (and any optional records) with an activated WLC compatible Listener and continues the protocol for the different charging modes. This step validates the Listener as fully WLC compatible. This function shall only be called when **ptx_RUL_WLC_t.State** is State_WCCA_ReadCapabilityRecord.	
Parameters	ctx	Pointer to an initialized WLC context.
Return Value	Status, indicating whether the operation was successful.	

7.13.3.5. ptx_RUL_WLC_StateWCCA_StartPollListenerReady

Declaration	`ptx_Status_t ptx_RUL_WLC_StateWCCA_StartPollListenerReady(` `ptx_RUL_WLC_t *ctx);`	
Description	If requested by the WLC Capability record, start waiting until the listener is ready to proceed with the WLC protocol for negotiated mode. This function shall only be called when **ptx_RUL_WLC_t.State** is State_WCCA_StartPollListenerReady.	
Parameters	ctx	Pointer to an initialized WLC context.
Return Value	Status, indicating whether the operation was successful.	

7.13.3.6. ptx_RUL_WLC_StateWCCA_PollListenerReady

Declaration	`ptx_Status_t ptx_RUL_WLC_StateWCCA_PollListenerReady(` `ptx_RUL_WLC_t *ctx);`	
Description	Checks if the requested waiting time from the listener elapsed and if it is ready to proceed with the WLC protocol for negotiated mode. This function shall only be called when **ptx_RUL_WLC_t.State** is State_WCCA_PollListenerReady.	
Parameters	ctx	Pointer to an initialized WLC context.
Return Value	Status, indicating whether the operation was successful.	

7.13.3.7. ptx_RUL_WLC_StateWCC_WriteInfoRecord

Declaration	`ptx_Status_t ptx_RUL_WLC_StateWCC_WriteInfoRecord(` `ptx_RUL_WLC_t *ctx);`	
Description	Writes the WLC Information record to the listener (without any optional records) for negotiated mode. This function shall only be called when **ptx_RUL_WLC_t.State** is State_WCC_WriteInfoRecord.	
Parameters	ctx	Pointer to an initialized WLC context.
Return Value	Status, indicating whether the operation was successful.	

7.13.3.8. ptx_RUL_WLC_StateWCC_ReadControlRecord

Declaration	`ptx_Status_t ptx_RUL_WLC_StateWCC_ReadControlRecord(` `ptx_RUL_WLC_t *ctx);`	
Description	Reads and parses the WLC Control record from the listener (and any optional records) for negotiated mode. This function shall only be called when **ptx_RUL_WLC_t.State** is State_WCC_ReadControlRecord.	
Parameters	ctx	Pointer to an initialized WLC context.
Return Value	Status, indicating whether the operation was successful.	

7.13.3.9. ptx_RUL_WLC_StateWPT_StartStaticCharging

Declaration	`ptx_Status_t ptx_RUL_WLC_StateWPT_StartStaticCharging(` `ptx_RUL_WLC_t *ctx);`	
Description	Starts a WPT cycle in static charging mode. This function shall only be called when **ptx_RUL_WLC_t.State** is State_WPT_StartStaticCharging.	
Parameters	ctx	Pointer to an initialized WLC context.
Return Value	Status, indicating whether the operation was successful.	

7.13.3.10. ptx_RUL_WLC_StateWPT_StartNegotiatedCharging

Declaration	`ptx_Status_t ptx_RUL_WLC_StateWPT_StartNegotiatedCharging(` `ptx_RUL_WLC_t *ctx);`	
Description	Starts a WPT cycle in negotiated charging mode. This function shall only be called when **ptx_RUL_WLC_t.State** is State_WPT_StartNegotiatedCharging.	
Parameters	ctx	Pointer to an initialized WLC context.
Return Value	Status, indicating whether the operation was successful.	

7.13.3.11. ptx_RUL_WLC_StateWPT_PollCharging

Declaration	`ptx_Status_t ptx_RUL_WLC_StateWPT_PollCharging(` `ptx_RUL_WLC_t *ctx);`	
Description	Checks if the currently ongoing WPT cycle is finished. This function shall only be called when **ptx_RUL_WLC_t.State** is State_WPT_PollCharging.	
Parameters	ctx	Pointer to an initialized WLC context.
Return Value	Status, indicating whether the operation was successful.	

7.13.3.12. ptx_RUL_WLC_StateSleep_StartSleep

Declaration	`ptx_Status_t ptx_RUL_WLC_StateSleep_StartSleep(` `  ptx_RUL_WLC_t *ctx);`	
Description	Starts the sleep mode of the Poller if requested by the listener. In this state, the Poller deactivates its RF-field for a certain amount of time (specified by the listener). This for example happens when the listener reports 'battery full' in the capability record. This function shall only be called when **ptx_RUL_WLC_t.State** is State_Sleep_StartSleep.	
Parameters	ctx	Pointer to an initialized WLC context.
Return Value	Status, indicating whether the operation was successful.	

7.13.3.13. ptx_RUL_WLC_StateSleep_PollSleep

Declaration	`ptx_Status_t ptx_RUL_WLC_StateSleep_PollSleep(` `  ptx_RUL_WLC_t *ctx);`	
Description	Checks if the whole requested sleep duration is finished and if a single sleep interval elapsed. If an interval elapses, a presence check is executed to ensure that the Listener is still in the field. If it is still present, the sleep should continue with the next interval, otherwise it should start from the beginning by polling for new Listeners. This function shall only be called when **ptx_RUL_WLC_t.State** is State_Sleep_PollSleep.	
Parameters	ctx	Pointer to an initialized WLC context.
Return Value	Status, indicating whether the operation was successful.	

7.13.3.14. ptx_RUL_WLC_StateOff_DeactivateListener

Declaration	`ptx_Status_t ptx_RUL_WLC_StateOff_DeactivateListener(` `  ptx_RUL_WLC_t *ctx);`	
Description	Deactivates the currently activated listener and starts the WLC state machine again in state State_NFCLE_StartPollListener. This function shall only be called when **ptx_RUL_WLC_t.State** is State_Off_DeactivateListener.	
Parameters	ctx	Pointer to an initialized WLC context.
Return Value	Status, indicating whether the operation was successful.	

7.13.3.15. ptx_RUL_WLC_StateError_ProcessError

Declaration	`ptx_Status_t ptx_RUL_WLC_StateError_ProcessError(` `  ptx_RUL_WLC_t *ctx);`	
Description	Processes any errors that happened in the state machine either through a status or through the system state. This could either terminate the state machine if left unprocessed or start again in state State_NFCLE_StartPollListener if processed correctly. Processing these errors can be done by handling the corresponding notifications in the WLC notification callback function. This function shall only be called when **ptx_RUL_WLC_t.State** is State_Off_DeactivateListener.	
Parameters	ctx	Pointer to an initialized WLC context.
Return Value	Status, indicating whether the operation was successful.	

7.13.4. Proprietary WLC API

The proprietary WLC API is used to operate with a compatible Renesas WLC Listener device. With such a device it is possible to use proprietary features which are available on top of the NFC Forum compliant functionality. Before using these, it is always necessary to check with **ptx_RUL_WLC_GetListenerType** if the currently connected listener device can use these features.

7.13.4.1. ptx_RUL_WLC_Proprietary_ReadT2T

Declaration	`ptx_Status_t ptx_RUL_WLC_Proprietary_ReadT2T(`
	`  ptx_RUL_WLC_t *ctx,`
	`  uint8_t block,`
	`  uint8_t *rxData,`
	`  uint8_t *rxDataLen);`	

Description	Assembles and transmits the proprietary REN_READ_CMD capable of reading 64 bytes at once from the Listener. Be aware that it is only possible to read block-wise, it is therefore required to reserve enough memory in rxData to read all required whole blocks.	
Parameters	ctx	Pointer to an initialized WLC context.
	block	Starting Block number to read from.
	rxData	Pointer to a buffer for storing the received data.
	rxDataLength	Pointer to the length of the buffer for storing the received data as input and the actual size of the received data as output.
Return Value	Status, indicating whether the operation was successful.	

7.13.4.2. ptx_RUL_WLC_Proprietary_WriteT2T

Declaration	`ptx_Status_t ptx_RUL_WLC_Proprietary_WriteT2T(`
	`  ptx_RUL_WLC_t *ctx,`
	`  uint8_t block,`
	`  uint8_t *txData,`
	`  uint8_t txDataLen);`	

Description	Assembles and transmits the proprietary REN_WRITE_CMD capable of writing 64 bytes at once to the Listener.	
Parameters	ctx	Pointer to an initialized WLC context.
	block	Starting Block number to read from.
	txData	Pointer to a buffer which stores the to be written data.
	txDataLength	Length of the data to be written.
Return Value	Status, indicating whether the operation was successful.	

7.13.4.3. ptx_RUL_WLC_Proprietary_SetListenerGpios

Declaration	`ptx_Status_t ptx_RUL_WLC_Proprietary_WriteT2T(`
	`  ptx_RUL_WLC_t *ctx,`
	`  bool setGpio0High,`
	`  bool setGpio1High);`	

Description	Controls the output level of the Listener's GPIOs. The Listener's GPIOs must be configured to "WlcpControlled".	
Parameters	ctx	Pointer to an initialized WLC context.
	setGpio0High	Sets Listener's GPIO0 to HIGH when 1 and to LOW otherwise.
	setGpio1High	Sets Listener's GPIO1 to HIGH when 1 and to LOW otherwise.
Return Value	Status, indicating whether the operation was successful.	

7.13.4.4. ptx_RUL_WLC_Proprietary_SetListenerShippingModeEnable

Declaration	`ptx_Status_t ptx_RUL_WLC_Proprietary_SetListenerShippingModeEnable(` `  ptx_RUL_WLC_t *ctx);`	
Description	Enable the Shipping Mode ("Deep Sleep" Mode) on the Listener to force it into entering this state.	
Parameters	ctx	Pointer to an initialized WLC context.
Return Value	Status, indicating whether the operation was successful.	

7.13.4.5. ptx_RUL_WLC_Proprietary_GetListenerSystemStatus

Declaration	`ptx_Status_t ptx_RUL_WLC_Proprietary_GetListenerSystemStatus(`
	`  ptx_RUL_WLC_t *ctx,`
	`  ptx_RUL_WLC_Proprietary_ListenerSystemStatus_t *systemStatus);`	

Description	Retrieves a system status information from the listener (e.g. detailed temperature information of the NTC).	
Parameters	ctx	Pointer to an initialized WLC context.
	systemStatus	Pointer to store the status information of the Listener.
Return Value	Status, indicating whether the operation was successful.	

7.13.4.6. ptx_RUL_WLC_Proprietary_TDC_Write

Declaration	`ptx_Status_t ptx_RUL_WLC_Proprietary_TDC_Write(`
	`  ptx_RUL_WLC_t *ctx,`
	`  uint8_t *txData,`
	`  uint8_t txDataLen`
	`  uint32_t ackTimeoutMs);`	

Description	Writes messages to the Listener's TDC buffer. The customer can choose between NFC Forum compliant write access and proprietary (= faster) write access via compile switch.	
Parameters	ctx	Pointer to an initialized WLC context.
	txData	Pointer to a buffer which stores the to be written data.
	txDataLength	Length of the data to be written.
	ackTimeoutMs	Timeout to wait for an acknowledgement from the listener (if 0, there will be no acknowledgement check).
Return Value	Status, indicating whether the operation was successful.	

7.13.4.7. ptx_RUL_WLC_Proprietary_TDC_IsReceived

Declaration	`ptx_Status_t ptx_RUL_WLC_Proprietary_TDC_IsReceived(`
	`ptx_RUL_WLC_t *ctx,`
	`uint8_t *received);`	

Description	Checks if the Listener's host MCU has read the previously sent message and an acknowledgement was received.	
Parameters	ctx	Pointer to an initialized WLC context.
	received	Pointer to store the status, if the message has been read (1 if it was received and 0 if not).
Return Value	Status, indicating whether the operation was successful.	

7.13.4.8. ptx_RUL_WLC_Proprietary_TDC_Read

Declaration	`ptx_Status_t ptx_RUL_WLC_Proprietary_TDC_Read(`
	`ptx_RUL_WLC_t *ctx,`
	`uint8_t *rxData,`
	`uint8_t *rxDataLen`
	`uint32_t ackTimeoutMs);`	

Description	Reads messages from the Listener's TDC buffer. Customer can choose between NFC Forum compliant read access and proprietary (= faster) read access via compile switch.	
Parameters	ctx	Pointer to an initialized WLC context.
	rxData	Pointer to a buffer for storing the received data.
	rxDataLength	Pointer to the length of the buffer for storing the received data as input and the actual
	rxTimeoutMs	Timeout for the message to be read from the Listener.
Return Value	Status, indicating whether the operation was successful.	

7.13.5. WLC API States

7.13.5.1. WLC State Overview

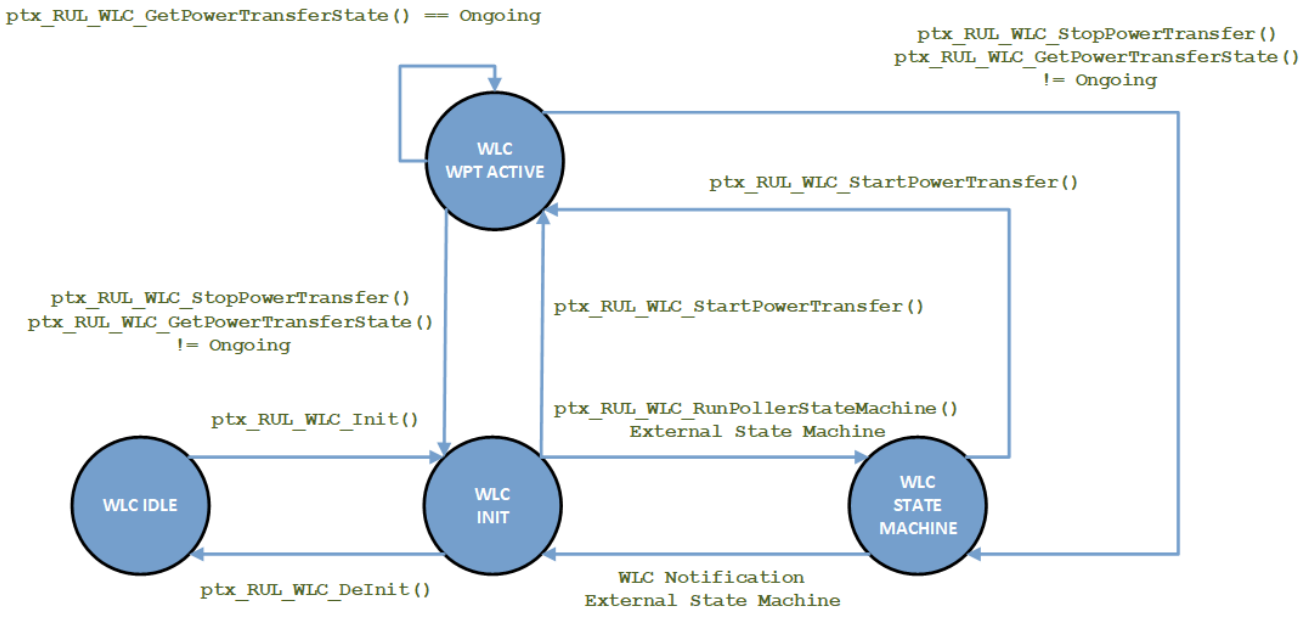

Figure 45. WLC API States

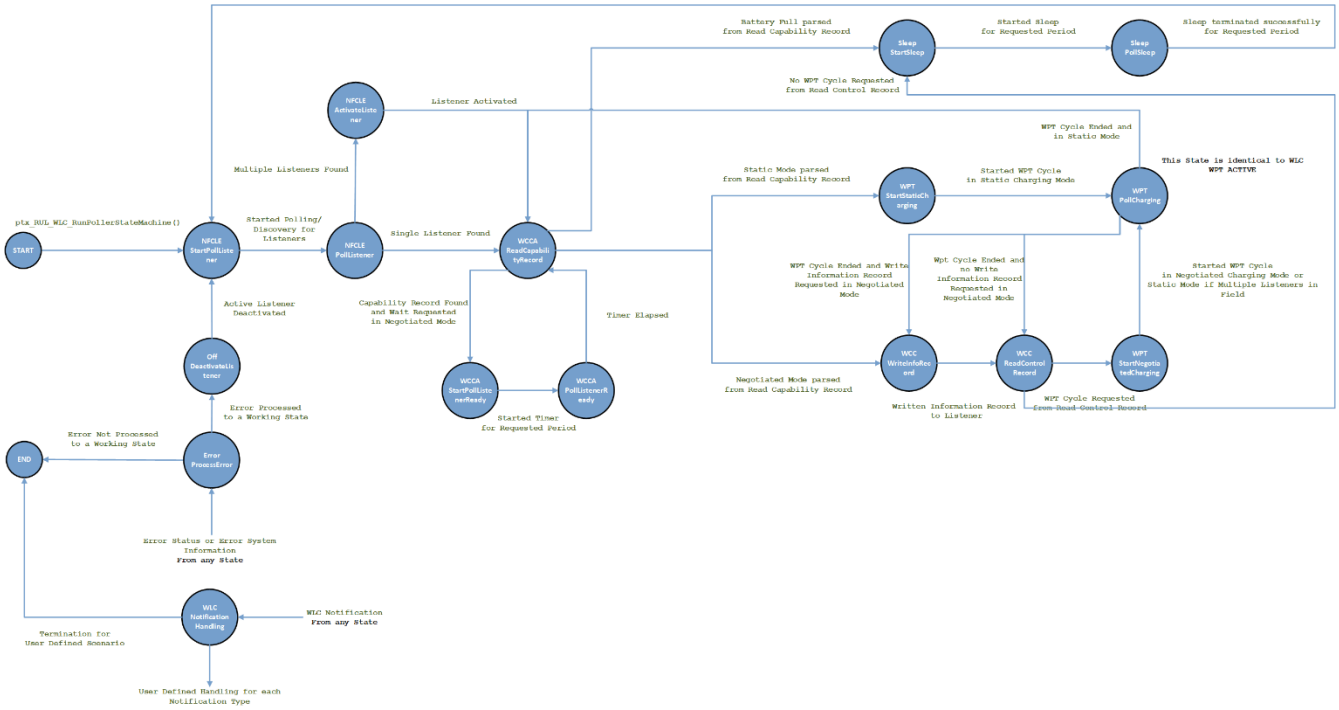

Figure 46. WLC State Machine API States

7.13.5.2. State Description WLC API States

The diagram from Figure 45 represents the general state machine of the WLC component, which is described in this chapter. There are substates when using a WLC state machine, which are described in the next chapter.

- **WLC IDLE**: This is the initial state if the WLC component is uninitialized. It can be entered by calling **ptx_RUL_WLC_DeInit()** and it can be left by calling **ptx_RUL_WLC_Init().**
- **WLC INIT**: This is the initialized state of the WLC component, from this, the whole functionality of the WLC component is available. It can be entered by calling **ptx_RUL_WLC_Init()** from **WLC IDLE** state but also from all other application states once these are finished.
- **WLC WPT ACTIVE**: This state is active, if a Wireless Power Transfer (WPT) Cycle is ongoing. In this state, no communication is ongoing and the RF field is constantly on, to transmit power to the activated Listener. It can be entered by starting the WPT cycle directly with **ptx_RUL_WLC_StartPowerTransfer()** in state **WLC INIT** or through a WLC state machine from state **WLC STATE MACHINE**. This can be either the internally implemented one as described later one or a user defined external state machine. For both cases also **ptx_RUL_WLC_StartPowerTransfer()** is used internally to enter this state as well. Leaving this state is possible for two scenarios, either the user directly terminates the transmission by calling **ptx_RUL_WLC_StopPowerTransfer()** or the WPT cycle finishes. It is possible to check with **ptx_RUL_WLC_CheckPowerTransferState()** for the current state of the WPT cycle.
- **WLC STATE MACHINE**: This represents the state if the WLC component has an active WLC state machine running. The substates for the internal variant are described in the next chapter. It can be entered for the internal state machine by calling **ptx_RUL_WLC_RunPollerStateMachine()**. This is a blocking state machine loop which can be terminated for certain scenarios (Error Status, Error System Information, WLC Notifications). The other option is to implement a user defined external state machine. In this, the user may use the state machine API similar as in the internal one and add certain additional features (for example, Proprietary API). It is also possible to realize a completely new state machine, which only uses the required API functions to start and stop a WPT cycle.

7.13.5.3. State Description WLC State Machine API States

The diagram from Figure 46 represents the substates when using the internal state machine. They can also be used to realize an external state machine by, for example adding user defined changes to the states. All the seen states are part of **WLC STATE MACHINE** from the general WLC states.

- **START**: This is the entering point of the internal WLC state machine (coming from **WLC INIT** of the general state machine). It can be started, by calling **ptx_RUL_WLC_RunPollerStateMachine()**, which will go to the **NFCLE StartPollListener** state. This is only applicable for the internal state machine, and for external implementations, there may be other ways to enter this state.
- **NFCLE StartPollListener**: This state starts a discovery/polling loop for listeners in the RF field for a preconfigured configuration. Once it starts, the state will switch to **NFCLE PollListener.**
- **NFCLE PollListener**: This state checks if listeners in the RF field have been detected. If there was not any detected tag, it would stay in this state until an exception happens, or a listener is detected. If there were multiple detected listeners, it would switch to the state **NFCLE ActivateListener**. If there was only one single listener, the machine would transition to the state **WCCA ReadCapabilityRecord**.
- **NFCLE ActivateListener**: In this state it is assumed that multiple listeners are detected. Therefore, it searches for a compatible card in the registry and activates it. Once this is done, it switches to the state **WCCA ReadCapabilityRecord.**
- **WCCA ReadCapabilityRecord**: At this point, there is an active listener in the RF field, but it is not certain that it is also capable of using the WLC protocol. For this reason, it is required to read an NDEF message from the listener and check for a WLC Capability record. If there is a valid record, it processes it and progresses to the next valid state. If the charging mode is set to static charging, it will transition to **WPT StartStaticCharging**, if it is set to negotiated charging and no waiting time is requested, it will transition to **WCC WriteInfoRecord**. If waiting time is requested, the state will transition to **WCCA StartPollListenerReady**. If the charging mode is set to battery full, it will transition to **Sleep StartSleep**.
- **WCCA StartPollListenerReady**: This state will start a timer, as requested by the WLC Capability record to wait for the listener to be ready to further process negotiated charging. It will transition into the state **WCCA PollListenerReady**.
- **WCCA PollListenerReady**: This state checks if the timer started in **WCCA StartPollListenerReady** is elapsed. It will stay in this state until this is true. Once finished, it will go back to **WCCA ReadCapabilityRecord.**
- **WCC WriteInfoRecord**: This states, builds a WLC Information record and writes it to the active listener. This is initially required to give the listener enough information about the charging capabilities of the poller. Once it is written, it will move to the state **WCC ReadControlRecord**.
- **WCC ReadControlRecord**: This state reads the current WLC Control record from the active listener. If there is a valid record, it will parse it further. If a WPT cycle is requested, it will transition to the state **WPT StartNegotiatedCharging**. If no cycle is required, it will start a sleep cycle by switching to **Sleep StartSleep**.
- **WPT StartStaticCharging**: In the static charging mode, a WPT cycle will be started with the same constant power level as the NFC communication, without any power level increase or decrease. Charging is started with the function **ptx_RUL_WLC_StartPowerTransfer().** Once the charging starts, it will transition into state **WPT PollCharging**.
- **WPT StartNegotiatedCharging**: In the negotiated charging mode, it will calculate the requested power level from the information received in the WLC Control record and start a WPT cycle with all required settings. Charging is started with the function **ptx_RUL_WLC_StartPowerTransfer().** Once the charging starts, it will transition into state **WPT PollCharging**.

- **WPT PollCharging**: This state is active if currently a WPT cycle is ongoing. It checks whether the cycle is still ongoing or if it is already terminated with the function **ptx_RUL_WLC_CheckPowerTransferState()**. Once it is finished, it will switch to various states depending on the previous information received. If the charging started in static mode, it would go back to **WCCA ReadCapabilityRecord** to start the process again from the beginning. If it started charging in negotiated mode, it will go back either to **WCC WriteInfoRecord** or **WCC ReadControlRecord**, which depends on if the Listener requested a WLC Information record in the last WLC Control record received.
- **Sleep StartSleep**: This state deactivates the activated listener and forces the PTX230W into standby mode to ensure low power consumption. When entering this state, information is received about the total sleep duration requested from the listener. It will then start a timer for a preconfigured sleep interval in which it will go to sleep. This interval should be smaller than the total sleep duration. Otherwise, will the presence check in **Sleep PollSleep** never happen. After this state, it will switch to **Sleep PollSleep.**
- **Sleep PollSleep**: After every preconfigured sleep interval, the Poller will wake up for a short time to start a discovery and to check if the previous activated listener is still in the field. This is the presence check. If this fails, the state machine will transition back to **NFCLE StartPollListener** and start from the beginning. If the listener is still present, a next sleep interval will start. This runs until the total sleep duration is elapsed. Then it will also start from the beginning in state **NFCLE StartPollListener.**
- **Off DeactivateListener**: This state deactivates the active listener and forces the PTX230W into standby mode to ensure low power consumption. Once this is done, it will transition to **NFCLE StartPollListener** and start from the beginning.
- **Error ProcessError**: This state is active whenever an error status or an error system information is returned from any state. It will then either process the corresponding error correctly to a working state again or it cannot process the error correctly to a working state. For the first variant, it will transition to **Off DeactivateListener** to reset the WLC component to a valid initial state and start the state machine again. The explicit processing of the errors can be done in the **WLC Notification Handling** state. For the second variant, the state will transition to **END.**
- **(WLC Notification Handling)**: This state represents an optional state, which needs to be implemented by the user. The user needs to create a function and assign it to the function pointer of type **ptx_RUL_WLC_NtfCb_t** in the context. In this function all available notifications can be handled once they are received. These notifications are sent from all the different state functions of the internal state machine. There are normal notifications, which give information about certain events in the WLC state machine, but there are also error notifications, which indicate that an error happened in the state machine. These are sent during the **Error ProcessError** state. Status and system information are sent as pointers for these cases to grant the ability to process these errors outside of the state machine and return valid values. Be aware that some errors (see API description) require to be processed further in the notification handler, otherwise the state machine will be terminated (state **END**). To set an error status to a valid processed state, it is required to set the erroneous status or system information to a valid value again. Be aware that it is also possible to terminate the state machine for any user specific scenario by setting **ptx_RUL_WLC_t.StateMachineActive** in the context. Once the processing is done, the notification handler returns to the position in the state machine it was called from.
- **END**: This state represents the termination point of the state machine, once this state is reached, the machine will end and go back to the **WLC INIT** state from the general state machine.

7.13.6. Implementation Guidelines/Suggestions

- When using the internal state machine functions, even if it is a user defined external state machine, it is always recommended to implement a **WLC Notification Handling** state/function to have the ability to get all information as well as handle all errors explicitly.
- It is only possible to run one state machine at a time. If using an external one, use also the internal flag to check whether it is already running.
- The WLC State Machine API functions should only be called inside of a valid state machine; everything else can lead to unknown behavior.
- To create a completely independent state machine, use the functions **ptx_RUL_WLC_StartPowerTransfer(), ptx_RUL_WLC_StopPowerTransfer()** and **ptx_RUL_WLC_CheckPowerTransferState()** to handle the actual power transmission. All other state/functions needs to be implemented by the user.
- Use **ptx_RUL_WLC_GetListenerType()** to check if the currently active Listener is a proprietary one before using the WLC Proprietary API.
- In any NDEF read operation of the internal state machine, also optional records are read from the Listener if they are available, these are available as dedicated notifications.
- When using the example **ptx_Example_WLC_DualPoller**, be aware that it currently is not allowing full synchronization between the two connected independent RRQ35230EB boards.

8. RF Configuration Updates

RF operations are generally very sensitive to the target environment, in other words, antenna size and shape, matching network and components, metal surroundings etc. To achieve the best possible performance for an application, the RF configuration settings need to be adapted accordingly. The PTX2xxR/W allows users to update RF configuration parameters using either statically generated RF configuration settings or to update the settings dynamically during runtime via an API call.

8.1 Static RF Configuration

The static RF configuration can be configured using the *PTX2xxR RUL Config Tool* from Renesas. Once configured, the settings are generated / exported into two **.c** and **.h** files called "**ptx_NSC_RfConfigVal.c**" and "**ptx_NSC_RfConfigVal.h**". Both files must be stored inside the RUL SDK in the folder "**\SRC\COMPS\NSC\RUL**" and must be integrated into the compile- and link-process.

Figure 47 shows a screenshot of the *PTX2xxR RUL Config Tool* and how the RF configuration can be generated.

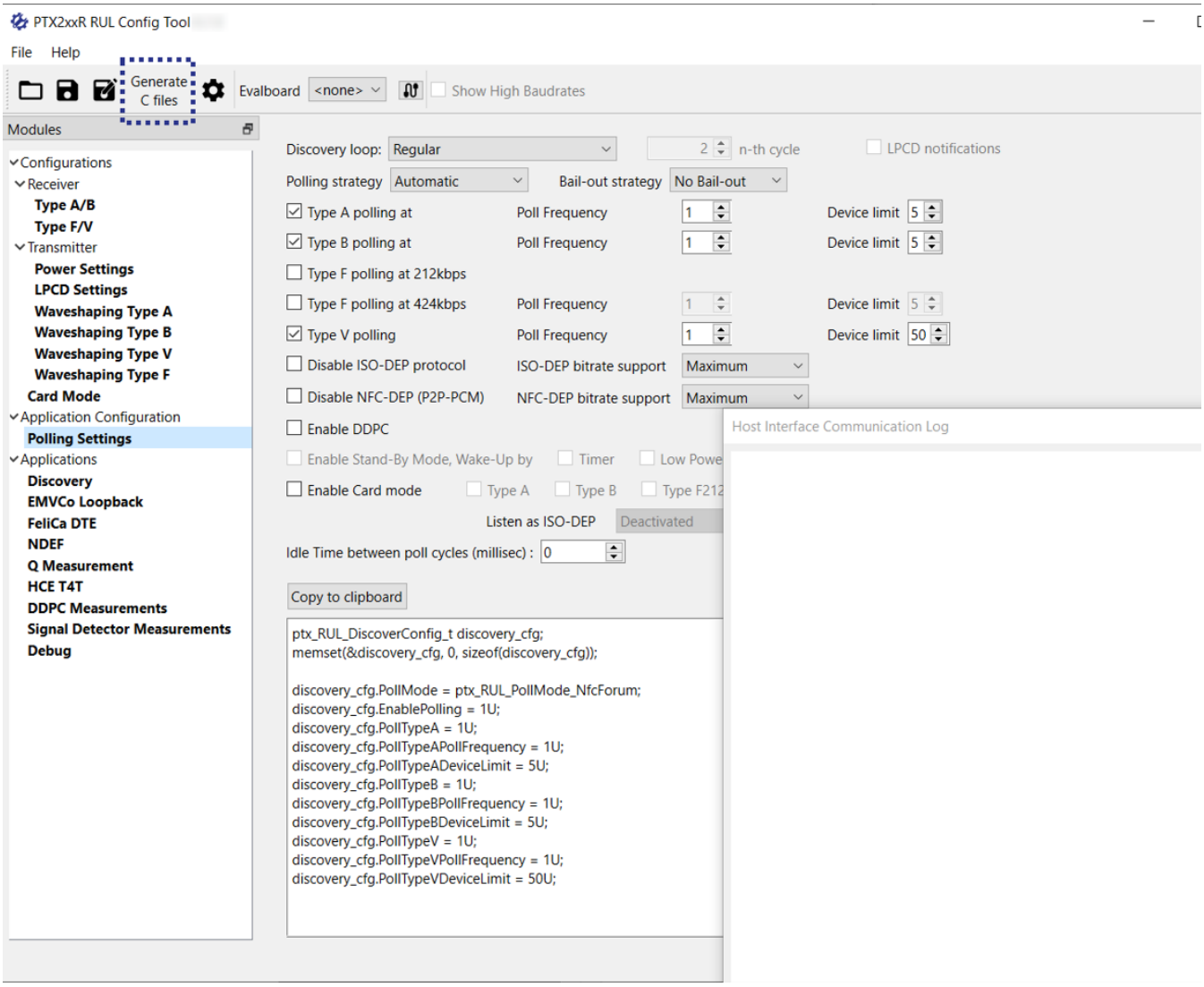

Figure 47. PTX2xxR RUL Config Tool Overview

To update RF configuration of the RUL SDK, click the toolbar button *Generate C files* and select the folder "SRC\COMPS\NSC".

Important: The RUL SDK already contains a default RF configuration (in other words, the two **.c** and **.h** files) to keep the SDK code and examples buildable. However, this default RF configuration needs to be overwritten at least once before the first integration activities of the RUL SDK, in other words, before the first execution/debugging. Without this mandatory initial step, the call to **ptx_RUL_InitHardware** returns error code **PTX_STATUS_E_NSC_INVALID_RFCONFIG**.

8.2 Dynamic at Runtime

The API function **ptx_RUL_Update_Hardware_Configuration** is used to change the RF configuration parameters at runtime. The function takes a struct-type as input parameter containing the ID, the value and the length of a given RF parameter set.

8.3 Platform Specific

If the RF parameters are stored in a platform-specific memory location (for example, EEPROM, internal, or extern Flash memory), the internal function **ptx_NSC_RfConfig_GetParamPointer** inside the source file **ptx_NSC_RUL_RfConfig.c** can be overwritten and adapted to specific target characteristics. The function is used to return a pointer to a selected set of RF parameter values and their length. The reference implementation provided with this SDK returns pointers and lengths to the constant fields of the files as described in section 5.

9. System Features

9.1 Power Management

The PTX2xxR/W and the RUL Stack supports various HW and SW mechanisms, such as:

- Stand-by mode(s) during NFC RF-Discovery and via API call
- “Low Power Card Detection (LPCD)”

These mechanisms optimize power consumption for power constraint devices where battery-life is crucial.

Important*:* The following power management HW and SW mechanisms can be combined as needed.

9.1.1. Stand-by Mode: NFC RF Discovery

Stand-by mode can be enabled between the polling cycles of the RF-Discovery procedure, in other words, when the PTX2xxR/W remains in IDLE-state or in Listen-mode (if enabled). To enable standby mode between the polling cycles, the following RF Discovery parameters need to be configured upon starting the RF Discovery procedure via API function **ptx_RUL_InitiateDiscovery**:

- **IdleTime**

  The value of this parameter indicates the time given in ms, when the PTX2xxR/W remains in IDLE state. Prior to that, the RF Discovery procedure has just finished polling for all enabled NFC technologies as shown in Figure 48.

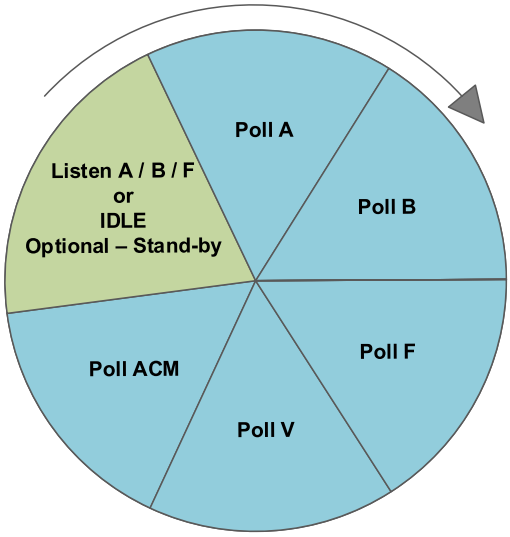

Figure 48. RF Discovery Procedure with Standby Mode

- **StandbyCfg**

  This parameter enables/disables stand-by mode during the RF Discovery procedure. The parameter itself is implemented as a C-structure and the structure members designate wake-up sources from stand-by mode. If any of the wake-up sources is set to 1, stand-by mode is enabled.

  The current implementation supports the following options to wake up from stand-by mode:

  - **Host Interface**

    Wakes up the system from stand-by mode whenever any activity on the physical host interface (I^2C, SPI, UART) gets detected.

    The system remains awake after a call to **ptx_RUL_Deactivate (IDLE)** via any host interface, which also stops the RF-Discover procedure.

    Important*:* Given that all wake-up options can be freely combined, it is strongly recommended to always enable at least the Host Interface wake-up option. Not enabling this option can lead to application-specific lock-states.

  - **(Internal) Timer**

    The internal timer is related to the **IdleTime** parameter. Once the timer expires, the system wakes up from stand-by mode and continues with the next polling cycle.

    The system remains awake when a remote device was detected or if the RF Discovery procedure was stopped (if enabled, see previous option). If no remote device was detected and no other wake up option was activated, the system returns to stand-by mode and stays in that mode for the next IdleTime milliseconds.

  - **Low Power Field Detection (LPFD)**

    If Listen mode is enabled for the RF-Discovery procedure, LPFD can be enabled to put the system into stand-by mode and to wake up again when the presence of an external RF field Is detected (in other words, a remote Poller-/Reader-device is detected).

    The system remains awake when a remote device was detected or if the RF Discovery procedure was stopped (if enabled, see previous option).

    Important*:* Figure 48 just shows an example where all NFC technologies are activated in Poll-/Reader-mode. It is up to the application to configure the RF Discovery procedure and to include/exclude different NFC technologies as needed.

- **GPIO**

  The system can be woken up by toggling the dedicated general purpose I/O pin. The system remains awake if a remote device was detected or if the RF Discovery procedure was stopped. Otherwise, the system turns to stand-by mode again.

  ***Important**:* The GPIO wake-up mechanism senses pin level transition from the default level, typically high-to-low. Exact pin connection and default level depends on the application. Please consult specific device datasheet.

9.1.1.1. Waking up via Host Interface

Any Host Interface activity wakes the system up from stand-by mode and the system turns to IDLE state as described in RUL state diagram Figure 2. This means that calling **ptx_RUL_Deactivate (DISCOVERY)** while the system is in DISCOVERY Stand-by mode would result in RUL state transition to IDLE, even though deactivation to DISCOVERY was requested. Note that calling the same function when Stand-by is not enabled would result in system staying in DISCOVERY state, as expected.

Therefore, it is strongly advised that if the system is in DISCOVERY with Stand-by mode enabled, the only Host Interface wake up is done by calling **ptx_RUL_Deactivate (IDLE)**.

9.1.2. Stand-by Mode: API Call

If required, the PTX2xxR/W can be explicitly put into stand-by mode or woken up via API call **ptx_RUL_SetPowerMode**. The input parameter **powerMode** designates whether the hardware enters stand-by mode or wakes up.

In this explicit state, the system supports two stand-by modes:

- **High-power Stand-by Mode**

  This mode is an intermediate stand-by mode where the on-chip FW tries to deactivate as many internal HW blocks before stand-by mode is entered. Depending on any potential ongoing background activities (for example, ongoing RF-communication), not all HW blocks may be switched off immediately. In this state, the FW regularly wakes up from the high-power stand-by mode and tries to deactivate the remaining HW blocks before finally entering low-power stand-by mode.

  This process is managed fully autonomously by the FW; the Application is not involved.

- **Low-power Stand-by mode**

  This mode is the targeted stand-by mode where the system fully entered stand-by mode and all corresponding HW blocks are disabled.

9.1.3. Low Power Card Detection (LPCD)

The PTX2xxR/W supports “Low Power Card Detection (LPCD)”. A regular polling cycle during the RF Discovery procedure gets replaced by a short power pulse which is sensitive to nearby changes in the RF field. If the PTX2xxR/W detects a change in the RF field, the device performs a regular polling cycle to check if there is a real Card/Tag nearby.

To enable LPCD, the following RF Discovery parameters need to be configured upon starting the RF Discovery procedure via API function **ptx_RUL_InitiateDiscovery**:

- **EnableLPCD**

  Enables/Disables LPCD and LPCD Hybrid mode as shown below:
  - 0 = LPCD disabled / regular RF-Discovery.
  - 1 = LPCD enabled.
  - 2–255 = Hybrid mode, where regular/full-power RF polling cycle is used in every n^{th} discovery period and LPCD is used in other periods.
- **EnableLPCDNtf**

  If enabled together with **EnableLPCD**, the PTX2xxR/W sends dedicated notifications to the RUL Stack whether the LPCD mechanism detected anything. If needed, the status of the detection can be retrieved via API call **ptx_RUL_GetStatusInfo**.

9.2 System Errors and Recovery

The PTX2xxR/W supports the detection of critical errors like over current and temperature. In case such an error occurs, the PTX2xxR/W shuts down automatically all relevant hardware blocks and informs the stack about the changed state. It is strongly recommended to call “**ptx_RUL_GetStatusInfo(System)**” periodically from the ready state onward!

9.2.1. Over-Temperature System Error

In case of an over temperature error, the PTX2xxR/W shuts itself down and reports the error **ptx_RUL_StatusInfoSystem_ErrorOvertemperature** to the stack. This error is reported to the user application via the **ptx_RUL_GetStatusInfo** function with the system parameter.

In case of an over temperature error, the stack tries to reinitialize the system using the **ptx_RUL_InitHardware** function until the system has cooled down enough to work again.

Important: Using the SEN-pin in this scenario is not allowed to reset the PTX2xxR/W.

9.2.2. Current-Limiter System Error

In case of an over current error, the PTX2xxR/W shuts itself down, and reports the error **ptx_RUL_StatusInfoSystem_ErrorOvercurrent** to the stack. This error is reported to the user application via the **ptx_RUL_GetStatusInfo** function with the system parameter.

In case of an over current error, the operation must stop and be shut down. The provided demo applications will provide a message for the user and exit the application gracefully.

Ensure that the **ptx_RUL_GetStatusInfo** is called periodically to detect such errors.

Important: An overcurrent error is usually related to potential issues with the target hardware, such as the PCB and the matching components/circuits.

9.2.3. Out-of-Memory System Error

In normal operation mode, the PTX2xxR/W exchanges various Control- and Data-packets with the Host using the RUL API. It is the responsibility of the Host and the corresponding user application to read pending packets from the PTX2xxR/W in time.

If the packets are not read in time, the internal packet buffers of PTX2xxR/W would overflow. Before this scenario happens, the PTX2xxR/W shuts itself down, and reports the error **ptx_RUL_StatusInfoSystem_ErrorOutOfMemory** to the stack. This error is reported to the user application via the **ptx_RUL_GetStatusInfo** function with the system parameter.

In case of an out-of-memory error, the Host and / or user application must reset the NFC system and restart.

Ensure that the **ptx_RUL_GetStatusInfo** is called periodically to detect such errors.

9.3 System Configuration

Depending on the application it might be required to adapt certain system features like the reference clock source or frequency, target frequency, over-current and over-temperature protection, etc. PTX2xxR/W SDK provides the default stable system configuration within the delivered SDK, but the users are also allowed to update system configuration parameters using either statically generated system configuration settings or to update the settings dynamically during runtime via an API call.

System configuration parameters with their default values are listed in Table 3.

Table 3. System Configuration Parameters

Parameter Identifier	Description
NSC_SystemConfig_Config_ClkSel	Clock Selection: 0 = XTAL, 1 = external clock using XTAL_IN, 2 = external clock using GPIO2. Default: 0.
NSC_SystemConfig_Config_ClkReqEn	External Clock Request Enable: 0 = disabled, 1 = enabled. Default: 0.
NSC_SystemConfig_Config_ClkReqGt	External clock request guard time, measured in microseconds, specifies the duration the PTX2K shall wait after a clock request is made before the HOST provides a stable clock signal. Default: 0.
NSC_SystemConfig_Config_RefFreq	Frequency of the system clock source. Default: 27.12MHz.
NSC_SystemConfig_Config_TargetFreq	System target Frequency. Default: 13.56MHz.
NSC_SystemConfig_Config_PrngSeed	PRNG seed value. Default: 0x44444444.
NSC_SystemConfig_Config_NAlmOffset	Offset used for PLL division word calculation, for ALM operation. Default: 0x000B237E (fits vco_offset 4720000Hz).
NSC_SystemConfig_Config_NalmMax	Maximum valid NALM value. Default: 0xFFFFFFFF.
NSC_SystemConfig_Config_NalmMin	Minimum valid NALM value. Default: 0x00000000.
NSC_SystemConfig_Config_XtalTrim	XTAL trimming value. Default: 12.
NSC_SystemConfig_Config_TempErrTresh	Over-temperature threshold, in °C. Value: 0–127 C. 0 - temperature protection disabled. Default: 85.
NSC_SystemConfig_Config_TxCurlimGain	Over-current gain selection. Value: 0 – over-current protection disabled, 1–12.5 (100mOhm resistor), 2–25 (50mOhm resistor). Default: 0.

Important: Although possible, changing the default system configuration should be done with great precaution because it might lead to unstable operation or even a complete malfunction if improper configuration is used.

9.3.1. Static System Configuration

Static system configuration can be done by directly modifying the default values of the parameters in **ptx_NSC_SystemConfig.c** file. After the parameters have been changed, stack needs to be recompiled. New configuration is loaded during system initialization by calling **ptx_RUL_InitHardware** API.

The screenshot below shows the constants defining default values for the system configuration parameters, located in **ptx_NSC_SystemConfig.c.** Apply new default values by simply changing provided constants.

```
/*
 * ###############################################################################################################
 * DEFINES / TYPES
 * ###############################################################################################################
 */
#define PTX_NSC_INIT_CLOCK_SELECT_DEFAULT          (0x00)          /* NSC INIT default clock selection (= crystal). */
#define PTX_NSC_INIT_EXT_CLOCK_GUARD_TIME          (0x00)          /* NSC INIT default external clock guard time. */
#define PTX_NSC_INIT_EXT_CLOCK_REQUEST_ENAB        (0x00)          /* NSC INIT default external clock request selection (= disabled). */
#define PTX_NSC_INIT_XTAL_TRIM_DEFAULT             (12)            /* NSC INIT default XTAL cap selection */
#define PTX_NSC_INIT_REFERENCE_FREQUENCY_DEFAULT   (27.12E6)       /* NSC INIT default crystal frequency (27.12MHz). */
#define PTX_NSC_INIT_TARGET_FREQUENCY_DEFAULT      (13.56E6)       /* NSC INIT default target frequency (13.56MHz). */
#define PTX_NSC_INIT_PRNG_SEED_DEFAULT             (0x44444444)    /* NSC INIT PRNG seed default. */
#define PTX_NSC_INIT_NALMMAX_DEFAULT               (0xFFFFFFFF)    /* NSC INIT NALM_MAX default. */
#define PTX_NSC_INIT_NALMMIN_DEFAULT               (0x00000000)    /* NSC INIT NALM_MIN default. */
#define PTX_NSC_INIT_NALMOFFSET_DEFAULT            (0x000B237E)    /* NSC INIT NHOST_ALM offset (div_offset) default (fits vco_offset 4720000Hz). *
#define PTX_NSC_INIT_TEMPERRTRESH_DEFAULT          85              /* NSC INIT Over-temperature protection default */
#define PTX_NSC_INIT_TXCURLIMGAIN_DEFAULT          0               /* NSC INIT Over-current protection disabled */
```

9.3.2. Dynamic (at Runtime) System Configuration

The API function **ptx_RUL_Update_Hardware_Configuration** is used to change the system configuration parameters at runtime. The function takes a struct-type as input parameter containing the ID, the value and the length of a given system parameter set. Not all system configuration parameters can be modified dynamically. Only the following parameters can be updated:

- ptx_RUL_SYS_CFG_ClkSel (mapped internally to NSC_SystemConfig_Config_ClkSel)
- ptx_RUL_SYS_CFG_RefFreq (mapped internally to NSC_SystemConfig_Config_RefFreq)
- ptx_RUL_SYS_CFG_TempErrTresh (mapped internally to NSC_SystemConfig_Config_TempErrTresh)
- ptx_RUL_SYS_CFG_TxCurlimGain (mapped internally to NSC_SystemConfig_Config_TxCurlimGain).

Important: After successful system configuration i.e. the API call returned success value, system turns to IDLE state without any RF configuration loaded. RF configuration should be then also loaded with the call to **ptx_RUL_Update_Hardware_Configuration**. Note that both RF and system configuration might be changed simultaneously by a single call to this API.

9.4 Tx-ESD Prevention Calibration

The Tx-ESD event prevention feature requires a one-time calibration during the design-in phase to determine the threshold (via RSSI-measurements) at which the on-chip FW shall limit Tx output power to prevent ESD clamping events.

The calibration itself can be performed via a call to **ptx_RUL_CalibrateTxEsdPreventionThreshold** as described in chapter 3.1.2.1.

After successful calibration:

- The function returns an array storing the measured (RSSI-)thresholds,
- These measured thresholds must be analyzed by the Application to identify the last TX multiplier value before a significant (RSSI) negative drop of the measured values occurs.

- A backoff of 3 LSB must be applied to the identified last TX multiplier value (before the drop) to account for component variations
- The resulting value represents **RFAmp_Max** for the antenna system,
- **RFAmp_Max** must then be stored into the RF Configuration value "**WPT_RSSI_MAX**" which gets then loaded by default during the standard RUL / System initialization.

***Attention*:** The results are applicable to all devices using the same antenna system configuration and it is therefore not required to perform per-device calibration!

10. General Integration Hints

10.1 Compliance / Certification Testing

Once the PTX2xxR/W solution (incl. the RUL SDK) gets integrated into a (final) product or a system, the product may need to undergo compliance- or certification testing before it can be released.

The necessity of such tests is strongly dependent on the target application, and which standardization schemes are applicable (for example, NFC Forum, EMVCo etc.).

The RUL SDK contains various libraries to support the test activities, such as:

- the NFC Forum DTA API,
- the POS-DTE API,
- a concrete NFC Forum WLC Poller application,
- …

The PTX2xxR/W incl. the RUL SDK support additional features which are typically outside the scope any compliance-/certification tests or standardization in general.

Such features are (for example):

- Stand-by Mode,
- Low Power Card Detection (LPCD),
- …

Enabling these features during compliance-/certification testing can negatively impact the test-results (for example, timings, unexpected RF output from a test point of view) and it is therefore **strongly recommended to disable any non-standard features** during compliance-/certification testing of a (final) product.

11. References

11.1 Standards and Regulations

- [NFC DIGITAL] - NFC Forum Digital Protocol Specification, NFC Forum
- [NFC NDEF] – NFC Data Exchange Format, NFC Forum
- [NFC T2T] – NFC Forum Type 2 Tag Specification, NFC Forum
- [NFC T3T] – NFC Forum Type 3 Tag Specification, NFC Forum
- [NFC T4T] – NFC Forum Type 4 Tag Specification, NFC Forum
- [NFC T5T] – NFC Forum Type 5 Tag Specification, NFC Forum
- [NFC WLC] – NFC Forum Wireless Charging, NFC Forum
- [NFC DI] – NFC Forum Device Information Record Type Definition, NFC Forum
- [NFC TEXT] – NFC Forum Text Record Type Definition, NFC Forum
- [NFC URI] – NFC Forum URI Record Type Definition, NFC Forum
- [NFC DTA] – NFC Forum Device Test Application, NFC Forum
- [MFCC 2017] – MIFARE Classic EV1 4K – Mainstream contactless smart card IC for fast and easy solution development
- [EMV DTE] – EMV® Contactless Terminal Level 1 Type Approval Device Test Environment
- [FELICA PROT] – FeliCa Reader/Writer Digital Protocol Requirements Specification Ver.1.22
- [FELICA PERF] – FeliCa Reader/Writer RF Performance Certification Specification Ver.1.5

12. Revision History

Revision	Date	Description
1.10	Jun 11, 2026	▪ Reworked intro section (added clarification which NFC Products are covered by term PTX2xxR/W) ▪ Replaced old product names by new names ▪ Removed obsolete HAL-methods that are out of scope for porting the RUL onto different target platforms ▪ Added “Attention” notes in section 6.5.8.
1.09	Dec 12, 2025	Added new Flags description in **ptx_TransparentMode_RFParams_t** : SOF-Only and Skip 1st Byte CRC
1.08	Oct 14, 2025	Fixed typo of SDK version number in document title to 1.2.0 from 1.1.1.
1.07	Sep 09, 2025	Added documentation of Card registry detailed information
1.06	Jul 3, 2025	Fixed typo of SDK version number in document title to 1.1.1 from 1.1.0.
1.05	Jun 20, 2025	▪ Updated sections 6.3, 6.5, 7.4, 7.5, and 7.13. ▪ Updated the description of ptx_NFCForumDTA_Init in 7.9.2.1.1. to reflect the required buffers to be initialized for reader and listen modes ▪ Numerous minor edits, updates, and formatting throughout document.
1.04	Feb 6, 2025	▪ Added **FeliCa-DTE** bullet and description to section 2.2.4. ▪ Added sub-section 6.5.8.
1.03	Jan 28, 2025	Removed function **ptx_PSL_DigMuxSelectSignal** and all related TestBus0 and TestBus1 signal vector tables.
1.02	Dec 10, 2024	▪ Updated for SDK version 1.0.0. ▪ Updated section 7.12.2.12. ▪ Added Figure 8 and Figure 9. ▪ Removed previous section 10 (“Current RUL SDK Feature Set and Known Limitations”) – moved to dedicated Release Notes document. ▪ Updated description of MFCC module (custom context / handle to be initialized now at application level). ▪ Updated description of NFC Forum DTA module (removal of automatic initialization of RUL component. Initialization must be handled at application level. Additional initialization parameters for the internal storage of NDEF messages must be provided by the application. Addition of NFC Forum device type/class selection to test configuration to enable/disable features according to required capabilities.).
1.01	Oct 1, 2024	Changed SDK version to 0.9.1 from 0.9.0.
1.00	Jul 25, 2024	Initial release.

IMPORTANT NOTICE AND DISCLAIMER

RENESAS ELECTRONICS CORPORATION AND ITS SUBSIDIARIES (“RENESAS”) PROVIDES TECHNICAL SPECIFICATIONS AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT OF THIRD-PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for developers who are designing with Renesas products. You are solely responsible for (1) selecting the appropriate products for your application, (2) designing, validating, and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. Renesas grants you permission to use these resources only to develop an application that uses Renesas products. Other reproduction or use of these resources is strictly prohibited. No license is granted to any other Renesas intellectual property or to any third-party intellectual property. Renesas disclaims responsibility for, and you will fully indemnify Renesas and its representatives against, any claims, damages, costs, losses, or liabilities arising from your use of these resources. Renesas' products are provided only subject to Renesas' Terms and Conditions of Sale or other applicable terms agreed to in writing. No use of any Renesas resources expands or otherwise alters any applicable warranties or warranty disclaimers for these products.

(Disclaimer Rev.1.01)

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-us/.

Trademarks

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