PTX1xxR NFC IoT-Reader API OS Stack Integration (SDK v7.2.0)

This document describes the PTX NSC IOT-Reader API software stack (referred to as IOTRD API) for the PTX1xxR-Reader (PTX100R, PTX105R, PTX130R) products, and the steps needed to build and integrate it into a target platform.

Contents

1. Introduction 4
1.1 Audience 4
1.2 Requirements 4
1.2.1. Building the IOTRD API Library 4
1.2.2. Running the IOTRD API Library 4
1.3 Terminology and Abbreviations 4
2. IOTRD API Software Architecture 5
2.1 Layer Description 5
2.1.1. Application Layer 5
2.1.2. Add-on APIs 6
2.1.3. Integration Layer 6
2.1.4. Core Component Layer 6
2.1.5. Hardware- and Operating System Abstraction Layer 6
3. IOTRD API Description 7
3.1 ptxIoTRd_Allocate_Stack 7
3.2 ptxIoTRd_Init_Stack 8
3.3 ptxIoTRd_Close_Stack 8
3.4 ptxIoTRd_Init_NSC 8
3.5 ptxIoTRd_Get_Revision_Info 9
3.6 ptxIoTRd_Initiate_Discovery 9
3.7 ptxIoTRd_Get_Card_Registry 9
3.8 ptxIoTRd_Activate_Card 10
3.9 ptxIoTRd_Data_Exchange 10
3.10 ptxIoTRd_Bits_Exchange_Mode 10
3.11 ptxIOTRd_Bits_Exchange 11
3.12 ptxIoTRd_RF_PresenceCheck 11
3.13 ptxIoTRd_T5T_IsolatedEoF 12
3.14 ptxIoTRd_T3T_SENSFRequest 12
3.15 ptxIoTRd_Reader_Deactivation 13
3.16 ptxIoTRd_Update_ChipConfig 13
3.17 ptxIoTRd_Set_Power_Mode 13
3.18 ptxIoTRd_Enable_RT 14
3.19 ptxIoTRd_Get_System_Info 14
3.20 ptxIoTRd_SWReset 14
3.21 ptxIoTRd_TempSensor_Calibration 15

© 2020-2024 Renesas Electronics

3.22 ptxIoTRd_Get_Status_Info 15
3.23 ptxIoTRd_ConfigHBR 15
3.24 ptxIoTRd_Set_RSSI_Mode 16
3.25 ptxIoTRd_Get_RSSI_Value 16

4. IOTRD API States 17

5. HCE API Description 22
5.1 ptxHCE_Init 22
5.2 ptxHCE_Deinit 22
5.3 ptxHCE_GetEvent 22
5.4 ptxHCE_SendData 23

6. HCE API States 24

7. IOTRD API SDK Deliverable 25

8. IOTRD API Target System Integration 27
8.1 Introduction 27
8.2 Build System 27
8.3 Integration Flow–IOTRD API => Stand-alone 29
8.4 Integration Flow–IOTRD API => Component 30
8.5 Target Platform Abstraction Layers 31
8.5.1. Hardware Abstraction Layer (HAL) 31
8.5.2. Operating System Abstraction Layer (OSAL) 34
8.6 Integration Notes/Hints 34
8.6.1. Performance Optimization 34
8.6.2. Artificial Delays/Sleep-Operations 34
8.6.3. Logging System Types 35
8.6.4. Maximum Number of Supported Cards 35
8.6.5. Reference Implementation–Code Size and Memory Consumption 36

9. RF and System Configuration Updates 37
9.1 Default RF Configuration 37
9.2 Default System Configuration 38
9.2.1. Temperature Sensor Calibration 38
9.3 Dynamic RF and System Configuration 38

10. Add-on Libraries/APIs 39
10.1 Native-Tag API 39
10.1.1. Supported Type 2 Tag Commands 39
10.1.2. Supported Type 3 Tag Commands 40
10.1.3. Supported Type 4 Tag Commands 40
10.1.4. Supported Type 5 Tag Commands 40
10.2 NDEF API 40
10.3 GPIO API 41
10.4 RF-Test API 42
10.5 FeliCa-DTE API 42
10.6 Transparent-Mode API 43
10.7 Transparent Data Channel (TDC) API 46

11. References 47
11.1 General..47
11.2 Standards and Regulations 47
12. Revision History 47

Figures

Figure 1. IOTRD API and PTX NSC Software Stack Architecture 5
Figure 2. IOTRD API Flow Example..17
Figure 3. State TEST..18
Figure 4. HCE API Flow Example...24
Figure 5. IOTRD API SDK Folder Structure 25
Figure 6. IOTRD API Integration Flow: Stand-alone System...29
Figure 7. IOTRD API Integration Flow: (sub-)Component within Existing Application 30
Figure 8. HAL and OSAL Architecture...31
Figure 9. PTX1xxR RRA-Operation using I2C (Simplified)..33
Figure 11. PTX1xxR IOT Config Tool..37
Figure 12. Native Tag API Overview 39
Figure 13. NDEF API Overview..41

1. Introduction

This document describes the PTX NSC IOT-Reader API software stack (referred to as IOTRD API) for the PTX1xxR-Reader products, and the steps needed to build and integrate it into a target platform.

The IOTRD API represents a set of library functions that allow users to implement generic reader applications according to the NFC-Forum.

In addition to the IOTRD API, the delivery also contains a demo implementation that shows the correct usage of the API and some typical examples like starting the RF discovery, RF data exchanges, handle multiple cards, etc.

1.1 Audience

This document is intended to be used by:

- Software architects
- Software engineers
- Software integrators

1.2 Requirements

1.2.1. Building the IOTRD API Library

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

- CMake 3.15 or higher (see cmake.org)
- Any C-compiler (depending on target platform) supporting C99 standard
  - Tested with gcc (Raspbian 8.3.0-6+rpi1) 8.3.0

1.2.2. Running the IOTRD API Library

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

- A general-purpose or real-time Operating System like Windows, (embedded) Linux, FreeRTOS, etc.
- Permission to access the file system of the Operating System to:
  - Read configuration file for the PTX1xxR
  - Read/write general configuration parameters from/to files
- Driver access to physical host interfaces

1.3 Terminology and Abbreviations

Terminology and Abbreviations	Description
HW	Hardware
Integrator	Developer who builds and/or integrates the IOTRD API into a target application
NDEF	NFC Data Exchange Format
NFC	Near Field Communication
NSC	NFC Soft Controller
RF	Radio Frequency
RTOS	Real-time Operating System

Terminology and Abbreviations	Description
SDK	Software Development Kit
SW	Software

2. IOTRD API Software Architecture

The IOTRD API is based on the PTX NSC Software Stack (referred to as NSC Stack) which consists of multiple layers and components.

The IOTRD API and the NSC Stack are implemented in ANSI C and are therefore independent of the underlaying target platform (see the following figure):

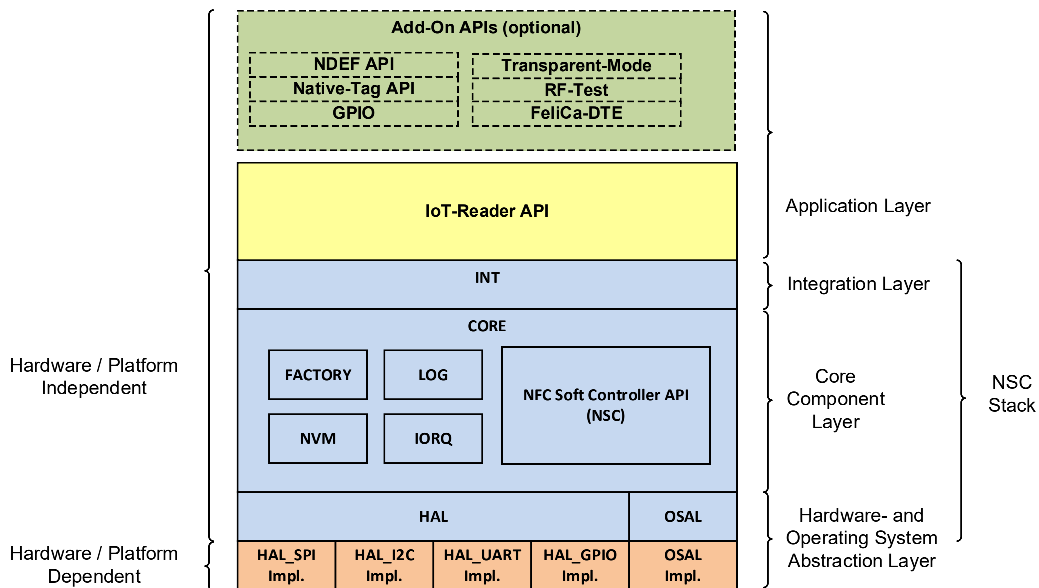

Figure 1. IOTRD API and PTX NSC Software Stack Architecture

2.1 Layer Description

2.1.1. Application Layer

This layer implements the actual IOTRD API which is the main interface to the target application. The IOTRD API provides a set of functions to:

- Initialize the IOTRD API and NSC Stack
- Initialize the PTX1xxR
- Discover cards according to NFC Forum incl. support for “Low-Power-Card-Detection”
- Select a specific card in case multiple cards and/or protocols were discovered
- Retrieve card details like technical and/or activation parameters
- Exchange RF-data and -bitstreams
- Stop RF communication
- Optionally update RF- and System-configuration parameters at runtime
- Optionally put PTX1xxR into and wake up from stand-by mode
- Perform Host Card Emulation using the HCE API (see section 5)
- Control various GPIO pins of the PTX1xxR (input, output, can be used any state from READY onwards)

- Perform FeliCa performance and Digital Protocol Requirement tests
- Enable various RF test-sequences (for example, PRBS9, PRBS15, ...)
- Exchange low-level RF-commands (Transparent-Mode), etc.
- Perform arbitrary NFC data exchange with a Renesas NFC Forum WLC Listener device (for example, PTX30W) using the “Transparent Data Channel”-API

The IOTRD API itself is described in section 3, and an example flow is discussed in section 4.

2.1.2. Add-on APIs

This layer contains various optional add-on APIs that can be used on top of the IOTRD API such as:

- To access native commands of a Tag
- NDEF-Tag operations
- GPIO-Operation
- FeliCa-DTE tests
- Generic RF tests
- Transparent-Mode operations
- Transparent Data Channel operations

See section 10 for descriptions of these APIs.

2.1.3. Integration Layer

This layer implements a collection of all APIs of the NSC Stack including its sub-components.

2.1.4. Core Component Layer

This layer represents the actual core of the NSC Stack and provides the following functionalities:

- PTX1xxR configuration and initialization
- RF-configuration
- RF-communication including call-back functions for asynchronous events and error handling
- Simplified NSC Stack initialization and parametrization via Factory sub-component
- Extensive Logging-capabilities to ease system integration and debug support
- Internal hardware access dispatcher (“IORQ”)
- Abstracted file access (“NVM”)

2.1.5. Hardware- and Operating System Abstraction Layer

This layer implements the following Software interfaces:

2.1.5.1. Hardware Abstraction Layer

This component is dependent on the used physical hardware interface (SPI, I^2C, or UART) between the application processor of the target platform and the PTX1xxR.

The Software interface requires function implementations to:

- Open/close an HARDWARE interface
- Configure an HARDWARE interface (for example, speed, timeouts)
- Exchange data via the hardware interface
- Cancel operations/access

Important: The delivered SDK contains a reference implementation for SPI, I^2C, and UART for the Linux OS. If an implementation for a different target platform is required, the reference implementation must be adapted (see section 8) or see the Renesas website to check for available reference implementations for other target platforms.

2.1.5.2. Operating System Abstraction Layer

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

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

Important: The delivered SDK contains a reference implementation the Linux OS. If an implementation for a different target platform is required, the reference implementation needs to be adapted (see section 8) or see the Renesas website to check for available reference implementations for other target platforms.

3. IOTRD API Description

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

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

3.1 ptxIoTRd_Allocate_Stack

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

3.2 ptxIoTRd_Init_Stack

Declaration	`short ptxIoTRd_Init_Stack ( void *stackComp, uint8_t interfaceType, char *deviceName, uint32_t deviceSpeed, uint32_t gpioNum char *fsPath);`	
Description	Initializes the IOTRD API stack component with given parameters	
Input Parameters	stackComp	Pointer to stack component
	interfaceType	Host-interface selection (UART, SPI, I^2C)
	deviceName	System name of host-interface (for example, \COM3)
	deviceSpeed	Host-interface speed
	gpioNum	Number of GPIO-pin used for SPI or I^2C
	fsPath	Path location of file "NSC_RF_CONFIG.dat"
Return Value	Status of operation	-

3.3 ptxIoTRd_Close_Stack

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

3.4 ptxIoTRd_Init_NSC

Declaration	`short ptxIoTRd_Init_NSC (void *stackComp, ptxIoTRd_NSCInitConfig_t *initConfig);`	
Description	Initializes the PTX1xxR	
Input Parameters	stackComp	Pointer to stack component
	initConfig	NSC initialization parameters
Return Value	Status of operation	-

3.5 ptxIoTRd_Get_Revision_Info

Declaration	`short ptxIoTRd_Get_Revision_Info (void *stackComp, ptxIoTRd_RevisionType_t *revisionType, uint32_t *revisionInfo);`	
Description	Reads various revisions of system like SW-version, uCode-version, hardware-revision, Product-ID etc. *Note*: HW-/Chip-ID and Product-ID can only be read after successful call to ptxIoTRd_Init_NSC. Product ID: 0x00 -> PTX100x 0x01 -> PTX105x 0x02 -> PTX130x 0xFF -> Unknown/Invalid Product-ID	
Input Parameters	stackComp	Pointer to stack component
	revisionType	Revision type
	revisionInfo	Pointer to variable holding revision information
Return Value	Status of operation	-

3.6 ptxIoTRd_Initiate_Discovery

Declaration	`short ptxIoTRd_Initiate_Discovery (void *stackComp, IoTRd_DiscConfig_t *discConfig);`	
Description	Starts the NFC Forum RF discovery	
Input Parameters	stackComp	Pointer to stack component
	discConfig	Pointer to RF-discover structure
Return Value	Status of operation	-

3.7 ptxIoTRd_Get_Card_Registry

Declaration	`short ptxIoTRd_Get_Card_Registry (void *stackComp, IoTRd_CardRegistry_t **cardRegistry);`	
Description	Returns internal card registry to get card information	
Input Parameters	stackComp	Pointer to stack component
	cardRegistry	Pointer to pointer to keep reference to card registry
Return Value	Status of operation	-

3.8 ptxIoTRd_Activate_Card

Declaration	`short ptxIoTRd_Activate_Card (void *stackComp, IoTRd_CardParams_t *cardParams, IotRd_CardProtocl_t protocol);`	
Description	Selects / Activates a given card in case of multiple available cards	
Input Parameters	stackComp	Pointer to stack component
	cardParams	Pointer to card (within registry) to select / activate
	protocol	RF-protocol to activate
Return Value	Status of operation	-

3.9 ptxIoTRd_Data_Exchange

Declaration	`short ptxIoTRd_Data_Exchange (void *stackComp, uint8_t *tx, uint32_t txLength, uint8_t *rx, uint32_t *rxLength, uint32_t msAppTimeout);`	
Description	Protocol-based or raw data exchange	
Input Parameters	stackComp	Pointer to stack component
	tx	Pointer to buffer holding data to send
	txLength	Length of data to send
	rx	Pointer to buffer holding received data
	rxLength	Size of buffer holding received data / length of received data
	msAppTimeout	Application timeout
Return Value	Status of operation	-

3.10 ptxIoTRd_Bits_Exchange_Mode

Declaration	`short ptxIoTRd_Bits_Exchange_Mode (void *stackComp, uint8_t enable);`	
Description	Enables/Disables the bit-exchange mode required to call ptxIoTRd_Bits_Exchange.	
Input Parameters	stackComp	Pointer to stack component
	enable	Enable/Disable flag

3.11 ptxIOTRd_Bits_Exchange

<table>
<tr><td>Declaration</td><td colspan="2"><pre>short ptxIoTRd_Bits_Exchange
              (void *stackComp,
               uint8_t *tx,
               uint8_t *txPar,
               uint32_t txLength,
               uint8_t *rx,
               uint8_t *rxPar,
               uint32_t *rxLength,
               uint32_t *numTotBits,
               uint32_t msAppTimeout);</pre></td></tr>
<tr><td>Description</td><td colspan="2">Exchanges a bitstream based on NFC-A technology</td></tr>
<tr><td rowspan="9">Input Parameters</td><td>stackComp</td><td>Pointer to stack component</td></tr>
<tr><td>tx</td><td>Pointer to buffer holding data bytes to send</td></tr>
<tr><td>txPar</td><td>Pointer to buffer holding parity bits to send</td></tr>
<tr><td>txLength</td><td>Length of tx- and txPar-buffers</td></tr>
<tr><td>rx</td><td>Pointer to buffer holding received bytes</td></tr>
<tr><td>rxPar</td><td>Pointer to buffer holding received parity bits</td></tr>
<tr><td>rxLength</td><td>Length of received bytes / parity bits</td></tr>
<tr><td>numTotBits</td><td>Total number of received bits</td></tr>
<tr><td>msAppTimeout</td><td>Application timeout</td></tr>
<tr><td>Return Value</td><td>Status of operation</td><td>-</td></tr>
</table>

3.12 ptxIoTRd_RF_PresenceCheck

<table>
<tr><td>Declaration</td><td colspan="2"><pre>short ptxIoTRd_RF_PresenceCheck
    (void *stackComp,
     ptxIoTRd_CheckPresType_t presCheckType);</pre></td></tr>
<tr><td>Description</td><td colspan="2">Executes a presence check method on ISO-DEP cards or NFC-DEP targets</td></tr>
<tr><td rowspan="2">Input Parameters</td><td>stackComp</td><td>Pointer to stack component</td></tr>
<tr><td>presCheckType</td><td>Presence-check method type.</td></tr>
<tr><td>Return Value</td><td>Status of operation</td><td>-</td></tr>
</table>

3.13 ptxIoTRd_T5T_IsolatedEoF

Declaration	`short ptxIoTRd_T5T_IsolatedEoF (void *stackComp, uint8_t *rx, uint32_t *rxLength, uint32_t msAppTimeout);`	
Description	Sends an EoF-packet according to T5T protocol	
Input Parameters	stackComp	Pointer to stack component
	rx	Pointer to buffer holding received bytes
	rxLength	Size of buffer holding received data / length of received data
	msAppTimeout	Application timeout
Return Value	Status of operation	-

3.14 ptxIoTRd_T3T_SENSFRequest

Declaration	`short ptxIoTRd_T3T_SENSF_Request( void *stackComp, uint16_t systemCode, uint8_t requestCode, uint8_t tsn, uint8_t *rx, uint32_t *rxLength, uint32_t msAppTimeout);`	
Description	Sends a SENSF_REQ-packet according to T3T protocol.	
Input Parameters	stackComp	Pointer to stack component
	systemCode	T3T System-code
	requestCode	T3T Request-code
	tsn	T3T Number of timeslot(s)
	rx	Pointer to buffer holding received bytes
	rxLength	Size of buffer holding received data / length of received data
	msAppTimeout	Application timeout
Return Value	Status of operation	-

3.15 ptxIoTRd_Reader_Deactivation

Declaration	`short ptxIoTRd_Reader_Deactivation (void *stackComp, uint8_t deactivationType);`	
Description	Stops any finished RF-communication and deactivates the reader and/or remove device.	
Input Parameters	stackComp	Pointer to stack component
	deactivationType	Type of deactivation (IDLE, DISCOVERY, Sleep)
Return Value	Status of operation	-

3.16 ptxIoTRd_Update_ChipConfig

Declaration	`short ptxIoTRd_Update_ChipConfig (void *stackComp, uint8_t nrConfigs, ptxIoTRd_ChipConfig_t *configParams);`	
Description	Updates RF- and System-parameters at runtime.	
Input Parameters	stackComp	Pointer to stack component
	nrConfigs	Number of RF-/System-configurations to set
	configParams	Pointer to n-configuration parameters sets
Return Value	Status of operation	-

3.17 ptxIoTRd_Set_Power_Mode

Declaration	`short ptxIoTRd_Set_Power_Mode (void *stackComp, uint8_t newPowerMode);`	
Description	Puts chip into stand-by or wakes it up from stand-by	
Input Parameters	stackComp	Pointer to stack component
	newPowerMode	Type of stand-by operation
Return Value	Status of operation	-

3.18 ptxIoTRd_Enable_RT

Declaration	`short ptxIoTRd_Enable_RT (void *stackComp, uint8_t *mode, const char *logFile);`	
Description	Enables / Disables immediate writing to a given log file	
Input Parameters	stackComp	Pointer to stack component
	mode	Enable or disable writing
	logFile	Filename where log entries should be written to
Return Value	Status of operation	-

3.19 ptxIoTRd_Get_System_Info

Declaration	`ptxStatus_t ptxIoTRd_Get_System_Info (void *stackComp, ptxIoTRd_SysInfoType_t infoType, uint8_t *infoBuffer, uint8_t *infoBufferLength);`	
Description	Optional command to retrieve system relevant information like VDPA-calibration result.	
Input Parameters	stackComp	Pointer to stack component
	infoType	Information identifier
	infoBuffer	Buffer to store information
	infoBufferLength	Length of information
Return Value	Status of operation	-

3.20 ptxIoTRd_SWReset

Declaration	`short ptxIoTRd_SWReset (void *stackComp);`	
Description	Performs a soft-reset of the PTX1xxR	
Input Parameters	stackComp	Pointer to stack component
Return Value	Status of operation	-

3.21 ptxIoTRd_TempSensor_Calibration

Declaration	`short ptxIotRd_TempSensor_Calibration (void *stackComp, uint8_t Tambient, uint8_t *Tshutdown);`	
Description	Performs offset calibration of PTX1xxR temperature sensor. Calculates compensated temperature shutdown threshold.	
Input Parameters	stackComp	Pointer to stack component
	Tambient	Ambient temperature at which calibration takes place
	Tshutdown	Pointer to temperature shutdown value
Return Value	Status of operation	-

3.22 ptxIoTRd_Get_Status_Info

Declaration	`short ptxIoTRd_Status_Info (void *stackComp, ptxIoTRd_StatusType_t statusType, uint8_t *statusInfo);`	
Description	Retrieves current operating state of chip	
Input Parameters	stackComp	Pointer to stack component
	statusType	Status type identifier
	systemState	Pointer to variable holding system state
Return Value	Status of operation	-

3.23 ptxIoTRd_ConfigHBR

Declaration	`short ptxIoTRd_ConfigHBR (void *stackComp, ptxIoTRd_HBRConfig_t *configParams);`	
Description	Retrieves current operating state of chip	
Input Parameters	stackComp	Pointer to stack component
	configParams	Pointer to configurations
Return Value	Status of operation	-

3.24 ptxIoTRd_Set_RSSI_Mode

Declaration	`short ptxIoTRd_Set_RSSI_Mode (void *stackComp, ptxIoTRd_RSSI_Mode_t rssiMode, uint8_t *rssiRefreshPeriodInt);`	
Description	Enables or Disables the RSSI-Mode in State "Ready"	
Input Parameters	stackComp	Pointer to stack component
	rssiMode	Enables / Disables RSSI-Mode
	rssiRefreshPeriodInt	RSSI-Calculation Refresh-Integer. Follows the equation "RSSI Refresh Rate in ms = 2 ^ (rssiRefreshPeriodInt - 1)". If set to null or a value higher than 16, 1ms will be set as default.
Return Value	Status of operation	-

3.25 ptxIoTRd_Get_RSSI_Value

Declaration	`short ptxIoTRd_Get_RSSI_Value (void *stackComp, uint16_t *rssiValue);`	
Description	Reads the current RSSI-value in State "Ready"	
Input Parameters	stackComp	Pointer to stack component
	rssiValue	Current RSSI value
Return Value	Status of operation	-

4. IOTRD API States

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

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

Figure 2. IOTRD API Flow Example

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

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

Details of the status word definition can be found in the API-documentation (see chapter 6) or directly in the source file “**\SRC\COMPS\ptx_Status.h**”.

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

State Description:

- **State: START**
  The "Stack Component" object needs to be allocated first via a call to "**ptxIoTRd_Allocate_Stack**".
- **State: RESET**
  Initialize the IOTRD API and the internal NSC components via a call to "**ptxIoTRd_Init_Stack**".
  This call takes parameters to initialize the intended host-interface including characteristics and the file path to the mandatory RF-configuration files "**NSC_RF_CONFIG.dat**" and "**NSC_SYS_CONFIG.dat**".
- **State: INITIALIZED**
  Once the IOTRD API and the NSC components themselves have been initialized, the PTX1xxR needs to be initialized and configured (handled automatically) via a call to "**ptxIoTRd_Init_NSC**".

- **State: READY**
  After IC initialization, the IOTRD API is ready to poll for cards in the field via a call to "**ptxIoTRd_Initate_Discovery**".
  The behavior of the discovery-loop is configurable where a user can change for example:

- poll for Type-A
- poll for Type-B
- poll for Type-F (212 kBit/s or 424 kBit/s)
- poll for Type-V
- use Low-Power card-detection ("LPCD"), etc.

Note: The call to "**ptxIoTRd_Initate_Discovery**" is non-blocking (i.e. it returns immediately to the caller) while the actual polling operation is handled in the background.

Important: If enabled in the System-configuration parameters, the PTX1xxR supports detection of critical errors like over current and temperature. In cases where an error occurs, the PTX1xxR shuts down automatically all relevant hardware-blocks and informs the stack about the changed state. This state can be read via a call to "**ptxIoTRd_Get_Status_Info (System)**".

Important: It is recommended to call "**ptxIoTRd_Get_Status_Info (System)**" periodically from state READY onwards.

In addition, the following optional functions can be executed *exclusively* in this state:

- **ptxIoTRd_Update_ChipConfig**
- **ptxIoTRd_Set_Power_Mode**
- **ptxIoTRd_Get_Revision_Info**
- **ptxIoTRd_*_RSSI_***
- **ptxFeliCa_DTE*** (except API/component (un)initialization
- **ptxRF_Test*** (except API/component (un)initialization
- **ptxTransparentMode *** (except API/component (un)initialization

Note: A call to "**ptxIoTRd_Update_ChipConfig**" requires a following call to "**ptxIoTRd_SWReset**" and "**ptxIoTRd_Init_NSC**" to apply the changed configuration.

The IOTRD API also supports higher bitrates for Type-A and Type-B discovery. The default configuration only supports 106kbit/s. Higher bitrates can be configured by a call to "**ptxIoTRd_ConfigHBR**". The higher bitrate is only used if the card supports it. If the card does not support the configured higher bit rates, the reader uses the default speed of 106kbit/s.

- **State: TEST (Optional)**

This optional state allows the application to perform various system and/or RF-tests using the add-on APIs described in this document as well as low-level RF-exchanges using the Transparent-Mode API. The transition from and back to state READY is shown in the following figure.

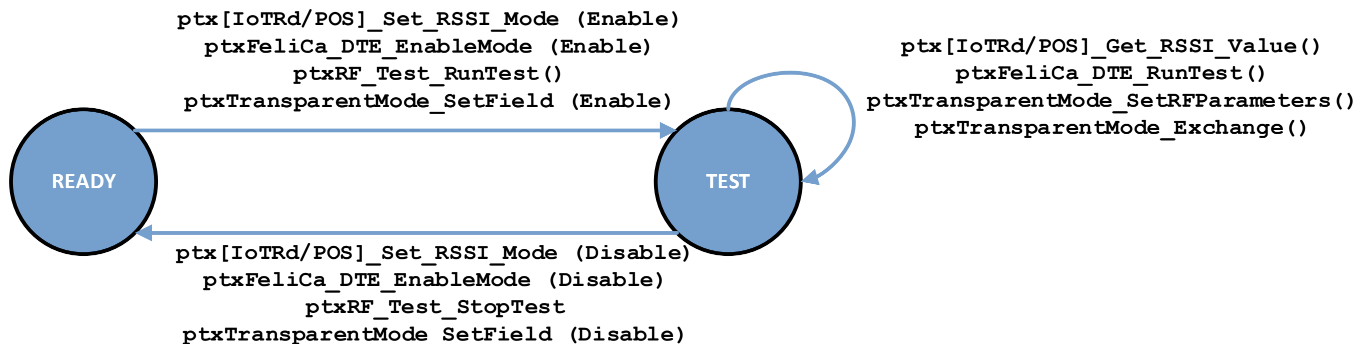

Figure 3. State TEST

- **State: POLLING**

The status of the polling operation can be retrieved via a call to "**ptxIoTRd_Get_Status_Info (Discover)**".

This function returns whether:

  - A single card was discovered and activated.
    A call to function "**ptxIoTRd_Get_Card_Registry**" should be used to retrieve details on the discovered card (for example, serial number, information on used RF-protocol, etc.) to determine for example, if a card supports the ISO-DEP protocol (see ISO 14443-4, T = CL).
    multiple cards were discovered, and the RF discovery is still ongoing.
    This state is for information purposes only.
  - Multiple cards were discovered, and the RF discovery has finished.
    A call to "**ptxIoTRd_Get_Card_Registry**" should be used to retrieve the internal card registry to get the number of discovered cards including detailed information.
  - Nothing was discovered.
  - The low power card detection (LPCD) triggered.

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

The IoT-Reader also supports higher bitrates for Type-A and Type-B polling. The default configuration only supports 106kbit/s. Higher bitrates can be configured by a call to “**ptxIoTRd_ConfigHBR**”. The higher bitrate is only used if the card supports it. If the card does not support the configured higher bit rates, the reader uses the default speed of 106kbit/s.

- **State: HOST CARD EMULATION**

This state is reached if an external field is discovered. Once entered this state, the application communicates with the external reader device and only leaves this state if the external field is turned off again. All data and notifications are written into a message queue and can then be processed by the HCE API. The application uses "ptxHCE_GetEvent" to receive the oldest event in the queue and processes it according to the event type. Possible event types can be

  - External Field On
  - External Field Off
  - Activated Listen for Type A
  - Data Exchange

Note: Once the external field is turned off, the application returns to the POLLING state

- **State: WAIT FOR SELECTION**

This state is reached if multiple cards were previously discovered or if a specific card was deactivated. The application can use the function "**ptxIoTRd_Get_Card_Registry**" to gather information about all available cards.

To select a specific card and/or to activate a specific RF-protocol, a call to "**ptxIoTRd_Activate_Card**" is required before an actual RF data exchange can be executed.

- **State: DATA EXCHANGE**

In this state RF data exchanges can be exchanged with the active card via:

  - Calls to "**ptxIoTRd_Data_Exchange**"
  - Calls to “**ptxIoTRd_Bits_Exchange**” (only if T2T protocol is active)
  - Calls to the functions from the **NativeTag**- and **NDEF-API**.

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

The RF-protocols ISO- and NFC-DEP are handled internally by the PTX1xxR; all other (NFC-Forum) protocols like T2T, T3T, and T5T must be handled by the application and/or the add-on APIs on top.

The API contains additionally the following functions that can be used in this state:

- **ptxIoTRd_RF_PresenceCheck**
  This is an optional function to perform a presence during an active data-exchange session with ISO-DEP cards or NFC-DEP targets.
- **ptxIoTRd_T5T_IsolatedEoF**
  This is an optional function which may be used to send an isolated EoF-packet to a T5T card which may be required for certain commands.
- **ptxIoTRd_T3T_SENSFRequest**
  This is an optional function which allows to send a T3T-SENSF_REQ with given parameters in the current state. The data contained in the provided output buffer contains the concatenated responses of each detected card with a prepended length.

Note: If the function "**ptxIoTRd_Data_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 how to treat 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 delivered demo application provides examples of how the different RF-technologies and -protocols are handled for single and multiple available cards.

The following output was taken from the demo application console output and shows example data exchanges for single cards / protocols.

Example Data Exchange for T3T / FeliCa => Read Block 0

```
Card activated ... OK!
01. RF-Technology = Type-F; SENSF_RES: 1201012E3D23BA0BA14100F1000000014300; Protocol....: T3T
=========== DATA EXCHANGE ================
TX = 06012E3D23BA0BA14100F1010900018000
RX = 07012E3D23BA0BA141FFA100
=========================================
```

Example Data Exchange for T2T / Mifare => Read Block 0

```
Card activated ... OK!
01. RF-Technology = Type-A; SENS_RES: 4400; NFCID1_LEN: 07; NFCID1: 0489FF02E53F80; SEL_RES: 00;
Protocol: T2T
=========== DATA EXCHANGE ================
TX = 3000
RX = 0489FFFA02E53F8058480000E110120000
=========================================
```

Example Data Exchange for T4T / ISO-DEP.B => Select PPSE APDU

```
Card activated ... OK!
01. RF-Technology = Type-B; SENSB_RES: 5057ABD7DC0000000080817100; Protocol....: ISO-DEP;
ATTRIB_RES: 00
=========== DATA EXCHANGE ================
TX = 00A404000E325041592E5359532E444446303100
RX = 00B20104009000
=========================================
```

Example Data Exchange for T5T / ISO 15693 => Read Block 0

```
Card activated ... OK!
01. RF-Technology = Type-V; DSFID: 00; RES_FLAGS: 00; UID: E0 04 01 50 96 13 9B 04 ; Protocol:
T5T
=========== DATA EXCHANGE ================
TX = 2220049B1396500104E000
RX = 000000000000
==========================================
```

In the case where multiple cards are present, the output looks like the following example where three ISO 15693 (T5T, Type-V) cards are present.

```
Multiple Card(s) detected - resolved ... OK!
01. RF-Technology = Type-V; DSFID: 00; RES_FLAGS: 00; UID: E0 04 01 50 96 13 3F 72
02. RF-Technology = Type-V; DSFID: 00; RES_FLAGS: 00; UID: E0 04 01 50 96 13 9B 04
03. RF-Technology = Type-V; DSFID: 00; RES_FLAGS: 00; UID: E0 04 01 08 0A 1B 25 D6
Selecting first detected card/protocol (RF-Protocol == T5T)...  ... OK!
01. RF-Technology = Type-V; DSFID: 00; RES_FLAGS: 00; UID: E0 04 01 50 96 13 3F 72 ; Protocol:
T5T
=========== DATA EXCHANGE ================
TX = 2220723F1396500104E000
RX = 000000000000
==========================================
```

The demo application selects the first card found in the field which is also stored first in the internal card registry.

A special case is RF-technology Type-A where the card response parameter SEL_RES (also referred to as SAK-byte in ISO 14443-3) may indicate support for the RF-protocols ISO-DEP and NFC-DEP. In this case, the single remote NFC-peer device is treated internally as two devices. Similar to cards, a specific protocol can be selected via a call to “**ptxIoTRd_Activate_Card**” in state “**WAIT FOR SELECTION**”.

The output of the demo application for this scenario looks as follows.

```
Multiple Card(s) detected - resolved ... OK!
01. RF-Technology = Type-A; SENS_RES: 0803; NFCID1_LEN: 04; NFCID1: 01020304; SEL_RES: 60
Selecting first detected card/protocol (RF-Protocol == NFC_DEP)...  ... OK!
01. RF-Technology = Type-A; SENS_RES: 0803; NFCID1_LEN: 04; NFCID1: 01020304; SEL_RES: 60;
Protocol: NFC-DEP; ATR_RES:
26D50101FE83DC567B35DF0000000000073246666D010112020207FF03020013040164070103
=========== DATA EXCHANGE ================
TX = 0000
RX = 0000
==========================================
```

Note: The default delivery of the IoT API activates the LLCP-protocol on top of NFC-DEP.

Similar to the "POLLING" state, completed RF-exchanges can be stopped via a call to "**ptxIoTRd_Reader_Deactivation**".

- **State: FINISHED**

  Once the complete application shall be stopped or shut down, it is required to call function "**ptxIoTRd_Close_Stack**" to free previously allocated system resources like memory, drivers, etc.

5. HCE API Description

This section provides an overview of the functions supported by the HCE API from the \COMPS folder.

Note: The HCE API consists of additional functions that are not listed here. These functions are used internally and are not intended to be used at application level.

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

5.1 ptxHCE_Init

Declaration	`ptxStatus_t ptxHCE_Init (void *stackComp, ptxHCE_t *hceCtx);`	
Description	Host Card Emulation Component initialization. Already handled during IoT API main component initialization.	
Input Parameters	stackComp	Pointer to component
	hceCtx	Pointer to initialization parameters.
Return Value	Status of operation	-

5.2 ptxHCE_Deinit

Declaration	`ptxStatus_t ptxHCE_Deinit (ptxHCE_t *hceCtx);`	
Description	Host Card Emulation Component deinitialization. Already handled during IoT API main component deinitialization.	
Input Parameters	hceCtx	Pointer to an allocated instance of the HCE stack controller record.
Return Value	Status of operation	-

5.3 ptxHCE_GetEvent

Declaration	`ptxStatus_t ptxHCE_GetEvent (ptxHCE_t *hceCtx, ptxHCE_EventRecord_t **event );`	
Description	This function allows the user to request latest event notification data received from the PTX card emulation device.	
Input Parameters	hceCtx	Pointer to an allocated instance of the HCE stack controller record.
	event	Reference to an event record supplied by the AP into which event details can be entered
Return Value	Status of operation	-

5.4 ptxHCE_SendData

Declaration	`uint16_t ptxHCE_SendData (void *stackComp, uint8_t *tx, uint32_t txLength, uint32_t msAppTimeout);`	
Description	This function sends Data to the main component in the stack.	
Input Parameters	stackComp	Pointer to an initialized instance of the main component in the stack.
	tx	Buffer containing the data to transmit.
	txLength	Length of "tx".
	msAppTimeout	Application-timeout in ms that the function is going to wait for receiving data from the card.
Return Value	Status of operation	-

6. HCE API States

Figure 4 shows an example flow of how the HCE API should be used.

The internal Host Card Emulation (HCE) component is initialized together with the IOTRD API and ready to use when the IOTRD API is. The HCE component will be active if in the polling configuration the ListenTypeA parameter is set to 1. This parameter enables the listen mode and allows the API to receive listen events. The HCE component uses a message queue, to queue received events and processes them in “**ptxIoTRdInt_DemoState_HostCardEmulation**”.

Note: An application can also directly use the functions of the HCE component (usage of the DemoState-extension is not mandatory).

The following section describes the states used in “**ptxIoTRdInt_DemoState_HostCardEmulation**”. The states START, RESET, INIT, and READY are the same as in IOTRD API State Machine. All transitions between the states are done via “**ptxHCE_GetEvent**”.

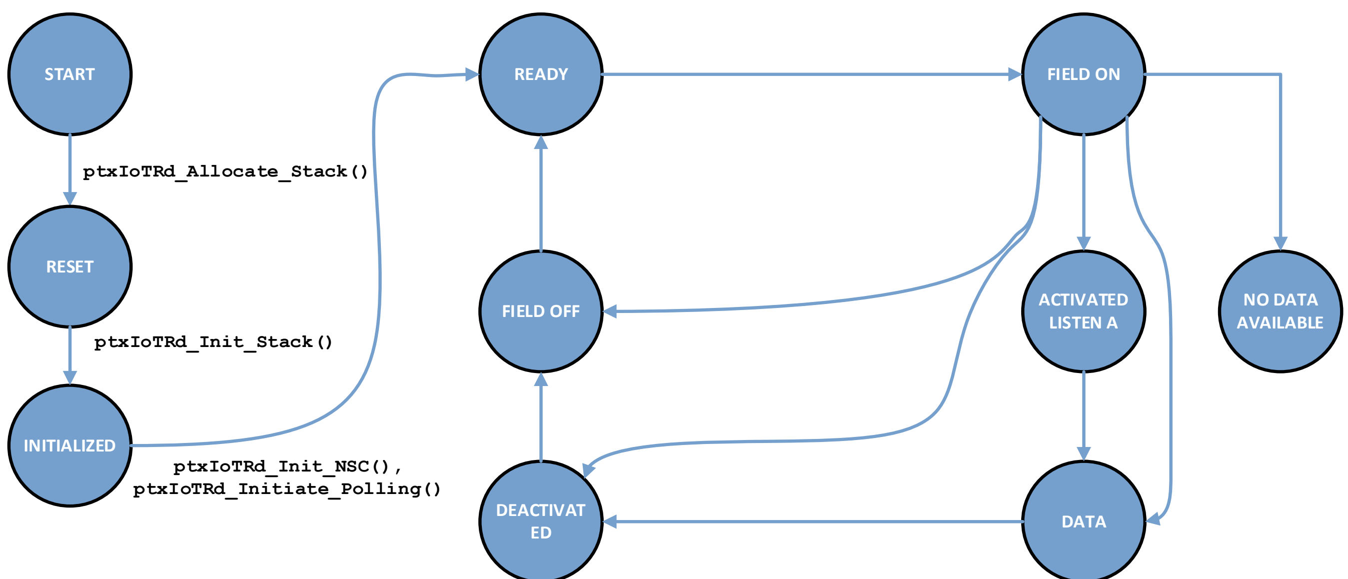

Figure 4. HCE API Flow Example

State Description:

- **State: FIELD ON**

  The message queue contains this state if the PTX1xxR detects an external field from a reader. The PTX1xxR can now receive and send data.

- **State FIELD OFF**

  The message queue contains this state if the PTX1xxR does not detect an external field. No more transactions are possible.

- **State: ACTIVATED LISTEN A**

  The message queue contains this state if the external reader has successfully activated the emulated card. The PTX1xxR receives the protocol information which should be activated.

- **State: DATA**

  The message queue contains this state if the external reader is sending data to the emulated card on a protocol level. The emulated card receives the command from the reader, processes it, and returns the requested data together with a status code.

- **State: DEACTIVATED**

  The message queue contains this state if the emulated card is deactivated. Currently there are three possible reasons:

  - A DESELECT command is received
  - A RELEASE command is received
  - The reader has turned off the external field

- **State: NO DATA AVAILABLE**

  The message queue contains this event if the transmission from the reader does not contain any data.

7. IOTRD API SDK Deliverable

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

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

The SDK is structured as shown in the following figure.

Root Folder	"SRC" Folder
BUILD DOCS FILE_SYSTEM SRC	APIs COMPS EXAMPLE 1_CMake_TB.txt 1_CMakeLists_Aux.txt CMakeLists.txt

Figure 5. IOTRD API SDK Folder Structure

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

- **\BUILD**

  Output folder of the build process containing IOTRD API as shared library and/or the demo application as executable and a setup file for the Raspberry-Pi.

  (Initial) Content:

\setupRasbPi.sh	Setup file for Raspberry-Pi reference platform to map the GPIO-pin (= IRQ) into user space. This file also performs a quick check for available hardware interfaces needed for the SDK (SPI/I2C/UART).

- **\DOCS**

  HTML-based description of the IOTRD API.

  Note: The landing page for the description is "**\DOCS\html\index.html**".

- **\FILE_SYSTEM**

  Contains the various configuration files for NSC Stack and PTX1xxR.

  Content:

\NSC_RF_CONFIG.dat	RF-configuration for PTX1xxR
\NSC_SYS_CONFIG.dat	System-configuration for PTX1xxR

- **\SRC\APIs**

  Contains the source code for the IOTRD API.

  Content:

\IOT_READER\ptx_IOT_READER.*	IOTRD API
\IOT_READER\ptx_IOT_RD_Int.h	IOTRD API data structures
\NATIVE_TAG\ptxNativeTag*.*	Native-Tag API
\NDEF\ptxNDEF*.*	NDEF API
\GPIO\ptxGPIO*.*	GPIO API
\FELICA_DTE\ptxFeliCa_DTE*.*	FeliCa-DTE API
\RF_TEST\ptxRF_Test*.*	RF-Test API
\TRANSPARENT_MODE\ptxTransparentMode*.*	Transparent-Mode API
\COMMON*.*	Generic Helper functions and additional code examples

- **\SRC\COMPS**

  Source code of NSC Stack including (sub-)components and HAL and OSAL.

  Content:

\FACTORY*.*	Factory component
\HAL*.*	Hardware Abstraction Layer (HAL) including a reference implementation based on I^2C or SPI for Linux \INT*.*
\INT*.*	Integration Layer
\IORQ*.*	Hardware access dispatcher for PTX1xxR
\LOG\	Logging component
\NSC\	NSC Stack Core component
\NVM\	NVM access component (file access)
\OSAL\	Operating System Abstraction Layer (OSAL) including reference implementation for Linux
*.h	Generic headers (status information, compile switches etc.)

  Important: The files "**ptxNSC_uCODE.c**" and '**.h**' in folder \SRC\COMPS\NSC\ contain the FW image for the PTX1xxR chip which is required for proper functionality of the PTX1xxR chip. Do not modify the content of these two files.

- **\SRC\EXAMPLE**

  Contains a demo example implementation of the EMV Loopback mode based on the IOTRD API.

  Content:

\ptx_IOT_RD_Main.*	Demo application

- **\SRC\CMake*.txt**

  CMake-based scripts to build the IOTRD API and the demo application (see section 8).

8. IOTRD API Target System Integration

This chapter describes the required steps for a Software integrator to:

- Compile the IOTRD API together with an example application as binary to work stand-alone
- Integrate the source code of the IOTRD API as (sub-)component into an existing application
- Implement the abstraction layers for HAL and OSAL
- Use the CMake build system

8.1 Introduction

As described in section 6, the IOTRD API and all the other components are available as source code of the delivered SDK.

Important: The SDK contains a reference implementation for the target platform dependent abstraction layers HAL and OSAL based on the Linux OS using SPI or I^2C. If another target platform and/or host interface is used, HAL and OSAL must be adapted as described in section 8.5.

While the source code of the IOTRD API can be directly integrated into existing applications (see section 8.4), the SDK contains ready-to-use build-scripts based on CMake which supports quick creation of the following to allow fast prototyping and integration:

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

8.2 Build System

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

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

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

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

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

  Important: If the existing Cmake scripts are used and new files get added to the application or existing files get renamed, these changes must be added/changed in this file.
- CMakeLists.txt
  Defines which source files/lists (see 1_CMakeLists.Aux.txt) belong to which build-target.

Currently, the following build targets are defined:

- **IOTRD_EXAMPLE_EXE**

  This target builds the IOTRD API together with the demo application as executable binary. The resulting executable is named IOTRDExample and uses the corresponding file extension of the target platform (for example, .exe on Windows, .a on Linux etc.).

  Build-steps:
  1. **"cmake -g . -DPTX_SDK_HAL=xxx" (xxx = SPI, I2C or UART)**
  2. **"cmake --build . --target IOTRD_EXAMPLE_EXE --config BUILD_TYPE"**

- **IOTRD_LIB**

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

  Build-steps:
  1. **"cmake -g . -DPTX_SDK_HAL=xxx" (xxx = SPI, I2C or UART)**
  2. **"cmake --build . --target IOTRD_LIB --config BUILD_TYPE"**

Important:

- Build-step 1 of each target is only needed once after installation of SDK or once every time one of the CMake-scripts get changed.
- If the definition for **PTX_SDK_HAL** is not set or not supported in build-step 1, SPI will be used by default.
- Ensure '**./setupRasbPi.sh**' is run first if Linux is used. This script checks for needed hardware interfaces (SPI/I2C/UART) and configures the IRQ pin used for SPI and I^2C.

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

8.3 Integration Flow–IOTRD API => Stand-alone

Figure 6 shows the integration flow for the IOTRD API as stand-alone library (Path 1), and the IOTRD API combined with the demo application as executable (Path 2) based on the CMake-build process.

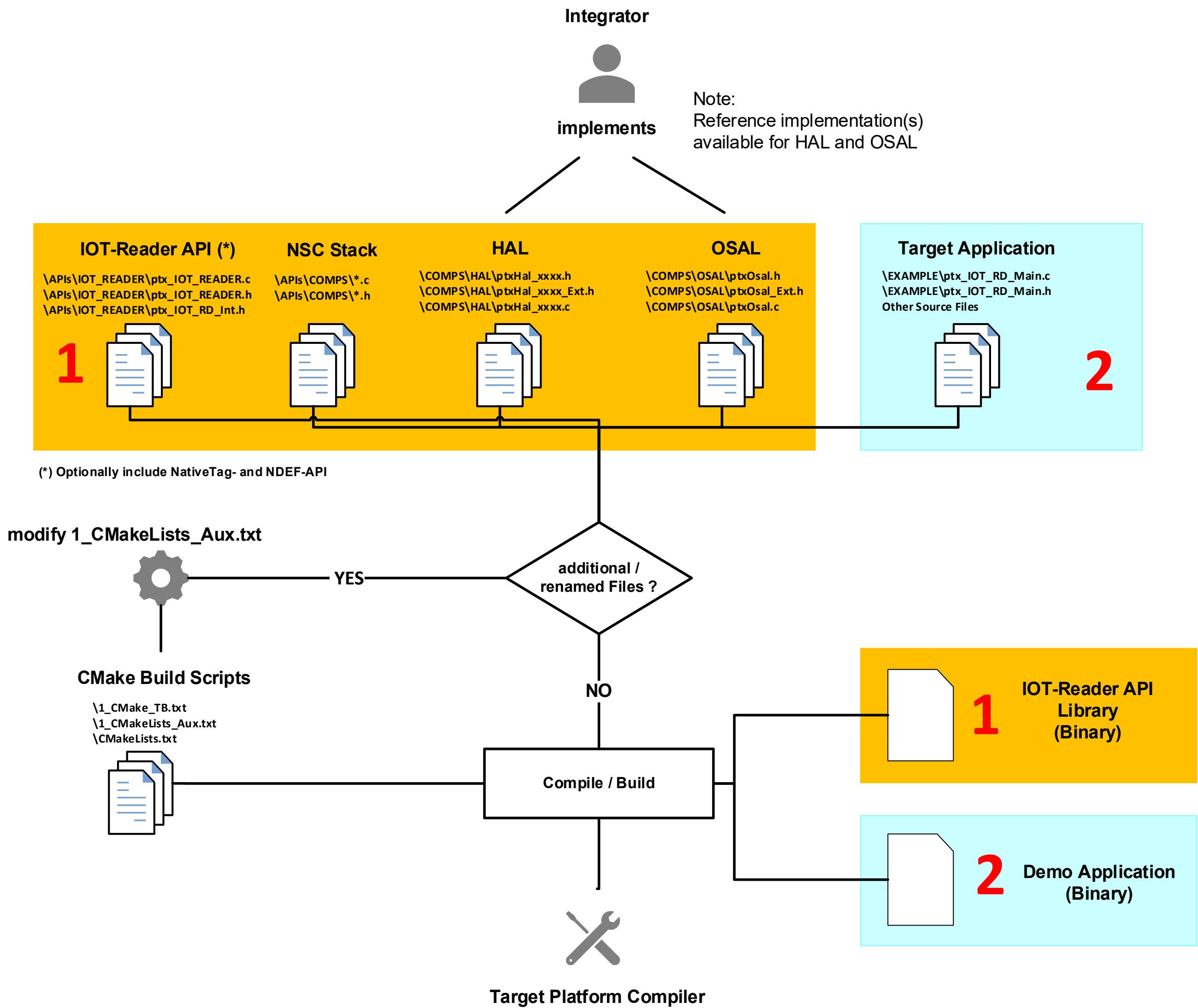

Figure 6. IOTRD API Integration Flow: Stand-alone System

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

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

8.4 Integration Flow–IOTRD API => Component

Figure 7 shows the example flow when the source code of the IOTRD API and the other components get directly integrated into an existing system / application.

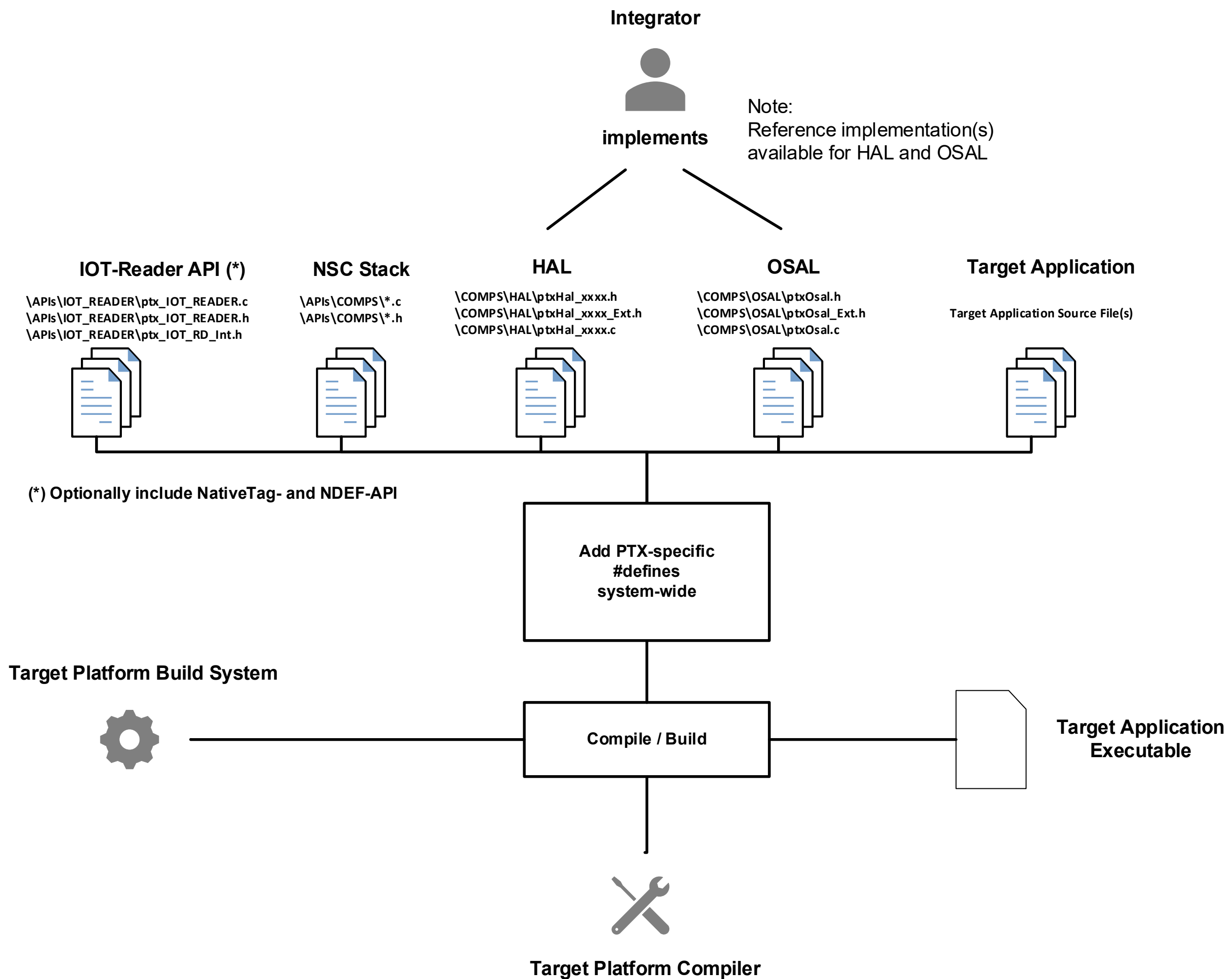

Figure 7. IOTRD API Integration Flow: (sub-)Component within Existing Application

The approach for the IOTRD API, the NSC Stack, and the abstraction layers HAL and OSAL is the same than described in section 8.3.

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

- “RD_ONLY”
- “PTX_FEATURES_NSC_READER_ONLY”
- “PTX_FEATURES_HAL_YYY” (YYY = SPI, I2C, or UART)
- “PTX_PRODUCT_TYPE_IOT_READER”
- “PTX_SDK_HAL=xxx” (xxx = SPI, I2C or UART)

8.5 Target Platform Abstraction Layers

The NSC Stack contains the two components that are both split into a target platform independent - and dependent as highlighted in the red boxes below in Figure 8.

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

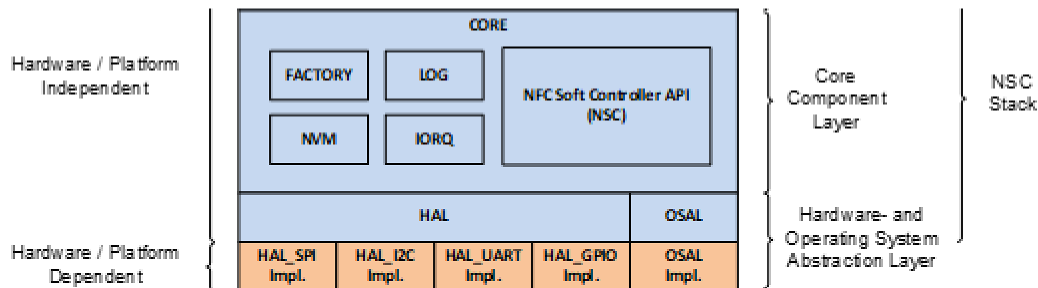

Figure 8. HAL and OSAL Architecture

The target platform independent part is directly used by the NSC Stack and, therefore, should not be changed. **The target platform dependent part of HAL and OSAL needs to be adapted for the specific platform as described in the following chapters.**

8.5.1. Hardware Abstraction Layer (HAL)

The HAL serves as the abstraction layer for the dedicated physical host interface between the application processor and the PTX1xxR which can be either SPI, I^2C or UART. It implements basic functions to send and receive data over the given host interface or to configure its parameters.

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

Target Platform Independent Part

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

Target Platform Dependent Part

Note: “xxxx” in the following paragraphs stands for either “UART”, “I^2C” or “SPI”.

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

  Important: The file “**ptxHal_xxxx_Ext.h**” defines the data structure type “**ptxHal_xxxx**” which is used by the target platform independent part.

The current definition of this structure including the content is used by the reference implementation (see below) but can be customized depending on the requirements and/or implementation of the target platform.

- **\SRC\COMPS\HAL\ptxHal_xxxx_Linux.c**
  OS-specific (Linux) reference implementation for specific host interface.
- **\SRC\COMPS\HAL\ptxHal_Gpio_Linux.c**
  OS-specific (Linux) reference implementation for handling IRQ pin used by SPI and I²C (see description in next section).

  Important: The provided reference implementations for HAL-SPI and -I2C may contain single API functions that are empty and/or are not implemented. This may have various reasons such as the functionality may not be explicitly required by the example reference target platform (for example, the Raspberry Pi 4) or the functionality is RFU and the concrete implementation is not required yet.

8.5.1.1. Sending and Receiving Data via Host Interface

The PTX1xxR hardware uses two types of communication via the physical host-interface:

- Synchronous communication used for commands and responses
- Asynchronous communication for events / notifications and RF data exchanges

The HAL and the NSC component supports both communication types by implementing mechanisms to synchronously send and receive data to/from the PTX1xxR and asynchronously receive data from the PTX1xxR.

The NSC component on top of the HAL layer manages both communication types internally using dedicated Threads.

The actual data exchange (in other words, sending and receiving data) is managed by HAL API function "**ptxHAL_TRx (…)**", which takes parameters for Tx and Rx operations. The implementation of "**ptxHAL_TRX**" must also support separated Tx and Rx operations by setting either the Tx or Rx parameters to NULL/0. This allows the API function for both communication types.

To detect if the PTX1xxR hardware wants to send data to the host (valid for both communication types), the HAL component must implement support for one of the following mechanisms depending on the used host-interface:

- **SPI and I²C**
  These host-interfaces use an additional IRQ-pin to indicate to the host that data is available.
  A reference / example implementation for Linux can be found in "**\SRC\COMPS\HAL\ptxHal_Gpio_Linux.c**"
- **UART**
  This host interface works in push-mode (in other words, data can be sent at any time to the host) after chip initialization. The HAL needs to implement means to collect all incoming data packets from the PTX1xxR hardware, the actual processing – for example, whether the data is complete or not – is managed by the higher-level NSC component.

8.5.1.2. Implementation and Verifying Low-Level Host-Interface Driver

An elementary step for porting the PLAT component to a new platform is to implement/provide access to low-level host-interface drivers for either SPI, I²C or UART. In many applications, these drivers provided by third parties.

To verify the correct functionality of the drivers in combination with the PTX1xxR chip, the following hard-coded sequences can be used to assess the correct connection, framing, timings etc.

Host-IF	Tx-Sequence	Expected Rx-Sequence	Comment
SPI	nSS = 0, 0x30, 0xFF, 0xFF, 0xFF	0x00, 0x00, 0x00, 0x21, nSS = 1	Use up to 10 Mbit/s
I2C	S, SA (W), 0x30, 0xFF, RS	SA (R), 0x21, P	Use 100 kHz/s S = Start Condition SA = Slave Address RS = Repeated Start Cond. P = Stop Condition
UART	0x55, 0x02, 0x30, 0xFF	0x01, 0x21	Use 115200 kBit/s

Note: For more information on the packet format, see References item [6].

8.5.1.3. Implementation Hints for Hardware Abstraction Layer (HAL)

This section provides additional information about the HAL implementation for specific host interfaces (if applicable).

- **I2C**

  The PTX1xxR hardware implements a few low-level host commands that require to send a repeated start condition on the I^2C bus during a data exchange via the HAL API function "**ptxHAL_TRx**". This is exemplified for the RRA operation in the Figure 9 (simplified). The RRA operation requires first to set/write the address where to read from, followed by a repeated start condition before the actual read operation takes place. Once the read operation is done, the host stops the data exchange by sending a stop condition on the I^2C bus.

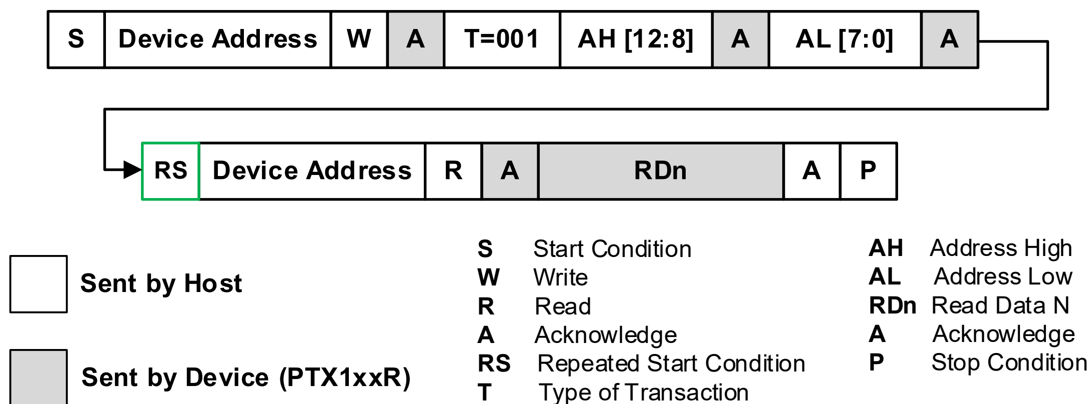

Figure 9. PTX1xxR RRA-Operation using I^2C (Simplified)

The handling of when a repeated-start condition required for a specific hardware-command is completely managed by the upper layers of the SDK and is controlled through the HAL I^2C API functions "**ptxHAL_SetI2C_NoStopFlag**" and "**ptxHAL_ClearI2C_NoStopFlag**". Both API functions are used to indicate (via a flag) to the lower layer HAL I^2C implementation if a repeated-start condition is necessary during the call to "**ptxHAL_TRx**".

As "**ptxHAL_TRx**" must be implemented/ported by the user, care must be taken to tell the target platform dependent I^2C driver implementation to not send a stop condition when the last entry of array parameter "**txBuf**" was sent in case the flag is set.

Note: Many target platform driver implementations and its API interfaces are using different terms and/or ways for handling repeated-start conditions or no-stop conditions like the following:

- Not sending a stop-condition for the current I^2C transfer

- Using a repeated start-condition for the next I²C transfer (prevents the stop condition internally)

The scenario as shown and required by the example in Figure 9. PTX1xxR RRA-Operation using I²C (Simplified)

can be achieved with each of the mentioned examples but may need some adaptations in the logic of the actual HAL I²C implementation.

8.5.2. Operating System Abstraction Layer (OSAL)

The OSAL serves as the abstraction layer for the underlying Operating System. It implements support for various OS-specific mechanisms such as Threads, Semaphores, Mutexes, Timers, dynamic Memory Allocation, etc.

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

Target Platform Independent Part

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

Target Platform Dependent Part

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

8.6 Integration Notes/Hints

8.6.1. Performance Optimization

The SDK contains various code examples to demonstrate the usage of the provided SDK API. As such, the code examples sometimes enables features and/or extended capabilities like artificial delays for better readability of the output in the console-application or serial terminal, (real-time) logging, etc.

Depending on the target system, these features/capabilities may impact the overall performance and can be adapted for the integration into the final application.

The following list provides a typical overview on which features/capabilities can be adapted to improve overall performance of the SDK APIs.

8.6.2. Artificial Delays/Sleep-Operations

To improve the readability of the output in the provided console-application and/or serial application, the example code contains several artificial delays/sleep-operations (referred to as “delays”).

There are two types of (artificial) delays:

- **Delay after exemplary RF data exchanges**

  These delays are used to improve readability and can be disabled by either defining the compile-flag/-switch “**PTX_DISABLE_EXAMPLE_DELAYS**” globally or by simply removing them
- **Delays after polling various states**

  Polling certain states/variables may increase the overall CPU load of an application or produce a high amount of unwanted console output. To avoid this, several delays were introduced (for example, card state during RF polling/discovery and no card was detected). Performance can be improved by adapting these delays for the target application or to replace the polling by using platform-specific stand-by mode(s).

  Example: The delays for polling the card-status is referenced in the example code by the “**PTX_*_NO_CARD_SLEEP_TIME**” definition.

```
/*
 * Example Code-Delays/-Sleeps; used to prevent high-CPU loads on target system if a certain status gets polled frequently.
 * May be adapted on target system
 */
#define PTX_      _NO_CARD_SLEEP_TIME          (10u)
```

8.6.3. Logging System Types

The internal stack contains a powerful logging system that is helpful for system integration and debugging. The logging system consists of two logger types:

- **Standard Logger**

  This standard/default Logger is based on a ring-buffer implementation in RAM with a configurable size. It can be optionally written to a given file once the function "***_Close_Stack**"-function gets called. Even though the Logger itself uses only RAM, a lot of entries can impact the system performance. Additionally, the size (in other words, the number of logging entries) have a direct impact on overall RAM-consumption. The size itself is configurable via the define "**POS_LOG_DEPTH**" or "**IOTRD_LOG_DEPTH**". Configuring a size of 0 disables the Logger completely.

- **Realtime Logger**

  While the standard/default Logger writes an output file only upon the call to "***_Close_Stack**", it can sometimes be very helpful to write the current logging entries immediately to the underlying file system. This can have a huge impact on the system's performance. The code examples enable this Logger by default by calling "***_Enable_RT_Log**". To disable this Logger, simply remove the function call in the main applications.

8.6.4. Maximum Number of Supported Cards

The IoT Reader SDK implementation uses an internal card registry to store card data (for example, technical and activation parameters) for up to 50 cards by default. The memory for card data entries in the registry is allocated statically and has a direct impact on the consumed memory.

The number for maximum supported cards can be adapted depending on the target application. The number can be changed by adapting the #define "**PTX_IOTRD_MAX_SUPPORTED_DEVICES**" in the header file of the IOTRD API component.

Note: On MCU systems, reducing the number from 50 down to 5 can free up approximately 1KB of RAM.

8.6.5. Reference Implementation–Code Size and Memory Consumption

The reference implementation provided with the SDK is built (Release-version) on a RaspberryPi-4 system based on Raspbian GNU/Linux 10 and GCC V8.3 which produces the following output in terms of code size and memory consumption.

Code Size Binaries:		**Size**
	IOTRDExample	~ 1,174MB
	libIOTRD.so	~ 1,053MB
Peak Heap/Memory Consumption of Example Application	Standard Logger with 10000 entries	~ 3,4 MB
Peak Heap/Memory Consumption of Example Application	Standard Logger disabled	~ 61KB

Note: The reference implementation of the SDK gets provided with the Standard Logger enabled using 10000 log entries which results in a relatively high memory consumption. This can be significantly reduced by reducing the number of log entries as described in section 8.6.3 or disabling the Standard Logger completely by setting the number of log entries to 0.

Important: The values listed in the table are reference values. Depending on the target system integration, build/environment properties such as the following can vary the numbers significantly:

- 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, Logger)
- Tested use case/scenario, etc.

9. RF and System Configuration Updates

The PTX1xxR allows configuration of RF and System configuration parameters. A default configuration is downloaded to the PTX1xxR during the initialization phase and can be changed during runtime via an API call.

9.1 Default RF Configuration

The default RF configuration is stored in a binary file called "**NSC_RF_CONFIG.dat**" and is stored inside the IOTRD API SDK in the folder "**FILE_SYSTEM**". The location of the files can be changed by setting the parameter "**fsPath**" of the API function "**ptxIoTRd_Init_Stack".**

The default RF configuration (in other words, the binary file itself) can be generated using the "PTX1xxR IOT Config Tool" from Renesas.

The following figure shows a screenshot of the "PTX1xxR IOT Config Tool". It allows to configure the RF-configuration parameters and to generate the required .dat-files accordingly.

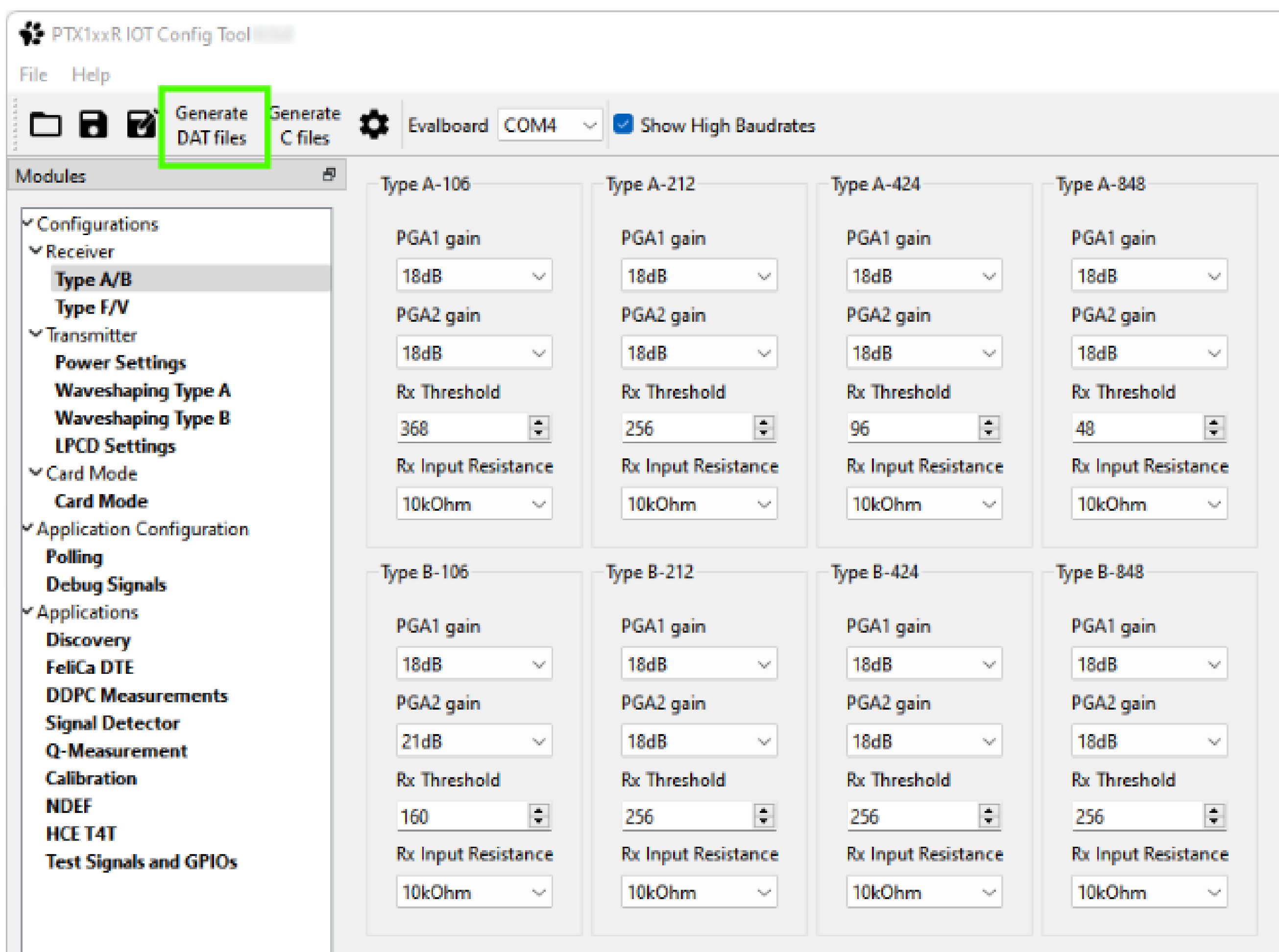

Figure 10. PTX1xxR IOT Config Tool

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

9.2 Default System Configuration

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

Configuring the parameters and generating the .dat-file works the same way than described in section 9.1.

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

9.2.1. Temperature Sensor Calibration

The PTX1xxR features an on-chip temperature sensor that continuously monitors the die temperature. If the temperature exceeds a configurable threshold, the transmitter is automatically disabled.

To get expected accuracy, temperature sensor requires calibration. Do this **once for a given PTX1xxR**. It is available to the user via the API **"ptxIoTRd_TempSensor_Calibration".**

Along with sensor calibration, the API calculates compensated temperature threshold value which is then used as one of the input parameters in a call to "**ptxIoTRd_Init_NSC**".

Temperature compensation steps:

- Set the system into INITIALIZED state (see section 4)
  - Call "**ptxIoTRd_Allocate_Stack**"
  - Call "**ptxIoTRd_Init_Stack**"
- Perform temperature calibration:
  - Set ambient temperature to a desired value (for example, 25°C). Provide this value as **"Tambient"** input parameter in **"ptxIoTRd_TempSensor_Calibration".**
  - Set the value of expected temperature shutdown threshold in **"Tshutdown"** parameter (for example, 100°C, the value is provided in the PTX1xxR Datasheet). Please note that **"Tshutdown"** is a pointer and returns the calculated threshold value from the calibration routine.
  - Call "**ptxIoTRd_TempSensor_Calibration**" with provided parameters.
- Permanently store the value returned in **"Tshutdown"** parameter for future use (for example, in a configuration file).

  Use that stored value to fill **"CalibratedTempThreshold"** member of the **"initConfig"** input parameter in every call to "**ptxIoTRd_Init_NSC**" during system initialization.

  Important:
  - Temperature sensor calibration (temperature threshold compensation) must be executed once per PTX1xxR in controlled environment conditions. Once done, it does not need to be started all over again before NSC initialization.
  - The resulting value should be stored and re-used in all future calls to "**ptxIoTRd_Init_NSC**".

9.3 Dynamic RF and System Configuration

To change the parameters at runtime, the API function "**ptxIoTRd_Update_ChipConfig**" can be used in state "**READY**". The function takes a struct-type as input parameter containing the ID, the value, and the length of a given parameter set.

10. Add-on Libraries/APIs

The IOTRD API contains additional support libraries that extend the existing functionality of the IOTRD component at application layer. The use of these libraries is optional (in other words, included or excluded from the build process).

The current SDK version contains the following add-on libraries:

- NativeTag API for NFC Forum Type Tags 2–5 (T2T, T3T, T4T, T5T)
- NDEF API for NFC Forum Type Tags 2–5 Tag (T2T, T3T, T4T, T5T)
- GPIO API for PTX1xxR
- RF-Test
- FeliCa-DTE
- Transparent-Mode

Details about each of API can be found in the following chapters.

10.1 Native-Tag API

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

As shown in the following figure, the Native-Tag API is located on top of the IOT-Reader API, and it uses internally the function “**ptxIoTRd_Data_Exchange**” to exchange RF data with the corresponding Tag.

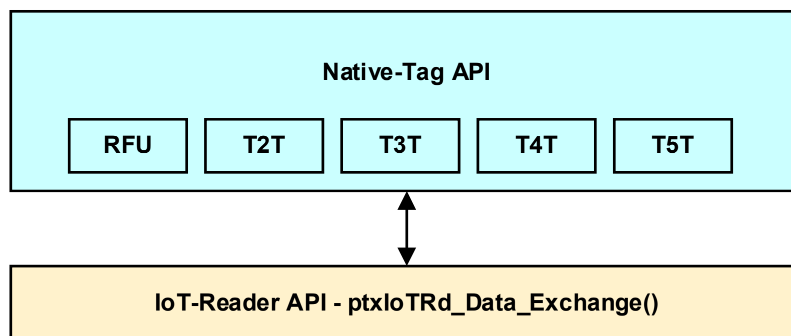

Figure 11. Native Tag API Overview

Each of the individual Tag-specific APIs follows the same approach where:

- An initial call to “**ptxNativeTag_TxTOpen**” is required to initialize the component
- A final call to “**ptxNativeTag_TxTClose**” is required to free potentially allocated resources
- All other functions can be used to send a Tag-specific command

Note: While the component initialization can also take place at the beginning (for example, after stack initialization), the actual Tag-specific command can only be sent in API state “DATA EXCHANGE”. For protocols which 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.

10.1.1. Supported Type 2 Tag Commands

- READ
- WRITE
- SECTOR_SELECT

10.1.2. Supported Type 3 Tag Commands

- SENSF_REQ
- CHECK
- UPDATE

10.1.3. Supported Type 4 Tag Commands

- SELECT
- READ_BINARY
- UPDATE_BINARY

10.1.4. Supported Type 5 Tag Commands

- READ_SINGLE_BLOCK _REQ
- WRITE_SINGLE_BLOCK_REQ
- LOCK_SINGLE_BLOCK_REQ
- READ_MULTIPLE_BLOCK _REQ
- EXTENDED_READ_SINGLE_REQ
- EXTENDED_WRITE_SINGLE_BLOCK _REQ
- EXTENDED_LOCK_SINGLE_BLOCK _REQ
- EXTENDED_READ_MULTIPLE_BLOCK _REQ
- SELECT_REQ
- SLPV_REQ

10.2 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

Important (*): 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 like:

- Tag memory size
- Timing parameters
- Certain file systems (may also use cryptographic algorithms for creation)
- Certain format of memory blocks, etc.

which are proprietary/manufacturer specific. Although prepared in the NDEF-API(s), the final implementation must be managed by the integrator (if required).

As shown in Figure 12, the NDEF API is located on top of the Native-Tag API. The NDEF API is split into two sub-modules consisting of the following:

- Tag-Type specific NDEF operations
- Generic NDEF operation layer

The generic NDEF operation layer determines which Tag/RF-Protocol is currently active and calls the corresponding Tag-Type specific operation.

Note: Each **READ**, **WRITE,** and **LOCK** operation requires an initial execution of operation **CHECK**, otherwise the operations/functions will return an error.

Note: Handling of IDs as described in the previous chapter, is handled automatically by the NDEF API.

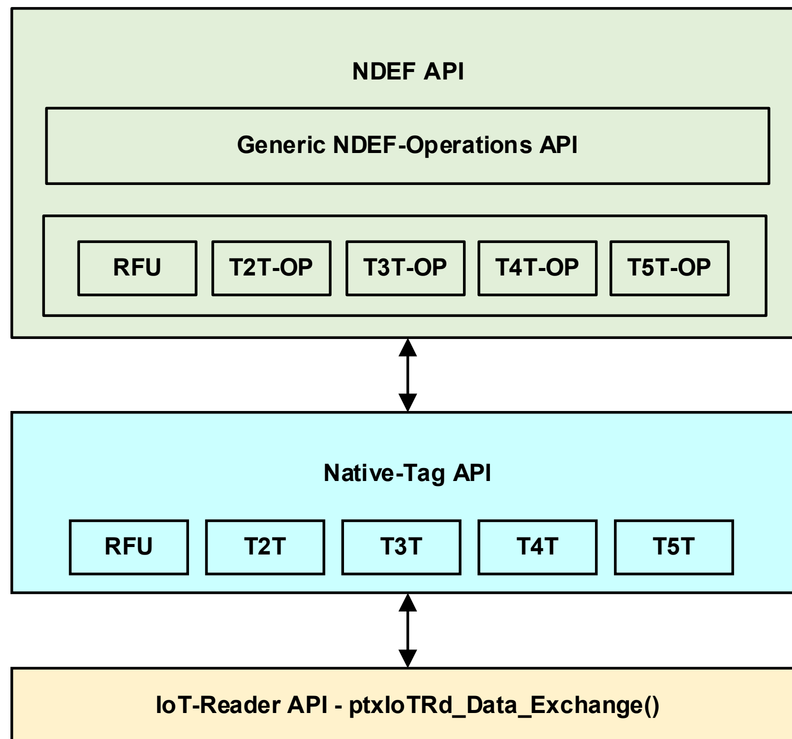

Figure 12. NDEF API Overview

Each of the individual Tag-Operation APIs or the general NDEF APIs follows the same approach where:

- An initial call to "**ptxNDEF_[TxT]Open**" is required to initialize the component
- A final call to "**ptxNDEF_[TxT]Close**" is required to free potentially allocated resources
- All other functions can be used to execute NDEF operations

Note: While the component initialization can also take place at the beginning (for example, after stack initialization), the NDEF operations can only be used in API state "DATA EXCHANGE". Setting Tag-specific parameters like ID (see previous section) are managed internally.

10.3 GPIO API

The GPIO API grants access to the GPIO pins 5 to 12 of the PTX1xxR which consists of the following set of functions:

- **ptxGPIO_Init**
  Initializes the GPIO component (possible to be called in any state)
- **ptxGPIO_Deinit**
  De-initializes the GPIO component (possible to be called in any state)

- **ptxGPIO_Config**
  Configure an individual GPIO pin as:
  - Input (+optional flag to enable an internal pull-up resistor; see PTX1xxR datasheet)
  - Output (+optional flag to increase the internal driver strength; see PTX1xxR datasheet)
- **ptxGPIO_Write**
  Sets / Writes a GPIO pin
- **ptxGPIO_Read**
  Gets / Reads a GPIO pin
- **ptxGPIO_Write_DAC**
  Writes a value to 5-bit DAC-0

Note: GPIO access can be performed in any state from **READY** onwards.

10.4 RF-Test API

The RF-Test API allows to perform various general RF tests such as the PRBS9 or PRBS15 tests.

- **ptxRF_Test_Init**
  Initializes the RF-Test component (possible to be called in any state)
- **ptxRF_Test_Deinit**
  Deinitializes the RF-Test component (possible to be called in any state)
- **ptxRF_Test_RunTest**
  Executes a selected RF-Test. The test configuration gets passed as test input parameter structure which also contains an ID field to identify/select the desired test.

Currently supported RF-Tests:

- PRBS9 (needs to be stopped manually (see below)
- PRBS15 (needs to be stopped manually (see below)
- Carrier-on without modulation (needs to be stopped manually (see below)
- **ptxRF_Test_StopTest**
  Stops an ongoing RF test.

Note: RF test execution and stopping is only allowed in state **READY**.

10.5 FeliCa-DTE API

The FeliCa-DTE API allows the user to perform various FeliCa-related tests to pass the FeliCa M-Class certification and can also run the test-sequences defined in the Reader/Writer Digital Protocol Requirements specification.

- **ptxFeliCa_DTE_Init**
  Initializes the FeliCa-DTE component (possible to be called in any state)
- **ptxFeliCa_DTE_Deinit**
  Deinitializes the FeliCa-DTE component (possible to be called in any state)
- **ptxFeliCa_DTE_EnableMode**
  Configures the PTX1xxR to work in FeliCa-DTE mode. The mode must be enabled before any test gets carried out and disabled afterwards.

- **ptxFeliCa_DTE_RunTest**
  Executes a selected RF-test. The test configuration gets passed as test input parameter structure which also contains an ID-field to identify/select the desired test.

Currently Supported FeliCa-Tests:

- M-Class Performance Tests according to References item [4]
- Test-sequences defined in Reader/Writer Digital Protocol Requirements according to References item [5]

Note: FeliCa-test execution is only allowed in state **READY**.

10.6 Transparent-Mode API

The optional Transparent-Mode API enables an application to implement proprietary protocols based on low-level RF commands.

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 with the standard API functions provided by this SDK.

- **ptxTransparentMode_Init**
  Initializes the Transparent-Mode component (possible to be called in any state)
- **ptxTransparentMode_Deinit**
  Deinitializes the Transparent-Mode component (possible to be called in any state)
- **ptxTransparentMode_SetField**
  Enables or disables the Transparent-Mode by turning the RF-field on or off in state **TEST**. An application must call this function before any further call to "**ptxTransparentMode_SetRFParameters()**" as well as "**ptxTransparentMode_Exchange()**" with input parameter **state = 1**. If the mode shall be exited to return to state **READY**, this function must be called again with input parameter state = 0.
- **ptxTransparentMode_SetRFParameters**
  This API function configures the hardware for the following call(s) to "**ptxTransparentMode_Exchange()**".

  The following configuration parameters are supported:

- RF-Technology: A, B, F or V
- Tx- and Rx-Bitrates: 106/212/424/848/26.5 kBit/s
- Flags (Parity, CRC): Enable / Disable Tx-/Rx-CRC handling
  Enable / Disable Tx-/Rx-Parity-Bit handling
- Number of Tx-Bits: Number of (residual) Bits to be sent for last byte
- RES-Limit: Maximum number of responses to receive

- **ptxTransparentMode_Exchange**
  This API function performs the actual data exchange via RF. Received data always contains one additional byte that contains a contactless status byte. The contactless status byte is defined as follows:

- Bit 7: If set to 1, a contactless error occurred (for example, CRC- or Parity-error).
- Bit 6–0: Number of valid bits in last received byte.
  Example: If the answer to a REQ-A/SENS_REQ is 0x44 0x03, the API function returns
  rxLength = 3 (2 byte received data + 1 byte contactless status byte)
  rx = 0x44 0x03 0x00 (last byte = contactless status byte)

Implementation Guidelines/Hints

- If a protocol should be implemented that always uses the same RF-parameters, a single call to this function is sufficient. The function "**ptxTransparentMode_Exchange()**" can be used to overwrite the parameters for a single call.
- Tx-/Rx-Parity-Bit handling is only applicable for RF-Technology A; ignored for others.
- Number of Tx-Bits can be set from 0–7 where 0 means that all 8 bits shall be sent.
- RF-Technology F works only for bitrates 212 and 424 kBit/s.
- RF-Technology F normally prepends the LEN-byte to a Tx- or Rx-frame. The LEN-byte is automatically managed by the hardware. 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 to re-enable the HW-Receiver n-times (i.e., 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 should be set to 0. In this case, the HW-Receiver gets re-enabled until the timeout-parameter expires (see "**ptxTransparentMode_Exchange()**"). For all other cases, RES-Limit should be set to 1.

Examples

- **Type A – REQ-A/SENS_REQ:**
  - RF-Technology = A
  - Tx/Rx Bitrate = 106 kBit/s
  - Flags = Tx/Rx Parity = Enabled, Tx-/Rx-CRC = Disabled
  - Number of Tx-Bits = 7
  - RES-Limit = 1
  - Timeout = 1ms
  - tx[0] = 0x26
  - txLength = 1

- **Type A – SELECT/SEL_REQ:**
  - RF-Technology = A
  - Tx/Rx Bitrate = 106 kBit/s
  - Flags = Tx/Rx Parity = Enabled, Tx-/Rx-CRC = Enabled
  - Number of Tx-Bits = 0
  - 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 Bitrate = 106 kBit/s
  - Flags = Tx/Rx Parity = d.c., Tx-/Rx-CRC = Enabled
  - Number of Tx-Bits = 0
  - 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 Bitrate = 212 or 424 kBit/s
  - Flags = Tx/Rx Parity = d.c., Tx-/Rx-CRC = Enabled
  - Number of Tx-Bits = 0
  - RES-Limit = 1
  - Timeout = 5ms
  - tx[…] = 0x00 0xFF 0xFF 0x00 0x0F
  - txLength = 5
- **Type F – SENSF_REQ (Multiple Cards or TSN != 0):**
  - RF-Technology = F
  - Tx/Rx Bitrate = 212 or 424 kBit/s
  - Flags = Tx/Rx Parity = d.c., Tx-/Rx-CRC = Enabled
  - Number of Tx-Bits = 0
  - RES-Limit = 0
  - Timeout = 2.4ms + TSN * 1.2 (*Note*: Timeout is always given in Integers)
  - tx[…] = 0x00 0xFF 0xFF 0x00 0x0F
  - txLength = 5
- **Type V – Inventory Command:**
  - RF-Technology = V
  - Tx/Rx Bitrate = 26.5 kBit/s
  - Flags = Tx/Rx Parity = d.c., Tx-/Rx-CRC = Enabled
  - Number of Tx-Bits = 0
  - RES-Limit = 1
  - Timeout = 10ms
  - tx[…] = 0x26 0x01 0x00
  - txLength = 3
- **Type B Prime – APGEN:**
  - RF-Technology = BPrime
  - Tx/Rx Bitrate = 106 kBit/s
  - Flags = Tx/Rx Parity = d.c., Tx-/Rx-CRC = Enabled
  - Number of Tx-Bits = 0
  - RES-Limit = 1
  - Timeout = 10ms
  - tx[…] = 0x01, 0x0B, 0x3F, 0x80
  - txLength = 4

10.7 Transparent Data Channel (TDC) API

The Transparent Data Channel (TDC) enables arbitrary data transfers between the PTX1xxR and a Renesas NFC Forum WLC Listener device (further referenced as **Listener** in this section) such a the Renesas PTX30W. The transport protocol is built upon ISO14443-3 (or NFC Forum T2T) - Type A 106kbps frames.

The TDC component consists of the following API functions:

- **ptxTDC_Init**

  Initializes the TDC component (possible to be called in any state).

- **ptxTDC_Deinit**

  Deinitializes the TDC component (possible to be called in any state).

- **ptxTDC_Write**

  Writes a TDC message with a max. of 63 payload bytes to the Listener. If the function input parameter "**ackTimeoutMs**" is set to a value larger than 0, the function call waits internally for an acknowledgement from the Listeners Host MCU by internally calling "**ptxTDC_IsReceived**" (see below).

- **ptxTDC_IsReceived**

  Checks if the previously sent message using "**ptxTDC_Write**" has been received/read by the Listener Host MCU.

Data transfers are done in packets of max 63 bytes. Every data transfer (no matter whether it's a TX or RX operation) must always be initiated by the PTX1xxR - it is the master of the communication channel. If data shall be sent to the Listener, the PTX1xxR can write this data directly to the T2T memory of the Listener. If data shall be received from the Listener, the Listener CANNOT start a data transfer directly to the PTX1xxR, instead it must wait until it is read by the PTX1xxR.

The application can choose between two different modes of operation:

- **NFC Forum Compliant Mode**:

  The NFC Forum Compliant transfer mode uses the T2T command set, defined in [7] to read and write data from/to the Listener.

  With the T2T_WRITE command, the PTX1xxR can transfer 4 bytes of payload to the Listener, whereas with the T2T_READ command, the PTX1xxR can read 16 bytes from the Listener device with a single RF transaction.

- **PTX Proprietary Mode:**

  Using the PTX proprietary transfer mode, 64 bytes can be transferred at once, either from or to the Listener, allowing an increased data throughput.

  By default the PTX Proprietary Mode is active. NFC Forum Compliant Mode can be enabled by setting the define, shown below.

```
#define TDC_NFC_FORUM_COMPLIANT
```

The TDC add-on API consists of the following files:

- \SRC\APIs\TDC\ptxTDC.c
- \SRC\APIs\TDC\ptxTDC.h

11. References

11.1 General

[1] Renesas *PTX100R Datasheet*

[2] Renesas *PTX105R Datasheet*

[3] Renesas *PTX130R Datasheet*

[4] *FeliCa Reader/Writer RF Performance Certification Specification Ver.1.5*

[5] *FeliCa Reader/Writer Digital Protocol Requirements Specification Ver.1.22*

[6] Renesas *Host-Interface Reference PTX1xxR Reader IC*

[7] NFC Forum, Tag Type 2 Specifications 1.2, 2021.

11.2 Standards and Regulations

- NFC Forum: https://nfc-forum.org/

12. Revision History

Revision	Date	Description
1.01	May 3, 2024	Updated document for SDK Release 7.2.0: ▪ Added support for B-Prime RF-Technology in Transparent-Mode API. ▪ Improved description of API Init-/Open-functions (clarified usage of allocated vs. initialized). ▪ Fixed issue related to EMV collision detection for Type-B (POS-SDK only).
1.00	Oct 18, 2023	▪ Updated the document to the latest template. ▪ Completed minor updates throughout; however, no technical changes were made.

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 Jan 2024)

Corporate Headquarters

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

Trademarks

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

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/.

© 2024 Renesas Electronics Corporation. All rights reserved.

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

Trademarks

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

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/.

© 2025 Renesas Electronics Corporation. All rights reserved.