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

This document describes the PTX NFC IOT-Reader API non-OS software stack (referred to as IoT-Reader (Non-OS)) for the PTX1xxR-Reader (PTX100R, PTX105R, PTX130R) products. The IoT-Reader (Non-OS) SDK provides a reference implementation of an IoT-Reader with multiple card type detection capability, running on a RA4M2 ARM Cortex® M33 (R7FA4M2AD) as an example host target platform.

Contents

Figures

1. Introduction

The provided SDK is implemented in C and the source code can be easily ported to any target MCU or host platform. The RA4M2 reference should be regarded only as an example implementation. Information about the porting to other MCU and host platforms is provided in this document and is supported by the Renesas application support team.

This solution is intended for platforms without an operating system and without a filing system.

In addition to the IoT-Reader (Non-OS) stack, the delivery also contains an example application for simultaneous discovery of multiple card types, thereby showing the correct usage of the IoT-Reader (Non-OS) stack.

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 IoT-Reader SDK

For building the IoT-Reader projects, the following tools are required:

- Renesas E2 Studio 2023.10 (or higher)
  - FSP Package v5.1 (or higher)
- C-Compiler supporting C99 standard

1.3 Abbreviations and Terminology

Abbreviation	Terminology
APDU	Application Protocol Data Unit.
HW	Hardware.
Integrator	Developer who build/integrates the IoT-Reader API into a target application.
IoTRd	IoT-Reader profile.
IoT-Reader	IoT-Reader stack controller.
NDEF	NFC Data Exchange Format.
NFC	Near Field Communication.
NSC	NFC Soft Controller.
RF	Radio Frequency.
RTOS	Real-time Operating System.
SDK	Software Development Kit.
SW	Software.

2. Architecture of IoT-Reader (Non-OS) System

The IoT-Reader (Non-OS) System follows component-based approach that increases modularity and usability. The components are divided into two main groups:

1. Hardware/Platform Independent:

It is suitable to run on application processors where there is neither operating system nor file system. Those components are IoT-Reader (IoT Rd) and NSC.

2. Hardware/Platform Dependent:

This part must be adapted to a specific MCU/platform used as application processor. Reference implementation is provided in this SDK (for more information, refer to section 8). PLAT component belongs to this group.

The IoT-Reader (Non-OS) stack is implemented in ANSI C in order to maximize its portability. Figure 1 shows the main components of this system.

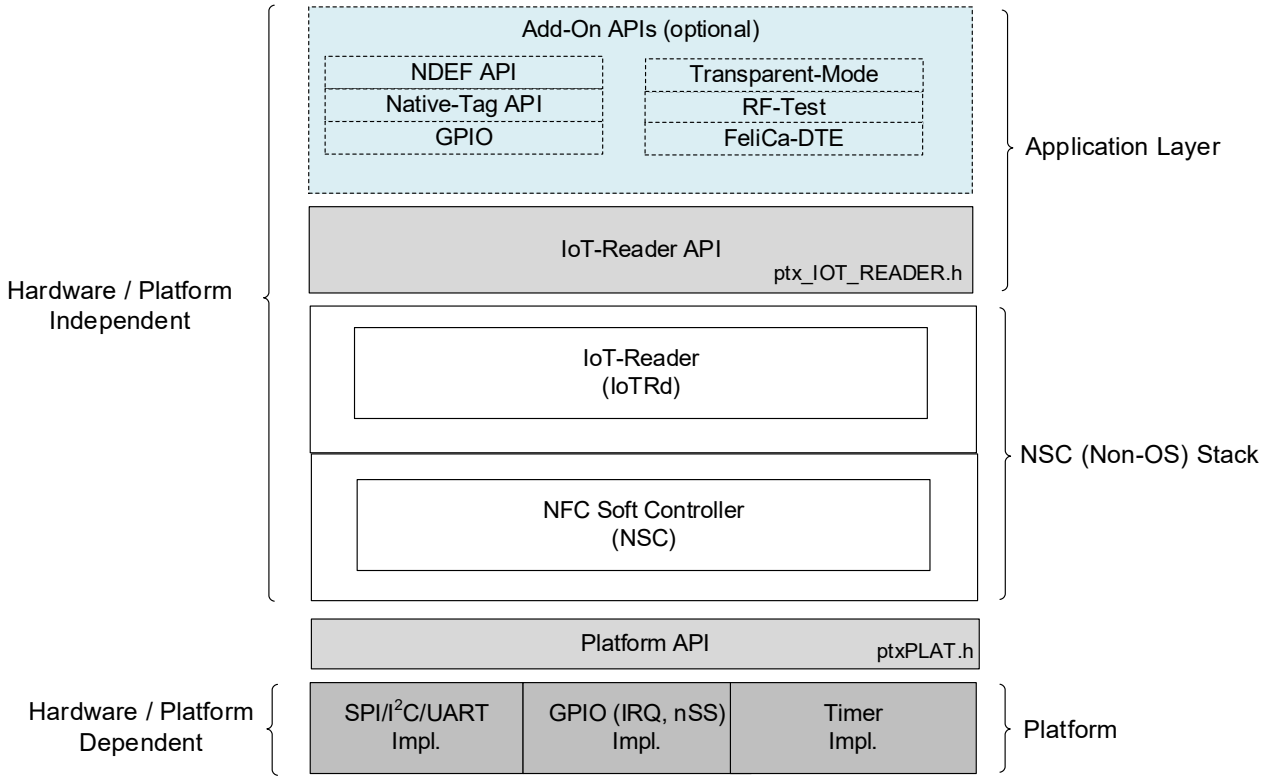


Figure 1. IoT-Reader (Non-OS) Stack Architecture

2.1 Component Description

2.1.1 IoT-Reader API

This API defines the system functionalities exposed by the IoT-Reader (Non-OS) Stack to the application on top. The IoT-Reader API provides the following features:

- Initializes the IOTRD API and NSC stack
- Initializes the PTX chip
- Discovers cards according to NFC Forum including support for “Low-Power-Card-Detection”
- Selects a specific card in case multiple cards/protocols were discovered
- Retrieves card details like technical/activation parameters
- Exchanges RF data and bitstreams

- Stops RF communication
- Optionally updates RF and System configuration parameters at runtime
- Optionally puts the PTX chip into and wake up from stand-by mode
- Controls various GPIO pins of the PTX chip (input, output, can be used any state from **Ready** onwards)
- Performs Host Card Emulation using the HCE API (see section 5)
- Performs FeliCa performance and Digital Protocol Requirement tests
- Enables various RF test-sequences (for example, PRBS9, PRBS15, ...)
- Exchanges low-level RF commands (Transparent Mode)
- Perform arbitrary NFC data exchange with a Renesas NFC Forum WLC Listener device (for example, PTX30W) using the “Transparent Data Channel”-API

The IoT-Reader API itself is described in detail in section 3, and an example flow is described 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 NSC IoT-Reader 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

These APIs are described in section 10.

2.1.3 IoT-Reader Component

This layer contains the actual implementation of the IoT-Reader API functions. It configures the NSC component to operate as multiprotocol NFC Reader device using the PTX NFC chip.

2.1.4 NSC Component

This component exposes to the upper layer the set of functions that abstract the PTX NFC functionality. This layer represents the actual core of the NSC (Non-OS) Stack and provides mainly the following functionalities:

- PTX Chip configuration and initialization
- PTX Chip RF discovery loop
- PTX Chip RF card activation detection
- PTX Chip RF data exchange

2.1.5 Platform API

This API defines the platform/MCU dependent functionalities required by the NSC (Non-OS) stack. The Platform API enables:

- Byte transfer to/from the PTX chip
- The “waiting on” synchronous (blocking) events, interrupts driven, triggered by PTX chip
- The capture of asynchronous (non-blocking) events, interrupts driven, triggered by PTX chip
- Sleep functionality which puts the software execution to sleep for certain time

2.1.6 Platform Component

This component is dependent on the platform/MCU which is used as application processor. It depends as well on the used physical hardware interface (SPI, I²C, or UART) between the application processor platform and the PTX chip.

The Platform component includes the following sub-modules:

- **Interface:** Implements the driver for the physical interface used for communication with the PTX chip
- **GPIO:**. Implements the driver for the GPIO used for IRQ. This submodule is needed when SPI or I²C are used as physical interface; if UART is used, IRQ is not required
- **Timer**: Implements a wrapper for a hardware Timer in order to provide time-out functionality for the IoT-Reader (Non-OS) stack

Note: The delivered SDK contains a reference implementation for Platform component used on R7FA4M2AD application processor. This layer must be adapted/ported for different targets.

3. IoT-Reader API Description

This section contains an overview of the functions provided by the IoT-Reader API.

Note: A detailed description of all functions including parameters and types can be found in the file: **\SRC\COMPS\IOT_READER\ptx_IOT_READER.h**

3.1 ptxIoTRd_Init

Declaration	`ptxStatus_t ptxIoTRd_Init(`
	`  ptxIoTRd_t *iotRd,`
	`  ptxIoTRd_InitPars_t *initParams);`	

Description	Initialize software and hardware components for IoT-Reader operation. This function has to be called before any other API functions. It performs software initialization and configuration for the PTX chip.	
Input Parameters	iotRd	Pointer to stack component (to be allocated by user).
	initParams	NSC initialization parameters.
Return Value	Status	Success or failure, refer to **ptxStatus_t** for details.

3.2 ptxIoTRd_InitNSC

Declaration	`ptxStatus_t ptxIoTRd_InitNSC(`
	`  ptxIoTRd_t *iotRd,`
	`  ptxIoTRd_ComInterface_Params_t *initParams);`	

Description	Initializes the NSC component / the PTX chip.	
Input Parameters	iotRd	Pointer to initialized stack component.
	initParams	Pointer to communication initialization parameter structure
Return Value	Status	Success or failure, refer to **ptxStatus_t** for details.

3.3 ptxIoTRd_Update_ChipConfig

Declaration	**ptxStatus_t** ptxIoTRd_Update_ChipConfig (**ptxIoTRd_t** *iotRd, **uint8_t** nrConfigs, **ptxIoTRd_ChipConfig_t** *configParams);	
Description	Updates the RF- and System-Configuration parameters of the NFC hardware. This function allows to change RF and System configuration parameters at runtime.	
Input Parameters	iotRd	Pointer to stack component.
	nrConfigs	Number of RF/System configurations to set.
	configParams	Pointer to n-configuration parameters sets.
Return Value	Status	Success or failure, refer to **ptxStatus_t** for details.

3.4 ptxIoTRd_Get_Revision_Info

Declaration	**ptxStatus_t** ptxIoTRd_Get_Revision_Info (**ptxIoTRd_t** *iotRd, **ptxIoTRd_RevisionType_t** revisionType, **uint32_t** *revisionInfo);	
Description	Reads various revisions of system like software version, uCode-version, hardware revision, product ID, etc. *Note*: HW/Chip ID and product ID can only be read after successful call to ptxIoTRd_InitNSC. Product ID: 0x00 = PTX100x 0x01 = PTX105x 0x02 = PTX130x 0xFF = Unknown/Invalid product ID	
Input Parameters	iotRd	Pointer to stack component (to be allocated by user).
	revisionType	Revision type.
	revisionInfo	Pointer to variable holding revision information.
Return Value	Status	Success or failure, refer to **ptxStatus_t** for details.

3.5 ptxIoTRd_Initiate_Discovery

Declaration	**ptxStatus_t** ptxIoTRd_Initiate_Discovery (**ptxIoTRd_t** *iotRd, **ptxIoTRd_DiscConfig_t** *discConfig);	
Description	This function starts the RF Discovery procedure as defined in the NFC Forum.	
Input Parameters	iotRd	Pointer to stack component.
	discover config	Pointer to RF Discovery structure (if set to NULL, default values will be used).
Return Value	Status	Success or failure, refer to **ptxStatus_t** for details.

3.6 ptxIoTRd_Get_Card_Registry

Declaration	**ptxStatus_t** ptxIoTRd_Get_Card_Registry (**ptxIoTRd_t** *iotRd, **ptxIoTRd_CardRegistry_t** **cardRegistry);	
Description	Access the internal card registry. This function can be used to access the internal card registry to retrieve a card's detailed information.	
Input Parameters	iotRd	Pointer to stack component.
	cardRegistry	Pointer to pointer to keep reference to card registry.
Return Values	Status	Success or failure, refer to **ptxStatus_t** for details.

3.7 ptxIoTRd_Activate_Card

Declaration	**ptxStatus_t** ptxIoTRd_Activate_Card (**ptxIoTRd_t** *iotRd, **ptxIoTRd_CardParams_t** *cardParams, **ptxIoTRd_CardProtocol_t** protocol);	
Description	Selects/Activates a given card in case of multiple available cards.	
Input Parameters	iotRd	Pointer to stack component.
	cardParams	Pointer to card (within registry) to select/activate.
	protocol	RF protocol to activate.
Return Value	Status	Success or failure, refer to **ptxStatus_t** for details.

3.8 ptxIoTRd_Data_Exchange

Declaration	**ptxStatus_t** ptxIoTRd_Data_Exchange (**ptxIoTRd_t** *iotRd, **uint8_t** *tx, **uint32_t** txLength, **uint8_t** *rx, **uint32_t** *rxLength, **uint32_t** msAppTimeout);	
Description	Protocol-based or raw RF data exchange.	
Input Parameters	iotRd	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 Values	Status	Success or failure, refer to **ptxStatus_t** for details.

3.9 ptxIoTRd_Bits_Exchange_Mode

Declaration	**ptxStatus_t** ptxIoTRd_Bits_Exchange_Mode (**ptxIoTRd_t** *iotRd, **uint8_t** enable);	
Description	Enables/Disables the bit exchange mode required to call ptxIoTRd_Bits_Exchange.	
Input Parameters	iotRd	Pointer to stack component.
	enable	Enable/Disable flag.
Return Values	Status	Success or failure, refer to **ptxStatus_t** for details.

3.10 ptxIoTRd_Bits_Exchange

Declaration	**ptxStatus_t** ptxIoTRd_Bits_Exchange (**ptxIoTRd_t** *iotRd, **uint8_t** *tx, **uint8_t** *txPar, **uint32_t** txLength, **uint8_t** *rx, **uint8_t** *rxPar, **uint32_t** *rxLength, **uint32_t** *numTotBits, **uint32_t** msAppTimeout);	
Description	Exchanges a bitstream based on NFC-A technology.	
Input Parameters	iotRd	Pointer to stack component.
	tx	Pointer to buffer holding data bytes to send.
	txPar	Pointer to buffer holding parity bits to send.
	txLength	Length of tx and txPar buffers.
	rx	Pointer to buffer holding received bytes.
	rxPar	Pointer to buffer holding received parity bits.
	rxLength	Length of received bytes/parity bits.
	numTotBits	Total number of received bits.
	msAppTimeout	Application timeout.
Return Value	Status of operation	Success or failure, refer to **ptxStatus_t** for details.

3.11 ptxIoTRd_RF_PresenceCheck

Declaration	**ptxStatus_t** ptxIoTRd_RF_PresenceCheck (**ptxIoTRd_t** *iotRd, **ptxIoTRd_CheckPresType_t** presCheckType);	
Description	Executes a presence check method on ISO-DEP cards or NFC-DEP targets.	
Input Parameters	iotRd	Pointer to stack component.
	presCheckType	Presence check method type.
Return Value	Status of operation	-

3.12 ptxIoTRd_T5T_IsolatedEoF

Declaration	**ptxStatus_t** ptxIoTRd_T5T_IsolatedEoF (**ptxIoTRd_t** *iotRd, **uint8_t** *rx, **uint32_t** *rxLength, **uint32_t** msAppTimeout);	
Description	Sends an EoF-packet according to T5T protocol.	
Input Parameters	iotRd	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	-

3.13 ptxIoTRd_T3T_SENSFRequest

Declaration	**ptxStatus_t** ptxIoTRd_T3T_SENSFRequest (**ptxIoTRd_t** *iotRd, **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/FeliCa protocol.	
Input Parameters	iotRd	Pointer to stack component.
	systemCode	System code.
	requestCode	Request code.
	tsn	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	-

3.14 ptxIoTRd_Reader_Deactivation

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

3.15 ptxIoTRd_Update_ChipConfig

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

3.16 ptxIoTRd_Set_Power_Mode

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

3.17 ptxIoTRd_Get_System_Info

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

3.18 ptxIoTRd_SWReset

Declaration	`ptxStatus_t ptxIoTRd_SWReset ( ptxIoTRd_t *iotRd);`	
Description	Performs a soft-reset of the PTX chip.	
Input Parameters	iotRd/	Pointer to stack component.
Return Value	Status	Success or failure, refer to **ptxStatus_t** for details.

3.19 ptxIoTRd_Deinit

Declaration	`ptxStatus_t ptxIoTRd_Deinit ( ptxIoTRd_t *iotRd);`	
Description	Close the IoT-Reader component. This function closes the IoT and releases the resources used. It must be called as the last function before the stop of the library usage.	
Input Parameters	iotRd	Pointer to stack component.
Return Value	Status	Success or failure, refer to **ptxStatus_t** for details.

3.20 ptxIoTRd_Get_Status_Info

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

3.21 ptxIoTRd_ConfigHBR

Declaration	**short** ptxIoTRd_ConfigHBR (**ptxIoTRd_t** *iotRd, **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	-

3.22 ptxIoTRd_Set_RSSI_Mode

Declaration	**short** ptxIoTRd_Set_RSSI_Mode (**ptxIoTRd_t** *iotRd, **ptxIoTRd_RSSI_Mode_t** rssiMode, **uint8_t** *rssiRefreshPeriodInt);	
Description	Enables or Disables the RSSI Mode in State “**Ready**”.	
Input Parameters	iotRd	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	-

3.23 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	iotRd	Pointer to stack component.
	rssiValue	Current RSSI value.
Return Value	Status of operation	-

4. IoT-Reader API States

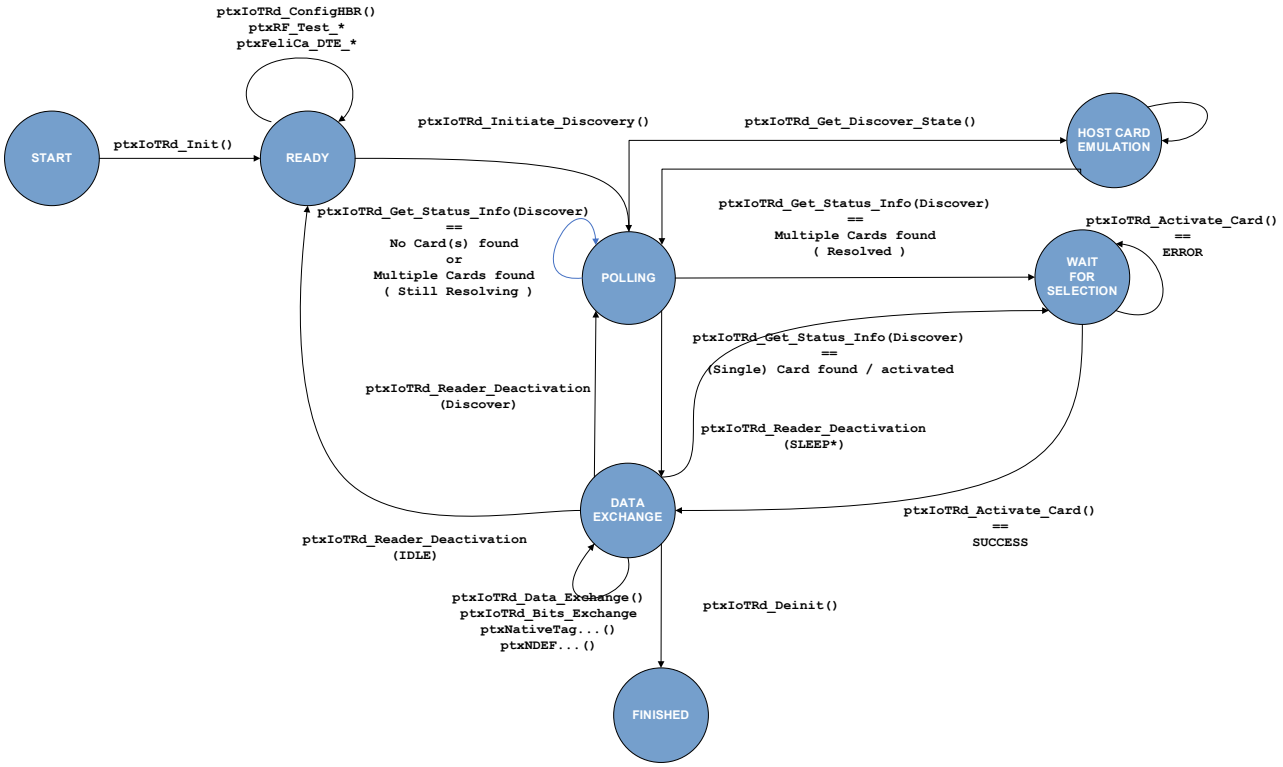


Figure 2. IoT-Reader API Flow Example

Note: All IoT-Reader API functions return a 16-bit status word indicating the status of the requested operation. If an operation succeeded, the status word is set to 0x0000 (= SUCCESS). In any other case the upper 8 bits of the status word indicate 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 source file “**ptxStatus.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.

4.1 State Descriptions

- **State: Start**
  The system must be initialized first via a call to "**ptxIoTRd_Init**". This will initialize the software Stack and the PTX chip to be ready for NFC operations.

- **State: Ready**
  After IC initialization, the IoT-Reader API is ready to poll for cards in the field via a call to "**ptxIoTRd_Initiate_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”)

  Note: The call to "**ptxIoTRd_Initiate_Discovery**" is non-blocking; that is, 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 PTX chip supports detection of critical errors like overcurrent and temperature. In case such an error occurs, the PTX chip 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)**".

  It is strongly 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_ChipConfi**g
  - **ptxIoTRd_Set_Power_Mode**
  - **ptxIoTRd_Get_Revision_Info**
  - **ptxIoTRd_*_RSSI_***
  - **ptxFeliCa_DTE*** (except API/component (un)initialization
  - **ptxRF_Test*** (except API/component (un)initialization

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

  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: Test** (Optional)
  This optional state allows the application to perform various system/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 Figure 3.

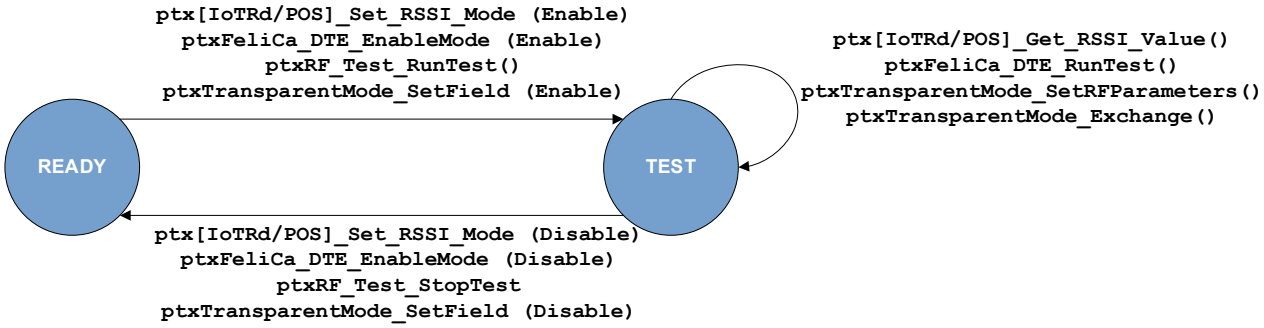


Figure 3. State: Test

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

  This function returns that:

  - 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 "**ptxIoTRd_Reader_Deactivation**".

- **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_Get_Event**" 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: When 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/activate a specific card or 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 must 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 PTX chip, all other (NFC Forum) protocols like T2T, T3T, and T5T must be handled by the application/add-on APIs on top.

  The API contains additionally the following functions which may 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 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 delivered demo application provides a few examples of how the different RF technologies and protocols are handled for single and multiple available cards. 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. like with cards, a specific protocol can be selected via a call to "**ptxIoTRd_Activate_Card**" in state "**WAIT FOR SELECTION**"

  Note: The default delivery of the IoT API activates the LLCP-protocol on top of NFC-DEP. Like in state "**Polling**", completed RF exchanges can be stopped via a call to "**ptxIoTRd_Reader_Deactivation**".

- **State: Finished**
  Once the complete IoT-Reader application should be stopped or shut down, it is required to call function "**ptxIoTRd_Deinit**" to free previously allocated system resources like memory, drivers, etc.

5. HCE API Description

This section contains an overview of the functions provided by the HCE API from the **\COMPS** folder.

Note: The HCE API consists of additional functions which 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.	
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 IoT-Reader API and ready to use when the IoT-Reader 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 use the functions of the HCE component also directly (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 IoT-Reader API state machine. All transitions between the states are done via "**ptxHce_Get_Event()**".

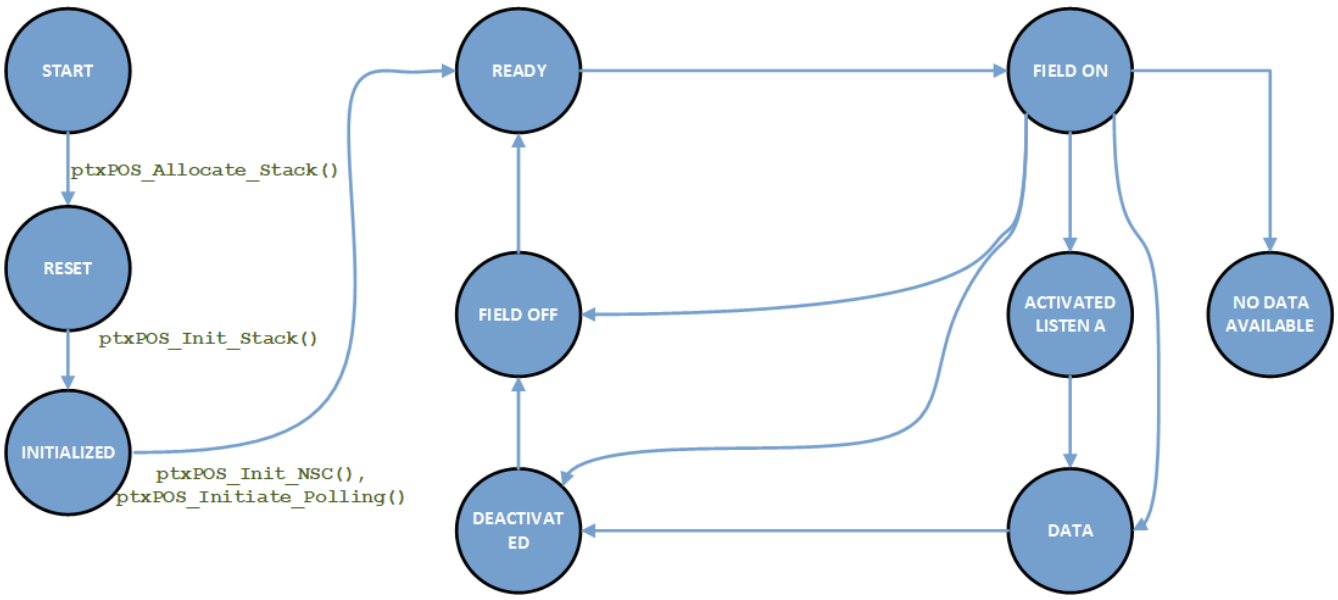


Figure 4. HCE API Flow Example

6.1 State Descriptions

- **State: Field On**
  The message queue contains this state if the PTX chip detects an external field from a reader. The PTX chip can now receive and send data.

- **State Field Off**
  The message queue contains this state if the PTX chip 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 PTX chip 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. IoT-Reader (Non-OS) SDK Delivery

The IoT-Reader (Non-OS) SDK delivery contains the source code for the IoT-Reader API and the NSC (Non-OS) Stack. A demo application is also included in the delivery in order to show how to communicate to various cards/tags via different protocols.

The R7FA4M2AD (Cortex-M33 based MCU) has been the platform used for reference implementation provided in the SDK. More detailed overview of platform specific files is given in section 8.

Important: The provided IoT Reader demo application project is based on Renesas FSP Platform. To build the project and execute the application, Renesas Flexible Software Package (FSP) and Renesas e2 studio development tool is required to be installed. Installation of those tools is out of scope of this document. Refer to the vendor website.

Although platform-specific code for R7FA4M2AD microcontroller is not included in the delivery, it can be easily generated after importing and building the project in Renesas e2 studio IDE. The SDK also contains a tool enabling the update of the RF Configuration used by the PTX1xxR. For more information, see section 9.

The SDK structure at root folder is:

- **Root Folder**

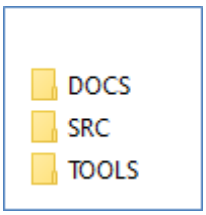


Figure 5. IoT-Reader (Non-OS) SDK Folder Structure

- **\DOCS**
  Doxygen-based API-description

- **\SRC**
  This folder contains the source code for both Application Example and IoT-Reader (Non-OS) Stack including add-on APIs.

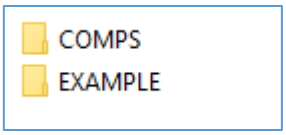


Figure 6. SRC Folder Structure

Important: The file "**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.

- **\TOOLS**
  PLAT contains the project to be imported in Renesas e2 studio.

Figure 7. Tools Folder Structure

- **\TOOLS\PLAT\RENESAS\RA4M2\E2STUDIO_WORKSPACE\PTX_IOTRD**
  This is the location where project files are located. When importing project in Renesas e2 studio, just point to this location. All the files required to build the project are included.

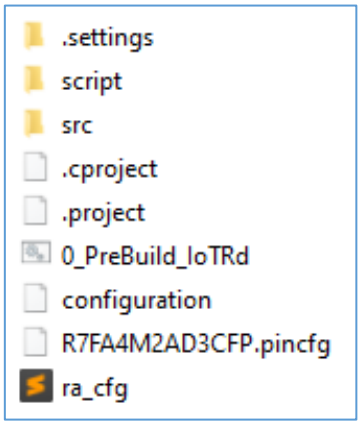


Figure 8. Project Files

8. Platform Specific

The platform/MCU dependent code has been encapsulated in the PLAT Component. This section describes the platform-specific code and project settings used for R7FA4M2AD as the reference implementation. This reference code can be used as guidance for porting the platform-specific code to any other platform/MCU used as application processor, in which case a general rule should be followed: keep PLAT API declarations the same while changing functions (or data structure) implementation to fit specific MCU requirements.

RA4M2 MCU-specific third-party code is not included in this delivery since the source code is automatically generated during every build process.

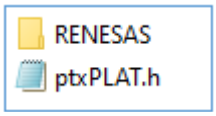


Figure 9. PLAT Folder Structure

The following items can be found in the PLAT folder:

- **\RENESAS**
  Folder containing source code specific for R7FA4M2AD platform
- **ptxPLAT.h**
  Platform API definition.

Figure 10 shows the content of RENESAS folder.

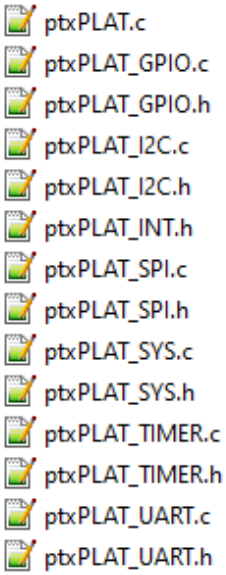


Figure 10. RENESAS Folder Structure

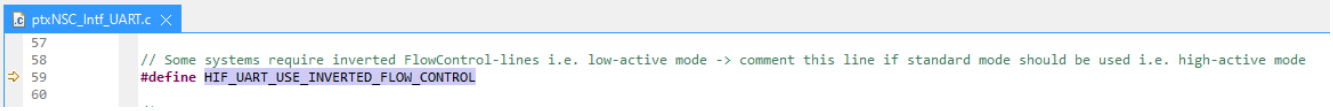
In the RENESAS folder there are the following items:

- **ptxPLAT.c**
  Platform-specific wrapper that adapts the PLAT API functions for the submodules: TIMER, GPIO and the host interfaces (I^2C, SPI and UART).
- **ptxPLAT_GPIO.h**
  Platform-specific header where the GPIO-IRQ set of functions are defined.
- **ptxPLAT_GPIO.c**
  Platform-specific implementation of the GPIO-IRQ for R7FA4M2AD.
- **ptxPLAT_INT.h**
  Platform-specific header where the PLAT component for R7FA4M2AD implementation is defined.
- **ptxPLAT_SPI/_I2C/_UART.h**
  Platform-specific header where the SPI/I2C/UART features are defined.
- **ptxPLAT_SPI/_I2C/_UART.c**
  Platform-specific implementation for SPI/I2C/UART features.

***Important**:* Depending on the used host-interface, a global compile switch / #define must be set depending on the used host-interface which is either “**PTX_INTF_SPI**”, “**PTX_INTF_I2C**”, or “**PTX_INTF_UART**”. If the switch / #define is not used, the provided demo code will not compile.

***Important**:* Using the UART-interface also requires using the Flow Control (RTS and CTS, see Datasheet). Some target systems may require changing the polarity of the Flow Control pins, which requires also to set/unset a preprocessor #define in the stack.

The #define is called **HIF_UART_USE_INVERTED_FLOW_CONTROL** (file “**ptxNSC_Intf_UART.c**”) and if set, the Flow Control pins are inverted (default) value.



- **ptxPLAT_TIMER.h**
  Platform-specific header where the Timer features are defined.

- **ptxPLAT_TIMER.c**
  Platform-specific implementation for Timer features.

8.1 Project Quick-Start Guide

As discussed in section 5, to build the delivered project and execute the demo application on EK-RA4M2 (RA4M2 MCU development board), Renesas Flexible Software Package (FSP) and Renesas e2 studio development IDE must be installed.

Step 1: Import project into workspace with *File >Import.*

Extract the archive file (.zip) containing the SDK first to a local folder and import the project via *File >Import* select “Existing Projects into Workspace”. In the next window, select the option “Select root directory” and point to the root folder with the extracted SDK. The project located in **\TOOLS\PLAT\RENESAS\RA4M2\E2STUDIO_WORKSPACE\PTX_IOTRD** gets found automatically by the tool. After clicking the “Finish” button at the bottom of the Window, the SDK gets finally imported and is ready to be built.

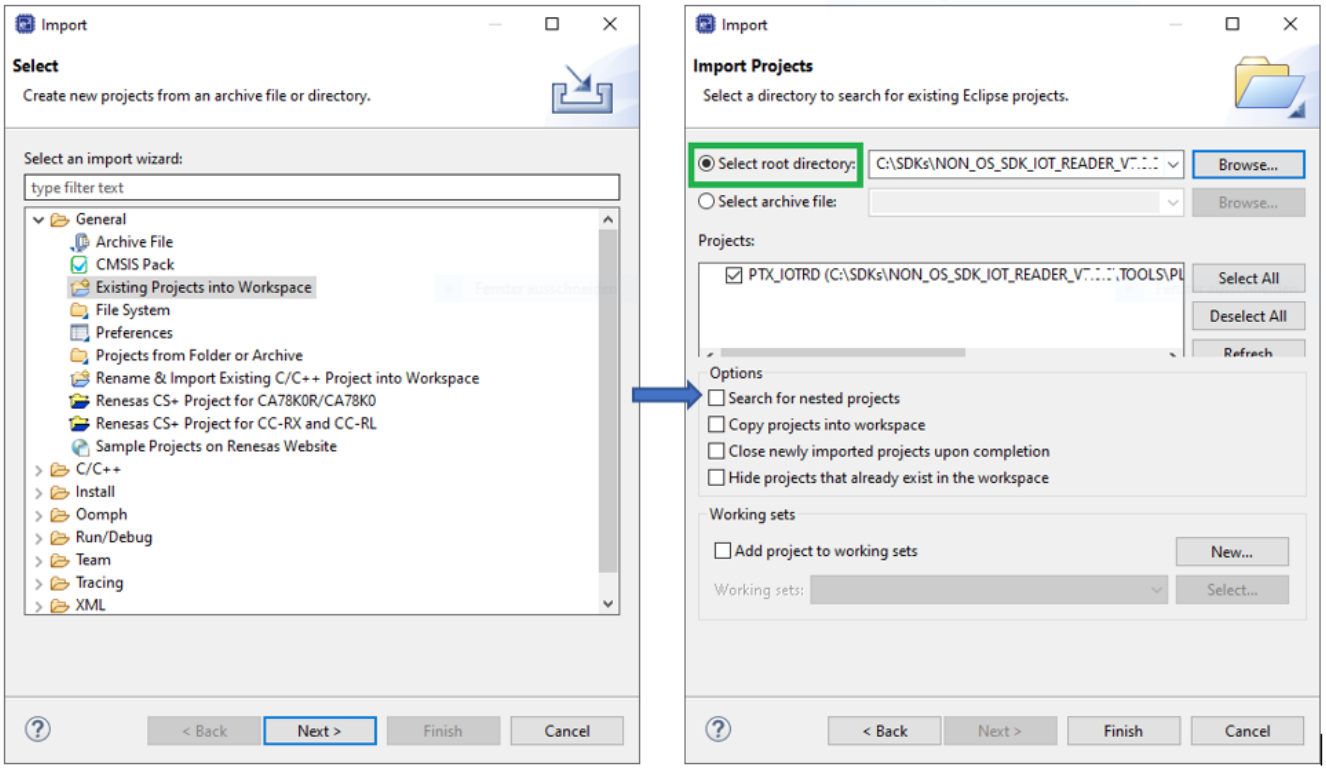


Figure 11. Importing the Project into Workspace

After successfully importing the project, the contents of the Project Explorer in e2 studio is similar to what is shown in Figure 12 (a).

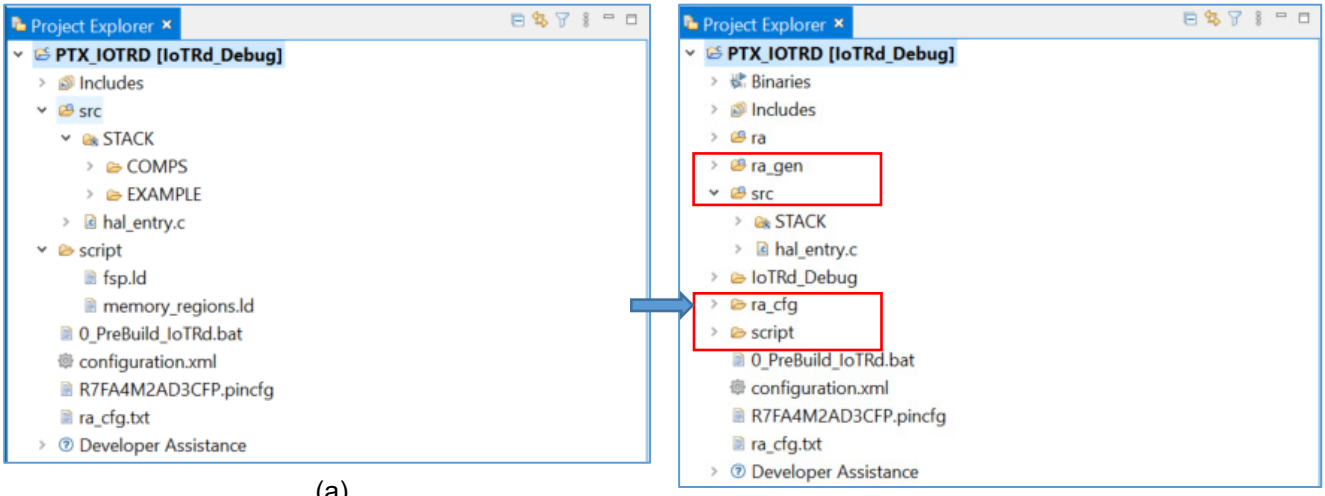


Figure 12. Project Explorer Contents

Note: (a) After Importing, Before Build; (b) After the Build.

The following items can be found in Project Explorer:

- **src/STACK**
  Contains SDK source files. However, project is organized so that source files are used as linked resources. This means that source files are not physically located in workspace folder but, are located in the SDK´s SRC folder as described in section 5.

- **hal_entry.c**
  This is the file containing the call to demo application.

- **script/fsp.ld**
  Linker script for R7FA4M2AD MCU.

- **0_PreBuild_IoTRd.bat**
  Batch file used in pre-build process.

- **configuration.xml**
  Configuration file used/modified in FSP Configuration tool to set up various platform related features. Based on these settings project content will be generated (board configuration and drivers source files). Hint: double-clicking on this file loads the FSP Configuration tool.

- **R7FA4M2AD3CFP.pincfg**
  Pin configuration file containing current MCU pins configuration. This is also updated in FSP Configuration tool.

- **ra_cfg.txt**
  File presenting current MCU and board configuration.

Step 2: Build Project

Select build configuration and click *Project* > *Build Projec*t.

There are multiple build configurations available. All of them will generate a **.elf** binary file (named of the build-configuration) which is then used to download into MCU.

In Figure 13, folders marked in red are automatically generated with every build process:

- **ra_gen**, **ra** and **ra_cfg** folders contain third-party source code which includes support for all used (configured) MCU and board features for example, clock settings, connectivity drivers, timers, IO drivers, etc.

  Example:

  IoTRd_App_SPI_Debug folder contains build output files for **IoTRd_App_SPI_Debug** build configuration. If another build configuration is chosen, a new folder with the same as build configuration will be created.

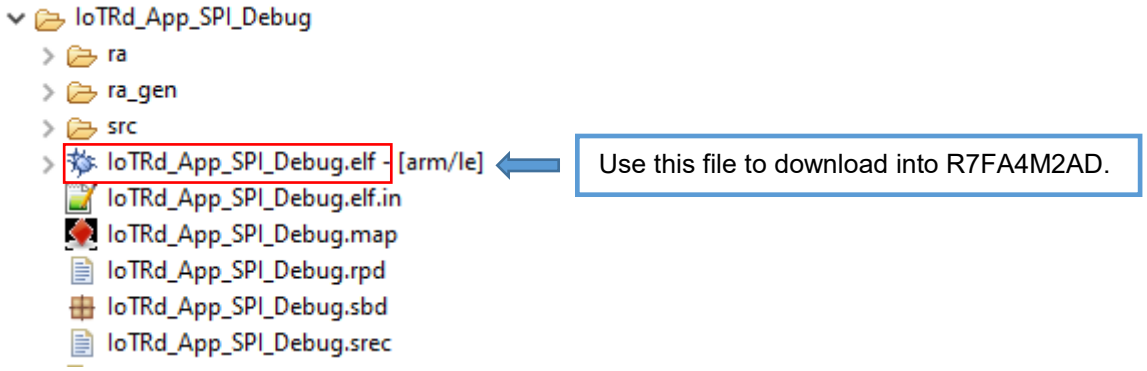


Figure 13. Build Output Folder

Step 3: Debugging

As already discussed, project has been developed based on the R7FA4M2AD MCU and deployed on the EK-RA4M2 development kit board. For this specific board, debugging is started as follows:

1. Connect the board´s J-Link OB port to USB on host PC
2. Select **.elf** binary to download to MCU
3. Click Run > Debug As > Renesas GDB Hardware Debugging
   a. Select J-Link ARM
   b. Select target R7FA4M2AD from the list

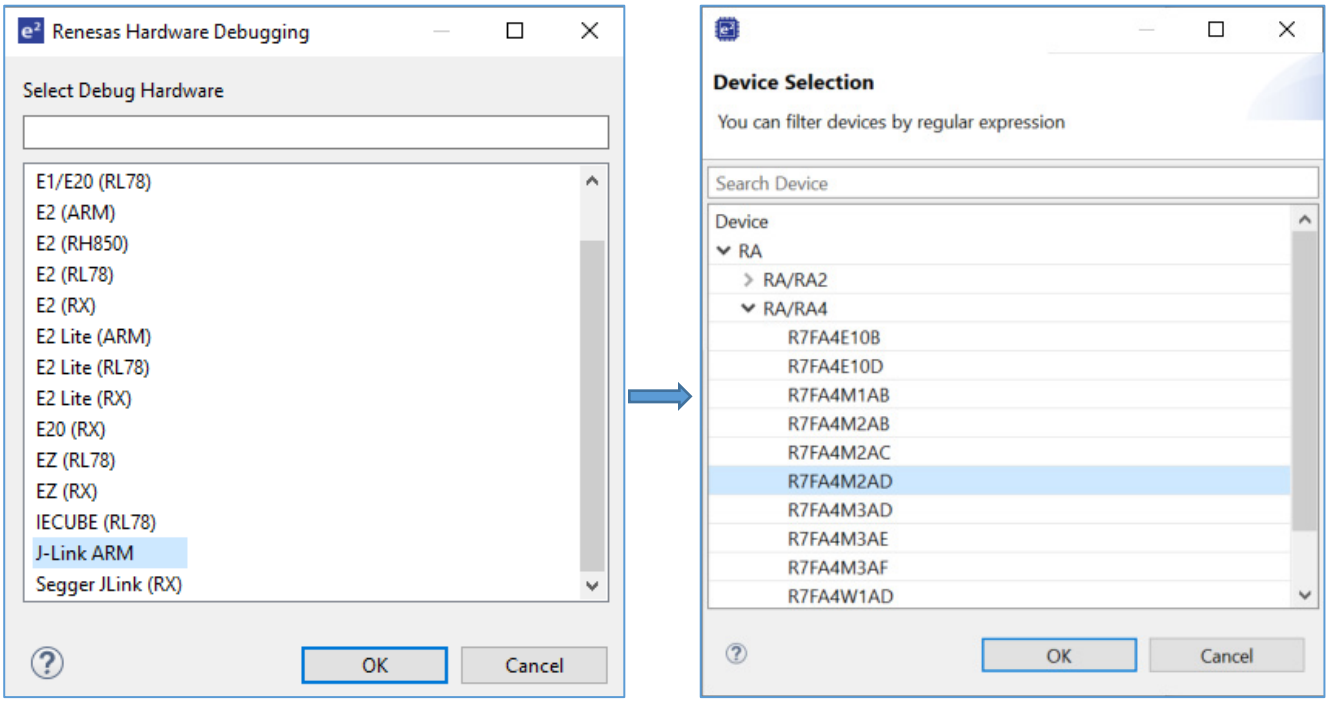


Figure 14. Selecting Debug Target

Important: The provided SDK contains all the required files and project settings to build and execute application on EK-RA4M2 board by following the steps described in this section. If a different board, a different host interface or a different R7FA4M2AD MCU configuration will be used to evaluate provided demo application, pins and peripherals of the MCU are or might need to be reconfigured. This means a different **.pincfg** file has to be provided, or pins and peripheral usage has to be set appropriately in FSP Configurator by opening and modifying **configuration.xml** file. Detailed procedure how this is done is provided in vendor´s documentation and is out of scope of this document.

- **Current SDK Pin Assignment (Renesas RA4M2-EVK)**

Interface	Pins	Comment
SPI	MOSI = P4.11 MISO = P4.10 SCK = P4.12 CS = 4.13	Requires Pin-Config RA4M2 EK SPI in **configuration.xml**.
I^2C	SDA = P4.09 SCL = P4.08	Requires Pin-Config RA4M2 EK I2C in **configuration.xml**.

Interface	Pins	Comment
UART	RXD = P2.06 TXD = P2.05 CTS = P4.08 RTS = P4.01	Requires Pin-Config RA4M2 EK UART in **configuration.xml**. CTS and RTS connected using 10K pull-down resistors.
GPIO (IRQ)	P4.14	Only required for I2C and SPI.
Console UART	RXD = P1.13 TXD = P1.12	Optional, only for Console-Output.

8.2 Porting Guide

8.2.1 Introduction

As described in the introduction to section 8, the PLAT component encapsulates all the hardware platform-dependent code and serves as the “Hardware Abstraction Layer“ (HAL) of the overall software-architecture as shown in the following figure.

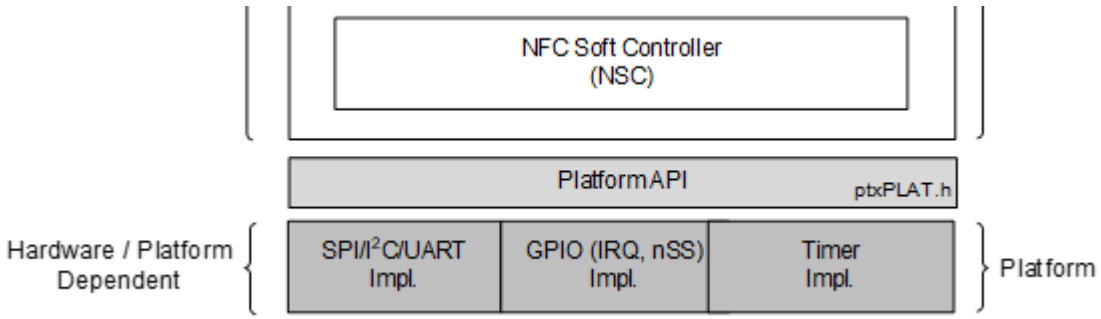


Figure 15. System Architecture

The higher software-layers are only using API functions from the PLAT layer that are defined in “**ptxPLAT.h**” and are agnostic to the concrete implementation of the PLAT component itself.

When the SDK (in other words, the PLAT component) should be ported to a new hardware-platform, all API functions from “**ptxPLAT.h**” must be implemented/ported.

“**ptxPLAT.h**” itself exposes the following low-level functionalities for the upper software layers:

- Host-Interface access (for example, SPI, I^2C, or UART)
- GPIO access (for example, IRQ pin used for SPI and I^2C, NSS pin for SPI)
- Timer access (for example, general purpose timers to implement delays, timeouts, etc.)
- Interrupt controlling

The SDK contains example/reference implementations of the PLAT component based on the Renesas RA4M2-MCU using the EK-RA4M2 eval-kit covering all above listed low-level functionalities.

8.2.2 PLAT Architecture

Figure 16 shows the architecture/structure of the PLAT component.

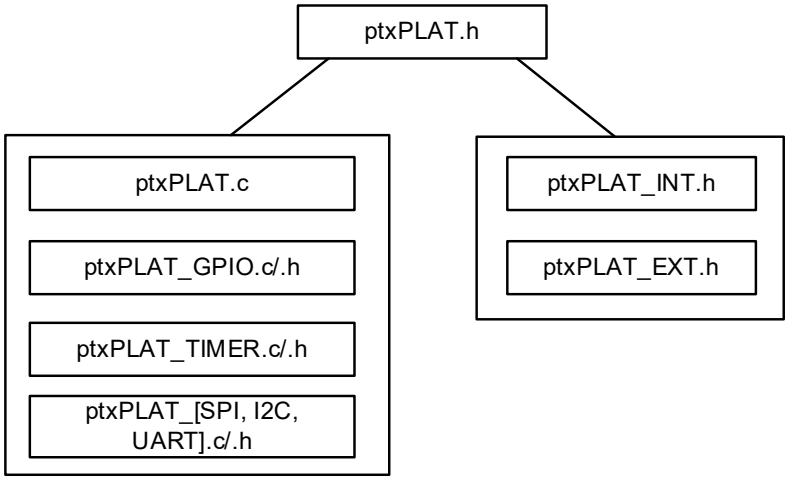


Figure 16. PLAT Component Architecture

The central part of the PLAT component is ptxPLAT.h which exposes all required low-level functionalities to the upper software layers.

"**ptxPLAT.c**" implements already some of the basic API functions defined in "**ptxPLAT.h**", such as:

- Component (de-)initialization
- IRQ control handling
- Dispatching the host-interface related API calls by using #defines as described in section 7 (see **PTX_INTF_SPI, PTX_INTF_I2C** or **PTX_INTF_UART**)
- Timer related API calls
- Checking for new available Rx data from the PTX1xxR chip

Despite being dependent on the underlying hardware, most of the API functions implemented in "**ptxPLAT.c**" are common functions that just refer to API functions from the GPIO/Timer and Host-Interface components. Therefore, it should not be needed to change big portions of this file except a few exceptions like enabling/disabling IRQs or MCU-related operations. Changes are however possible/allowed (for example, code optimization reasons).

All API functions of the PLAT component use a component specific data structure called "**ptxPlat_t**" which is defined in "**ptxPLAT_INT.h**". This data structure contains a minimum set of base variables/members which are uses across the entire PLAT component. If proprietary variables are needed (or any kind of other user context), this data structure should be used for that.

"ptxPLAT_EXT.h" contains #include-directives and compiler-specific disabling of warnings for third-party code which is outside the scope of the SDK provided by PTX.

The three modules "**ptxPLAT_GPIO.c/.h**", "**ptxPLAT_TIMER.c/.h**" and "**ptxPLAT_[SPI, I2C, UART].c/.h**" contain the concrete low-level API-functions which are then commonly called within "**ptxPLAT.c**". These low-level API functions must be implemented/ported and require – depending on the underlying hardware – the most adaptations. To simplify the implementation/porting process, the SDK contains reference implementations that should be used as an example. Some parts of the implementation can even be reused.

8.2.3 PLAT Host-Interface Implementation

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 PLAT component must support both communication types by implementing mechanisms to do the following:

- Synchronously send and receive data to / from the PTX1xxR
- Asynchronously receive data from the PTX1xxR

This mechanism is supported by the PLAT API function "**ptxPLAT_TRx (…)**", which takes parameters for Tx and Rx operations. As mentioned above, the PLAT implementation must support separated Tx and Rx operations by setting either the Tx or Rx parameters to NULL/0. This allows to use the API function to be used for both communication types.

To detect if the PTX1xxR hardware wants to send data to the host (both communication types), the PLAT 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.
- **UART**
  This host-interface works in push-mode (for example, data can be sent at any time to the host after chip initialization). It is the responsibility of the implementation in "**ptxPLAT_UART.c/.h**" to collect the incoming data and to provide means to signal to the upper software layers that the received data/frame is complete.

  Note: UART Rx-frames from the PTX1xxR are always starting with 1 byte length information followed by the actual data -> [LEN] [DATA …].

  A frame is complete when the number of received DATA bytes equals the value in [LEN]. For more information on the structure of UART frames, refer to References item [6].

8.2.3.1 Example: Exchanging Commands with PTX1xxR

Figure 17 shows a simplified example how the SDK exchanges commands and responses with the PTX1xxR. The application on the host calls an API function (for example, Discover command) from the SDK (valid for POS or IoT SDK). Every API function call that requires access to the PTX1xxR calls a function from the NSC library which handles all the calls to the PLAT component.

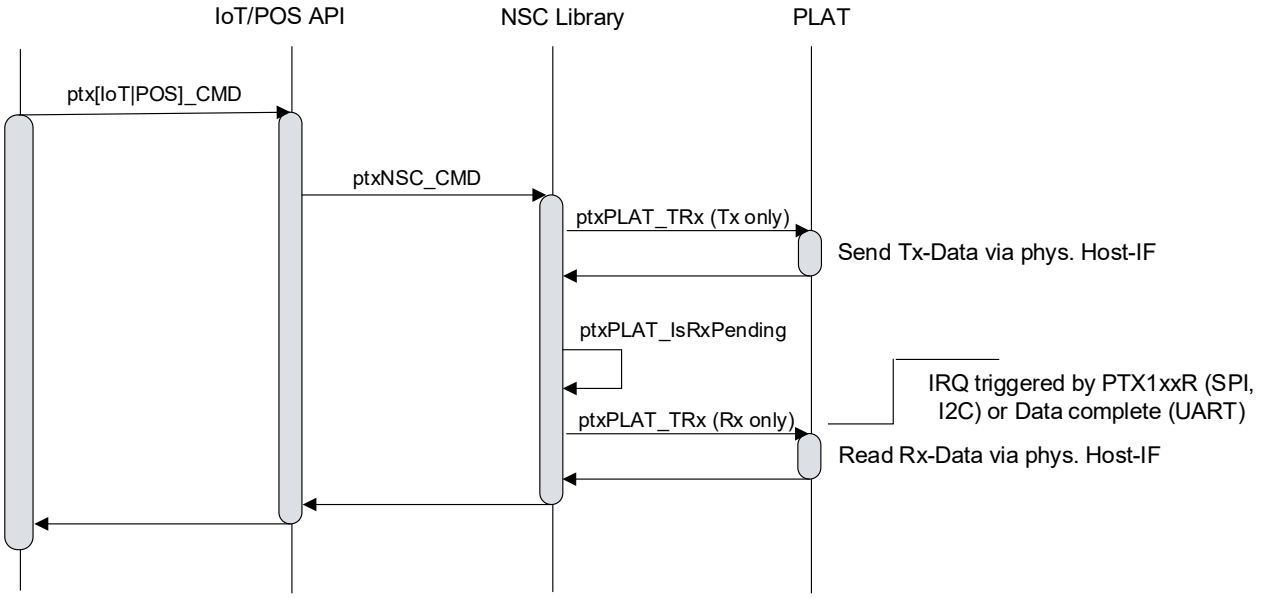


Figure 17. Synchronous Command Handling (Simplified)

In this case, the NSC library function first transmits the command to the PTX1xxR by calling the "**ptxPLAT_TRx**" function with only Tx-parameters provided. Afterwards, it waits until it receives a response from PTX1xxR by calling "**ptxPLAT_IsRxPending**" by implementing/handling one of the mechanisms described above for SPI, I^2C, or UART.

When data is available, the NSC library calls again the "**ptxPLAT_TRx**" function but only with Rx parameters provided to read the actual data from the PTX1xxR.

8.2.3.2 Example: Receiving Events/Notifications from PTX1xxR

Figure 18 shows a simplified example of how the SDK and in particular the PLAT component reads events/notifications from the PT1xxR. Atypical example for an event/notification is the status whether a card was found and activated during the RF-discovery process or an RF error occurred.

The detection if PTX1xxR has data to send:

- Is managed internally by the SDK API functions (POS or IoT) or the NSC library, or
- Can implicitly be triggered by the application through various API calls (see below)

When the PLAT component gets initialized, the upper software layers provide a callback function which is called when an event/notification or RF data is available from the PTX1xxR.

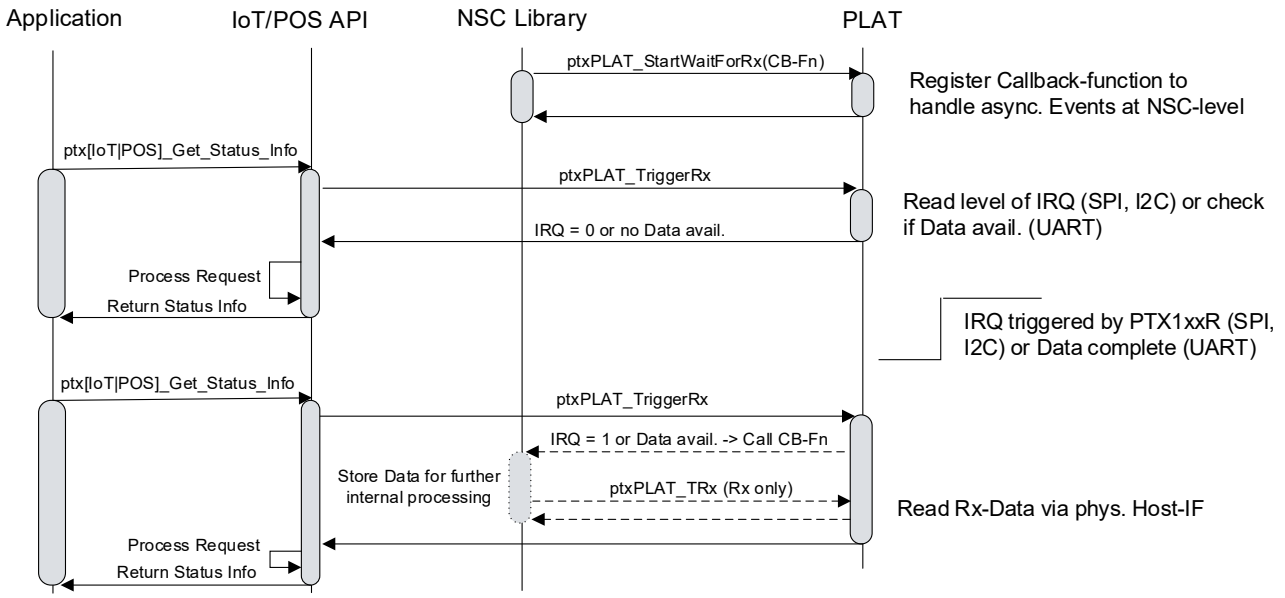


Figure 18. Asynchronous Command-Exchange (Simplified)

To check if new data is available, many of the SDK API functions and/or the NSC library functions are internally calling "**ptxPLAT_TriggerRx**". This function works in a similar manner as "**ptxPLAT_IsRxPending**" which is used for exchanging commands and responses. If data is available, the PLAT component will call the initially registered callback function implemented in the NSC library.

The callback function calls the "**ptxPLAT_TRx**" function, but only with Rx parameters provided to read the actual data from the PTX1xxR and stores the data internally for further processing.

Apart from other SDK API or NSC library functions, one important example of an SDK API function which calls "**ptxPLAT_TriggerRx**" is "**ptx[IoTRd | POS]_Get_Status_Info**" (further referenced as "**ptxGet_Status_Info**") as shown in Figure 18. Assuming the application has started the RF-Discovery process and frequently polls for new cards by calling "**ptxGet_Status_Info (Discover)**". Every time it gets called, "**ptxPLAT_TriggerRx**" gets called internally before it returns the status whether a card was found or not. During the first call in Figure 18, no event (IRQ or data) is available, but immediately afterwards. When "**ptxGet_Status_Info (Discover)**" gets now called the second time, "**ptxPLAT_TriggerRx"** sees that an event is pending and calls the callback function which then reads the data from PTX1xxR and stores the event data internally for further processing. In this example, it could have been the event "Card Type x Activated" which gets then returned by "**ptxGet_Status_Info (Discover)**".

Important: In order not to miss any asynchronous events from the PTX1xxR, it is crucial that the application regularly calls "**ptxGet_Status_Info**" in its main loop or thread. It is strongly recommended to check regularly for critical system errors via "**ptxGet_Status_Info (System)**" as mentioned in section 4.

8.2.4 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, I^2C 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 test 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 100kHz/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

Important: The provided implementations for "**ptxPLAT_TRx**" and the concrete implementations for SPI, I^2C, and UART implement the third-party drivers in a blocking manner. If blocking behavior is not suitable for the given target system, adaptations to the implementation may be required.

Example:
When sending data to the PTX1xxR using "**ptxPLAT_TRx**", a "Write" function from the third-party driver is internally called to send out the data on the physical bus. Once all data was sent out, the driver generates an IRQ and calls a callback function in the ISR-context to signal a "Tx Complete" event. The "**ptxPLAT_TRx**" function waits for this "Tx-Complete" event in a blocking manner by polling until the event status changes and resumes afterwards. Depending on the used host-interface, the same or a similar mechanism is implemented for "Rx-Complete" events (see previous section).

If the current blocking/polling type of implementation is not suitable for the available target system, other mechanisms can be implemented as well as shown in the following examples:

- The polling mechanism can be replaced by putting the host into stand-by-mode while waiting for the event and wake-up when an IRQ occurs, thus avoiding high CPU load.
- In an RTOS environment, a semaphore can be used which suspends/pauses the current thread while waiting for the event ("consume") and resumes when an IRQ occurs ("produce").
- The blocking behavior of "**ptxPLAT_TRx**" may be replaced by:
  - Adding user-specific callback functions after the Tx/Rx-operation, thus giving the application all means to either wait for the end of an operation or execute other code parts.
  - Changing the function in a way that it returns immediately after the Tx/Rx operation. In addition, the PLAT component would need to be extended by an additional API-function to check whether the Tx/Rx operation has finished. This would also require some adaptation in the higher software layers. This would allow an asynchronous implementation of "**ptxPLAT_TRx**".

8.2.5 Implementation Suggestions for PLAT Layer

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

- **I^2C**
  The PTX1xxR hardware implements a few low-level host commands which require to send a repeated-start condition on the I^2C bus during a data exchange via the PLAT API function "**ptxPLAT_TRx**". This is exemplarily shown for the RRA-operation in Figure 19 (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. When the read operation is done, the host stops the data exchange by sending a stop condition on the I^2C bus.

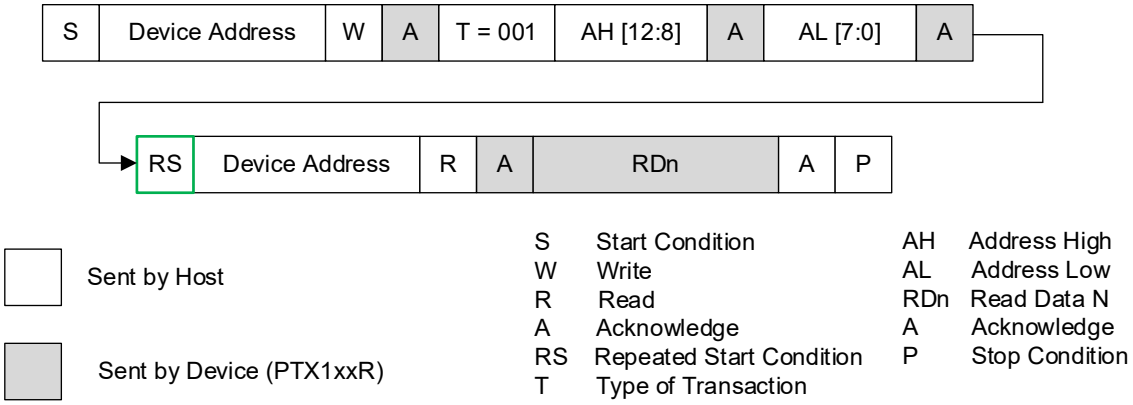


Figure 19. PTX1xxR RRA-Operation using I²C (Simplified)

The handling of when a repeated-start condition is required for a specific HW-command is completely managed by the upper layers of the SDK and is controlled through the parameter "**flags**" of "**ptxPLAT_TRx**" . One bit of this parameter is used to indicate to the lower layer PLAT I²C implementation if a repeated-start condition is necessary during the call to "**ptxPLAT_TRx**".

As "**ptxPLAT_TRx**" must be implemented / ported by the user, care must be taken to tell the target platform-dependent I²C 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: Unfortunately, many target platform driver implementations and its API interfaces are using different terms/ways for handling repeated start conditions or no stop conditions, such as:

- Not sending a stop condition for the current I²C 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 19 can be achieved with each of the mentioned examples but may need some adaptations in the logic of the actual PLAT I²C implementation.

- **UART**
  The provided reference low-level UART implementation in "**ptxPLAT_UART.c**" is based on a circular buffer implementation to store all the received messages from the PTX1xxR hardware.

  UART Messages are defined as follows:

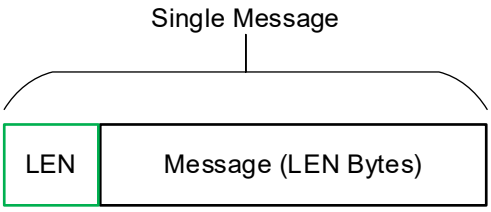


Figure 20. UART Message Format

The first byte "LEN" defines the length of the message, and the remaining "LEN"-bytes define the actual message itself. The circular buffer implementation allows to concatenate multiple messages consecutively as shown in the following example.

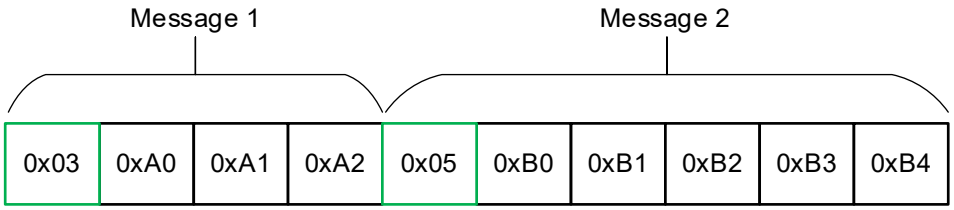


Figure 21. Concatenated UART Messages

In this example, the circular buffer has a total size of 10 bytes which exemplarily contains two messages.

The first message has a length of 3 bytes and payload value range from 0xA0 to 0xA2 and the second message following right after has a length of 5 bytes and payload value range from 0xB0 to 0xB4.

If the circular buffer would just consist of 8 bytes, the 2 remaining bytes would overwrite the first 2 bytes of the first message as shown below.

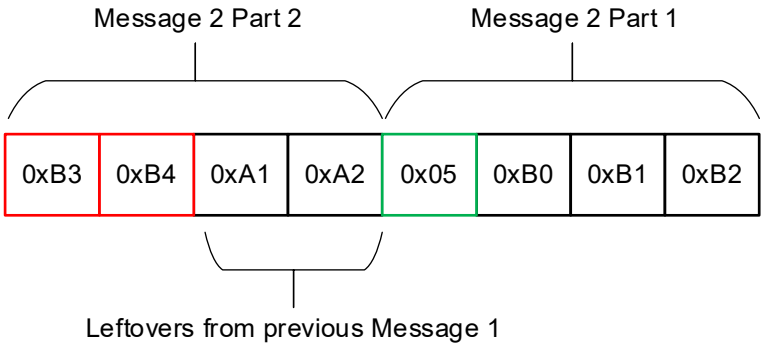


Figure 22. UART Messages with Circular Buffer Overflow

With every received byte from the UART, the upper layers in the SDK check if a message has been completely received (for example, via call to "**ptxPLAT_(UART)_TriggerRx**").

If a message is complete, the upper layers in the SDK calls "**ptxPLAT_GetReceivedMessage**" to copy the next pending (complete) message into a dedicated buffer for further processing. This is happening in the background while the next bytes can be received from the UART.

As the UART interface is relatively slow compared to the speed of the host-processor, the first message would already be completely processed while the reception of the second message is still ongoing, therefore the overflow (in other words, overwriting) of the bytes from a previous message is not an issue.

As "**ptxPLAT_(UART)_GetReceivedMessage**" must be implemented/ported by the user, it is strongly recommended to:

- Re-use the approach of a circular buffer implementation (as provided in the reference implementation) by keeping track of the current message index as well as the index of the currently received byte. Furthermore, the implementation must be able to handle circular buffer overflows.
- Use a size of at least 256 bytes for the circular buffer.

If an implementation other than a circular buffer approach is chosen (not recommended), adaptations are needed in the upper SDK layers as well, like in function "**ptxNSC_GetRx**" in "**ptxNSC_Intf_UART.c**".

8.3 Integration Notes/Suggestions

8.3.1 Performance Optimization

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

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

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

8.3.2 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 (further referenced 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)
```

8.3.3 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 IoT-Reader API component.

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

8.3.4 Reference Implementation–Code Size and Memory Consumption

The reference implementation provided with the SDK was exemplarily built (release version) on a Renesas RA4M2 system using GCC V9.3.1 which produces the following output in terms of code size and memory consumption.

Memory Type (Target IoTRd_App_xxx_Release)	Size (SPI)	Size (I^2C)	Size (UART)
PTX Code with third-party code [1]	~ 53KB	~ 52KB	~ 52KB
PTX Code without third-party code	~ 39KB	~ 39KB	~ 39KB
Total Memory Consumption (Data + BSS, Example Application)	~ 3KB	~ 2,8 KB	~ 2,9 KB
Stack Consumption (Example Application)	~ 3,4 KB	~ 3,4 KB	~ 3,4 KB

1. The third -party code consists of low-level drivers and libraries to access peripherals on the RA4M2 platform such as host interfaces (SPI, I^2C, …), GPIOs, timers, etc.

***Important**:* The values listed in the table are reference values. 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.

8.3.5 Switching Reference Host-Interface Implementation

The SDK contains example/reference implementations for the three supported host-interfaces SPI, I²C, and UART for the Renesas RA4M2-platform based on the EK-RA4M2 evaluation kit.

To switch between the reference host-interface implementations in E2 Studio, complete the following steps:

1. Open file **configuration.xml** (1), select tab “Pins” (2) and select the targeted pin configuration via the drop-down menu (3).

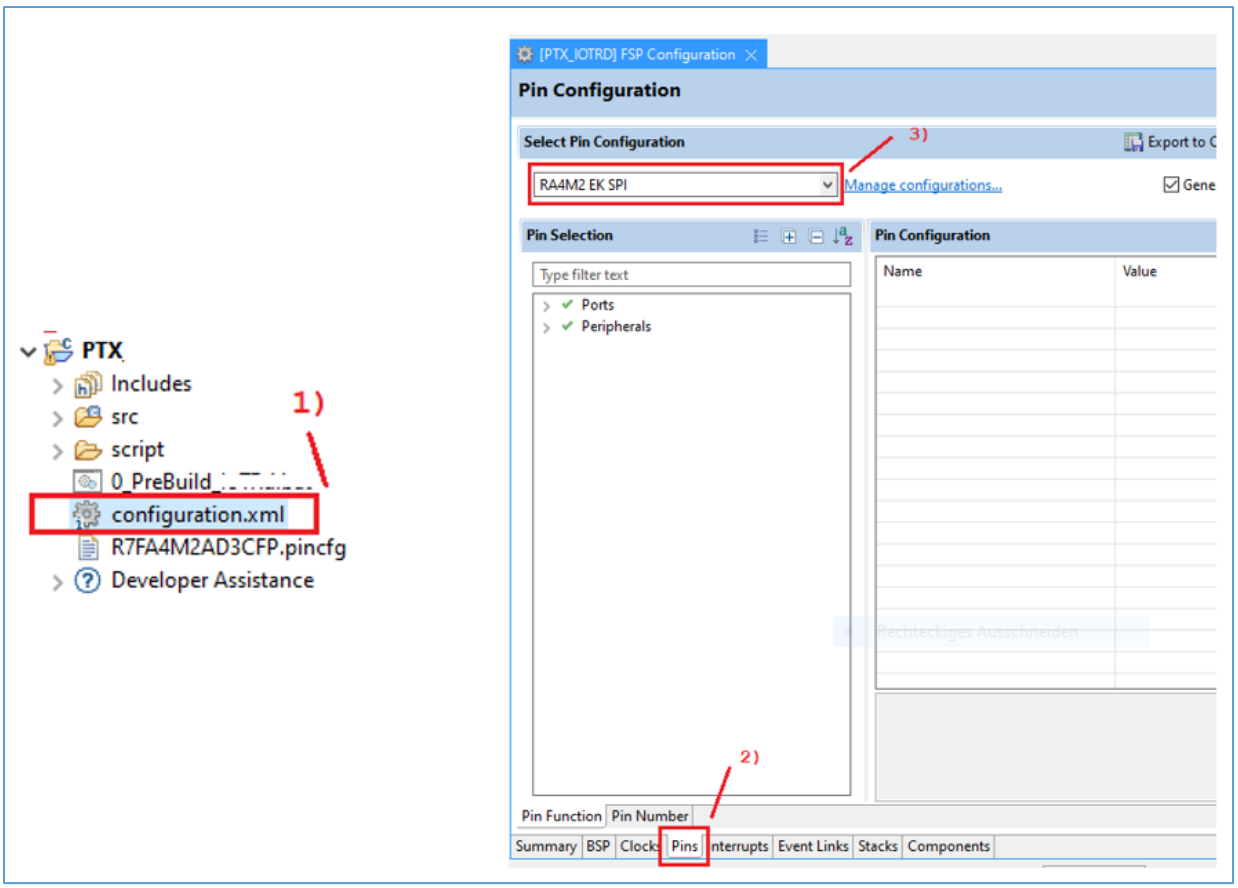


Figure 23. Pin Configuration in “configuration.xml”

2. Set project-wide/global pre-processor definition **PTX_INTF_SPI**, **PTX_INTF_I2C** or **PTX_INTF_UART** as described in section 8 for the targeted host-interface implementation.

   ***Important**:* This step is only mandatory if a project gets ported to a new target platform, that is, if none of the provided example reference implementations gets used. When using one of the existing implementations, the definitions are automatically set when the desired build-target gets selected (see next step).

3. Select corresponding build-target.

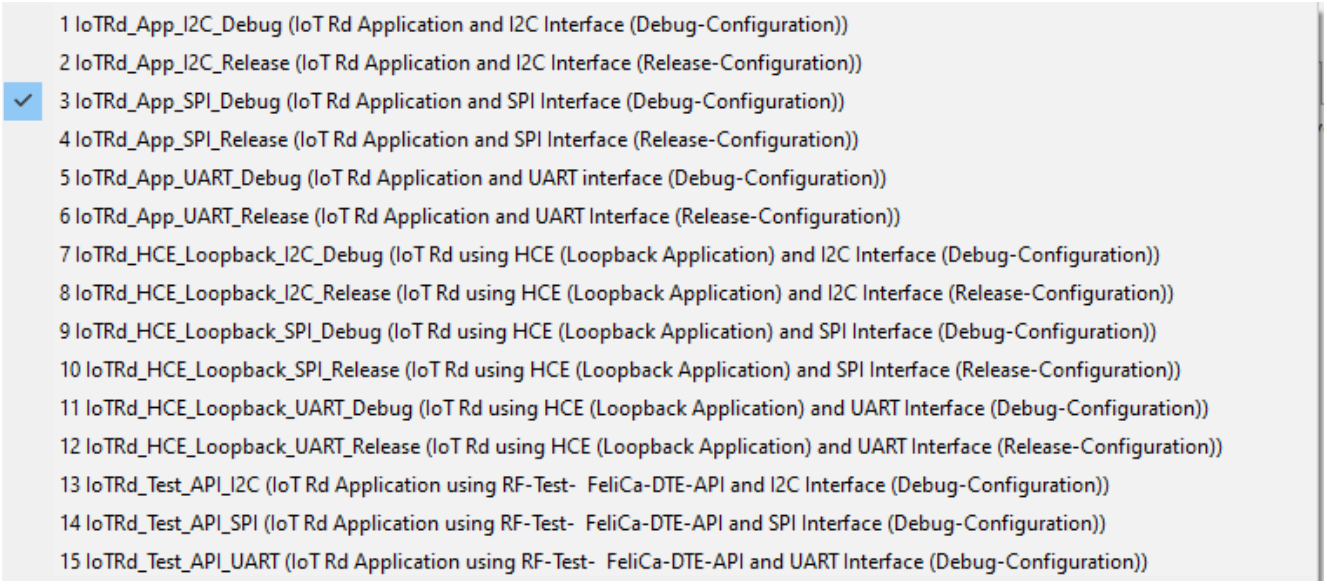


Figure 24. Build Target Selection

4. Wire/connect the physical pins from the EK-RA4M2 board and the PTX board as described in section 8.1.
5. Reboot the PTX board, build the target, and run the application.

9. RF Configuration Updates

The PTX chip allows to update RF configuration parameters. A default configuration is downloaded to the PTX chip during the initialization phase and can be optionally changed during runtime via an API call.

9.1 Default RF Configuration

The default RF configuration is provided as source files "**ptxNSC_RfConfigVal.c**" and "**ptxNSC_RfConfigVal.h**" and is stored inside the IoT-Reader (Non-OS) SDK in the folder "**SRC\COMPS\NSC**".

The default RF configuration, that is the source files themselves, can be generated using the *PTX1xxR IOT Config Tool* from Renesas (for reference documentation, see section 11).

Figure 25 shows a screenshot of the *PTX1xxR IOT Config Tool*. It allows the user to configure the RF configuration parameters and to generate the required '**.c**' and '**.h**' files accordingly.

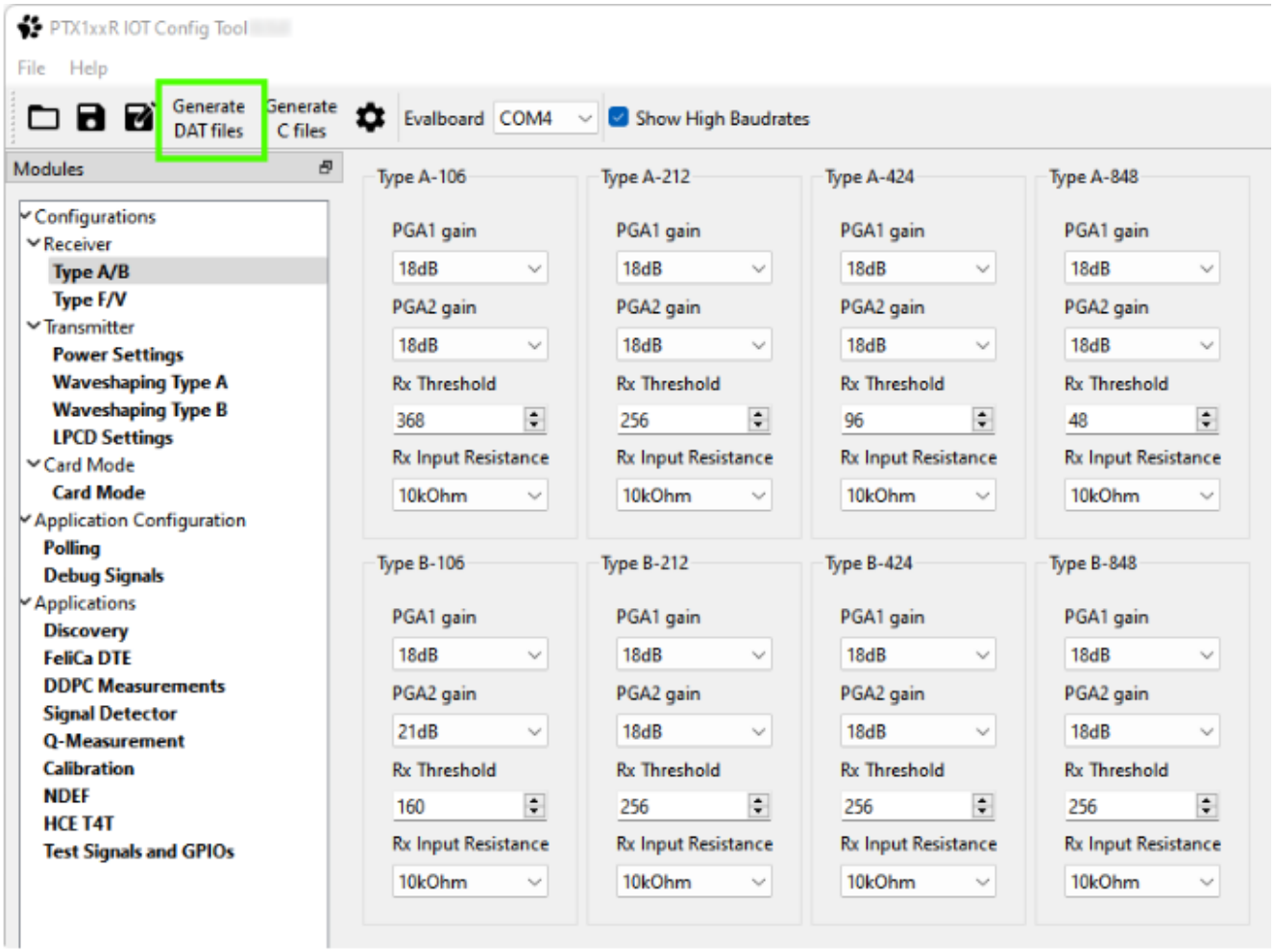


Figure 25. PTX1xxR IOT Config Tool

To update RF configuration of the IoT-Reader (Non-OS) SDK, click the toolbar button “Generate C files” and select the folder “SRC\COMPS\NSC”.

Important: The default RF configuration provided with the SDK works best with the accompanying product evaluation kit/board. It may also work well out of the box with a custom setup/application using a different antenna setup and matching components.

In most cases, updating (in other words, adapting) the default RF configuration is mandatory for a custom/final target application setup. As a reminder, the SDK implements an artificial compiler warning printing the following message:

```
Attention - Default SDK RF-Config settings used - please adapt values for
target application platform!
```

The warning disappears automatically when the RF configuration is regenerated using the *PTx1xxR Config Tool*.

9.1.1 Temperature Sensor Calibration

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

To get expected accuracy, the temperature sensor requires calibration. This should be done once for a given PTX chip. Calibration can be triggered by appropriate setting of the following input parameters in a call to "**ptxIoTRd_Init**":

ptxIoTRd_TempSense_Params_t *TemperatureSensor

- **Calibrate**
  Start temperature sensor calibration or not: 1 = start, 0 = don´t start
- **Tshutdown**
  Expected thermal shutdown threshold value
- **Tambient**
  Ambient temperature at which temp. sensor calibration takes place

Along with sensor calibration, the API calculates the compensated temperature threshold value that is then used as one of the input parameters in subsequent calls to "**ptxIoTRd_Init**".

Temperature Compensation Steps:

1. Set the system into **Start** state (according to section 4)
2. Start system initialization and perform temperature calibration:
   a. Set ambient temperature to a desired value for example, 25°C. Provide this value as "**Tambient**" input parameter in "**ptxIoTRd_Init**"
   b. Set the value of expected temperature shutdown threshold in "**Tshutdown**" parameter (for example, 100°C, the value is provided in the PTX1xxR datasheet). Note that "**Tshutdown**" returns the calculated threshold value from the calibration routine.
   c. Set "**Calibrate**" to 1.
   d. Call "**ptxIoTRd_Init**" with provided parameters.
3. Permanently store the value returned in "**Tshutdown**" parameter for future use (for example, in a configuration file).

   Use that stored value to fill "**TemperatureSensor::Tshutdown**" member of the "**initParams**" input parameter in every call to "**ptxIoTRd_Init**", during system initialization.

   ***Important**:* Temperature sensor calibration (temperature threshold compensation) must be executed once per PTX chip in controlled environment conditions (with parameter "**Calibrate**" set to 1). When done, it does not need to be started all over again.

   The resulting value should be stored and re-used in all future calls to "**ptxIoTRd_Init**" (with parameter "**Calibrate**" set to 0).

9.2 Dynamic at Runtime

The API function "**ptxIoTRd_Update_ChipConfig**" can be used to change the RF and System 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.

9.3 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 "**ptxNSC_RFConfig_GetConfigPointer**" inside the source file "**ptxNSC_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 described in section 7.

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; that is, they can be included in or excluded from the build process.

The current SDK version 2.0.0 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 for each of the APIs can be found in the following sections.

10.1 Native-Tag API

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

As shown in Figure 26, 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.

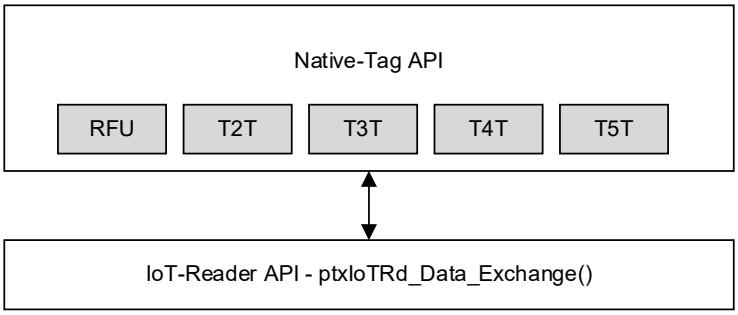


Figure 26. Native-Tag API Overview

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 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 Exchang**e" 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/sequences, such as:

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

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

As shown in Figure 27, the NDEF API is located on top of the Native-Tag API. The NDEF API is split into two sub-modules consisting of:

- The Tag-Type specific NDEF-operations
- A 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 of the operations **READ**, **WRITE**, and **LOCK** requires an initial execution of operation **CHECK**, otherwise the operations/functions will return an error.

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

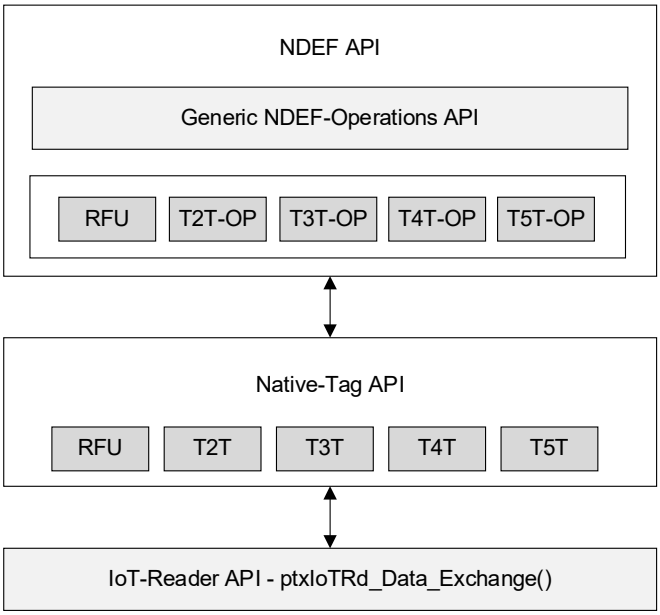


Figure 27. NDEF API Overview

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 handled internally.

10.3 GPIO API

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

- **ptxGPIO_Init**
  Initializes the GPIO component (possible to be called in any state).
- **ptxGPIO_Deinit**
  Deinitializes 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 PTX chip datasheet)
  - Output (+optional flag to increase the internal driver strength; see PTX chip 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 (must be stopped manually (see below)
  - PRBS15 (must be stopped manually (see below)
  - Carrier-on without modulation (must 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 to perform various FeliCa-related tests to pass the FeliCa M-Class certification and/or to 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 PTX chip 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 up 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()**" / "**ptxTransparentMode_Exchange()**" with input parameter state = 1. If the mode should be exited to return to state **Ready**, this function must be called again with input parameter state = 0.

- **ptxTransparentMode_SetRFParameter**s
  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:
    Max. number of responses to receive

- **ptxTransparentMode_Exchange**
  This API function performs the actual data exchange via RF. Received data always contains one additional byte which 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/Suggestions:

- 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 should 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 hardware 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 hardware receiver gets re-enabled until the timeout parameter (see "**ptxTransparentMode_Exchange()**") expires. For all other cases, RES-Limit should be set to 1.

Examples:

- **Type A – REQ-A / SENS_REQ**:
  - RF Technology = A
  - Tx/Rx Bitrate = 106kBit/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 = 106kBit/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 = 106kBit/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 = 26.5kBit/s
  - Flags = Tx/Rx Parity = d.c., Tx/Rx CRC = Enabled
  - Number of TxBits = 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).
- **ptx**TDC_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).
- ptx**TDC_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 is 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 References item [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 as shown below.

```
#define TDC_NFC_FORUM_COMPLIANT
```

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

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

10.7.1 Transparent Data Channel (TDC) Timer API

The non-OS implementation of the TDC API requires a stop-watch functionality which is implemented in a dedicated component called TDC Timer API.

The TDC API is **hardware-dependent** and must therefore be adapted/ported to the target platform. The reference implementation provided with the SDK is based on the App-Timer implementation located in:

\SRC\COMPS\PERIPHERALS\RA4M2\ptxPERIPH_APPTIMER.*

The App-Timer implementation uses a general-purpose Timer of the RA4M2 reference MCU. For porting the implementation, any timer supporting a resolution given in [Milliseconds] is suitable.

The TDC Timer component consists of the following API functions:

- **ptxTDC_Timer_Start**
  Starts a timer with a given number of milliseconds.
- **ptxTDC_Timer_Start**
  Stops the previously started timer.
- **ptxTDC_Timer_Status**
  Returns whether the timer has expired or not.

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

- \SRC\COMPS\TDC\ptxTDC_Timer.c
- \SRC\COMPS\TDC\ptxTDC_Timer.h

11. References

11.1 General

- [1] Renesas *PTX100R Datasheet*
- [2] Renesas *PTX105R Datasheet*
- [3] Renesas *PTX130R Datasheet*
- [4] Renesas *PTX1xxR – POS-Reader API Application Note: "Device Test Environment"*
- [5] *FeliCa Reader/Writer RF Performance Certification Specification Ver.1.5*
- [6] *FeliCa Reader/Writer Digital Protocol Requirements Specification Ver.1.22*
- [7] NFC Forum, Tag Type 2 Specifications 1.2, 2021.

11.2 Standards and Regulations

- NFC Forum

12. Revision History

Revision	Date	Description
1.01	May 3, 2024	Updated manual for SDK Release 7.2.0: ▪ 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). ▪ Added new member "ReadCCVariant" to T2T-NDEF-OP initialization parameters to select if the Capability Container CC gets read initially via Block 3 (default) or Block 0. ▪ Extended description of PLAT-UART reference implementation ▪ Added support for B-Prime RF-Technology in Transparent-Mode API ▪ Integrated “Transparent Data Channel (TDC)”-API (previously located in PTX130WN WLC Poller SDK)
1.00	Oct 24, 2023	SDK Release 7.1.0 ▪ Upgraded project files to Renesas E2 Studio 2022-10 and FSP 4.2 ▪ Added new API “Transparent-Mode”.

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

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.

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.