PTX1xxW NFC Forum Wireless Charging Poller API Non-OS Stack Integration (v2.1.0)

This document describes the NFC Forum Wireless Charging (WLC) Poller SDK for the PTX1xxW chip, further referred to as “WLCN”. The WLCN SDK provides a reference application of the NFC Forum Wireless Charging Protocol (see References item [1]) for the WLC Poller device running on a RA4M2 Arm Cortex® M33 (R7FA4M2AB) as an example host target platform.

Contents

Figures

1. Introduction

This solution is intended for platforms without an Operating System (Non-OS) and without a file system. For details of the “Wireless Charging NFC Forum Protocol”, refer to References section item [1].

1.1 Audience

This document is intended to be used by:

- SW architects
- SW engineers
- SW integrators

1.2 Abbreviations and Terminology

HW	Hardware
Integrator	Developer who builds/integrates the IoT-Reader API into a target application
IoTRd	IoT-Reader profile
IoT-Reader	IoT-Reader stack controller
NFC	Near Field Communication
NSC	NFC Soft Controller
RF	Radio Frequency
SDK	Software Development Kit
SW	Software
WLC	Wireless Charging
WLC Listener Device	Device being charged
WLC Poller Device	Charging device
WLCN	Wireless Charging NFC Forum

2. Architecture of the WLCN System

The WLCN System follows component-based approach which 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 an Operating System nor a File System available. This includes all components starting from the NSC Core upwards to the actual WLCN API.

2. **Hardware/Platform Dependent**:

   This part needs to be adapted to a specific MCU/platform used as Application processor (for more details, refer to section 6). This includes the PLAT component and PERIPHERALS with all its sub-components.

The WLCN Stack is implemented in ANSI C to maximize its portability. Figure 1 shows the main components of the stack.

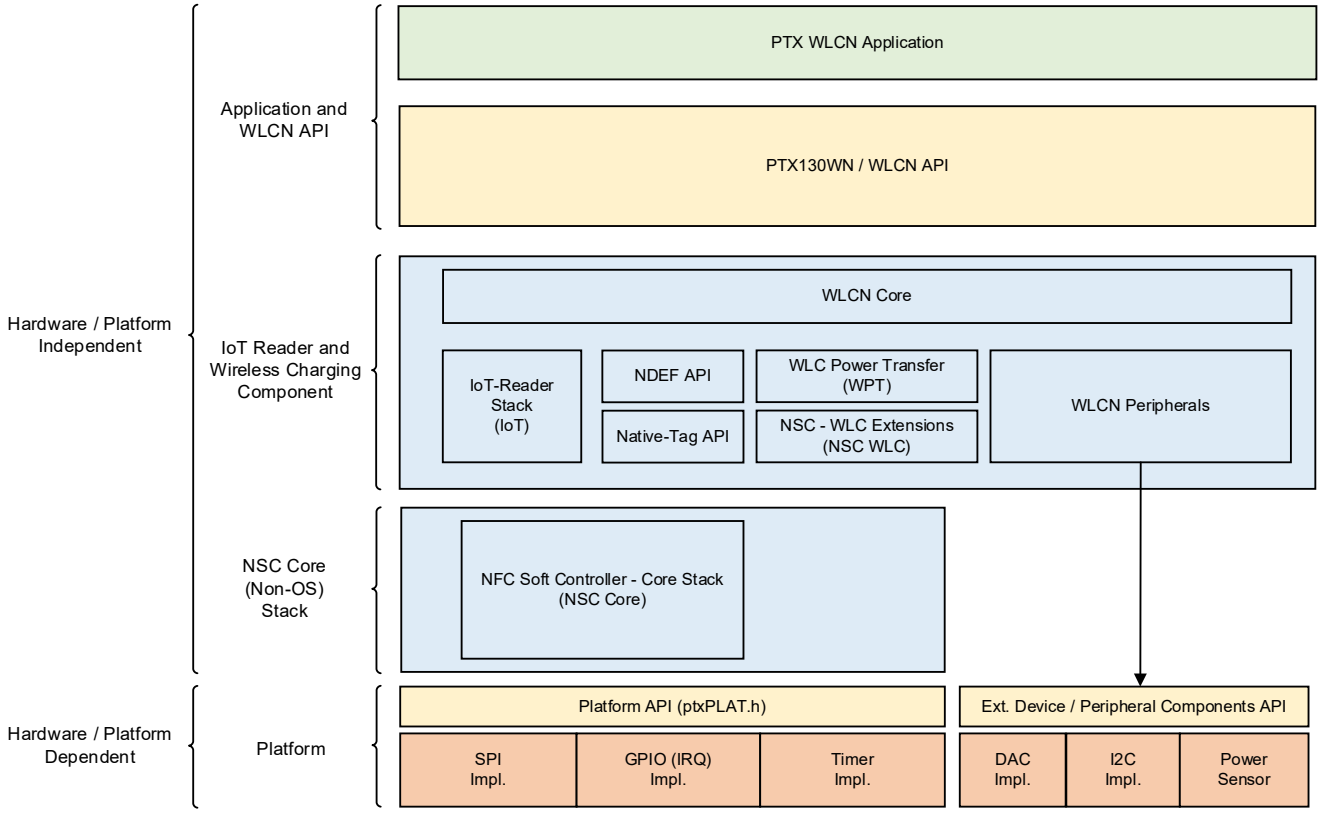

Figure 1. PTX Chip Software Stack Architecture

2.1 Component Description

2.1.1. PTX WLCN Application

It implements the actual charging application for the WLC Poller device according to the NFC Forum Wireless Charging Protocol (see References item [1]).

2.1.2. WLCN API

This API exposes all the wireless charging related functionalities of the NFC Forum Wireless Charging Protocol for the WLC Poller device to the application on top (see References item [1]).

The WLCN API provides the following features:

- System initialization of all hardware and software components
- Enabling NFC communication
- Perform wireless charging via NFC

Depending on the application, the WLCN API allows the actual wireless charging procedures to be managed directly:

- via the internal WLCN component through a single API call.
- by the application on a more granular level through multiple API calls.

By using the initial parameter, the application can provide a call-back function where the following event will be reported to main application:

- Listener detected on NFC link
- Charging Cycle Started
- Charging Cycle Finished
- Listener removed on NFC link

The WLCN API is described in detail in section 3 and an overview of the API states are described in section 4.

2.1.3. IoT-Reader & Wireless Charging Component

This component is split into multiple sub-components which provide access to the actual NFC communication functionalities as well as the wireless charging control blocks.

2.1.3.1. WLCN Core

This component implements all the functions required for the NFC Forum Wireless Charging Protocol and exposes a sub-set of it as the WLCN API.

2.1.3.2. IoT-Reader

This component implements the standard PTX IoT-Reader API which exposes the complete set of NFC Reader functionalities, except for Host Card Emulation (HCE), to higher software layers. A detailed description of the IoT-Reader can be found in References item [2].

2.1.3.3. Generic NDEF Operation API

This module implements the protocol to perform data exchange based on NFC Data Exchange Format (NDEF) messages between WLC poller and listener. It supports NFC Forum Tag Types 2–5.

2.1.3.4. NDEF Parser/Builder API

This module contains the functionalities for parsing and creating NDEF messages required for WLC communication.

2.1.3.5. WLC Power Transfer

This component implements functionality to enable wireless power transfers from the WLC Poller to the WLC Listener device with a given output power and duration.

2.1.3.6. NSC WLC Extensions

This component represents an extension of the NSC Core component and implements add-on functions for the WLC Power Transfer component and handling of wireless charging related call-back routines, such as notifications about charging states.

2.1.3.7. WLCN Peripherals

The WCLN peripherals are software components for initializing optional hardware components on the EVK PCB. These optional components include a DC/DC converter to supply the PTX1xxW with a variable voltage, an additional temperature sensor, and a power sensor for efficiency measurements.

Note: Customers with independent designs not based on the EVK circuit diagram do not need this software component.

2.1.4. NSC Component

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

- Configuration and initialization
- RF discovery loop
- RF card activation detection
- RF data exchange

2.1.5. Platform API

This API defines the platform/MCU dependent functionalities required by the NSC Stack.

The Platform API enables:

- Byte transfer to/from PTX chip
- The ‘waiting on’ synchronous (blocking) events, interrupts driven, triggered by the PTX chip
- The capture of asynchronous (non-blocking) events, interrupts driven, triggered by the PTX chip
- Sleep functionality placing the software execution to sleep for a certain time period

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, I^2C 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 (for example, SPI in this SDK).
- **GPIO**. Implements the driver for the GPIO used for IRQ. This submodule is needed when SPI or I^2C are used as physical interface; if UART is used, IRQ is not required.
- **Timer**. Implements a wrapper for a hardware timer to provide time-out functionality for the IoT-Reader Stack.
- *Note*: The delivered SDK contains a reference implementation for Platform component used on R7FA4M2AB application processor. This layer needs to be adapted/ported for different targets.

2.1.7. Peripherals Components/API

The WLCN SDK requires additional external peripherals such as:

- Digital-analog converter (DAC) for controlling the input power of the PTX chip.
- Power sensor used for current measurement.
- I^2C driver to interface the power sensor.

2.1.8. Poller system configuration

The WLCN SDK provides a header file for system configuration. The file can be found in:

\SRC\COMPS\WLC_POLLER\WLCN \ptxWLCN_PollerDefines.h

This file exposes a variety of customer-specific settings of the NFC WLC Poller. The following parameters can be adjusted:

- Enable/Disable support of BFOD
- Enable/Disable support of WPT Stop request
- Used power class
- Number of power levels
- Poll interval

A more detailed description can be found in the file itself.

3. WLCN API Description

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

3.1 ptxWLCN_Init

Declaration	`ptxStatus_t ptxWLCN_Init(`
	`  ptxWLCN_t *wlcn,`
	`  ptxWLCN_InitParams_t *initParam);`	

Description	Initializes the software and hardware components for WLCN operation. This function must be called before any other API function. It performs software initialization and configuration for the PTX chip.	
Input Parameters	`wlcn`	Pointer to WLCN component not initialized.
	`initParam`	Init parameters to configure software and hardware for proper operation.
Return Value	Status	Success or failure, refer to **ptxStatus_t** for details.

3.2 ptxWLCN_DeInit

Declaration	`ptxStatus_t ptxWLCN_DeInit(` `  ptxWLCN_t *wlcn);`	
Description	Deinitializes the WLCN component and all its sub-components.	
Input Parameters	`wlcn`	Pointer to initialized WLCN component.
Return Value	Status	Return Value.

3.3 ptxWLCN_State_WaitForListener

Declaration	`ptxStatus_t ptxWLCN_State_WaitForListener(`
	`ptxWLCN_t *wlcn,`
	`ptxWLCN_ChargeStates_t *chargingState);`	

Description	This function sets up the Listener discovery.	
Input Parameters	`wlcn`	Pointer to initialized WLCN component.
	`chargingState`	In/Out parameter: Current/Next charging state.
Return Value	Status	Return Value.

3.4 ptxWLCN_State_PollListener

Declaration	`ptxStatus_t ptxWLCN_State_PollListener(`
	`ptxWLCN_t *wlcn,`
	`ptxWLCN_ChargeStates_t *chargingState);`	

Description	This function polls for any device to be in the field.	
Input Parameters	`wlcn`	Pointer to initialized WLCN component.
	`chargingState`	In/Out parameter: Current/Next charging state.
Return Value	Status	Return Value.

3.5 ptxWLCN_State_ActivateListener

Declaration	`ptxStatus_t ptxWLCN_State_ActivateListener(`
	`ptxWLCN_t *wlcn,`
	`ptxWLCN_ChargeStates_t *chargingState);`	

Description	In case multiple Listener devices were discovered in ‘PollListener’-State this function activates one after the other until a WLC-CAP record was found.	
Input Parameters	`wlcn`	Pointer to initialized WLCN component.
	`chargingState`	In/Out parameter: Current/Next charging state.
Return Value	Status	Return Value.

3.6 ptxWLCN_State_GetCapRecord

Declaration	`ptxStatus_t ptxWLCN_State_GetCapRecord(`
	`ptxWLCN_t *wlcn,`
	`ptxWLCN_ChargeStates_t *chargingState);`	

Description	This function retrieves the WLC Capability record (and any optional WLC records) from the Listener-device and parses its parameters.	
Input Parameters	`wlcn`	Pointer to initialized WLCN component.
	`chargingState`	In/Out parameter: Current / Next charging state.
Return Value	Status	Return Value.

3.7 ptxWLCN_State_WaitForListenerReady

Declaration	`ptxStatus_t ptxWLCN_State_WaitForListenerReady(`
	`ptxWLCN_t *wlcn,`
	`ptxWLCN_ChargeStates_t *chargingState);`	

Description	This function waits for a certain amount of time, specified by the Listener until the WLC Capability record can be read again by the poller.	
Input Parameters	`wlcn`	Pointer to initialized WLCN component.
	`chargingState`	In/Out parameter: Current/Next charging state.
Return Value	Status	Return Value.

3.8 ptxWLCN_State_StaticCharging

Declaration	`ptxStatus_t ptxWLCN_State_StaticCharging(`
	`ptxWLCN_t *wlcn,`
	`ptxWLCN_ChargeStates_t *chargingState);`	

Description	This function performs a power transfer in static charging mode (see References item [1]).	
Input Parameters	`wlcn`	Pointer to initialized WLCN component.
	`chargingState`	In/Out parameter: Current / Next charging state.
Return Value	Status	Return Value.

3.9 ptxWLCN_State_SetInfoRecord

Declaration	`ptxStatus_t ptxWLCN_State_SetInfoRecord(`
	`ptxWLCN_t *wlcn,`
	`ptxWLCN_ChargeStates_t *chargingState);`	

Description	This function writes the WLC Information record to the listener.	
Input Parameters	`wlcn`	Pointer to initialized WLCN component.
	`chargingState`	In/Out parameter: Current/Next charging state.
Return Value	Status	Return Value.

3.10 ptxWLCN_State_GetControlRecord

Declaration	`ptxStatus_t ptxWLCN_State_GetControlRecord(`
	`ptxWLCN_t *wlcn,`
	`ptxWLCN_ChargeStates_t *chargingState);`	

Description	This function retrieves the WLC Control record from the listener device during the WCC phase.	
Input Parameters	`wlcn`	Pointer to initialized WLCN component.
	`chargingState`	In/Out parameter: Current/Next charging state.
Return Value	Status	Return Value.

3.11 ptxWLCN_State_FieldOff

Declaration	`ptxStatus_t ptxWLCN_State_FieldOff(`
	`ptxWLCN_t *wlcn,`
	`ptxWLCN_ChargeStates_t *chargingState);`	

Description	The poller turns off its RF-field, in case the listener reports 'BattFull' in its WLC Capability record or refuses another WPT via the WLC Control record. Hence. this function deactivates the field and waits for the required amount of time, specified by the listener.	
Input Parameters	`wlcn`	Pointer to initialized WLCN component.
	`chargingState`	In/Out parameter: Current/Next charging state.
Return Value	Status	Return Value.

3.12 ptxWLCN_State_NegoCharging

Declaration	`ptxStatus_t ptxWLCN_State_NegoCharging(`
	`ptxWLCN_t *wlcn,`
	`ptxWLCN_ChargeStates_t *chargingState);`	

Description	This function performs a power transfer in negotiated charging mode (see References item [1]).	
Input Parameters	`wlcn`	Pointer to initialized WLCN component.
	`chargingState`	In/Out parameter: Current/Next charging state.
Return Value	Status	Return Value.

3.13 ptxWLCN_State_PollCharging

Declaration	`ptxStatus_t ptxWLCN_State_PollCharging(`
	`ptxWLCN_t *wlcn,`
	`ptxWLCN_ChargeStates_t *chargingState);`	

Description	This function polls for the end of the charging state (static or negotiated). If not finished it keeps the state otherwise it updates the state accordingly.	
Input Parameters	`wlcn`	Pointer to initialized WLCN component.
	`chargingState`	In/Out parameter: Current/Next charging state.
Return Value	Status	Return Value.

3.14 ptxWLCN_State_PollReady

Declaration	`ptxStatus_t ptxWLCN_State_PollReady(`
	`ptxWLCN_t *wlcn,`
	`ptxWLCN_ChargeStates_t *chargingState);`	

Description	This function polls for the end of the wait state. If not finished it keeps the state otherwise it updates the state accordingly.	
Input Parameters	`wlcn`	Pointer to initialized WLCN component.
	`chargingState`	In/Out parameter: Current/Next charging state.
Return Value	Status	Return Value.

3.15 ptxWLCN_State_PollFieldOff

Declaration	`ptxStatus_t ptxWLCN_State_PollFieldOff(`
	`ptxWLCN_t *wlcn,`
	`ptxWLCN_ChargeStates_t *chargingState);`	

Description	This function polls for the end of the field-off state. If not finished, it keeps the state otherwise it updates the state accordingly.	
Input Parameters	`wlcn`	Pointer to initialized WLCN component.
	`chargingState`	In/Out parameter: Current/Next charging state.
Return Value	Status	Return Value.

3.16 ptxWLCN_State_DeactivateListener

Declaration	`ptxStatus_t ptxWLCN_State_DeactivateListener(`
	`ptxWLCN_t *wlcn,`
	`ptxWLCN_ChargeStates_t *chargingState);`	

Description	This function deactivates the Listener device.	
Input Parameters	`wlcn`	Pointer to initialized WLCN component.
	`chargingState`	In/Out parameter: Current/Next charging state.
Return Value	Status	Return Value.

3.17 ptxWLCN_State_PresenceCheck

Declaration	`ptxStatus_t ptxWLCN_State_PresenceCheck(`
	`ptxWLCN_t *wlcn,`
	`ptxWLCN_ChargeStates_t *chargingState);`	

Description	Validates if the Listener is still in the vicinity of the Poller. Turns the RF-Field on, runs the RF-discovery and turns the field off again.	
Input Parameters	`wlcn`	Pointer to initialized WLCN component.
	`isAvailable`	Out parameter: Pointer to uint8_t indicating if listener is still within proximity of the poller.
Return Value	Status	Return Value.

3.18 ptxWLCN_SetWptWbLuTable

Declaration	`ptxStatus_t ptxWLCN_SetWptWbLuTable(`
	`ptxWLCN_t *wlcn,`
	`const uint8_t *table);`	

Description	The user can pass a table/array with 32 entries (in range 0–255) to this function, to define a waveshape used for the WPT cycles.	
Input Parameters	`wlcn`	Pointer to initialized WLCN component.
	`table`	Pointer to the lookup table with 32 entries in range 0–255
Return Value	Status	Return Value

3.19 ptxWLCN_SetNfcWbLuTable

Declaration	`ptxStatus_t ptxWLCN_SetNfcWbLuTable(`
	`ptxWLCN_t *wlcn,`
	`const uint8_t *table);`	

Description	The user can pass a table/array with 32 entries (in range 0–255) to this function, to define a waveshape used for the NFC communication between the individual WPT cycles.	
Input Parameters	`wlcn`	Pointer to initialized WLCN component.
	`table`	Pointer to the lookup table with 32 entries in range 0–255.
Return Value	Status	Return Value.

3.20 ptxWLCN_GetWptWbLuTable

Declaration	`ptxStatus_t ptxWLCN_GetWptWbLuTable (`
	`ptxWLCN_t *wlcn,`
	`uint8_t *table);`	

Description	Retrieves the WPT lookup table (LUT) currently in use.	
Input Parameters	`wlcn`	Pointer to initialized WLCN component.
	`table`	Pointer to store the lookup table. (32 entries in range 0–255).
Return Value	Status	Return Value.

3.21 ptxWLCN_GetNfcWptWb

Declaration	`ptxStatus_t ptxWLCN_GetNfcWptWb (`
	`ptxWLCN_t *wlcn,`
	`uint8_t *wb);`	

Description	Retrieves the automatically scaled NFC wavebank currently in use.	
Input Parameters	`wlcn`	Pointer to initialized WLCN component.
	`wb`	Pointer to store the wavebank. (32 entries in range 0–31).
Return Value	Status	Return Value.

3.22 ptxWLCN_GetWptWb

Declaration	`ptxStatus_t ptxWLCN_GetWptWb (`
	`ptxWLCN_t *wlcn,`
	`uint8_t *wb);`	

Description	Retrieves the wavebank used for the WPT phase.	
Input Parameters	`wlcn`	Pointer to initialized WLCN component.
	`wb`	Pointer to store the wavebank. (32 entries in range 0–31).
Return Value	Status	Return Value.

3.23 ptxWLCN_GetNfcWb

Declaration	`ptxStatus_t ptxWLCN_GetNfcWb ( ptxWLCN_t *wlcn, uint8_t *wb);`	
Description	Retrieves the wavebank used for the communication phase (between the WPT cycles).	
Input Parameters	`wlcn`	Pointer to initialized WLCN component.
	`wb`	Pointer to store the wavebank. (32 entries in range 0–31).
Return Value	Status	Return Value.

3.24 ptxWLCN_IsPtxListener

Declaration	`ptxStatus_t ptxWLCN_IsPtxListener ( ptxWLCN_t *wlcn, uint8_t *isPtx);`	
Description	Checks if the activated tag/Listener is a PTX device.	
Input Parameters	`wlcn`	Pointer to initialized WLCN component.
	`isPtx`	True if connected listener is a PTX device.
Return Value	Status	Return Value.

3.25 ptxWLCN_SetListenerGpios

Declaration	`ptxStatus_t ptxWLCN_SetListenerGpios ( ptxWLCN_t *wlcn, bool setGpio0High, bool setGpio1High);`	
Description	Sets the GPIO levels of the PTX30W (PTX30W must be correctly configured).	
Input Parameters	`wlcn`	Pointer to initialized WLCN component.
	`setGpio0High`	Sets PTX30W's GPIO_0 to HIGH when true.
	`setGpio1High`	Sets PTX30W's GPIO_1 to HIGH when true.
Return Value	Status	Return Value.

4. WLCN API State Machine

Figure 2 shows an example flow of how to use the WLCN.

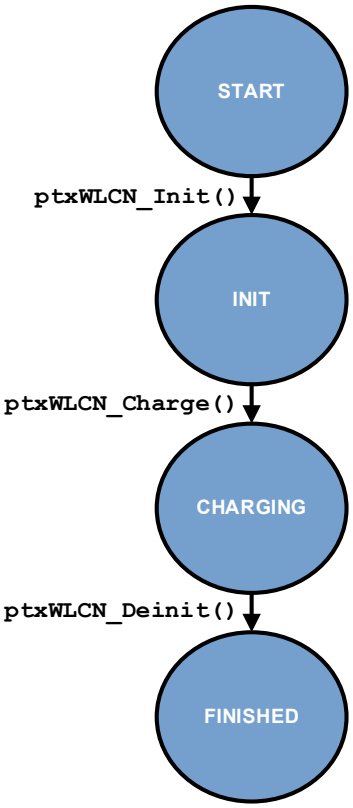

Figure 2. WLCN API Flow Example

Note: All WLCN API functions return a 16-bit status word indicating the status of the requested operation. If an operation succeeded, the status word is set to 0x0000 (= SUCCESS). In any other case, the upper 8 bit of the status word indicates the (sub-)component identifier of where the error occurred, and the lower 8 bit indicates the exact error code. Details of the status word definition can be found in the header file "**ptxStatus.h**".

Figure 2 shows a typical example flow of how to use the WLCN API assuming all functions return successfully. If an error occurs, it is handled internally within each state API to either trigger appropriate state transition or keep the system safe in the current state.

States Descriptions:

- **State: START**

  The system needs to be initialized first via a call to "**ptxWLCN_Init**". This initializes the software Stack and readies the PTX chip for NFC and wireless charging operations.

- **State: INIT**

  After Stack and hardware initialization, the WLCN API is ready to start the WLC application state machine implemented in the given code example.

- **State: CHARGING**

  This state is reached through the implementation of a state machine calling "**ptxWLCN_State_***" functions.

  The state machine implements the complete WLC logic, including:

  - Handling of listener RF activation process
  - Start of WLC static or negotiated charging according to the capabilities of listener
  - Charge listener device until battery is full

- Re-start the previous steps

Here is an overview of all existing state functions:

- **ptxWLCN_State_WaitForListener**
- **ptxWLCN_State_ActivateListener**
- **ptxWLCN_State_GetCapRecord**
- **ptxWLCN_State_WaitForListenerReady**
- **ptxWLCN_State_StaticCharging**
- **ptxWLCN_State_SetInfoRecord**
- **ptxWLCN_State_GetControlRecord**
- **ptxWLCN_State_NegoCharging**
- **ptxWLCN_State_FieldOff**
- **ptxWLCN_State_DeactivateListener**
- **ptxWLCN_State_PollListener**
- **ptxWLCN_State_PollReady**
- **ptxWLCN_State_PollFieldOff**
- **ptxWLCN_State_DeactivateListener**

  Important: It is highly recommended to check regularly (for example, with every change of the charging state) for critical system errors like thermal (over-temperature) errors. This can be achieved by using the function “**ptxIoTRd_Get_Status_Info**”. If the application is using the “**ptxWLCN_State**” functions, then regular checks are highly recommended.

  System error handling (and potential) recovery is shown in the provided SDK demo application.

- **State: FINISHED**

  Once the complete WLCN application is stopped or shutdown, it is required to call function "**ptxWLCN_DeInit"** to stop all RF activities and to free previously allocated system resources like memory, drivers, etc.

5. WLCN SDK Delivery

The WLCN SDK delivery contains the source code for the following:

- WLCN wireless charging application
- WLCN API
- IoT-Reader
- NSC Core Stack
- PLAT component with SPI as reference host interface to the PTX chip
- Reference implementation of the used TI INA219 power sensor

R7FA4M2AB (Cortex-M33 based MCU) has been the platform used for the reference implementation provided in the SDK. More detailed overview of platform specific files is given in section 6.

Important: The provided WLCN application project is based on the Renesas Flexible Software Package (FSP). To build the project and execute the application, install the required the Renesas FSP and Renesas e^2 studio development tools. Installation of those tools is out of scope of this document – refer to the vendor’s website.

Although platform specific code for the R7FA4M2AB microcontroller is not included in the delivery, it can be easily generated after importing and building the project in the Renesas e^2 studio IDE.

The SDK structure at root folder is as following.

- **Root-Folder**

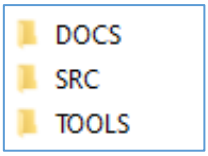

Figure 3. WLCN SDK Folder Structure

- **\DOCS**

  Doxygen-based API-description

- **\SRC**

  This folder contains the source code for WLCN application (“EXAMPLE”) and the WLCN stack (“COMPS”) plus all (sub-)components.

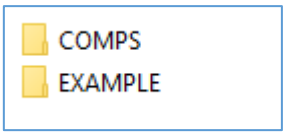

Figure 4. SRC Folder Structure

- **\TOOLS**

  PLAT contains the project to be imported in Renesas e^2 studio.

Figure 5. Tools Folder Structure

- **\TOOLS\PLAT\RENESAS\RA4M2\E2STUDIO_WORKSPACE\PTX130WN_SDK**

  This is the location where project files are located. When importing a project in the Renesas e^2 studio, just point to this location. All the files required to build the project are included.

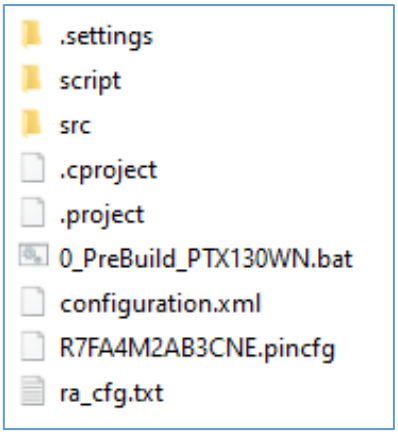

Figure 6. Project Files

5.1 Extensions

In order to provide the user with the capability to customize the final application or extend the bare wireless charging functionality of the Listener device, there are specific extension modules available through dedicated APIs. The following sections describe all the currently implemented extension features.

5.1.1. Transparent Data Channel

The Transparent Data Channel (TDC) enables arbitrary data transfers between the Poller and Listener. The transport protocol is built upon ISO14443/3–Type A 106kbps frames. The default SDK poller application displays an example usage of the data exchange APIs, as shown below.

```
 * \brief Writes messages to the listener's buffer.
ptxStatus_t ptxWLCN_TDC_Write(ptxWLCN_t *wlcn, uint8_t *txData, uint8_t txLen, uint32_t ackTimeoutMs);

 * \brief Reads messages from the listener's buffer.
ptxStatus_t ptxWLCN_TDC_Read(ptxWLCN_t *wlcn, uint8_t *rxData, uint8_t *rxDataLen, uint32_t rxTimeoutMs);

 * \brief Checks if the Listener's host MCU has read the previously sent message.
ptxStatus_t ptxWLCN_TDC_IsReceived(ptxWLCN_t *wlcn, uint8_t *received);
```

Data transfer is done in packets of max 63 bytes. Each and every data transfer (no matter whether it's a Tx or Rx operation) must always be initiated by the Poller – it is the master of the communication channel. If data shall be sent to the Listener, the Poller 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 Poller, instead it must wait until it is read by the Poller.

5.1.2. Data Exchange API

Three dedicated APIs provide all the required features for sending and receiving data on the Poller side.

5.1.2.1. ptxWLCN_TDC_Write

Declaration	`ptxStatus_t ptxWLCN_TDC_Write ( ptxWLCN_t *wlcn, uint8_t *txData, uint8_t *txLen, uint32_t *ackTimeoutMs);`	
Description	Writes messages with a maximum of 63 payload bytes to the PTX30W's buffer. The API requires a pointer to an initialized WLCN component, the pointer to the actual payload and the length information. Customers can choose between NFC Forum compliant write access and proprietary write access (= faster) via the compile switch.	
Input Parameters	`wlcn`	Pointer to initialized WLCN component.
	`txData`	Pointer to the data that shall be sent to the PTX listener.
	`txLen`	Length of the data to be sent (maximum of 63 bytes allowed).
	`ackTimeoutMs`	Timeout (in milliseconds) for the listener to read the transmitted message. If the Listener has not read the message withing `ackTimeoutMs` (for example, 50ms), the API will return a **ptxStatus_TimeOut** error.
Return Value	Status	Return Value.

5.1.2.2. ptxWLCN_TDC_Read

Declaration	`ptxStatus_t ptxWLCN_TDC_Read ( ptxWLCN_t *wlcn, uint8_t *rxData, uint8_t *rxLen, uint32_t *rxTimeoutMs);`	
Description	Reads messages with a maximum of 63 payload bytes from the PTX30W's buffer. The API requires a pointer to an initialized WLCN component, the pointer to a buffer for storing the received payload and a pointer to store the length information. Customers can choose between NFC Forum compliant read access and proprietary read access (= faster) via the compile switch.	
Input Parameters	`wlcn`	Pointer to initialized WLCN component.
	`rxData`	Pointer to store the received data.
	`rxLen`	Available buffer length of the rxData-buffer.
	`rxTimeoutMs`	Timeout for the Poller to wait for a message (in milliseconds). The API will block for the specified amount of time and repeatedly try to read a message from the Listener until it has found one.
Return Value	Status	Return Value.

5.1.2.3. ptxWLCN_TDC_IsReceived

Declaration	`ptxStatus_t ptxWLCN_TDC_IsReceived ( ptxWLCN_t *wlcn, uint8_t *received);`	
Description	Checks if the previously sent message using "**`ptxWLCN_TDC_Write()`**" has been received/read by the PTX30W's host MCU.	
Input Parameters	`wlcn`	Pointer to initialized WLCN component.
	`Received`	True if the listener's host MCU read the message.
Return Value	Status	Return Value.

Customers can choose between two different modes of operation:

- **NFC FORUM COMPLIANT Mode:**

  The NFC Forum Compliant transfer mode uses the T2T command set (see References item [3]) to read and write data from/to the listener.

  With the T2T_WRITE command, the poller can transfer 4 bytes of payload to the listener, whereas with the T2T_READ command, the Poller 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.

  The APIs are located within:

  - src/STACK/COMPS/WLC_POLLER/WLCN/ptxWLCN_Extension.c
  - src/STACK/COMPS/WLC_POLLER/WLCN/ptxWLCN_Extension.h

By default the PTX PROPRIATARY mode is active. NFC FORUM COMPLIANT mode can be enabled by setting the define, shown below.

```
#define TDC_NFC_FORUM_COMPLIANT
```

5.1.3. Example Use Case

In the wireless charging protocol, data transfers are possible only during the communication phase before the readout of the WLC_CTL record, as shown in the screenshot below.

```
215                         case GetControlRecord:
216         #ifdef ENABLE_TDC
217                         {
218                             /**
219                             * ############ TDC Example ############
220                             *
221                             */
222                             uint8_t is_ptx_device;
223                             (void) ptxWLCN_IsPtxListener(&wlcn, &is_ptx_device);
224
225                             if(is_ptx_device)
226                             {
227                                 /** Reserve some memory. */
228                                 uint8_t data[256];
229                                 uint32_t dataLen = sizeof(data);
230
231                                 /** Generate dummy data */
232                                 for(uint16_t i = 0; i < 256; ++i)
233                                 {
234                                     data[i] = (uint8_t) i;
235                                 }
236
237                                 /** Transmit the data using simple high level protocol. */
238                                 status = ptxWLCN_TDC_HlpTx(&wlcn, data, dataLen);
239                                 /** Check if transmission was successful. */
240                                 if (ptxStatus_Success == status)
241                                 {
242                                     /** Clear the TX buffer.*/
243                                     memset(data, 0x00, 256);
244                                     /** Reuse the TX buffer for RX payload. */
245                                     status = ptxWLCN_TDC_HlpRx(&wlcn, data, &dataLen);
246                                     /** Check if RX operation was successful. */
247                                     if (ptxStatus_Success == status)
248                                     {
249                                         /** Do something with the data*/
250                                     }
251                                 }
252                             }
253                         }
254         #endif
255                             status = ptxWLCN_State_GetControlRecord(&wlcn, &charging_state);
```

The WLCN main program example shows the (optional) transparent data exchange usage. An additional protocol was added on top of the TDC in the main example. It enables the data transfers of up to 16MB into either of the two directions by chunking the payload into 63 byte frames. The implementation of the protocol is available in the files:

- src/STACK/COMPS/WLC_POLLER/WLCN/ptxWLCN_TDC_HLP.c
- src/STACK/COMPS/WLC_POLLER/WLCN/ptxWLCN_TDC_HLP.h

Note: Provided example uses dummy data and does not store the transferred data.

6. Platform Specific

This section describes the platform specific code and project settings used for R7FA4M2AB as the reference implementation. The platform/MCU dependent code has been encapsulated in two components in this project. On one hand, PLAT Component implements the platform/MCU dependent code for interfacing the PTX chip; on the other hand, PERIPHERALS component implements the drivers for the external hardware devices needed for WLCN application – basically a Power Sensor and a DCDC power convertor. This reference code can be used as guidance for porting the platform specific code to any other platform/MCU used as the application processor, in which case, a general rule should be followed:

> *Keep API declarations the same while changing implementation of functions (or data structure) for the specific platform/MCU requirements*.

RA4M2 MCU specific 3rd party code is not included in this delivery since the source code is automatically generated during every build process.

6.1 PLAT Component

Figure 7 shows the PLAT Component folder structure:

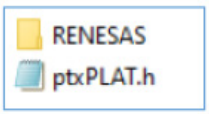

Figure 7. PLAT Folder Structure

The following items can be found in the PLAT folder:

- **\RENESAS\RA4M2**

  Folder containing source code specific for R7FA4M2AB MCU.

  The following items are in the RA4M2 folder:

  - **ptxPLAT.h**
    Platform API definition.
  - **ptxPLAT.c**
    Platform specific wrapper that adapts the PLAT API functions for submodules TIMER, GPIO and SPI.
  - **ptxPLAT_GPIO.h**
    Platform header where the GPIO-IRQ set of functions are defined.
  - **ptxPLAT_GPIO.c**
    Platform specific implementation of the GPIO-IRQ for R7FA4M2AB.
  - **ptxPLAT_INT.h**
    Platform specific header where the PLAT component for R7FA4M2AB implementation is defined.
  - **ptxPLAT_SPI.h**
    Platform header where the SPI features are defined.
  - **ptxPLAT_SPI.c**
    Platform specific implementation for SPI features for R7FA4M2AB.
  - **ptxPLAT_TIMER.h**
    Platform header where the Timer features are defined.
  - **ptxPLAT_TIMER.c**
    Platform specific implementation for Timer features for R7FA4M2AB.

In order to port PLAT Component to a new MCU/platform, the API functions in previous header files should keep the same definition as they are used from the upper layer. Implementations in ‘C’ files and data structure should be adapted for the specific MCU/platform.

6.2 PERIPHERALS Component

Figure 8 shows the PERIPHERALS folder files:

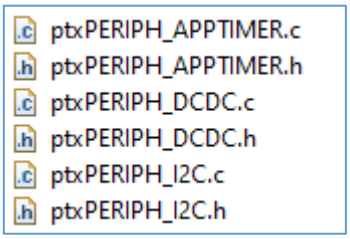

Figure 8. PERIPHERALS Folder Structure

The following items are in the PHERIPERALS folder:

- **ptxPERIPH_APPTIMER.h**
  Header where the Timer functions are defined.
- **ptxPERIPH_APPTIMER.c**
  Platform specific implementation of Timer functions for R7FA4M2AB.
- **ptxPERIPH_DCDC.h**
  Header where the functions needed to operate the DCDC Power Converter are defined.
- **ptxPERIPH_DCDC.c**
  Platform specific implementation of the DCDC functions for R7FA4M2AB.
- **ptxPERIPH_I2C.h**
  Header where the I^2C functions are defined.
- **ptxPERIPH_I2C.c**
  Platform specific implementation of the I^2C functions for R7FA4M2AB.

To port the PERIPHERALS component to a new MCU/platform, the API functions in previous header files should keep the same definition as they are used from upper layer. Implementations in C files and data structure should be adapted for the specific MCU/platform.

6.3 IRQ Handling

The PTX chip has one GPIO configured as an output (called IRQ) used to notify the host that there is some pending data to be read-out on its buffers; typically, a response or a notification in the NSC communication. This GPIO is set high by the PTX chip when there is some data pending, and the line is set low by PTX chip when the data has been read-out.

On the MCU there is one ISR configured for that GPIO in the rising edge, which means that the ISR is executed when IRQ line is set high. This ISR execution is used to unlock “_WFI_” events (for example, “Wait For Interrupt” events) where the MCU is entered to save some power. Once that MCU is out of WFI, it checks that the state of the IRQ line is high and performs the read operation of the buffer on PTX chip.

The read operation is executed from the main application software context and not from the ISR context that is just used to signal the IRQ raising change.

When this SDK is ported to a new MCU-based system, the user needs to set an ISR routine that is executed on the rising edge of the GPIO.

6.4 Project Quick Start Guide

To build the delivered project and execute the demo application on EK-RA4M2 (RA4M2 MCU development board), the Renesas FSP and e^2 studio development IDE are required to be installed (see section 5).

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

The project is in: **\TOOLS\PLAT\RENESAS\RA4M2\E2STUDIO_WORKSPACE\PTX130WN_SDK**

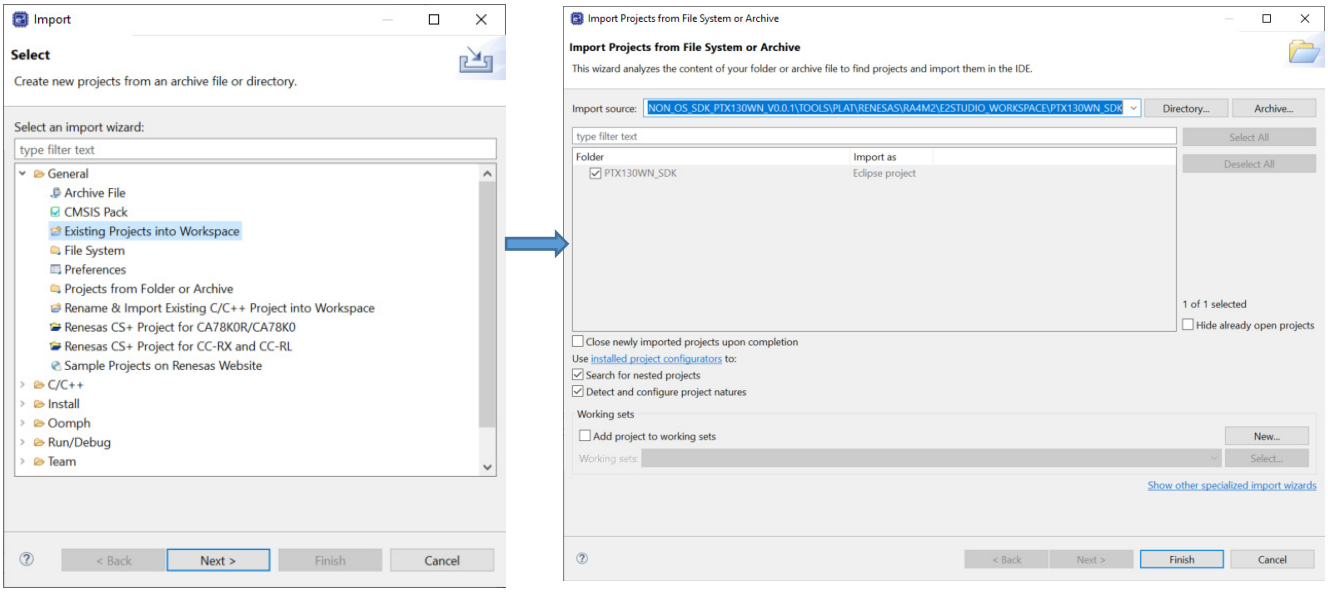

Figure 9. Importing the Project into the Workspace

After successfully importing the project, the contents of the Project Explorer in the e^2 studio appears like that in Figure 10 (a).

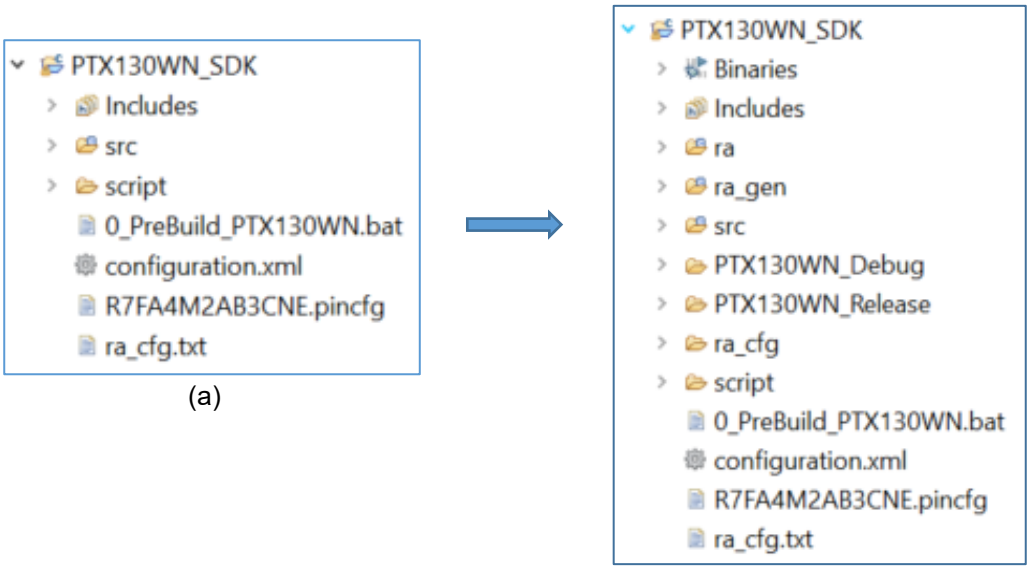

Figure 10. Project Explorer Contents: (a) After Importing, Before Build; (b) After the Build

The Project Explorer contains the following items:

- **src/STACK**

  Contains SDK source files. However, the 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 SDK´s SRC folder as described in section 5.

- **hal_entry.c**

  This file contains the call to demo application.

- **script/fsp.ld**

  Linker script for R7FA4M2AB MCU.

- **0_PreBuild_PTX130WN.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-click on this file to load the FSP Configuration tool.

- **R7FA4M2AB3CNE.pincfg**

  Pin configuration file containing current MCU pins configuration. This is also updated in the FSP Configuration tool.

- **ra_cfg.txt**

  File containing current MCU and board configuration.

Step 2: Build the Project

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

There are 2 build configurations available: PTX130WN_Debug and PTX130WN_Release. Both will generate .elf binary file which is then used to download into MCU.

In Figure 10 (b), folders marked in red are automatically generated with every build process:

- **ra_gen, ra and ra_cfg** folders contain 3rd party source code which includes support for all used (configured) MCU and board features, for example, clock settings, connectivity drivers, timers, IO drivers, etc.
- **PTX130WN_Debug** folder contains build output files for PTX130WN_Debug build configuration. If another build configuration is chosen (in other words, **PTX130WN_Release)**, then a corresponding output folder is created.

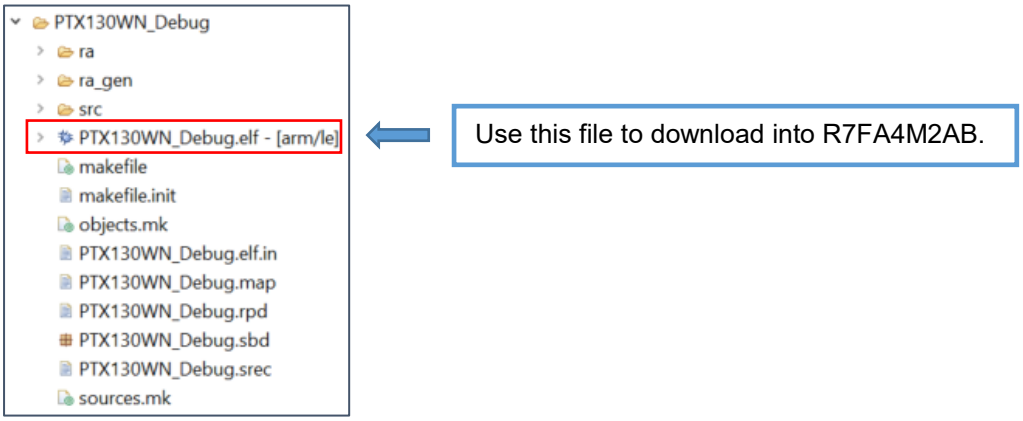

Figure 11. Build Output Folder

Step 3: Debugging

As previously mentioned, the project has been developed based on the R7FA4M2AB MCU and deployed on the PTX NFC-WLC_Poller board. The hardware required for setting up the system is shown in Figure 12 through Figure 16.

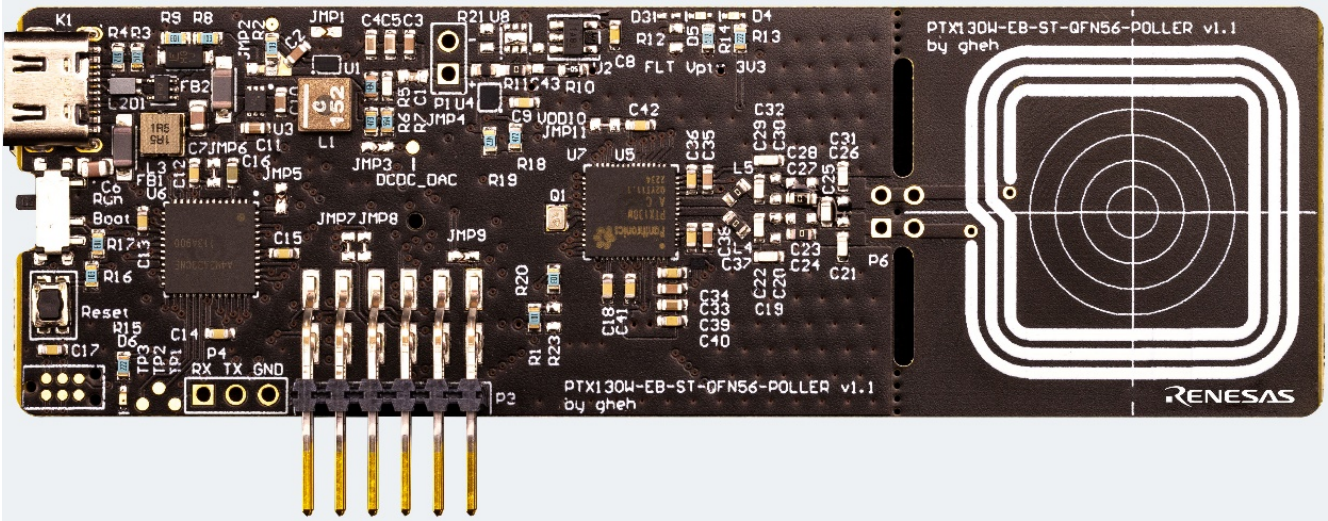

Figure 12. PTX NFC-WLC_Poller Board

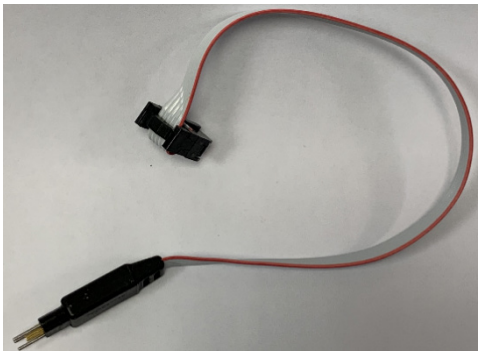

Figure 13. Tag-Connect Cable

Figure 14. Tag-Connect Clip

Figure 15. J-Link Debugger

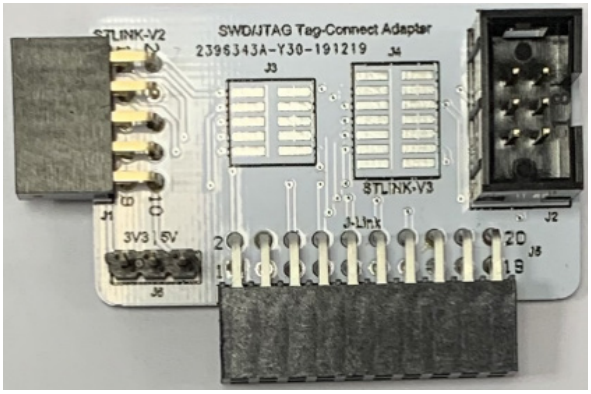

Figure 16. Tag-Connect Adapter

To program the MCU and set up the system, connect the J-link debugger through a USB port to the computer; and on the other side to the Tag-Connect adapter. The Tag-Connect cable is connected from the Tag-Connect adapter to the Debug Header on the PTX Poller board (the Tag-Connect clip is used to attach it properly to the board).

- Select **.elf** binary to download to MCU.
- Click *Run > Debug As > Renesas GDB Hardware Debugging*
  - Select J-Link ARM
  - Select target R7FA4M2AB from the list

After a successful start of the debugger, the WLC Poller starts to emit an NFC RF field to detect a WLC Listener device.

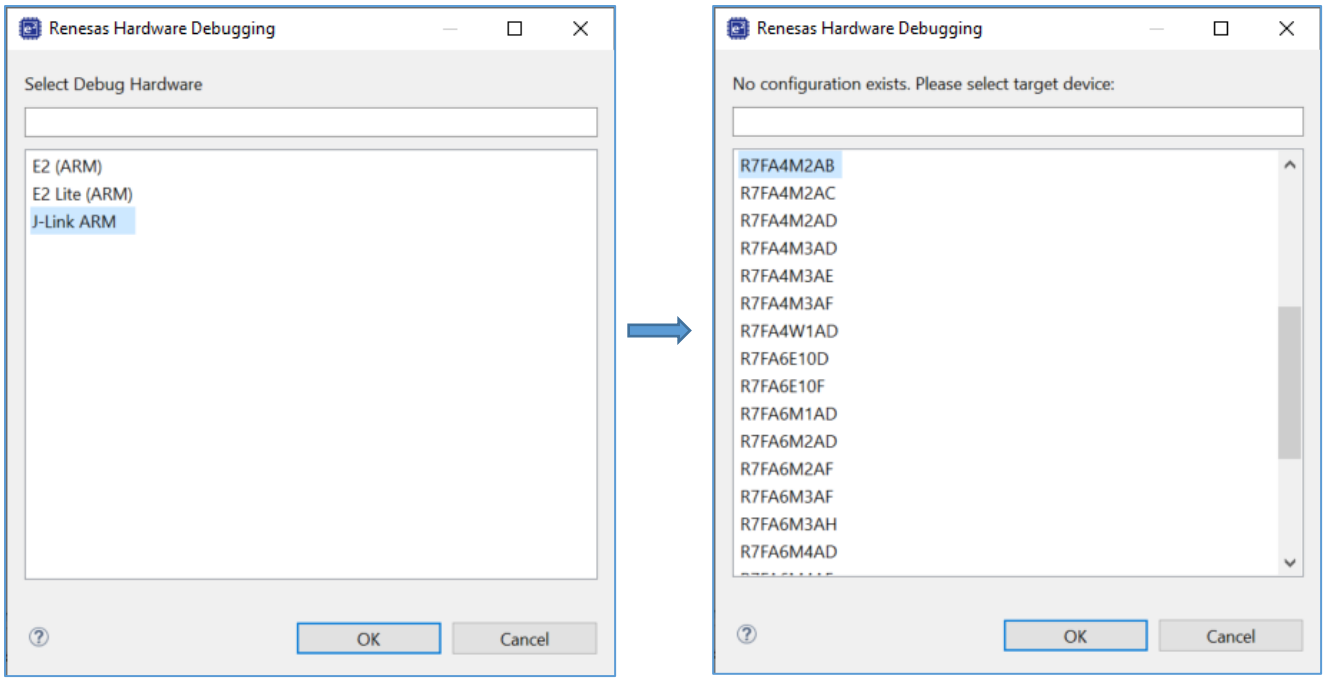

Figure 17. Selecting Debug Target

Important: The provided SDK contains all the required files and project settings to build and execute an application on the PTX Poller board by following the steps described in this section. If a different board or R7FA4M2AB MCU configuration will be used to evaluate the provided demo application, pins and peripherals of the MCU might need to be reconfigured. This means a different "**.pincfg**" file must be provided, or pins and peripheral usage need to be set appropriately in the FSP Configuration tool by opening and modifying the **configuration.xml** file. A detailed procedure of how this is done is provided in the vendor's documentation and is out of scope of this document.

7. References

[1] *NFC Forum, Wireless Charging Technical Specification 2.0, 2021.*

[2] PTX1xxR NFC IoT-Reader API for Non-OS Stack Integration (SDK v7.1.0) User Manual

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

8. Revision History

Revision	Date	Description
1.00	Feb 28, 2024	Initial release.

IMPORTANT NOTICE AND DISCLAIMER

RENESAS ELECTRONICS CORPORATION AND ITS SUBSIDIARIES (“RENESAS”) PROVIDES TECHNICAL SPECIFICATIONS AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT OF THIRD-PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for developers who are designing with Renesas products. You are solely responsible for (1) selecting the appropriate products for your application, (2) designing, validating, and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. Renesas grants you permission to use these resources only to develop an application that uses Renesas products. Other reproduction or use of these resources is strictly prohibited. No license is granted to any other Renesas intellectual property or to any third-party intellectual property. Renesas disclaims responsibility for, and you will fully indemnify Renesas and its representatives against, any claims, damages, costs, losses, or liabilities arising from your use of these resources. Renesas' products are provided only subject to Renesas' Terms and Conditions of Sale or other applicable terms agreed to in writing. No use of any Renesas resources expands or otherwise alters any applicable warranties or warranty disclaimers for these products.

(Disclaimer Rev.1.01 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.