PTX1xxX SDK RL78 Porting Process

Introduction

This document describes the usage of the PTX IoT Reader demo application using e² Studio and running it on the Renesas RL78/G23 evaluation board.

Contents

Figures

1. Requirements

This document applies to:

- IoT-Reader (Non-OS) SDK for PTX1xxR family v7.2.0
- RL78/G23 microcontrollers family

2. Porting Process

The provided SDK is implemented in C and represents the porting of RA4M2 reference to RL78/G23 device using SPI communication with the PTX100R chip.

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.

The porting of RA4M2 reference to RL78/G23 device encountered four main aspects:

- CC-RL Compiler does not support Variable Length Arrays.
- CC-RL Compiler does not consider local constant definitions as true constants.
- RL78/G23 unique memory architecture require mandatory pointers to constant usage for constant data.
- Platform layer has to be implemented using generated code API's for HW Drivers.

3. Software

In this section, all software related topics are analyzed in detail and relevant instructions are given to final user.

3.1 Software Documentation

To have a proper understanding of the provided SDK, reference the following documents:

- RL78/G23 User Manual
- PTX1xxR NFC IoT-Reader Non-OS Stack Integration Manual

3.2 Software Tools

- e2 Studio – version 2021-10 or above
- Flexible Software Package (FSP) for Renesas RA MCU Family – version 5.5.0 or above
- Toolchain Renesas CC-RL – v1.14.00 or above
- (Optional) Logic2 Software

3.3 Software Setup

This section describes the step-by-step guidelines for deploying and debugging the downloaded SDK content to the hardware boards.

3.3.1 Opening a Project in e2 Studio

Open the project in the Renesas provided IDE (integrated development environment) called e2 Studio. The IDE is downloadable from e2 Studio.

Note: The e2 Studio includes a free 30-day trial CC-RL compiler license. After this free trial period expires, the product must then be purchased.

3.3.1.1 Downloading the SDK Content

The SDK content download includes a **.ZIP** package containing a README.md, the source code, the e 2 Studio project configuration and Smart Configurator settings (see Figure 1). Extract the content.

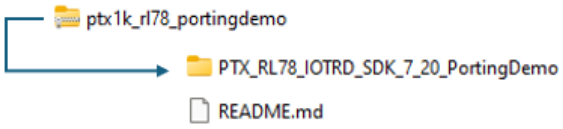

Figure 1. SDK ZIP Package Content

3.3.1.2 Importing the Project in e2 Studio

After extracting the project content, open it directly as an e2 Studio project as follows (see Figure 2):

1. Open e2 Studio.
2. Create a workspace (if one has yet to be created).

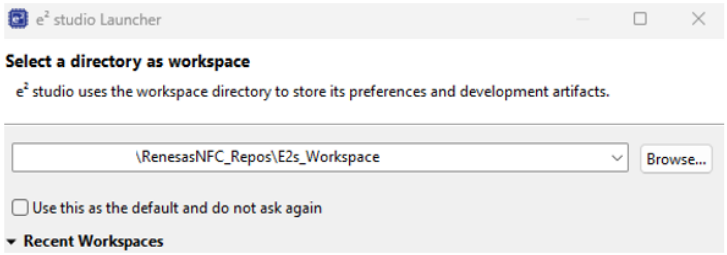

Figure 2. Select a Directory as Workspace

3. From the top menu bar, click **File > Import**.
4. Since the SDK package already contains an existing project, select *General* > *Existing Project into Workspace* (see Figure 3). Click **Next**.

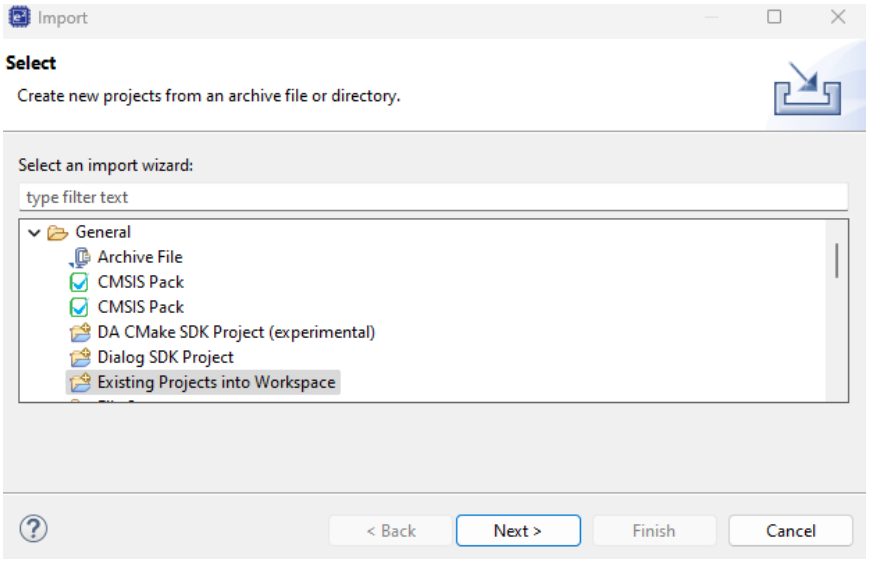

Figure 3. Import Existing Project

5. The root directory must be selected. Click on **Browse**, navigate to **ptx1k_rl78_portingdemo\PTX_RL78_IOTRD_SDK_7_20_PortingDemo\PTX_RL78_IOTRD_20240510_** and select this folder (see Figure 4).
6. In the *Projects* list, the new project has to be detected by e2 Studio. Once detected, click **Finish**.

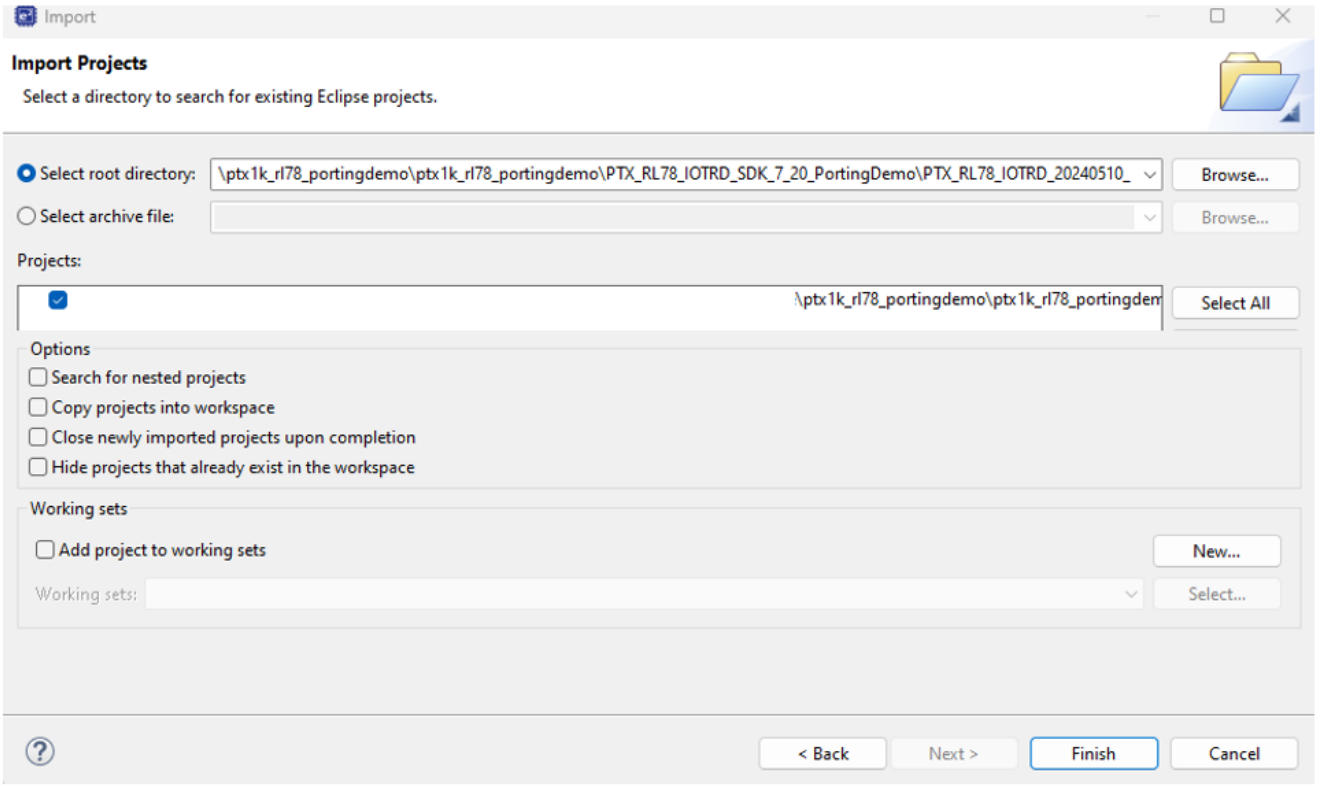

Figure 4. Browse to Project Location

7. In the *Project Explorer* tab, the imported project will be displayed as shown in Figure 5.

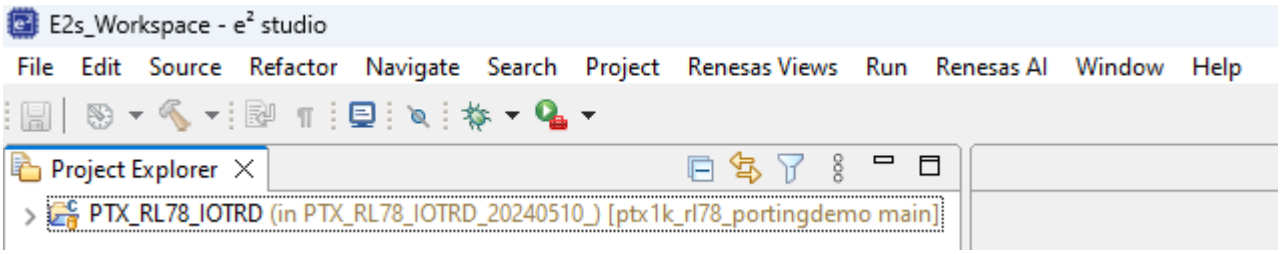

Figure 5. Project Explorer – Imported Project

3.3.1.3 Building the Project

To build the newly imported project, navigate to the *Project Explorer* tab, right-click on the project and select *Build Project*.

Note: For this demo, all build settings are already configured and the project from the delivered package is ready to build. If the user intends to change the build settings, refer to the procedure in the e2 Studio documentation.

In the event of a build failure, contact Renesas technical support.

3.3.1.4 Debug the Project

Upon successfully building the project, debug the project using the *On Board Debugger* of the RL78/G23 Evaluation Kit. For this step, refer to chapter 3 to perform the Hardware Set-up.

For debugging the demo, use Renesas GDB Hardware Debugging. The user must configure the right device and the COM port where the RL78/G23 device is connected.

To configure the debugger, navigate to the *Project Explorer* tab, right-click on the project **> Debug As > Debug Configurations**. The preconfigured **PTX_RL78_IOTRD.x** settings are available for debugging the Renesas GDB Hardware (see Figure 6).

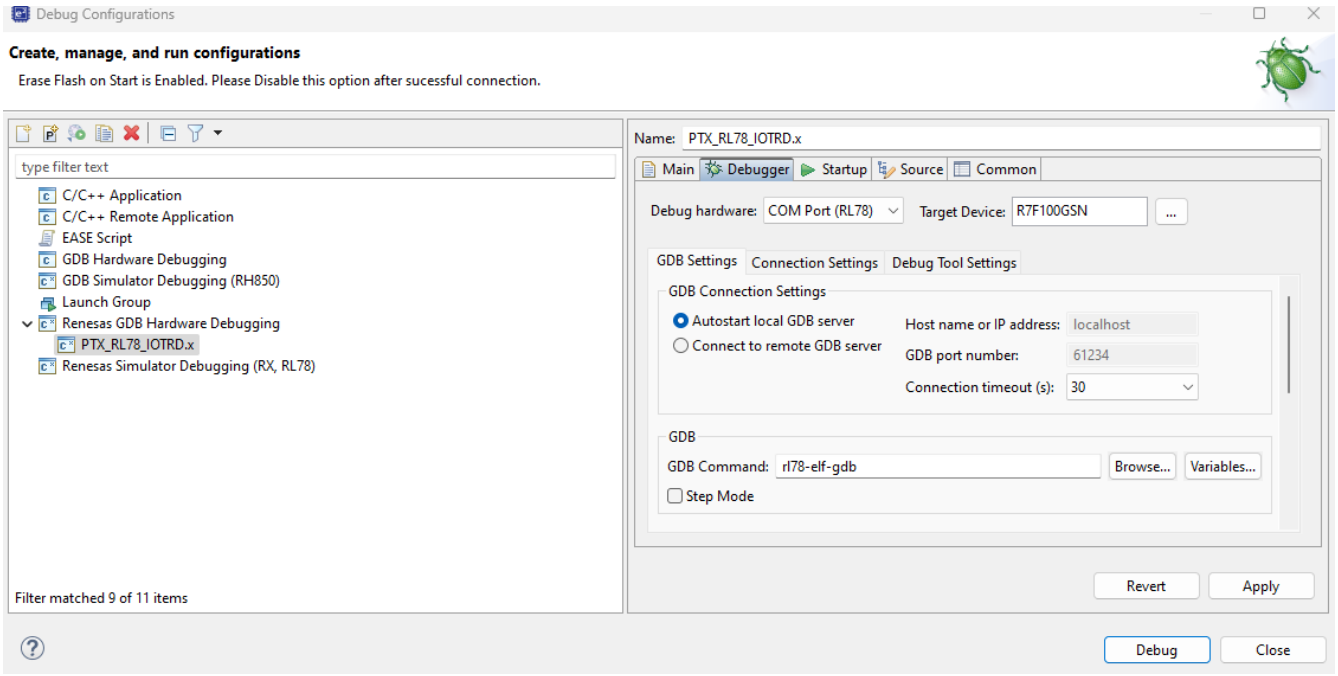

Figure 6. Debug Configurations

From the *Debugger* tab, the user can change the *Target Device*. The R7F100GSN is used for the purpose of this demo. Under *Debugger* > *Connection Settings*, the user can change the COM port option where the debugger is connected. After all the settings are entered, click on **Apply**.

To navigate to the *Project Explorer* tab, right-click on the project **> Debug As > PTX_RL78_IOTRD.x**.

If the debug fails, check the hardware connections and COM port allocation.

If the issue persists, contact Renesas technical support.

3.4 Using IoT Reader Library in the RL78/G23 Project

The IoT-Reader (Non-OS) system follows a component-based approach that increases modularity and usability. The components are divided into two main groups (see Figure 7):

- **Hardware/Platform Independent**: Suitable to run on application processors where there is neither an operating system nor a file system. Those components are IoT-Reader (IoT Rd) and NSC.
- **Hardware/Platform Dependent**: This must be adapted to a specific MCU/platform used as an application processor. Reference implementation is provided in this SDK. The PLAT component belongs to this group.

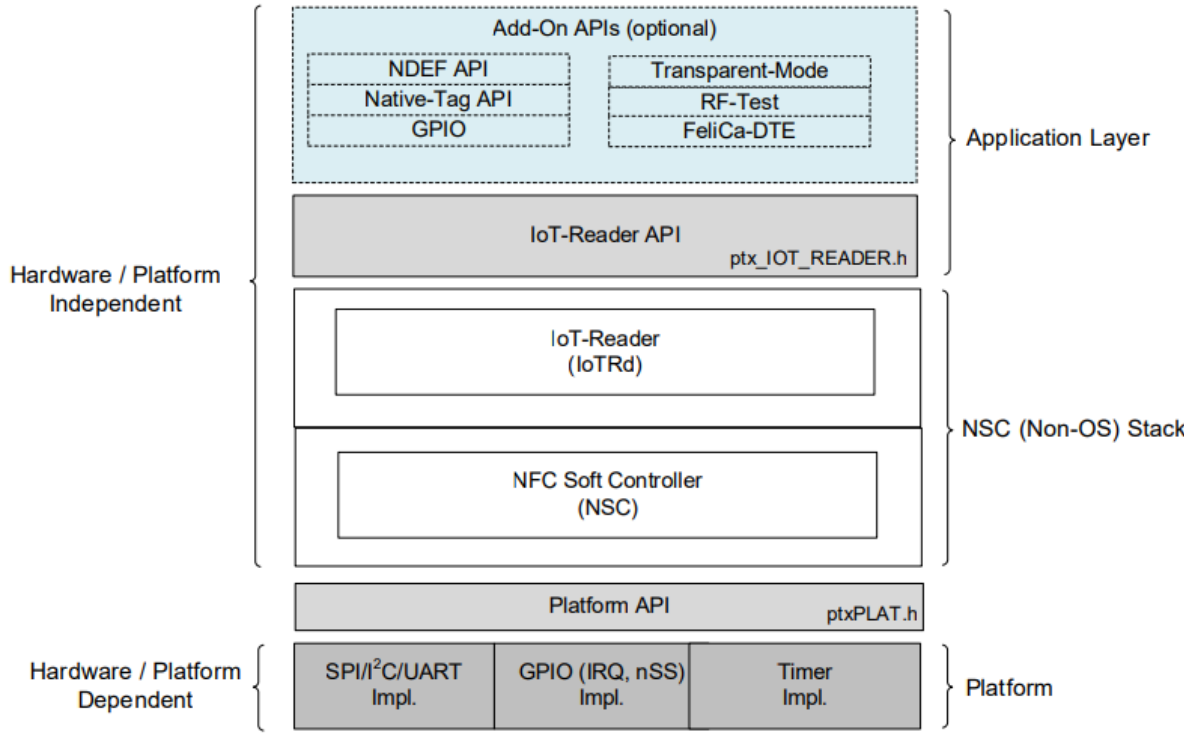

Figure 7. IoT-Reader (Non-OS) Stack Architecture

For detailed information on the Component and API descriptions, refer to the PTX1xxR NFC IoT-Reader API for Non-OS Stack Integration (SDK v7.2.0) User Manual.

3.4.1 Compiler-Specific Implementation

To port the original IoT-Reader (Non-OS) stack to RL78/G23, some compiler specific updates must occur (for more information, refer to IoT-Reader document link in section 1).

The Toolchain used for this demo project is Renesas CC-RL v1.14.00. All of the following updates apply only to Renesas CC-RL.

In order to improve the flexibility and structure of the SDK content, all modifications related to specific usage of the Renesas CC-RL Toolchain were isolated from the rest of the project using a compiler switch macro **PTX_RL78_CCRL**. All of the required modifications are explained in the following sections.

3.4.1.1 Variable Length Arrays(VLAs)

In CC-RL, Variable Length Arrays(VLAs) are not supported because they are a feature of the C99 standard; CC-RL primarily adheres to the C90 standard with limited C99 extensions.

In the original IoT-Reader (Non-OS) stack, a variable length for arrays is used to improve the stack memory usage at runtime. For CC-RL, these are compiler errors and solving them is required.

The solution for this porting demo was to use the maximum size arrays instead of variable runtime arrays.

An example of the RL78/G23 Porting in such cases is shown in Figure 8.

```
static ptxStatus_t ptxNSC_RfConfig_Send(ptxNSC_t *nscCtx, ptxNSC_RfConfigTlv_t *nsc_rf_cfg_params, uint8_t rfconfig_tlv_count)
{
    /*
     * This function parses TLV data and fills output buffer in an optimal way (to cut down the number of transmissions).
     */
    ptxStatus_t status = ptxStatus_Success;

    const uint8_t nsc_rfConfig_cmd_len = 255u;
    uint8_t nsc_rfconfig_minimum_length = 17u;      /* ptxNSC_RfConfig_RegsPollV_len (15) + config type (1) + empty config (1)*/
    uint8_t nsc_rfConfig_index = 0;

#ifdef PTX_RL78_CCRL
    uint8_t nsc_rfConfig_cmd[255u];
    /* Mark TLVs that were already sent. So far not anyone used. */
    uint8_t used_indices[MAX_BUFFER_SIZE];
#else
    uint8_t nsc_rfConfig_cmd[nsc_rfConfig_cmd_len];
    /* Mark TLVs that were already sent. So far not anyone used. */
    uint8_t used_indices[rfconfig_tlv_count];
#endif
```

Figure 8. Required Updates for VLAs

3.4.1.2 True Constant Usage

In CC-RL, not all constant definitions are allocated in ROM memory and considered by the compiler as “true” constants. The side-effect of not considering a constant as a “true” constant is again array declaration based on a constant size. For CC-RL this is a compiler error.

To improve modularity in the original IoT-Reader (Non-OS) stack, a lot of local constants definition was used instead of traditional **#defines**.

The solution for this porting demo was to remove all local constants definition and to replace it with local **#defines** or, depending by case, directly using the hardcoded value.

An example of the RL78/G23 Porting in such cases is shown in Figure 9.

```
void ptxNSC_GetRx (ptxNSC_t *nscCtx)
{

#ifdef PTX_RL78_CCRL
    uint8_t rx_buff[RX_BUFFER_MAX_SIZE];

    uint8_t *rxBuf[NUMBER_MAX_BUFFERS];
    size_t *rxLen[NUMBER_MAX_BUFFERS];
#else
    const size_t RX_BUFFER_MAX_SIZE = 256;
    const size_t NUMBER_MAX_BUFFERS = 1;

    uint8_t rx_buff[RX_BUFFER_MAX_SIZE];


    uint8_t *rxBuf[NUMBER_MAX_BUFFERS];
    size_t *rxLen[NUMBER_MAX_BUFFERS];
#endif
```

Figure 9. "True" Constant Solution

3.4.1.3 Ternary Operator Warning

When using a ternary operator (also referred to as “short-if”), Renesas CC-RL requires both operands to have the same type, otherwise a warning is returned. The quickest solution for this is to use traditional “if-else” implementations (see Figure 10).

```
#ifdef PTX_RL78_CCRL
                if(msg_len >= bytes_read)
                {
                    (void)memcpy(&msgBuffer[0],&t4tOpComp->RxBuffer[0],bytes_read);
                }
                else
                {
                    status = PTX_STATUS(ptxStatus_Comp_T4TOP, ptxStatus_InsufficientResources);
                }
#else
                (msg_len >= bytes_read) ?
                        ((void)memcpy(&msgBuffer[0],&t4tOpComp->RxBuffer[0],bytes_read)) :
                        (status = PTX_STATUS(ptxStatus_Comp_T4TOP, ptxStatus_InsufficientResources));
#endif
```

Figure 10. Short-if Usage

3.4.2 Device-Specific Implementation

The RL78 family of microcontrollers offers a unique memory segmentation by dividing the memory between near memory and far memory, primarily to optimize access time and to provide extended addressing capabilities.

- Near memory uses 16-bits address space, being accessed by default addressing mode
- Far memory is accessed using extended addressing instructions

To improve the memory access speed, the RL78 family offers a “Mirror” section, where a configured sector from constant data can be copied to “Mirror”. However, in this porting demo, this feature can’t be used because the required data sized stored in constant section is higher than the “Mirror” size.

The IoT-Reader (Non-OS) SDK proposal of a two stack solution requires the storage of large buffers in the constant memory section (addressed to FAR memory), buffers which must be exchanged via serial communication in order to implement the firmware download. This chosen solution combines flexibility and robustness in integrating the NFC Chip with Host Controller.

Due to the original IoT-Reader(Non-OS) SDK not being designed for such a memory architecture, some data accesses have to be modified in order to ensure the correct behavior.

The issue with the RL78/G23 is pointers handling. Because of its unique memory architecture, if a normal pointer is used, it automatically points to NEAR memory and the correct reference to constant section is lost. The proposed solution for this demo is to use pointer to constant instead of normal pointers (see Figure 11).

```
 * \brief Internal command to exchange NSC-commands using buffers.
ptxStatus_t ptxNSC_HAL_WriteBuffer(struct ptxNSC *nscCtx, ptxNscHal_BufferId_t bufferId, const uint8_t *txBuf[], size_t txLen[], size_t numBuffers);

 * \brief Internal command to write certain SFRs.
ptxStatus_t ptxNSC_HAL_Wra(struct ptxNSC *nscCtx, uint16_t address, uint8_t value);

 * \brief Internal command to write certain SFRs (operating-mode dependent).
ptxStatus_t ptxNSC_HAL_Wra_NoWait(struct ptxNSC *nscCtx, uint16_t address, uint8_t value);

 * \brief Internal command to write certain SFRs (operating-mode dependent).
ptxStatus_t ptxNSC_HAL_Wra_NoCheck(struct ptxNSC *nscCtx, uint16_t address, uint8_t value);

 * \brief Internal command to read certain SFRs.
ptxStatus_t ptxNSC_HAL_Rra(struct ptxNSC *nscCtx, uint16_t address, uint8_t *value);

 * \brief Internal command to write NSC-instructions.
ptxStatus_t ptxNSC_HAL_WriteInstruction(struct ptxNSC *nscCtx, uint16_t address, const uint8_t *pPayload, size_t txLen );
```

Figure 11. Updated APIs Declaration

The above two APIs declaration are just an example of required updates. For this porting demo, numerous changes like this, including the Smart Configurator generated files, are required.

3.4.3 Platform-Specific Implementation

As discussed in section 3.4, the system architecture of IoT-Reader(Non-OS) SDK requires two system components:

- **Host controller** – represented in the case of this porting Demo by RL78/G23 128p
- **NFC Chip** – represented by PTX105R-QC compact board

In order to ensure a flexible solution, the user is responsible to implement the interfacing between the two system components.

For the software architecture, this layer is represented by the Platform component.

This component is dependent on the platform/MCU application processor being used. It also depends on the physical hardware interface being used (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. For this specific porting demo, SPI was chosen as the connectivity interface.
- **GPIO**: Implements the driver for the GPIO used for IRQ. This submodule is needed when SPI or I²C are used as the 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.

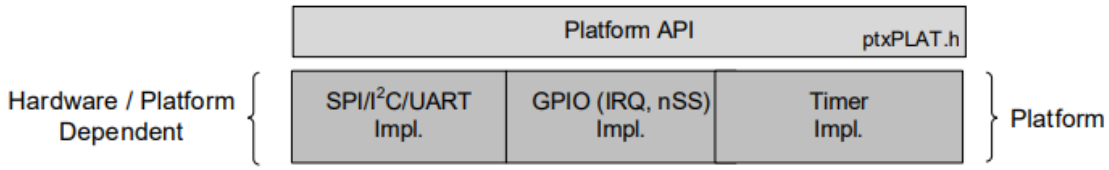

Figure 12. Platform Layer

A detailed description of every API can be found in PTX1xxR NFC IoT-Reader API for Non-OS Stack Integration (SDK v7.2.0) User Manual.

The delivered package contains a reference implementation of the platform layer applicable for RL78/G23 128p device. The user is responsible to adapt or implement the required APIs on a new device. Each layer is described in the following sections.

3.4.3.1 Timer

An interval timer is needed to implement fixed a duration sleep or non-blocking timer.

The timer resource is configured in Smart Configurator from the *Components* tab **> Add Component > Interval Timer**. The configuration name is important because it becomes the generated files name.

Operation mode is chosen to 16-bits count mode. One available resource must also be chosen.

After the timers driver is added, there are additional configurations to be performed. For the interval timer, a duration must be set and some APIs to start/stop the timers are generated.

If choosing to use the same timer name **Config_TAU0_1** and same resource **TAU0_1**, and the same configurations, the code doesn't need to be adapted since generation will not overwrite the one provided.

For the purpose of this demo, the timer name is **Config_TAU0_1**. Thus, three files based on this name are generated and adaption instructions are:

- **Config_TAU0_1.h**

  This is the public interfaces file. All the accessible APIs are available and within this file. Two more APIs were also added:

  - *uint16_t UserCode_R_Config_TAU0_1_Start(uint32_t Timeout_ms)* – used to start the timer with a desired duration received in milliseconds resolution.

- *uint32_t UserCode_R_Config_TAU0_1_GetTimerFrequency(void)* – used to compute the timer configured frequency.

The following APIs must also be added if the drivers are regenerated. It is recommended to add them under special comments to avoid overwriting.

- **Config_TAU0_1.c**

  In this file, all the APIs declared are defined in the public header file. A reference implementation is provided for custom APIs *UserCode_R_Config_TAU0_1_Start()* and *UserCode_R_Config_TAU0_1_GetTimerFrequency()*.

  These implementations must be adapted to the current configuration (for example, register TDR01 can be different and can be found in *R_Config_TAU0_1_Create()*).
- **Config_TAU0_1_user.c**

  In this file, the timer interrupt is handled. The flag **timeoutFlag** is set when ISR is invoked. Ensure that **timeoutFlag** is correctly handled.

3.4.3.2 Interrupt

A GPIO external interrupt is used to trigger the Host controller(RL78/G23) from NFC chip (PTX1xxR). The interrupt can be added from the Smart Configurator by selecting *Components* tab **> Add Component > Interrupt Controller**.

The configuration allows the selection of the specific interrupt, the edge detection and the priority. The specific interrupt is allocated to one pin, so choosing a specific interrupt does not require a pin to be chosen – it is automatically managed.

When using the same name **Config_INTC** and the same **INTP6** configurations, the code doesn't need to be adapted since generation will not overwrite the one provided.

For the purpose of this porting demo, the name is **Config_INTC**. Thus, three code files are generated as shown below. In the event of regenerating the files, ensure that **Intp6Flag** is well-handled by ISR.

- **Config_INTC.h**

  In this file, the public APIs are declared. No additional APIs are needed.
- **Config_INTC.c**

  In this file, the public APIs are defined. No additional implementations are needed.
- **Config_INTC_user.c**

  In this file, the ISR is handled and **Intp6Flag** is initialized and set.

3.4.3.3 Communication

To provide extensive flexibility over the hardware set-up, the firmware code for PTXxxR is included in the SDK and flashed at initialization.

The SPI role is to transfer the FW code to the PTXxxR and to exchange runtime commands. Since the FW code is saved in the **.constf** memory section, some adaption needs to be implemented for generated files.

The SPI driver is added from the *Components* tab **> Add Component > SPI (CSI) Communication**. A name must be given, the operation must be selected, and the resource must be chosen. For SPI, the pins must also be configured from the *Pins* tab.

If the same name and the same configurations are used for the SPI driver, the code doesn't need to be adapted since generation will not overwrite the one provided.

For the purpose of this demo, the file name is **Config_CSI30**. Thus, three code files are generated as shown below. In the event of regenerating or reconfiguration, ensure the files are adapted to the **ptxPlat.c** requirements.

- **Config_CSI30.h**

  In this file, the public APIs are declared. ***Important***: The following generated API prototype must be changed in this file.

  For *MD_STATUS R_Config_CSI30_Send_Receive(const uint8_t * const tx_buf, uint16_t tx_num, uint8_t * const rx_buf)*, **tx_buf** must be a constant pointer to a constant. If not, the CC-RL compiler will automatically allocate the address into NEAR RAM and will no longer point to the **.constf** location of the firmware code.
- **Config_CSI30.c**

  In this file, the public APIs are defined. In the event of regenerating the files, ensure that the following modifications are implemented.

  SPI transfer function *R_Config_CSI30_Send_Receive* uses a global address for storing the data to transfer. Ensure that the global address (*gp_csi30_tx_address*) is a pointer to constant. If not, the CC-RL compiler will automatically allocate the address into NEAR RAM and will no longer point anymore to the **.constf** location of the FW code.
- **Config_CSI30_user.c**

  In this file, the SPI interrupts are handled. In the event of regenerating, ensure that interrupt flags *txEndTransfer*, *transferError*, and *rxEndTransfer* are well handled. The same modification for global address(*gp_csi30_tx_address*) must also be implemented.

The implementation of additional modifications can not be separated by special comments sections and will be overwritten at each code regeneration. ***Important***: After SPI is properly functioning, generate only initialization APIs. The rest of APIs are copied and handled under special comments sections.

4. Hardware

This section describes all of the necessary hardware components required to run the IoT-Reader (Non-OS) stack on an RL78/G23 device and how to successfully discover a card.

For the purpose of this demo, the RL78/G23 with 128 pins is the reference device and uses the Host Controller side in the “two stack” solution provided by Renesas. The PTX105RQC is used as NFC Controller. The two devices are connected via SPI.

4.1 Hardware Documentation

- RL78/G23-128P Fast Prototyping Board: RTK7RLG230CSN000BJ – RL78/G23-128p
- PTX105R-QC: PTX105RQC – PTX105R PMOD™ Board for IoT

4.2 Hardware Tools

4.2.1 RL78/G23-128P Fast Prototyping Board

The RL78/G23 microcontroller group is a new generation of the RL78 family of microcontrollers with 41µA/MHz CPU operation. The RL78/G23 group has the industry's lowest power consumption with 210nA at stop (4KB SRAM retention) and a snooze mode sequencer which significantly reduces power consumption during intermittent operation. The RL78/G23 group features a wide operating voltage range of 1.6V to 5.5V (up to 32MHz), a broad range of package pin counts (from 30 pins to 128 pins), and up to 768KB of flash memory.

Figure 13. RL78/G23-128P Fast Prototyping Board

4.2.2 PTX105R-QC

The PTX105R PMOD™ board is fully integrated into the QuickConnect IoT ecosystem and provides a full-featured, high-end NFC reader solution for easy integration. Using the standard PMOD form factor allows NFC evaluation on many Renesas MCU evaluation kits. The NFC stack is pre-compiled on the RA2E1 48MHz Arm® Cortex®-M23 general-purpose microcontroller.

QuickConnect IoT addresses issues related to the creation of a custom solution of MCUs, connectivity, and sensors by providing standard hardware and software building blocks to enable quick prototyping of IoT systems.

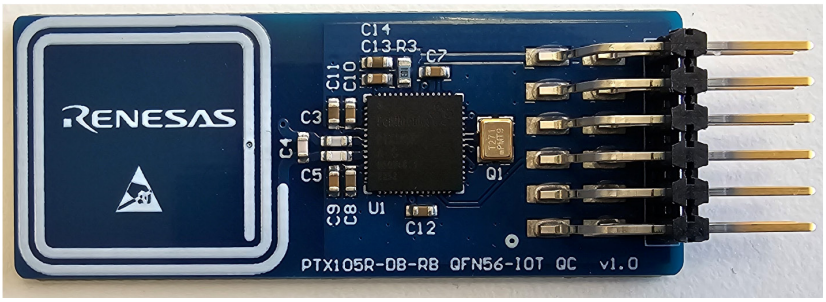

Figure 14. PTX105RQC PMOD Board

4.3 Hardware Setup

The physical connection between the two boards is achieved with a PMOD™ connector.

The RL78/G23-128P board has two PMOD™ connectors, PMOD1 and PMOD2. For the purpose of this demo, PMOD1 is configured for data exchange with the PTX105R-QC and the two boards are connected using the PMOD1 connector.

The RL78/G23-128P board also has an on-board Debugger. To flash and debug the device, connect the RL78/23-128P to a PC using a USB to microUSB cable.

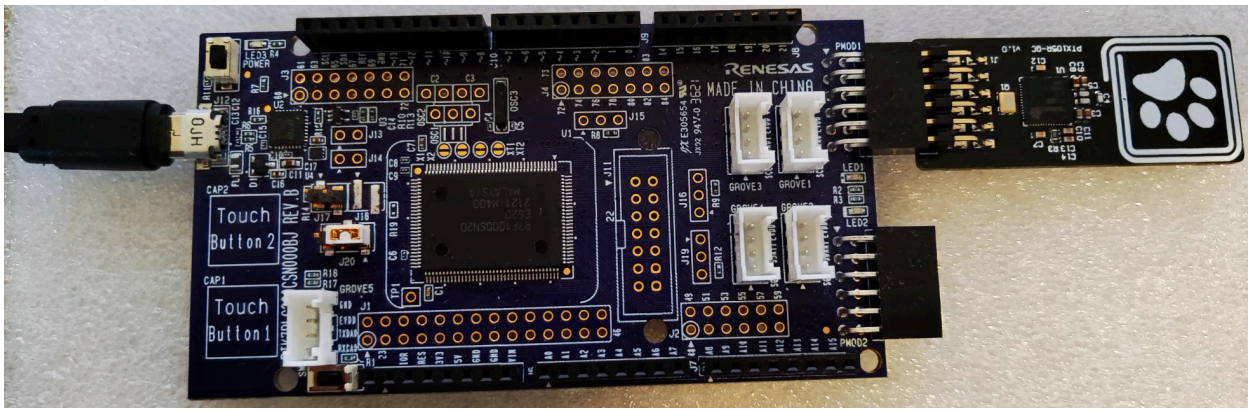

Figure 15. Complete Hardware Setup

Note: For PTX105R-QC, everything is connected via PMOD so no additional wiring is needed, including powering the device.

5. Porting from e2 Studio to CS+

To automatically port from e2 Studio to CS+, refer to the webpage Porting from the e² studio to CS+.

6. Conclusions

The RL78 memory architecture requires implementing some custom modifications to the generated files from *Smart Configurator*. To simplify the modifications handling process, the original generated code is provided in the delivery package.

The user can then view a comparison of exactly what and where changes, or modifications, occurred. The original package is **smc_gen_Unmodified**. This folder is excluded from the build.

All the required updates to the original IoT-Reader(Non-OS) Stack are applicable to the Renesas CC-RL Toolchain.

This porting demo implements the reference design applicable to the RL78/G23 128p device. Adapting and modifying the required APIs so that porting is successful on other devices in the RL78 family is the role of the user.

7. Revision History

Revision	Date	Description
1.00	Jan 28, 2025	Initial release.

IMPORTANT NOTICE AND DISCLAIMER

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

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

(Disclaimer Rev.1.01)

Corporate Headquarters

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

Contact Information

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

Trademarks

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