IoT-Reader (Non-OS) SDK V7.2.0 Integration to FreeRTOS-based System Process

Introduction

This document describes additional modifications required to achieve the integration of the IoT-Reader (Non-OS) SDK v7.2.0 into a FreeRTOS-based system using E^{2}Studio.

In the context of FreeRTOS scheduling, IoT-Reader stack is used as one single thread solution which has the advantages of using a simplified design, low resource consumption, and deterministic behavior of the NFC component.

Contents

Figures

1. Requirements

This document applies to:

- IoT-Reader (Non-OS) SDK for PTX1xxR family v7.2.0

The full documentation of IoT-Reader (Non-OS) SDK v7.2.0 can be found in the *PTX1xxR NFC IoT-Reader API for OS Stack Integration (SDK v7.2.0) User Manual*.

2. About FreeRTOS

FreeRTOS™ is a lightweight, open-source real-time operating system designed for embedded systems. It provides a simple and efficient kernel to manage tasks, scheduling, and resources, making it ideal for microcontrollers and small processors with limited resources.

The complete documentation for FreeRTOS is available at FreeRTOS documentation - FreeRTOS.

3. Integration Process

The integration process discussed in this document handles all aspects of correctly deploying the IoT-Reader (Non-OS) stack into a FreeRTOS-based system, starting with project creation and finishing with code adaptation required to fulfill a real-time operating system compliance.

In order to keep the integration as simple as possible, the objective is to create a single thread for the whole IoT-Reader stack. The one single thread solution provides numerous advantages. Beside simplified design which does not require any synchronization mechanism or data exchange between internal components, there is also the resources aspect and the deterministic behavior of the entire stack to be considered.

The required integration steps are summarized as follows:

1. Create and configure a FreeRTOS project using E^{2}Studio and FSP.
2. Import IoT-Reader (Non-OS) SDK v7.2.0 into the new created project.
3. Create the IoT-Reader thread and configure required stacks(drivers).
4. Update the HAL layer of IoT-Reader (Non-OS) SDK v7.2.0 to make it compliant with FreeRTOS.
5. Link IoT-Reader thread to IoT-Reader stack.

3.1 How to Start a FreeRTOS-based Project using E2Studio

If a FreeRTOS project is not yet created, the first step to integrate IoT-Reader (Non-OS) SDK v7.2.0 into a FreeRTOS-based system is to create a new E2Studio project proper configured for using FreeRTOS.

For demonstration purposes, an RA4M3 device is used as the host controller.

1. Open E^{2}Studio and navigate to File > New > Renesas C/C++ Project and select Renesas RA.
2. Select the C/C++ Template and click Next.
3. Choose an appropriate project name and click Next.

4. Select the desired device and desired settings and click Next.

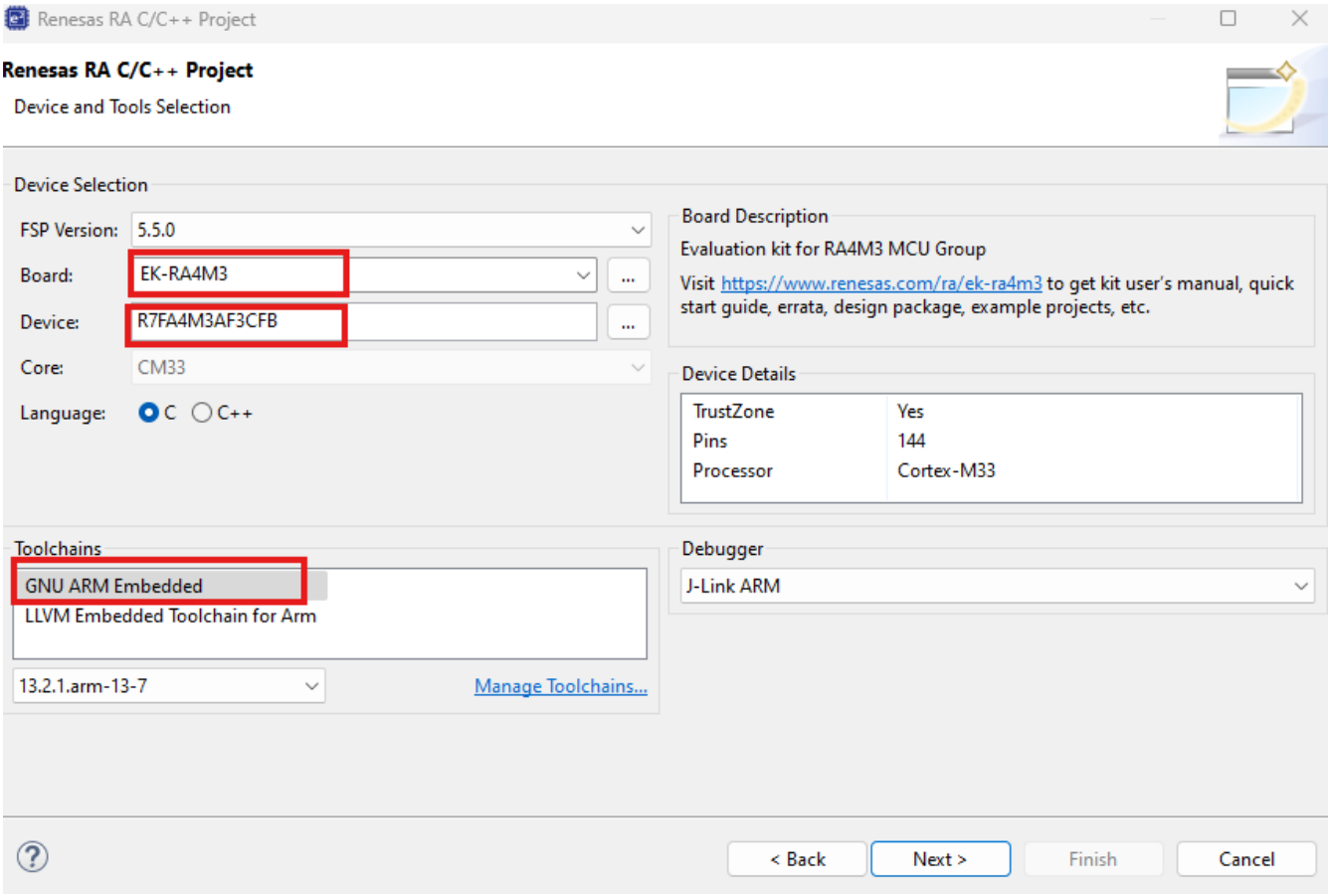

Figure 1. E²Studio Project Device Configuration

5. Select the project type and click Next.
6. At this step, it is necessary to select the build artifacts and RTOS, so this is the step were FreeRTOS selection occurs. Select FreeRTOS and click Next.

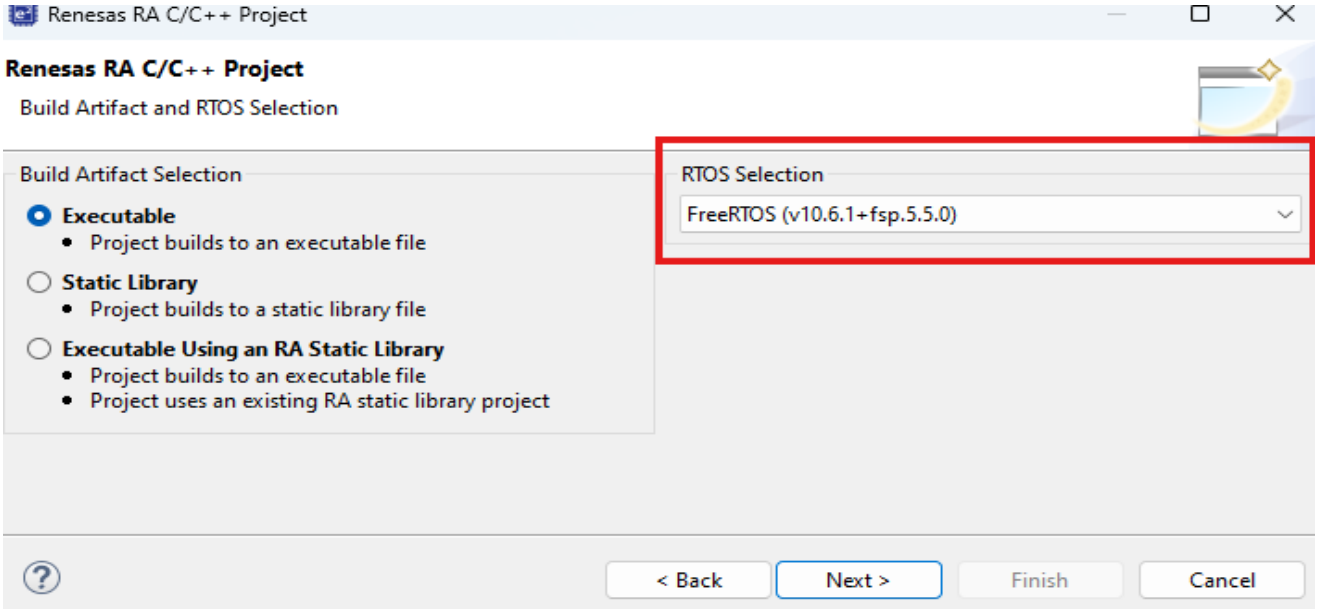

Figure 2. Project FreeRTOS Configuration

7. At this step it is recommended to choose the Blinky example as it will be more relevant for testing to have at least two threads active. Click on Finish and the project configuration is ready.

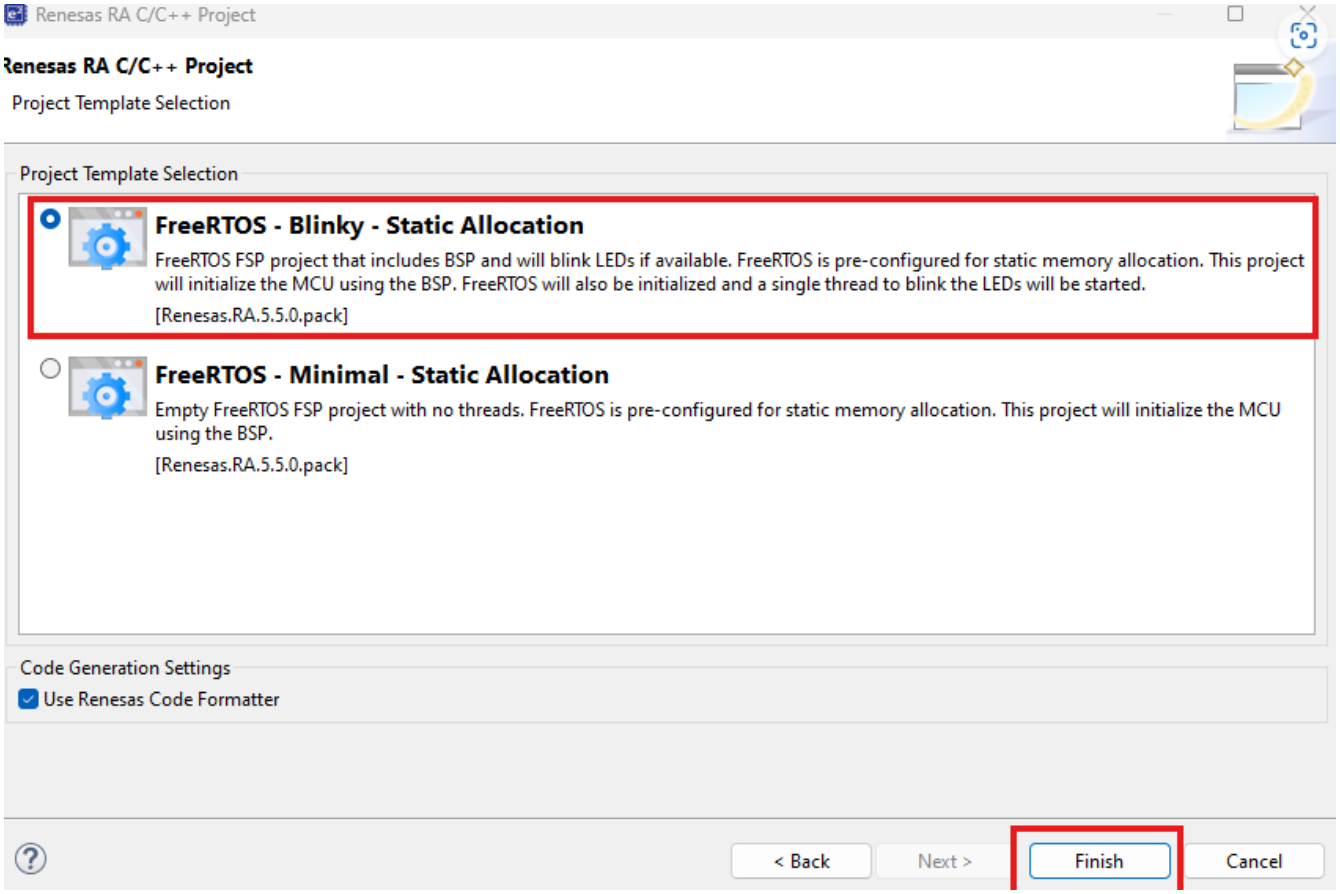

Figure 3. Project Template Selection

After completing the above steps the project is compilable and ready to be customized.

Note: The E^{2}Studio project provides only static memory allocation for FreeRTOS projects.

3.2 Create a New Thread for IoT-Reader Stack

New Threads can be created from the FSP Stacks tab. After creating the thread dedicated to IoT-Reader the most important configurations are priority and stack size.

The whole corresponding code to the thread configuration and thread initialization will be automatically generated by FSP. In addition, all the allocated stacks(drivers) will be configured and initialized in the thread context.

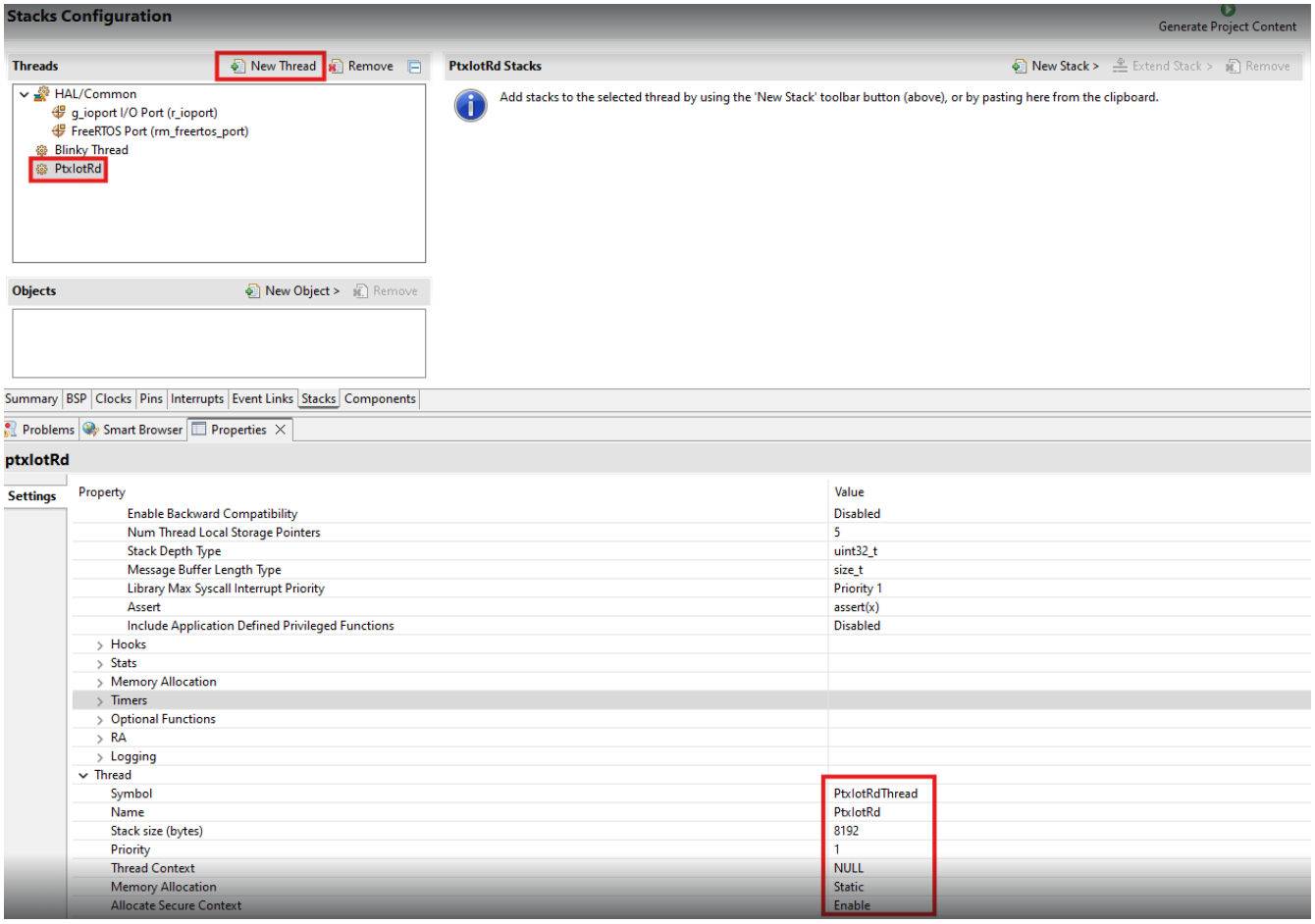

Figure 4. IoT-Reader Thread Configuration

Use a stack size of at least 8192 bytes. Once the thread is created, the code can be generated. A new file will be generated for each configured thread.

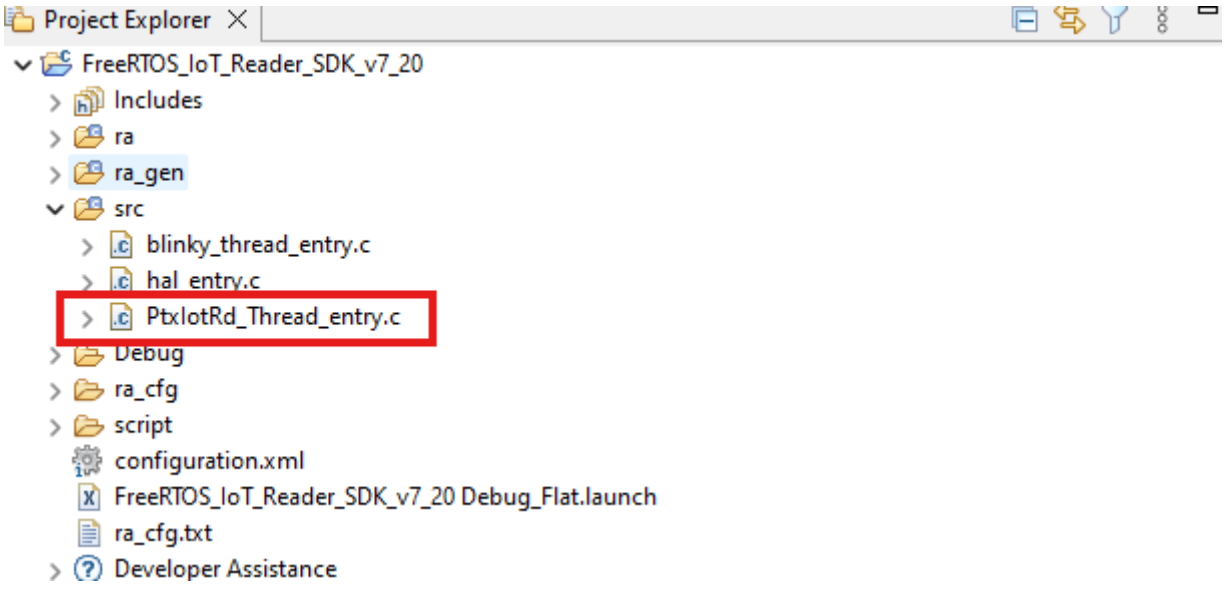

Figure 5. IoT-Reader Thread Generated Files

3.3 Configure Stacks for IoT-Reader Thread

In FSP Stacks tab, for each thread all the stacks(drivers) must be configured. For the IoT-Reader Thread the stacks(drivers) must be configured the same way as for the IoT-Reader (Non-OS) SDK v7.2.0 project.

Note: The full documentation of IoT-Reader (Non-OS) SDK v7.2.0 can be found in *PTX1xxR NFC IoT-Reader API for OS Stack Integration (SDK v7.2.0) User Manual*.

One thing worth noting, the stacks(drivers) allocated to a specific thread are “private” and cannot be accessed by other threads. If it is the case to define a driver that can be consumed by multiple threads, the HAL/Common section can be used. The recommendation is to try as much as possible to keep the project compact and well separated by allocating correctly the hardware resources to threads.

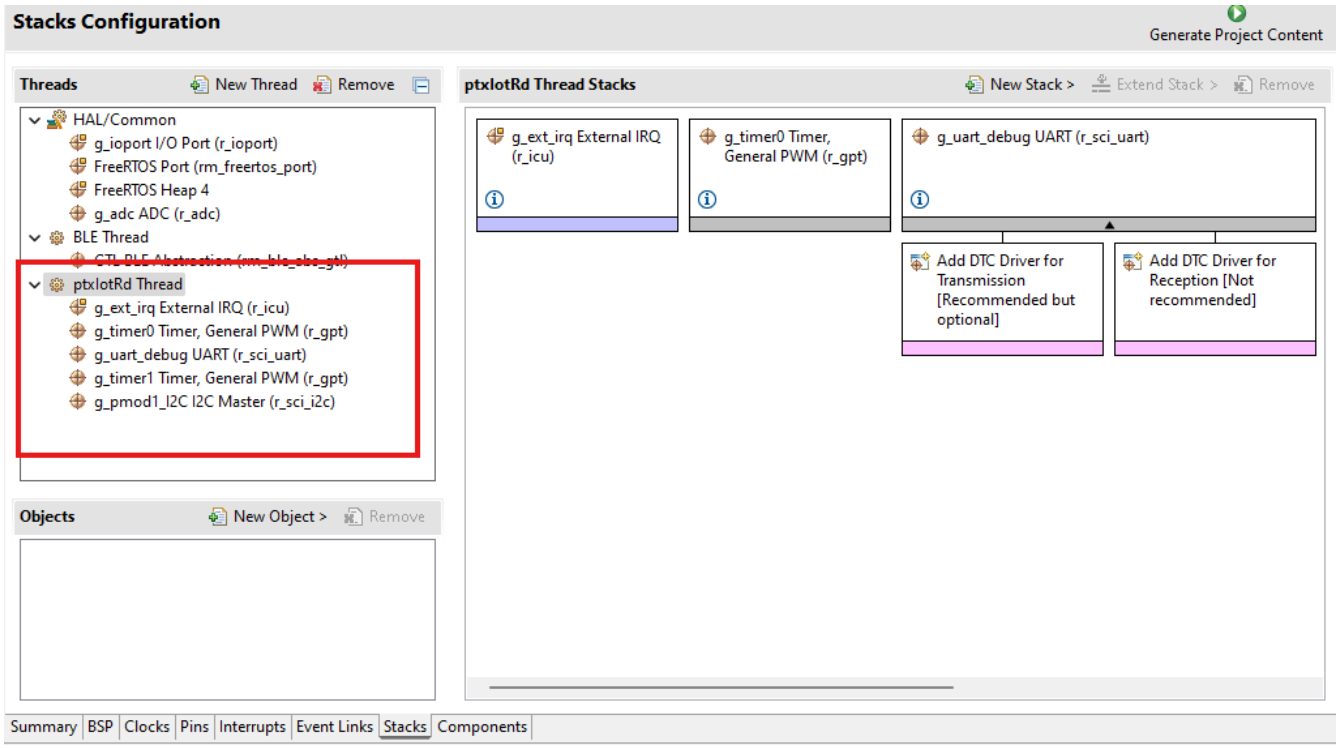

Figure 6. IoT Reader Stacks Configuration

The stacks(drivers) content and detailed configuration can be found in the original IoT-Reader (Non-OS) SDK v7.2.0 project.

3.4 Import the IoT-Reader (Non-OS) SDK v7.2.0 Code Content into the Project

After all the FSP configurations, readying the code generation is possible. FSP will generate all necessary infrastructure including the thread entry function for the newly created thread for IoT-Reader.

The next step is to import into the project the code sources from IoT-Reader (Non-OS) SDK v7.2.0. This step can be achieved by creating a new “Source folder” in the project and pasting the code folders inside. When doing so, however, the user must ensure the following:

- The files are not excluded from build.
- To add the new added source files to compiler include paths.
- To add the same project Maros as for IoT-Reader (Non-OS) SDK v7.2.0 project.

At the end of this section the whole project should compile, but from a functional point of view, it will not do anything because the IoT-Reader thread activity is not linked with IoT-Reader stack activity.

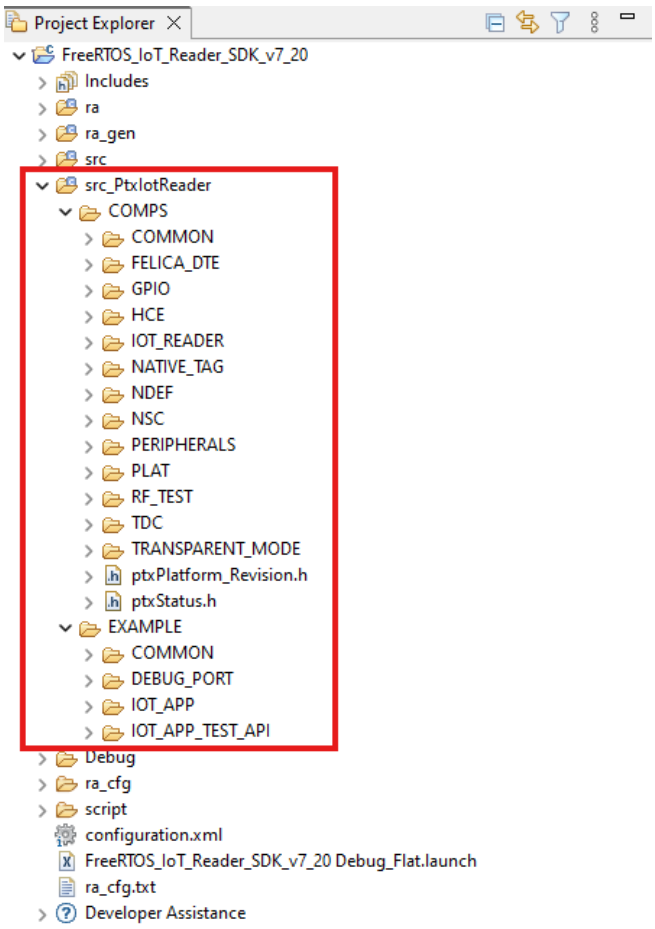

Figure 7. IoT-Reader Source Code Imported

3.5 IoT-Reader (Non-OS) SDK v7.2.0 Code Updates

After following the above steps, the project should be compilable. In order to make it work, there are some required updates to be completed.

FSP already generates the thread initialization and the thread entry point, so the IoT-Reader thread entry is already available. There are two design updates that should be completed in order to make the IoT-Reader stack working.

3.5.1 Implement “blocking-states” using Binary Semaphores

In the implementation of IoT-Reader (Non-OS) SDK v7.2.0, there are some use-cases where the host controller will wait for events in a blocking way, which effectively monopolizes the processor’s execution time and prevents the FreeRTOS scheduler from executing other tasks. Of course, this implementation is forbidden in an RTOS-based system because it will block the other tasks.

The simplest solution possible to achieve the expected behavior is to use the binary semaphore mechanism offered by FreeRTOS. This mechanism is very simple: using a “Take” operation from any place of the task context will block the task until the semaphore is freed from an ISR using a “Give” operation.

The semaphore must be declared and initialized before any usage, so before starting any activity make sure the semaphore was initialized correctly.

- ***ptxPLAT_WaitForInterrupt*** is implemented, like the name suggests, to wait for interrupts (from internal timers or external GPIO) in a closed loop, blocking the CPU. The new implementation just “takes” the semaphore and informs the scheduler to block the current thread until the semaphore is free.

```
ptxStatus_t ptxPLAT_WaitForInterrupt(ptxPlat_t *plat)
{
    ptxStatus_t status = ptxStatus_Success;

    if (PTX_COMP_CHECK(plat, ptxStatus_Comp_PLAT))
    {
        /* Wait for Interrupts */
        __DSB();
        __WFI();
        __ISB();

    } else
    {
        status = PTX_STATUS(ptxStatus_Comp_PLAT, ptxStatus_InvalidParameter);
    }

    return status;
}
```

```
ptxStatus_t ptxPLAT_WaitForInterrupt(ptxPlat_t *plat)
{
    ptxStatus_t status = ptxStatus_Success;

    if (PTX_COMP_CHECK(plat, ptxStatus_Comp_PLAT))
    {
        /* Wait for Interrupts */
        //RTOS Implementation. Lock the thread until the interrupt occurs
        xSemaphoreTake(xPtxIotRdSemaphore, portMAX_DELAY);

    } else
    {
        status = PTX_STATUS(ptxStatus_Comp_PLAT, ptxStatus_InvalidParameter);
    }

    return status;
}
```

Figure 8. ptxPLAT_WaitForInterrupt Implementation

- ***ptxPLAT_TIMER_Start*** is used to start a timer in both blocking and non-blocking modes. For non-blocking mode there is nothing to update. However, for the blocking mode, the same implementation strategy as for previous function must be done. One important item worth noting is that the second implementation was slightly modified to unlock the thread even if the semaphore is free from GPIO ISR. More about this item is provided later in this section where the semaphore “give” strategy is discussed.

```
ptxStatus_t ptxPLAT_TIMER_Start(ptxPlatTimer_t *timer, uint32_t ms, uint8_t isBlock, pptxPlat_TimerCallBack_t fnISRCb, v
{
    ptxStatus_t status = ptxStatus_Success;

    if ( (NULL != timer) && (ms > 0) && ((0 == isBlock)||(1u == isBlock)))
    {
        /** Clear IsElpased state */
        timer->IsElapsed = 0;

        /** Stop Timer, first. Then, set period. Finally, start timer counter. */
        timer_instance_t *timer_instance = (timer_instance_t *)timer->TimerInstance;
        fsp_err_t r_status = R_GPT_Stop(timer_instance->p_ctrl);

        if(FSP_SUCCESS == r_status)
        {
            uint32_t timer_freq_hz = R_FSP_SystemClockHzGet(FSP_PRIV_CLOCK_PCLKD) >> timer_instance->p_cfg->source_div;
            uint32_t period_counts = (uint32_t) (((uint64_t) timer_freq_hz * ms) / 1000);
            r_status = R_GPT_PeriodSet(timer_instance->p_ctrl, period_counts);
        }

        if (FSP_SUCCESS == r_status)
        {
            timer->ISRCallBack = fnISRCb;
            timer->ISRCxt = ISRCxt;

            r_status = R_GPT_Start(timer_instance->p_ctrl);

            if(FSP_SUCCESS == r_status)
            {
                timer->TimerState = Timer_InUse;
                if (1u == isBlock)
                {
                    * WFI until Timer is elapsed. Suitable for Pause() functionality.
                    while(0 == timer->IsElapsed)
                    {
                        __DSB();
                        __WFI();
                        __ISB();
                        __NOP();
                    }
                }
            }
        }
    }
```

```
ptxStatus_t ptxPLAT_TIMER_Start(ptxPlatTimer_t *timer, uint32_t ms, uint8_t isBlock, pptxPlat_TimerCallBack_t fnISRCb,
{
    ptxStatus_t status = ptxStatus_Success;

    if ( (NULL != timer) && (ms > 0) && ((0 == isBlock)||(1u == isBlock)))
    {
        /** Clear IsElpased state */
        timer->IsElapsed = 0;

        /** Stop Timer, first. Then, set period. Finally, start timer counter. */
        timer_instance_t *timer_instance = (timer_instance_t *)timer->TimerInstance;
        fsp_err_t r_status = R_GPT_Stop(timer_instance->p_ctrl);

        if(FSP_SUCCESS == r_status)
        {
            uint32_t timer_freq_hz = R_FSP_SystemClockHzGet(FSP_PRIV_CLOCK_PCLKD) >> timer_instance->p_cfg->source_div;
            uint32_t period_counts = (uint32_t) (((uint64_t) timer_freq_hz * ms) / 1000);
            r_status = R_GPT_PeriodSet(timer_instance->p_ctrl, period_counts);
        }

        if (FSP_SUCCESS == r_status)
        {
            timer->ISRCallBack = fnISRCb;
            timer->ISRCxt = ISRCxt;

            r_status = R_GPT_Start(timer_instance->p_ctrl);

            if(FSP_SUCCESS == r_status)
            {
                timer->TimerState = Timer_InUse;
                if (1u == isBlock)
                {
                    //RTOS Implementation. Lock the thread until the interrupt occurs
                    xSemaphoreTake(xPtxIotRdSemaphore, portMAX_DELAY);
                }
            }
        }
    }
```

Figure 9. ptxPLAT_TIMER_Start Implementation

- ***ptxPLAT_GPIO_IsrCallback*** is used to handle any interrupt detected by the IRQ(***INTPx***) pin. On the non-OS SDK, there is no specific action to do because the CPU is directly waked up by hardware mechanism; however, on the FreeRTOS implementation the semaphore must be free. There is no need for complex implementation to check if the semaphore is currently blocked before freeing it. It is very important here to use ***xSemaphoreGiveFromISR*** instead of ***xSemaphoreGive***.

```
void ptxPLAT_GPIO_IsrCallback(external_irq_callback_args_t *p_args)
{
    (void)p_args;
}
```

```
void ptxPLAT_GPIO_IsrCallback(external_irq_callback_args_t *p_args)
{
    (void)p_args;
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;
    xSemaphoreGiveFromISR(xPtxIotRdSemaphore, &xHigherPriorityTaskWoken);
}
```

Figure 10. ptxPLAT_GPIO_IsrCallback Implementation

- ***ptxPLAT_TIMER_IsrCallback*** is used to manage timer interrupts. One the same logic like for ISR callback, in FreeRTOS version the semaphore must be freed.

```
void ptxPLAT_TIMER_IsrCallback(timer_callback_args_t *p_args)
{
    * PERIODIC_MODE is used for timer operation:

    timer_ctx.IsElapsed = 1u;

    /*Let's call back if defined. */
    if (NULL != timer_ctx.ISRCallBack)
    {
        timer_instance_t *timer_instance = (timer_instance_t *)timer_ctx.TimerInstance;
        R_GPT_Stop(timer_instance->p_ctrl);

        if (NULL != timer_ctx.ISRCxt)
        {
            timer_ctx.ISRCallBack(timer_ctx.ISRCxt);
        }
    }

    (void)p_args;
}
```

```
void ptxPLAT_TIMER_IsrCallback(timer_callback_args_t *p_args)
{
    * PERIODIC_MODE is used for timer operation:

    timer_ctx.IsElapsed = 1u;
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /*Let's call back if defined. */
    if (NULL != timer_ctx.ISRCallBack)
    {
        timer_instance_t *timer_instance = (timer_instance_t *)timer_ctx.TimerInstance;
        R_GPT_Stop(timer_instance->p_ctrl);

        if (NULL != timer_ctx.ISRCxt)
        {
            timer_ctx.ISRCallBack(timer_ctx.ISRCxt);
        }
    }

    (void)p_args;
    xSemaphoreGiveFromISR(xPtxIotRdSemaphore, &xHigherPriorityTaskWoken);
}
```

Figure 11. ptxPLAT_TIMER_IsrCallback Implementation

3.5.2 Link IoT-Reader APIs to Thread Entry Function

In the previous section all the required updates to make the IoT-Reader stack compatible with a FreeRTOS-based system were presented. Now, the final topic is how to link the thread entry function generated from FSP to the actual APIs offered by the IoT-Reader stack.

The first option is a basic one but will ensure the same behavior as for IoT-Reader non-OS version: a continuous pooling loop. The only necessary update is to directly call ***ptxAPP_Entry*** from the generated thread entry function, of course after initializing the semaphore.

The second proposed option is more complex, and it depends on the project needs and designer choices. The proposal is to rework the ***ptx_IOT_RD_Main.c*** file to work with individual commands and “control” the closed loop behavior using the Queue mechanism offered by FreeRTOS.

Queue mechanism provide a lot of advantages because it allows commands and data transfer from one thread to another, in this way giving the user a solution to create complex applications.

The following is a simple example of how to start the polling loop using queue commands, but this is highly dependent of the project needs. However, it is important to mention that this layer is outside of the IoT-Reader core functionality and is intended to be reworked by users.

```
/* ptxIotRd Thread entry function */
/* pvParameters contains TaskHandle_t */
void ptxIotRdThread_entry(void *pvParameters)
{
    FSP_PARAMETER_NOT_USED (pvParameters);

    /* TODO: add your own code here */
    xPtxIotRd_Queue = xQueueCreate(QUEUE_LENGTH, QUEUE_ITEM_SIZE);
    xPtxIotRdSemaphore = xSemaphoreCreateBinaryStatic(&xSemaphoreBuffer);

    if(( xPtxIotRdSemaphore == NULL )
     ||( xPtxIotRd_Queue == NULL ))
    {
        /* Critical Error */
        ptxCommon_PrintF(" Critical error: Resources not allocated. ");
    }
    else
    {
        /* Initialize ptx1k */
        /* This event is raised here just for example purpose. It shall be raised by higher application level */
        ptxIotRd_QueueCommand.callback = NULL;
        ptxIotRd_QueueCommand.ptxIotRd_QueueCommand = e_ptxIotRd_Init;

        if(pdPASS != xQueueSend(xPtxIotRd_Queue, &ptxIotRd_QueueCommand, portMAX_DELAY))
        {
            /* Critical Error */
            ptxCommon_PrintF(" Critical error: Enqueue event failed.");
        }

        /* start Pooling */
        /* This event is raised here just for example purpose. It shall be raised by higher application level */
        ptxIotRd_QueueCommand.callback = ptxIotRd_NdefTagDetectedCallback;
        ptxIotRd_QueueCommand.ptxIotRd_QueueCommand = e_ptxIotRd_StartPooling;

        if(pdPASS != xQueueSend(xPtxIotRd_Queue, &ptxIotRd_QueueCommand, portMAX_DELAY))
        {
            /* Critical Error */
            ptxCommon_PrintF(" Critical error: Enqueue event failed.");
        }

        while (1)
        {
            /* Wait for events */
            if(pdPASS == xQueueReceive(xPtxIotRd_Queue, &ptxIotRd_QueueCommand, portMAX_DELAY))
            {
                ptxAPP_Entry(ptxIotRd_QueueCommand);
            }
        }
    }
}
```

Figure 12. ptxAPP_Entry Integration

In this example, the first step is to allocate the queue and the binary semaphore, and after successfully initializing them start loading events into the queue.

ptxIotRd_QueueCommand is a custom build object that can transmit any desired data to the thread using the queue mechanism.

Then in a closed loop, there should be a queue interrogation to check if there are any pending messages. The FreeRTOS scheduler will manage the thread execution, so if there is no message pending, the task will not be executed.

4. Improvements

The currently implemented solution does not fully reimplement the ***ptxIoTRdInt_Run_Demo_Loop*** function, so it is only possible to initialize the NFC stack and start the polling loop, but it is not possible to stop it.

The current solution was tested for I^2C and SPI communication between the host controller and a PTX1xxR device. For the UART, because of the “push” mode implementation strategy, the FreeRTOS semaphore mechanism does not work.

5. Conclusions

The integration of IoT-Reader (Non-OS) SDK v7.2.0 into a FreeRTOS-based system can be easily achieved by using binary semaphore mechanism, and even queue mechanism, if the project application layer requires complex thread interactions.

The current solution was tested for I^2C and SPI communication between the host controller and a PTX1xxR device.

It is important to remember that this is just a single thread solution, and to allow the NFC integration into complex RTOS-based projects, it is not a fully RTOS designed development.

6. Revision History

Revision	Date	Description
1.00	Jan 16, 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.