PTX1xxX SDK Porting Process

This document describes the PTX1xxX SDK porting process, as well as providing a vendor-independent reference project and tutorial for creating firmware applications for any microcontroller using the PTX1xxX Non-OS SDK.

Contents

Figures

1. Introduction

This document describes the following key porting activity processes and demonstrations:

- Provide an out-of-the-box working project independent from any MCU vendor and IDE
- Demonstrate the complete porting process of the PTX Hardware Abstraction Layer (HAL)
- Demonstrate adding generic code to run the demo application
- Demonstrate a powerful, portable and easy to reuse sample project concept
- Demonstrate low-power operation mode
- Describe proper usage of SDK API for user application
- Using tools independent of MCU vendors

1.1 Porting Process Overview

Developers need to implement certain glue-code for each new platform or project to enable the SDK to communicate with the PTX1xxX chip, manage dependencies, add application code, etc. To better understand the porting processes described in this document, it is recommended to familiarize yourself with the Renesas SDK. The accompanying project is designed to provide a deep insight of the porting steps and the following major activities are described:

- Setting up a flexible build-system based on **CMake** to achieve an IDE-independent build process
- Description of the system components used in the firmware
- Demonstration of porting the PTX HAL for I²C interface with several design-aspects being considered
- Integration of the SDK demo application into the project
- Pushing HW-dependence down to the CMSIS-layer
- Avoiding bad practices in SDK usage
- Configure and use Visual Studio Code (VSCode) for development and debugging
- Demonstrate vendor independent project (building the system, flashing/debugging process)
- Show code metrics of the generated firmware

To provide an out-of-the-box working firmware implementation that can be easily tested easily on a low-cost standard demo board for a low-end MCU, the NUCLEO-L010RB has been chosen from the STMicroelectronics.

2. Requirements

This project relies on the following hardware and software components and tools:

2.1 Software

- CMake
- Ninja
- ARM GCC (bare metal)
- CMSIS Core
- CMSIS Driver Implementation for Destination HW
- STM32 Cube Programmer
- Visual Studio Code (VSCode) (optional)

2.2 Hardware

- NUCLEO-L010RB Board (for testing the demo firmware)
- PTX100R Evaluation Board
- Segger J-link Debug Probe (optional)

  Note: If not used, then use the built-in debug probe of the NUCLEO-L010RB)

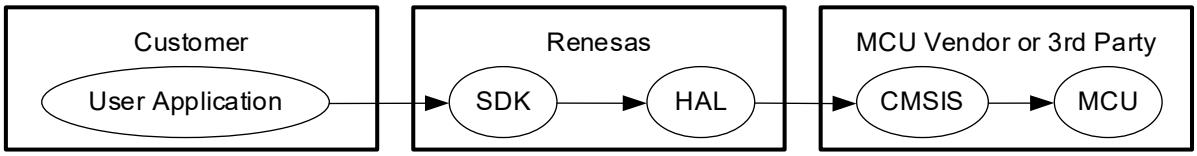
3. System Overview

This document provides an embedded hardware and software project. The wireless NFC communication used by the demonstrated system is standardized under the name ISO/IEC 14443 and consists of the following main parts:

- ISO/IEC 14443-1:2018 Part 1: Physical characteristics
- ISO/IEC 14443-2:2020 Part 2: Radio frequency power and signal interface
- ISO/IEC 14443-3:2018 Part 3: Initialization and anticollision
- ISO/IEC 14443-4:2018 Part 4: Transmission protocol

3.1 Software

The software project contains the following building blocks (grouped by the author):



The highest level component is the "User Application"; this is always implemented by the customer. This layer calls the API functions from the SDK that contain the implementation of the underlying I²C, SPI or UART protocols to communicate with the PTX1xxX chip, as well as some part of the RF protocol.

The SDK layer needs to access the underlying hardware (MCU peripherals and GPIO pins) from within the HAL module, is always hardware-dependent and is generally implemented by the customer. This project, however, uses the standard CMSIS-Core and CMSIS-Driver API (generally referred to as CMSIS) to provide the user with a ready-to-use hardware-independent HAL reference implementation.

The CMSIS libraries provide the lowest-level implementation to the MCU hardware and are available by several MCU vendors.

3.2 Hardware

The firmware development uses the NUCLEO-L010RB evaluation board (Figure 1) connected via an I²C interface to the PTX100R evaluation board (Figure 2). This provides a cost-effective and easy-to-use platform for testing and debugging the firmware application provided by this project.

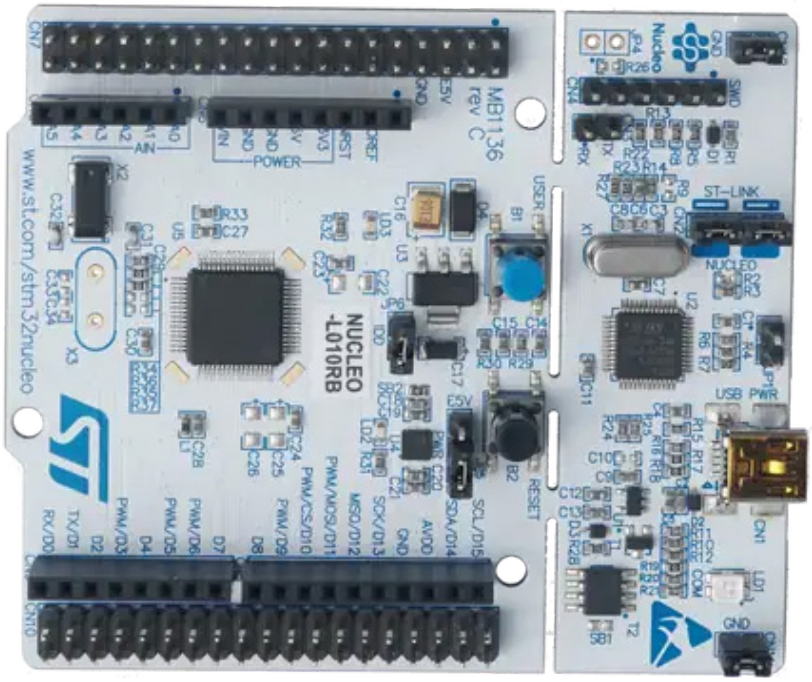


Figure 1. NUCLEO-L010RB Evaluation Board

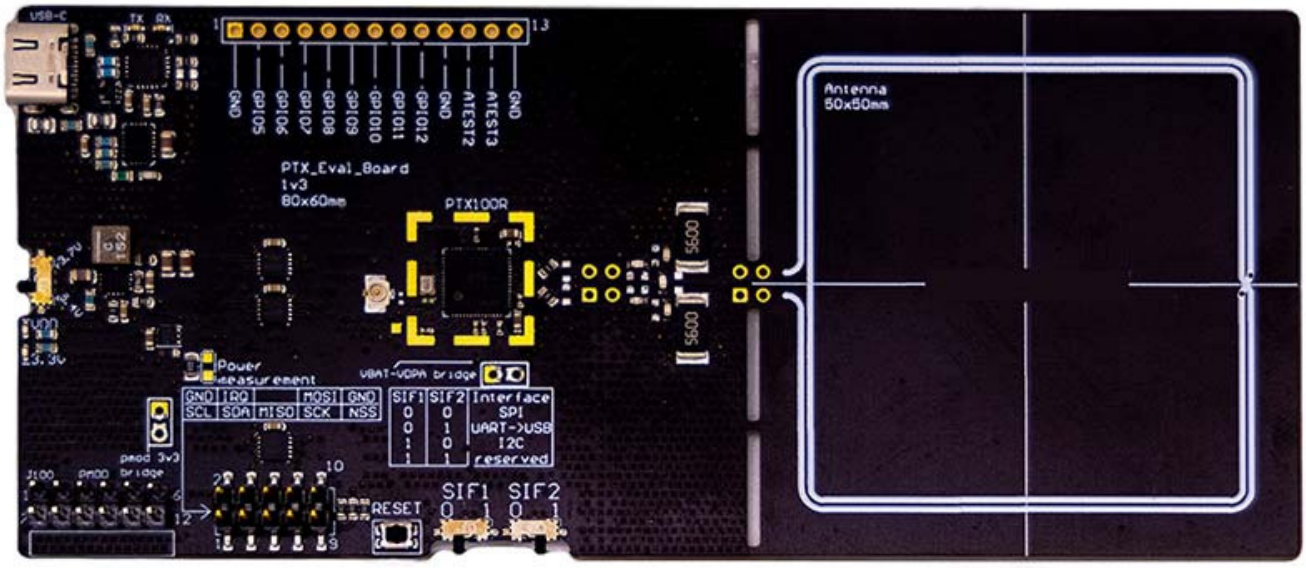


Figure 2. PTX100R Evaluation Board

3.3 Development System

Figure 3 represents the tools used for development. The Visual Studio Code is required for debugging as well as compiling the firmware.

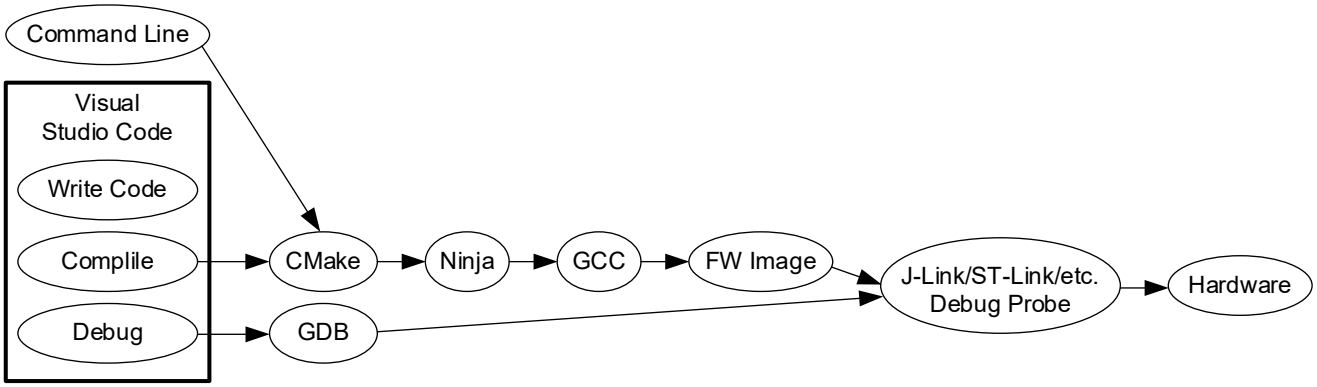


Figure 3. Development Tools System

4. Porting Considerations

When developing an embedded device, there are often several important factors, or trade-offs, involved that need to be considered. Two of these factors are:

- Time to market
- Current consumption

Most of these factors influence the required time, resources and cost of development. To optimize the development process, there are also several constraints that need to be considered when planning the porting process of the SDK. For example:

- The complete SDK code shall never be changed. It is designed to work in most environments with a proper HAL implementation (**ptxPLAT_* functions** defined in **ptxPLAT.h**).

4.1 Simple Approach

The design goal is to implement the interface functions as simple as possible and with minimal effort. This approach may also be used for an evaluation of the product so that a working system can be developed quickly. With only few lines of code and no hardware-interrupt usage, the absolute minimal implementation is very lightweight and can be done in less than an hour. However, such an implementation usually employs busy-wait and data transmission in polling mode which is typically not very efficient for the MCU.

4.2 Optimized Approach

There are several degrees of optimization in the HAL. The most relevant is to introduce the usage of HW-IRQ and the power-saving modes of the underlying MCU. The SDK functions have a blocking nature that allow the MCU to sleep instead of doing busy-waits during the waiting periods in an SDK function. This radically reduces the power consumption. For further optimization, the usage of DMA-enabled data transmission achieves even longer battery life.

4.3 Using with a Real-Time Operating System (RTOS)

Special considerations need to be taken for devices running a real-time operating system. As the SDK functions usually block the executing thread, either the application firmware needs to take this blocking action into account or a dedicated thread is needed for the NFC operations. An RTOS is usually enabled to use the power-saving features of the MCU, therefore, the waiting functions in the HAL (for example, **ptxPLAT_WaitForInterrupt**) need to be implemented in a way that does not conflict with the OS. Similarly, the functions managing the interrupts (for example, **ptxPLAT_DisableInterrupts**) need to be carefully implemented so as not to block the whole scheduler, but rather, the particular thread only. For sharing resources (for example, I²C bus instances, timers, etc.) or communicating among threads, the thread synchronization primitives (mutexes, events, etc.) provided by the RTOS are to be used.

5. Software Components

The demo NFC reader firmware consists of several components as described in the following sections. The 3rd party components used in the project are stored in their respective folder and have not been changed (with the exception of removing files that are not used) in order to optimize the final size of the project archive package.

5.1 CMSIS Core

The Common Microcontroller Software Interface Standard (CMSIS) is a vendor-independent abstraction layer for microcontrollers that are based on ARM® Cortex processors. CMSIS defines generic tool interfaces and enables consistent device support. The CMSIS software interfaces simplify software reuse, reduce the learning curve for microcontroller developers, and improve time-to-market for new devices.

CMSIS provides interfaces to the processor and peripherals, real-time operating systems, and middleware components. CMSIS includes a delivery mechanism for devices, boards, and software, and enables the combination of software components from multiple vendors. For more information, refer to the ARM Developer website.

The CMSIS Core package "**ARM.CMSIS.5.9.0.pack**" used in this application can be downloaded from the CMSIS packs page. The package is stored in the `contra/cmsis` folder and represented by the cmsis build target.

5.2 CMSIS Driver

The CMSIS Driver specification is a software API that describes peripheral driver interfaces for middleware stacks and user applications. The CMSIS Driver API is designed to be generic and independent of a specific RTOS, making it reusable across a wide range of supported microcontroller devices.

The standard peripheral driver interfaces connect microcontroller peripherals with middleware that implements communication stacks, file systems, or graphic user interfaces. Each peripheral driver interface may provide multiple instances reflecting the multiple physical interfaces of the same type in a device. For example, two physical SPI interfaces are reflected with separate Access Structs for SPI0 and SPI1. An Access Struct is the interface of a driver to the middleware component or the user application. For more information, refer to the Keil CMSIS-Driver website.

CMSIS provides the generic interface to use the peripherals in a standard way, but the implementation needs to be done separately. There are numerous MCU vendors that provide such a Software Pack to be used by a Middleware or user application. For more information and a list of official Driver-implementation by MCU families, refer to the Keil CMSIS-Driver website.

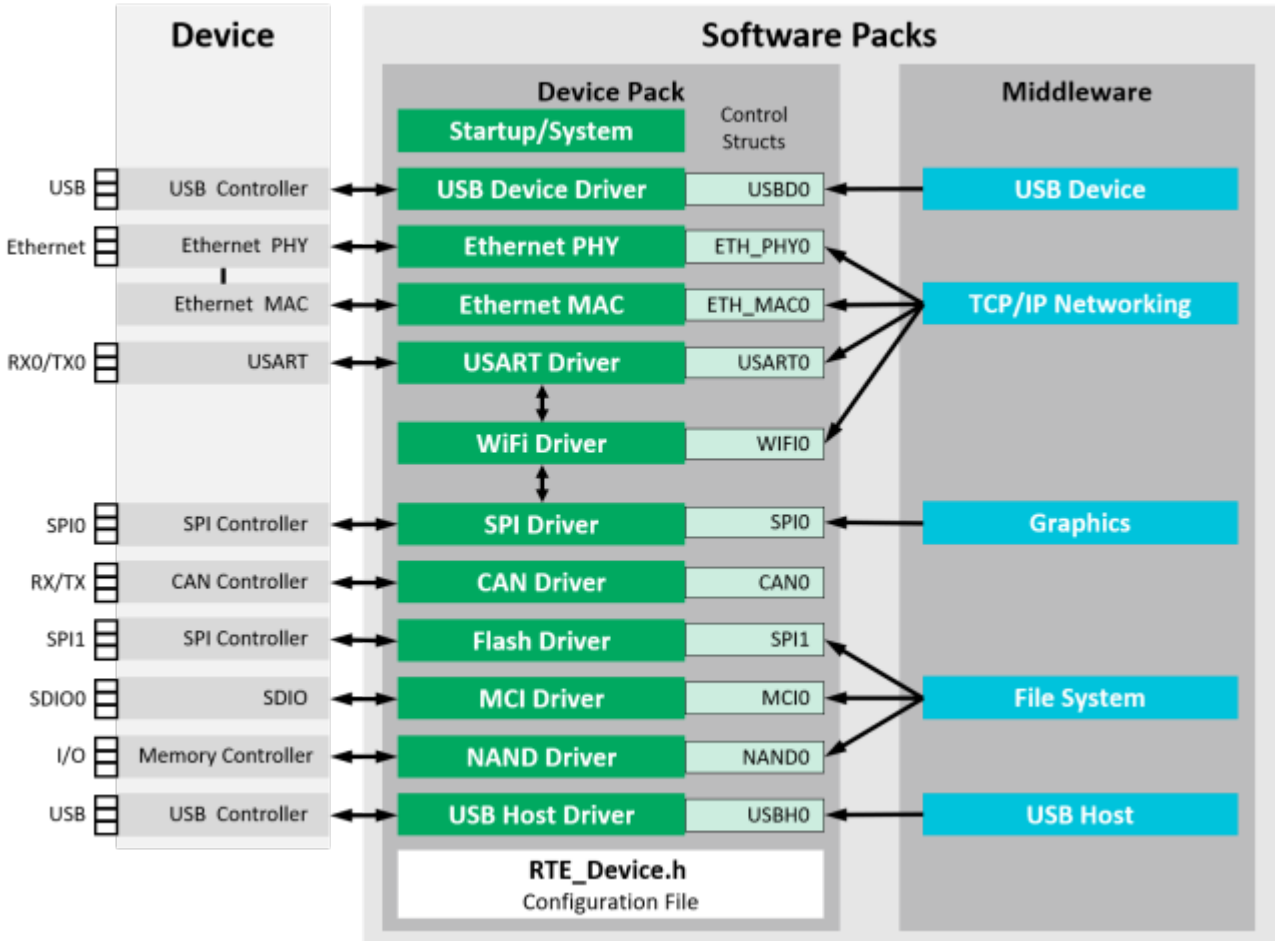


Figure 4. CMSIS Driver Package

The CMSIS-Driver package "**Keil.STM32L0xx_DFP.2.2.0.pack**" used in this application can be downloaded from Keil MDK5 Software Packs website. The driver package is stored in the `contrib/device` folder and represented by the build targets `cmsis_driver` and `driver`.

5.3 PTX Hardware Abstraction Layer (HAL)

The PTX Hardware Abstraction Layer (HAL) component provides the bridge between the PTX SDK and the underlying MCU and board. To make the SDK as platform-independent as possible, it forwards all hardware-access to the HAL using the interface header file **ptxPLAT.h** (build target **ptx_hal_interface**).

For this case, the HAL-implementation is unique for each and every device and needs to be implemented by the customer. The SDK contains a reference implementation for the Renesas RA4M2 MCU (in folder `COMPS/PLAT/RENESAS`) using the native drivers from Renesas to access the hardware.

The implementation provided by this project (`src/ptxPLAT.c` in build target `ptx_hal`) pushes the limits of the hardware-dependent domain even more down in the hierarchy due to relying exclusively on the CMSIS-Core and CMSIS-Driver API by accessing their standard interface.

The particular CMSIS Driver implementation provided by STMicroelectronics is actually an additional layer above their own driver-framework (ST-HAL). This means that each CMSIS function invoked by the PTX HAL will be routed down to the ST-HAL to access the MCU peripherals. The ST-HAL also requires some variables to be allocated in the global space. This is the reason for adding an additional build target `driver-dep`.

5.4 PTX SDK

The Renesas SDK is the source code library that presents an easy-to-use API to interact with the PTX1xxX. There are 2 variants available for download from the support portal:

1. OS-based SDK (`OS_SDK_IoTRD_V7.1.0.zip`): Designed for devices with an embedded operating system (Linux/Windows).
2. Non-OS SDK (`NON_OS_SDK_IOT_READER_V7.1.0.zip`): Designed to be used on microcontrollers and platforms with limited resources.

The demo firmware is based on the Non-OS SDK that is stored in the folder `contrib/ptxIotSdk` and is represented by the build targets `ptx_hal_interface`, `sdk_interface` and `sdk`.

Important: The content of the SDK *should not* be modified as it is a complex code base, and any modification may introduce side effects that may have an impact on the functionality. Projects with an altered SDK cannot be supported by the Renesas support team.

5.4.1 SDK Architecture

In order to maximize portability, the IoT-Reader SDK was implemented in "C" (requires C99-compatible compiler) and the system architecture follows a modular approach divided in platform dependent and platform independent modules.

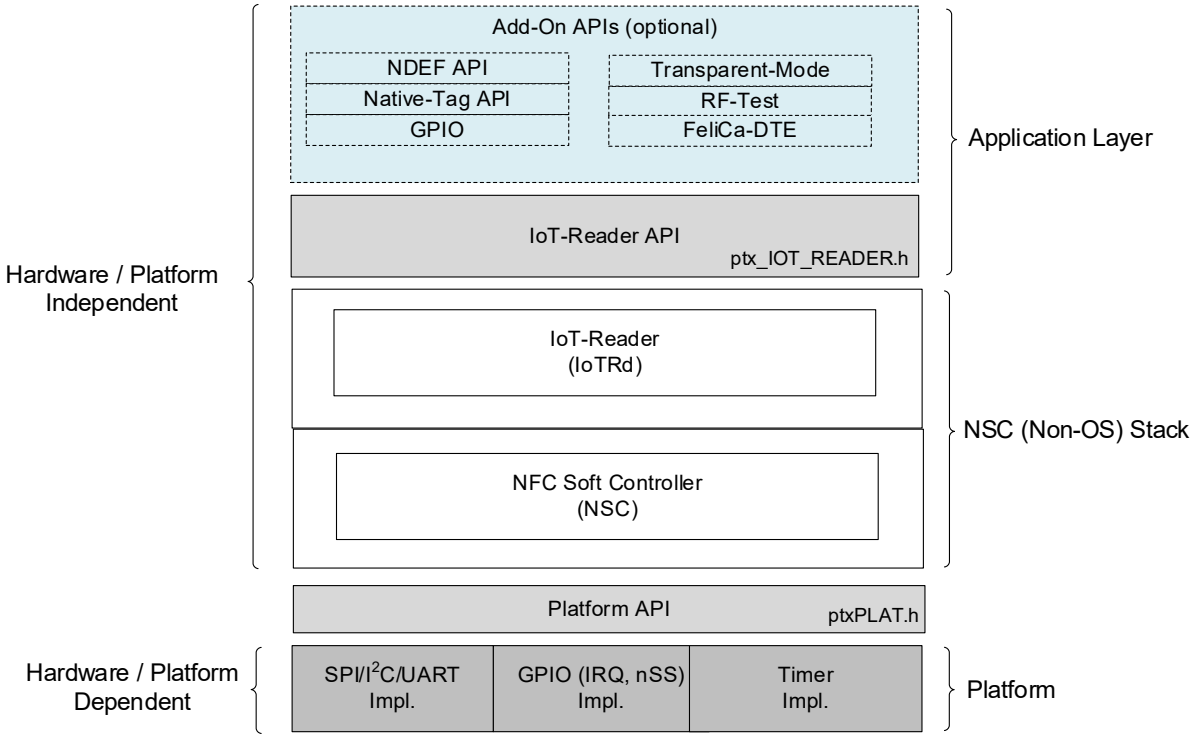


Figure 5. IoT-Reader (Non-OS) Stack Architecture

Platform dependent modules, such as SPI/I²C/UART, GPIO and Timer blocks (located at the bottom of the diagram) need to be implemented for the underlying hardware to successfully communicate with the PTX1xxX. For more information on the implementation for the demo firmware, see the HAL Implementation section.

The SDK supports all interface types (UART, I²C, SPI). The specific type for the application needs to be selected during compilation using the compile switch `-DPTX_INTF_UART`, `-DPTX_INTF_I2C` or `-DPTX_INTF_SPI` respectively. In this demo application, only the I²C interface is implemented.

The platform independent modules contain the publicly available APIs (see the top of Figure 5) that allow the application to perform the following operations on the chip:

- IoT-Reader
- Host card emulation
- NDEF
- Native-Tag
- GPIO
- Transparent Mode
- RF-Test
- FeliCa-DTE

Introduction of modules other than the IoT-Reader is not the scope of this documentation. For more information on optional modules, see the PTX1xxR NFC IoT-Reader API Non-OS Stack Integration (SDK v7.1.0) manual.

5.4.2 IoT-Reader API

The IoT-Reader API performs the following main operations:

- Initialize the IOT Reader API and the PTX1xxX
- Start a polling loop to discover NFC cards
- Select a specific card in case multiple cards were discovered (anticollision)
- Retrieve card data details and system information (for example, RSSI, critical error, etc.)
- Communicate with the card
- Stop RF communication

The following state diagram (Figure 6) shows an example of how the IoT Reader APIs can be called in order to operate successfully.

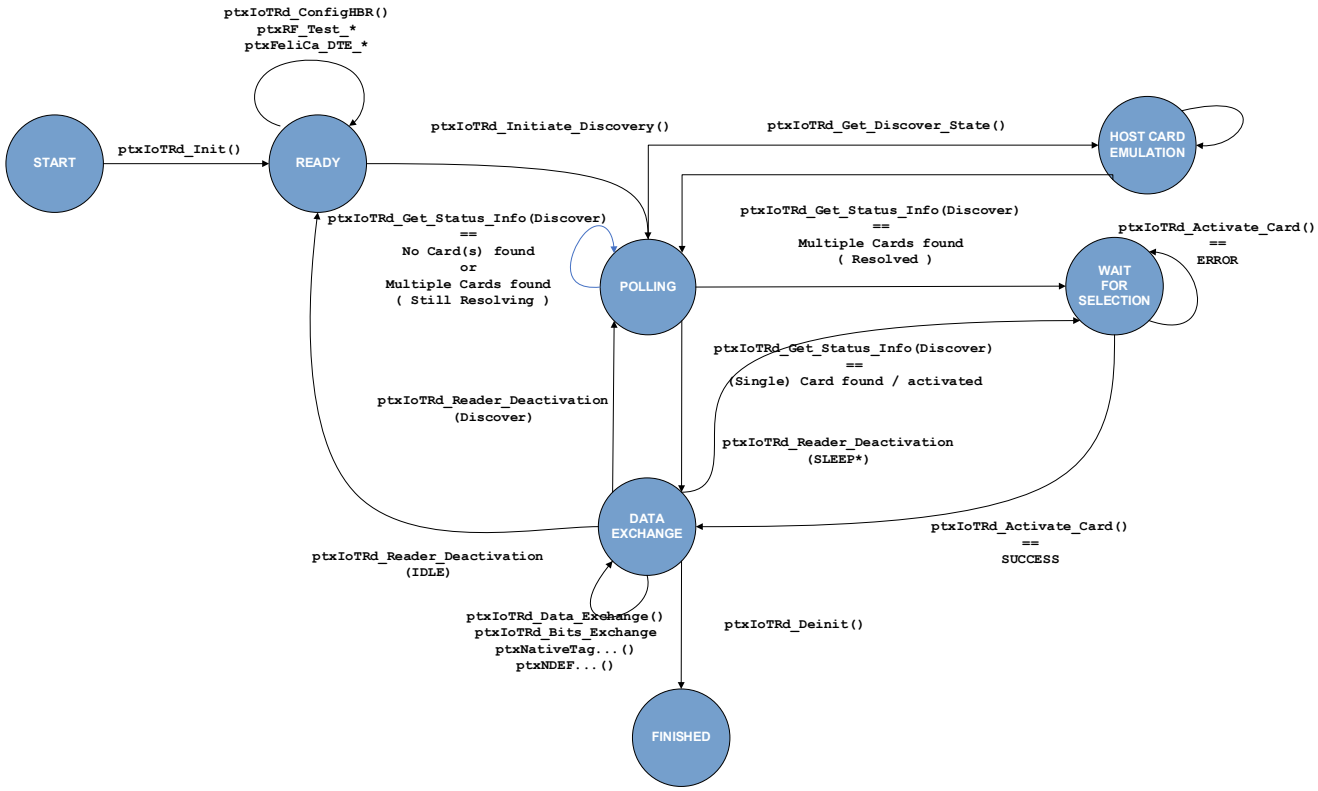


Figure 6. IoT-Reader API States

For more information regarding API functions and usage, see the PTX1xxR NFC IoT-Reader API Non-OS Stack Integration (SDK v7.1.0) manual, or look up the example demo firmware source code.

5.5 SEGGER Real-Time Transfer (RTT)

SEGGER's Real-Time Transfer (RTT) is the proven technology for system monitoring and interactive user I/O in high-performance embedded applications.

The RTT library is used to display messages from the demo application. It is also available for download at RTT Target Package. The library is stored in the folder `contrib/SEGGER_RTT` and represented by the build target `rtt`.

For more information, see the SEGGER RTT website.

5.6 APP

The demo firmware contains a small startup code that initializes the logging facility and invokes the example application that is part of the SDK.

To provide a simplified dependency-graph, the user-level example code (files `ptx_IOT_RD_Main.*` and `ptxHCE_Loopback.*`) have been moved out from the SDK to the `src/` folder.

This application component is considered as the user application that implements the desired reader-functionality. It is represented by the `ported_iot_rd_demo` build target.

The example application provides the following functionality in one file. This functionality needs to be selected via an external compile switch:

- **NFC Reader demo** activates the reader-functionality of the chip. To select it, the compiler flag `-DUSE_PTX_IOTRD_DEMO` variant is automatically selected by the build system.

- **Card emulation demo** activates the card-emulation functionality that can be used with an external NFC reader. Selected it using the compiler flag `-DUSE_PTX_HCE_LOOPBACK_DEMO`.

Similar to the SDK compilation, the required host communication interface needs to be specified during compilation of the example source files (see PTX SDK).

6. Project Overview

The demo firmware project uses **CMake** – a platform-independent build system to compile the code. CMake is an extremely flexible and capable software tool that enables easy definition of modules and build steps in textual form using multiple `CMakeLists.txt` files. The goal of the demonstration is to provide a firmware implementation that supports easy integration into any system, and usage of any MCU, with the least amount of changes to the code. To approximate this as much as possible, see the project's hierarchical design as represented in the following dependency graph (Figure 7):

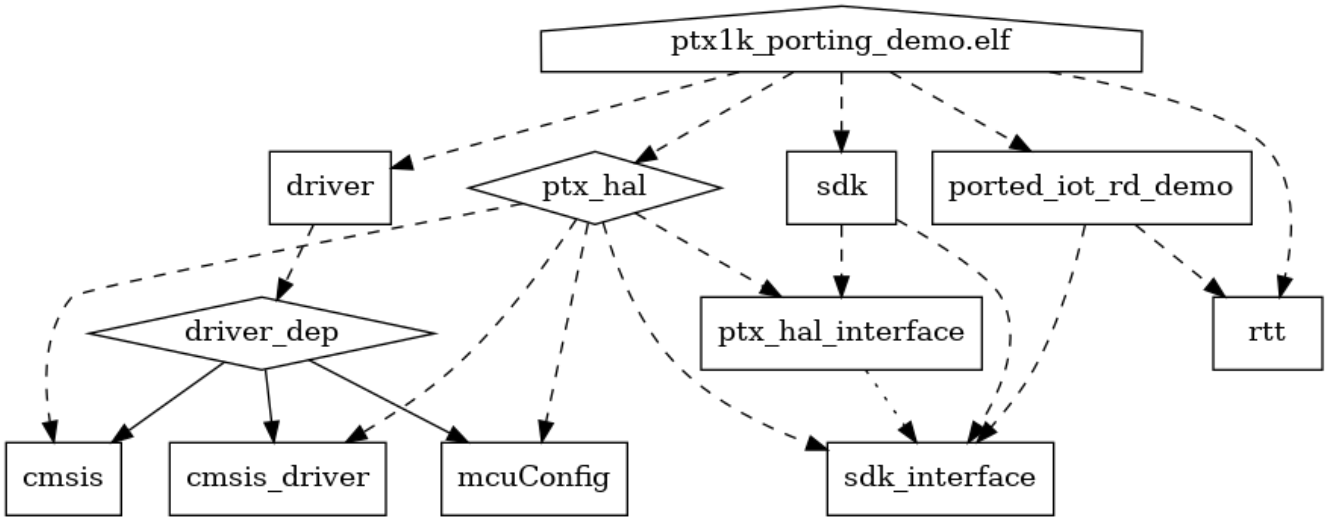


Figure 7. Firmware Modules Dependency Graph

6.1 Building Blocks

The main component is the `ported_iot_rd_demo` which is the example application `ptx_IOT_RD_Main.c` delivered in the SDK archive. For simplicity, the relevant high-level source files that belong to this example application have been moved into the `src` folder. These files also serve as a starting point for customers that are new to the SDK. As shown in Figure 7, the `ported_iot_rd_demo` is well separated from other parts of the system. It cannot directly access the hardware. This is accomplished by calling the API functions from the SDK via the `sdk_interface`.

The `sdk` represents the IoT SDK implementation which is used directly by the demo to communicate with the chip. The `ptx_hal` is the HAL implementation used by the SDK. Both modules have an interface (API) which are represented by the `.h` files. Figure 7 shows that the implementation of the functional modules are completely decoupled, and the only way they interact is via the `sdk_interface` and `ptx_hal_interface` libraries. This level of separation makes it easy to use the SDK with an arbitrary MCU by implementing a suitable `ptx_hal` module.

The goal of the demo is to minimize the effort spent during integration, therefore, the provided `ptx_hal` implementation uses the CMSIS-Core (`cmsis`) and CMSIS-Driver (`driver`) exclusively with a decoupled interface (`cmsis_driver`). These are standard driver templates and implementations that can be downloaded from the internet. The graph reveals that the `ptx_hal` has no dependency toward the hardware layer, other than these interfaces.

The lowest level in the hardware-abstraction is the STM-HAL and the CMSIS-Driver implementation (driver) and its external dependencies (`driver_dep`) that are required by the underlying STM-HAL driver framework. In practice, this enables the easy exchange of the whole `driver` and `driver_dep` implementation to another one provided by a different MCU vendor or 3rd party source. The only important condition is that the driver needs to implement the CMSIS-Driver API.

The `rtt` module represents the SEGGER_RTT library that provides the functionality to deliver log messages to the user.

6.2 Folder Structure

The project consists of the following folders:

6.2.1 src/

The `src/` folder contains the following custom implementations for modules:

- `ported_iot_rd_demo`
  - High-level main application (example application files moved out from the SDK EXAMPLE folder)
- `ptx_hal`
  - HAL implementation using the CMSIS API
- `driver_dep`
  - Global variable instance required by the CMSIS Driver implementation provided by the MCU vendor, but not belonging to the functional NFC-app
- `mcuConfig`
  - Header `.h` files required for low-level MCU configuration that are used exclusively by the underlying ST-HAL implementation

6.2.2 contrib/

This `contrib/` folder contains the following source code libraries provided by Renesas or any 3rd party:

- `sdk_interface`
  - Interface library that provides access to high-level SDK functions. Any user application needs this library linked so that they can be made available for inclusion via `CMake`
- `sdk`
  - Compilable SDK implementation located in the `ptxIotSdk` subfolder
  - The original unchanged source code
- `cmsis`
  - Official CMSIS Core library located in `cmsis` subfolder
  - Provides the standard low-level primitives to access the underlying ARM MCU core and peripherals in a generic way
  - Provides the standard API for the high-level peripheral access
- `driver`
  - CMSIS driver implementation located in the device subfolder. This particular implementation is provided by the MCU vendor STMicroelectronics and internally uses ST's own driver layer (ST-HAL)
- `cmsis_driver`
  - Interface library to the high-level CMSIS Core and CMSIS Driver interfaces (`.h` files)

- `ptx_hal_interface`
  - Interface library to the HAL API functions (`ptxPLAT.h` file) that are part of the SDK
- `rtt`
  - Segger RTT library from subfolder SEGGER_RTT

6.2.3 vscode/

This folder contains the project settings for the VSCode using the following files:

- **cmake-variants.yaml**
  - Holds the build settings for the possible target MCUs or boards
- **launch.json**
  - Contains the configuration(s) for the debug probes, so that the FW can be flashed and debugged on the MCU

6.3 Build Process

There is no requirement for a particular operating system to be used; all the tools employed in this project work on multiple platforms. The commands and screenshots in this document reflect the development in a Linux operating system.

To build the demo firmware, the following prerequisites need to be installed in the system (see Requirements):

- CMake
- Ninja
- Bare-metal ARM GCC

To flash the code to the target board, the *STM32CubeProgrammer* is also required.

Note: An existing installation of the *STM32CubeIde* already contains the command-line variant of the tool which can be used with the VSCode or directly from a terminal.

To make the compilation as efficient as possible, each batch of source files gets compiled with as much information provided as necessary. Therefore, the project contains the aforementioned libraries, modules with their own respective flags and dependencies linked. The `CMake` build system handles the dependencies among the modules correctly during compilation.

The build process consists of the following 2 steps:

1. **Configuration**: With the right amount of global information provided to `CMake`, it parses through all targets, builds up the internal dependency-graph and generates the proper files to the `make` program used to execute the compilation steps.
2. **Build**: Based on the generated files, the `make` program performs the compilation and linking for each target.

This source code and build system may also serve as a template for custom projects and can be changed or customized for the user's needs. `CMake` is very flexible and simplifies the process of adding new components to the system. However, care must be taken of the dependencies otherwise the dependency chain will be broken. If the dependency chain becomes broken, either the configuration or the build step will fail due to a missing header file or symbol.

6.4 Build from Command Line

Renesas recommends following the out-of-source building approach for compilation so that the original source folders do not get populated with generated files. Use the command line (bash or powershell) to create (and change to) a new subfolder `build/` in the project directory. The command line would appear as follows:

```
mkdir build
cd build
```

Figure 8. Command Line for Creating a New Build Subfolder

To build the demo firmware, the following parameters need to be specified for the configuration step:

- ARM bare-metal C-compiler path: `-DCMAKE_C_COMPILER=/usr/bin/arm-none-eabi-gcc`
  - The compiler is not limited to GCC. If it is not in the PATH, the absolute filename needs to be specified
- Cross-compiling related flags: `-DCMAKE_TRY_COMPILE_TARGET_TYPE=STATIC_LIBRARY`
  - During testing for functional compiler, `CMake` will not try to execute the generated file
- MCU-specific settings: `-DCMAKE_SYSTEM_PROCESSOR=cortex-m0plus`, `-DDEMO_MCU=STM32L010x8`
  - These flags inform the compiler of the target MCU type and architecture
- Firmware build type: `-DCMAKE_BUILD_TYPE=Debug`
  - Informs the build system which configuration to build. The `Debug` is optimized for debugging the target; `Release` and `MinSizeRel` apply different optimizations to reduce the FW footprint but lose the ability to debug. Should the MCU flash size be smaller than the resulting FW image, an optimized approach could be used to produce an image that still fits in the memory.
- Output folder: `-B {build_folder_name}`
  - Folder to create the artifacts
- Source folder: `-S {source_folder_name}`
  - Folder containing the top-level `CMakeLists.txt` file
- Generator: `-G Ninja`
  - Determines which maker program should be used for governing the compilation process

An example invocation of the configuration step (from the `build/` subfolder) would be:

```
cmake -DCMAKE_TRY_COMPILE_TARGET_TYPE=STATIC_LIBRARY -DCMAKE_C_COMPILER=/usr/bin/arm-
none-eabi-gcc -DCMAKE_SYSTEM_PROCESSOR=cortex-m0plus -DDEMO_MCU=STM32L010x8 -
DCMAKE_BUILD_TYPE=Debug -S.. -B. -G Ninja
```

After a successful configuration, the build step invokes the compiler to generate the firmware image:

```
cmake --build .
```

The following screenshot shows the complete process with results:

```
[redacted]:~/work/ptx1k_porting_demo$ mkdir build
[redacted]:~/work/ptx1k_porting_demo$ cd build/
[redacted]:~/work/ptx1k_porting_demo/build$ cmake -DCMAKE_TRY_COMPILE_TARGET_TYPE=STATIC_LIBRARY -DCMAKE_C_COMPILER=/usr/bin/arm-none-eabi-gcc
-DCMAKE_SYSTEM_PROCESSOR=cortex-m0plus -DDEMO_MCU=STM32L010x8 -DCMAKE_BUILD_TYPE=Debug -S.. -B. -G Ninja
-- The C compiler identification is GNU 9.2.1
-- The ASM compiler identification is GNU
-- Found assembler: /usr/bin/arm-none-eabi-gcc
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/arm-none-eabi-gcc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/[redacted]/work/ptx1k_porting_demo/build
[redacted]:~/work/ptx1k_porting_demo/build$ cmake --build .
[48/48] Linking C executable src/ptx1k_porting_demo.elf
[redacted]:~/work/ptx1k_porting_demo/build$
```

6.5 Flash and Run

Flash and run the firmware image after a successful build can easily be done using the STM32CubeProgrammer application.

After connecting the boards and wires, the *CubeProgrammer* can be used in 4 steps to flash the MCU:

1. Open the `ptx1k_porting_demo.elf`.
2. Press the *Connect* button.
3. Click the *Download* button.
4. Click *OK* to close the message about notifying the result of the flash operation

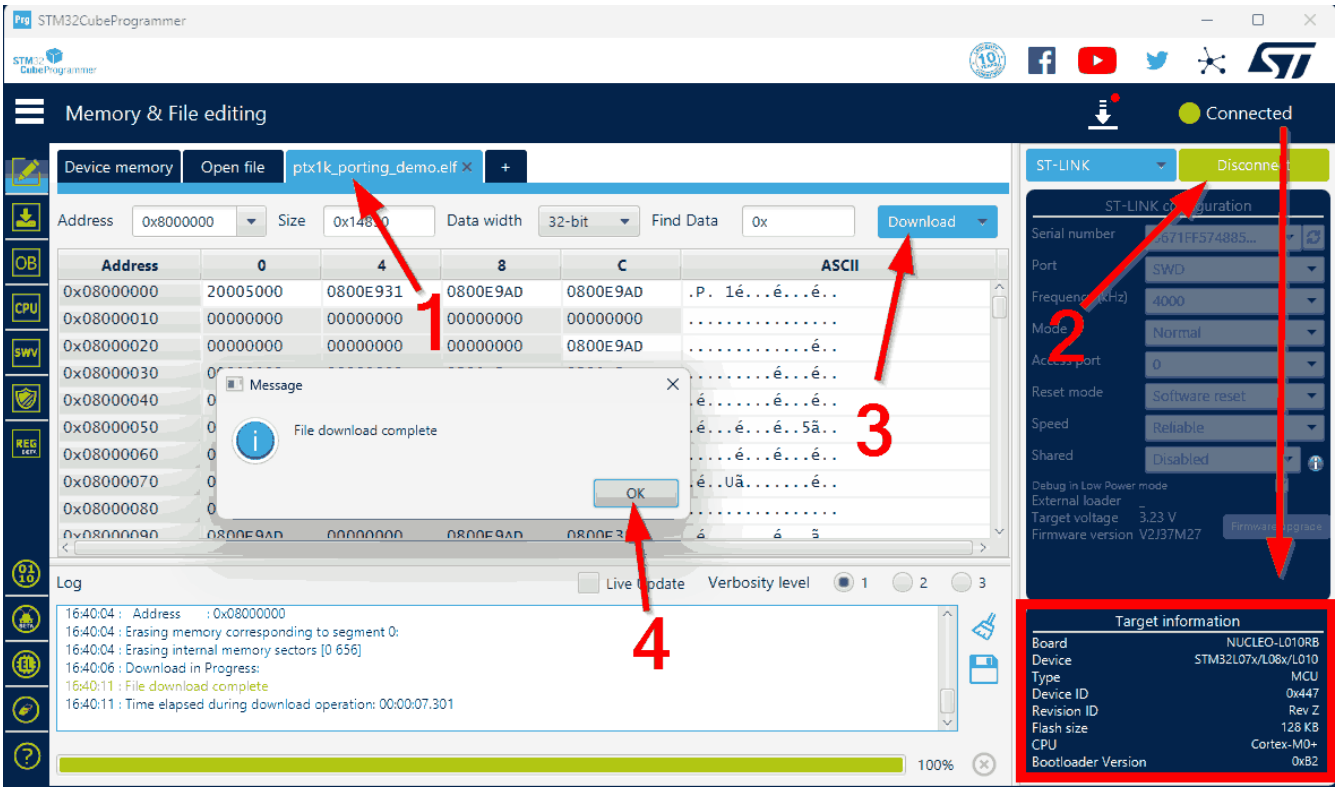


Figure 9. Memory and File Editing Window

After flashing, the MCU needs to be restarted by:

1. Changing to the **CPU** page.
2. Clicking the *Hardware Reset* button.

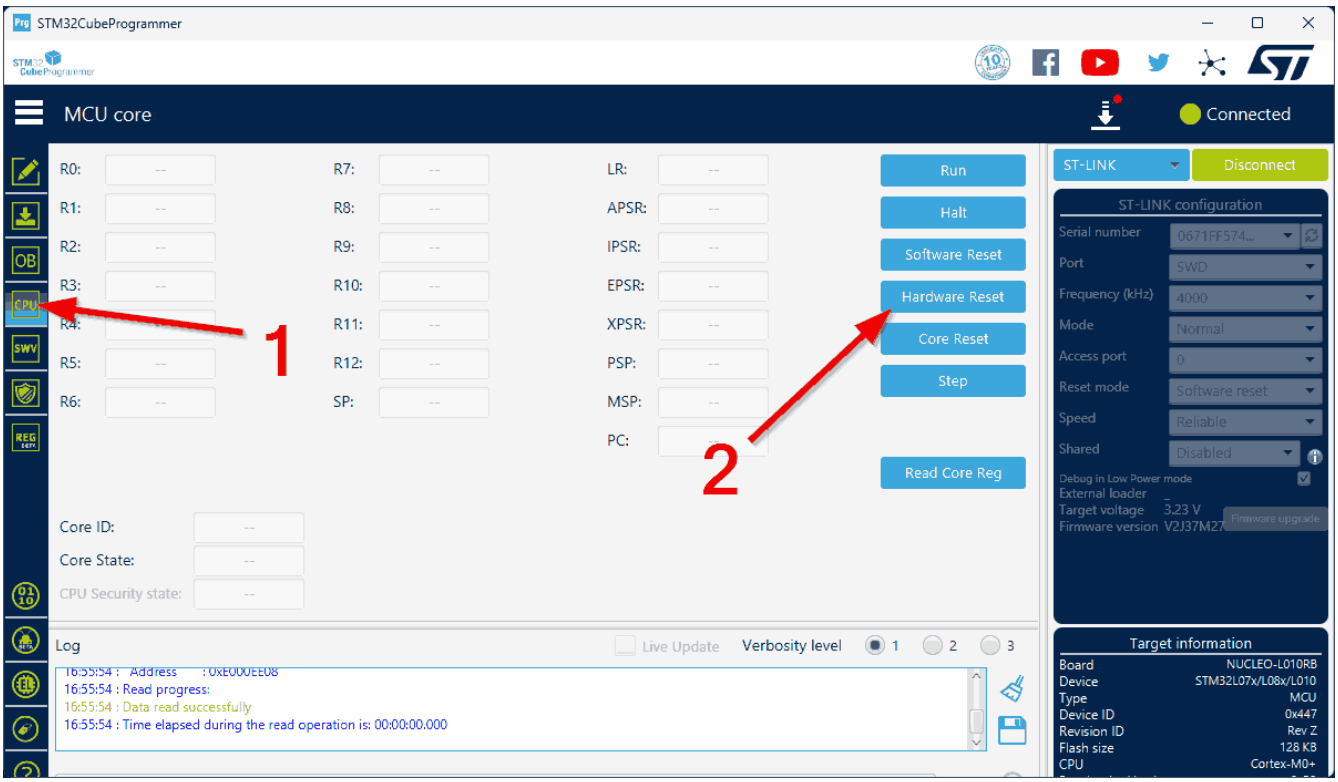


Figure 10. CPU Page – Restarting the MCU after Flashing

The demo is now ready to read cards. To learn how to read the log messages from the MCU, see the section Run in Debugger.

7. HAL Implementation

The following sub-sections describe each of the HAL functions and how to implement the `ptxPlat.c` file for a custom design in the demo application.

7.1 ptxPLAT.h Interface

The following subsections describe the implemented functions belonging to the PTX HAL.

7.1.1 ptxPLAT_AllocAndInit()

This function initializes the MCU peripherals and internal variables that are required for the NFC operation.

Peripheral	Usage
RCC	Responsible for MCU clocking.
I2C1	Used for communication with the PTX1xxX chip.
GPIOB	Used by PTX1xxX to request an IRQ by rising the pin.
LPTIM1	Used to wake-up the MCU from low-power mode in waiting operations.

7.1.1.1 RCC (Clock Module)

The MCU is clocked by the MSI clock at a frequency of 2MHz by default. However, the power regulator embedded in the MCU is initialized to the voltage scale range 2 which allows an operating frequency of 16MHz. With the clock running on a much lower frequency, further power consumption optimization can be achieved by switching the regulator to scale 3 mode, thus limiting its utilization until 4MHz.

7.1.1.2 I2C1 (I²C Module)

The I²C peripheral is clocked by PCLK1. The required GPIO pins are configured. The remaining I²C initialization is handled by the CMSIS Driver implementation.

A detailed explanation on configuring the I²C pins can be found on the Controllerstech website.

7.1.1.3 IRQ

The external GPIO interrupt is configured for the `PORTB.5` so that the PTX1xxX device can wake up the MCU once there is data to be read out.

7.1.1.4 LPTIM1 (Low Power Timer)

The SDK requires no more than 1 timer instance at a time making it is safe to hard-code it. The current implementation uses `LPTIM1` in waiting (for answer or simple delay) operations by providing a timer interrupt to wake up the MCU from the power-saving mode "Stop Mode" in this project.

While clocked by the Low Speed Internal (LSI) source, the `LPTIM1` can still continue to operate in "Stop Mode", making it the perfect peripheral for this scenario.

The frequency of the LSI clock is 37kHz. Setting the Prescaler to 32 generates an increment cycle of 0.86 milliseconds (ms). As the SDK functions have time-granularity of 1 millisecond, the provided waiting duration needs to be scaled by a factor of (100 / 86).

7.1.2 ptxPLAT_Deinit()

This function releases the resources allocated to communicate with the PTX1xxX. If not implemented, the NFC domain never shuts down.

7.1.3 ptxPLAT_TRx()

The transceive function performs the communication on the physical layer between the host MCU and the PTX1xxX. The chosen communication protocol is I²C. The implementation is divided into two paths: transmission (Tx) and reception (Rx).

Important: The communication is always initiated by the host even when the PTX1xxX requests to transmit data. In such situations, the PTX1xxX (I²C Slave) raises the IRQ line and notifies the MCU (I²C Master) that it is ready to communicate. The MCU then initiates the data exchange.

During read operation, the SDK passes the `PTX_PLAT_TRX_FLAGS_I2C_RESTART_CONDITION` flag to the function. In this case, a repeated start condition must be performed after the transmission of the last TX block before the data bytes can be read out from the PTX1xxX.

7.1.4 ptxPLAT_WaitForInterrupt()

To minimize the power consumption, the MCU enters "Stop Mode" during waiting for response from the PTX1xxX. The MCU should wake up and continue code execution when the PTX1xxX asserts the IRQ pin.

Note: To prevent stalling the MCU in an endless waiting state, the SDK uses a timer to set a maximum waiting period. If the time elapses, a corresponding IRQ will wake up the MCU. This generates an error where the actual SDK command terminates with a timeout error code. Only these two events result in returning from the function.

7.1.5 ptxPLAT_StartWaitForRx()

The purpose of the function is to register a callback function that will be invoked once the PTX1xxX rises the IRQ line. If the pin is already asserted, the registered function gets called directly.

7.1.6 ptxPLAT_StopWaitForRx()

The MCU stops waiting for the PTX1xxX to assert the IRQ line.

7.1.7 ptxPLAT_Sleep()

The MCU blocks the code execution for the designated period of time by entering "Stop Mode".

Note: This function is not permitted to return before the elapsed waiting time.

7.1.8 ptxPLAT_GetInitializedTimer()

The function prepares a timer for the SDK and also gets enabled for later usage in ptxPLAT_Sleep(), ptxPLAT_TimerStart(), ptxPLAT_TimersElapsed() and ptxPLAT_TimerDeinit().

7.1.9 ptxPLAT_TimerStart()

This function starts the timer obtained by the ptxPLAT_GetInitializedTimer().

The SDK may start the timer in blocking mode (where the function does not return until the timer has elapsed) or in asynchronous mode. In each case, the callback function, provided as an argument, will be invoked when the time elapses.

7.1.10 ptxPLAT_TimersElapsed()

Evaluates if the timer has elapsed.

7.1.11 ptxPLAT_TimerDeinit()

Disables the timer.

7.1.12 ptxPLAT_IsRxPending()

Evaluates if there is a pending incoming communication request from the PTX1xxX.

7.1.13 ptxPLAT_TriggerRx()

Invokes the previously registered state handler callback function if there is a pending IRQ.

Note: This function gets executed each time the `ptxIoTRd_Get_Status_Info()` is called from the main application loop. That is the "polling function" of the SDK internal state machine, therefore, it must be called regularly. Otherwise, the SDK will not be able to process the incoming events.

7.2 Auxiliary Functions

The following functions under section 7.2 are not part of the SDK interface. These belong to the HAL implementation as helpers called by multiple HAL-functions.

7.2.1 isIrqPending()

Checks if the IRQ pin is asserted.

7.2.2 goToSleep()

Executes the necessary steps to set the MCU in "Stop Mode".

7.2.3 EXTI4_15_IRQHandler()

Interrupt service for handling the IRQ request from the PTX1xxX. The interrupt event wakes up the MCU and continues the code execution at the point it entered power-saving mode.

7.2.4 LPTIM1_IRQHandler()

Interrupt service for handling the time elapse events. The interrupt event wakes up the MCU and continues the code execution at the point it entered power-saving mode.

7.3 Required Changes in 3rd Party Code

Ready-to-use 3rd party libraries may contain restrictions or limitations that require the use of a vendor-specific tool (for example, IDE) which can contradict the scope of this document. This subsection explains the required 3rd party code modifications in this project to reconcile unfavorable design decisions.

7.3.1 CMSIS Driver Implementation

STMicroelectronics® provides a CMSIS-compatible, ready-to-use I²C implementation in the driver package. However, it is designed to be used together with the STM32CubeIde or STM32CubeMX tools which generate source files with content computed from the project settings. In this instance, the value of the timing register gets calculated based on the clock settings. Using the driver without this calculation is possible but can take a considerable amount of time, therefore rendering it unusable.

The driver strategy to determine the value of the I²C timing register (`I2C_TIMINGR`) is to sequentially iterate and evaluate all possible values of the module's timing prescaler (`PRESC`), data setup time (`SCLDEL`) and data hold time (`SDADEL`) register fields. The iteration process is performed inside the `I2C_GetTimingValue` function. *Note*: The computation of this value can take several minutes.

To prevent this significant delay caused by the long-running function, the source code has been altered to skip the `I2C_GetTimingValue()` and directly write the proper value in the `TIMINGR` register.

Below is a snippet of the code modification on the `I2C_Control()` function in file `I2C_STM32L0xx.c`:

```
/* Get TIMING register values */
// val = I2C_GetTimingValue(&clk_setup, clk_spec);
/* Set register values */
I2C->reg->CR1 &= ~I2C_CR1_PE;
I2C->reg->TIMINGR = 0x708;
I2C->reg->CR1 |= I2C_CR1_PE;
```

Figure 11. Code Modification on the I²C Control Function

7.4 Additional Dependencies

The CMSIS driver implementation from STMicroelectronics uses their custom HAL drivers to access the peripherals. These drivers are designed in a way that some files are generated by their IDE. As this project does not use any vendor-specific tools, the recommended approach for these files is to look up the template of the particular file or take an example project for an evaluation board of the vendor, then copy and adapt the file contents to meet the required specifications.

7.4.1 stm32l0xx_hal_conf.h

Configuration file with macros to enable certain selected peripherals, including their respective interface files. This file also contains MCU clock configuration.

This file is available as a template in `contrib/device/Drivers/STM32L0xx_HAL_Driver/Inc/stm32l0xx_hal_conf_template.h` and could be modified according to the requirements.

7.4.2 MX_Device.h

The CMSIS driver implementation requires the presence of this file that contains a series of MX_ macros defining properties of the configured peripherals. An altered copy of this file may be taken from an example project generated by the vendor IDE (or internet).

7.4.3 st_hal_dependencies.c

For the I²C driver, the ST-HAL requires a global context variable and the interrupt service routine (`I2C1_IRQHandler()`) to be implemented and calls the appropriate event handlers from the ST-HAL.

8. NFC Reader Demo Application

The firmware entry point is the `main()` located in the `main.c` file. This initializes the RTT module to enable the output messages being transferred to host PC and get printed in the RTT-Viewer (terminal) window.

The next call will invoke the main function of the example application (`ptxIOT_READER_App()`) which is not expected to return.

The function `ptxDBGPORT_Write()` defined in the `main.c` file is responsible for forwarding the messages to be printed to the RTT module. This function gets invoked by the `ptxCommon_PrintF()` from various parts of the SDK and the demo application when there is a message or debug log to be printed to the screen.

The build system will automatically compile the **NFC Reader demo** variant of the example application with the I²C communication interface active. The demo uses API functions from the following header files:

```
ptx_IOT_READER.h
ptxCOMMON.h
ptxIoTRd_COMMON.h
ptxNDEF.h
ptxNDEF_T2TOP.h
ptxNDEF_T3TOP.h
ptxNDEF_T4TOP.h
ptxNDEF_T5TOP.h
ptxNativeTag_T5T.h
ptxT4T.h
```

Figure 12. NFC Reader Demo Header Files

Entering the function `ptxIOT_READER_App()` in `_ptx_IOT_RD_Main.c`, the variable `start_temperature_sensor_calibration` will be set to 1, telling the SDK to execute the temperature calibration on the PTX1xxX chip.

Important: By default, the temperature calibration is always enabled although it shall be done exactly once and not more for each chip with the resulting shutdown temperature value saved into NVM. This value shall be used for any subsequent initialization of the SDK. Executing the temperature calibration with invalid or improper parameters (for example, Ambient temperature = 25°C by default) on a chip that is much warmer or colder will either result in an invalid setting or an error code returned by the function.

After preparing the initialization structure, the SDK can be initialized by the command:

```
st = ptxIoTRd_Init(&iotRd, &initParams);
```

Figure 13. SDK Initialization Command

The SDK API functions usually return a status code `st`, that is a `uint16_t` with the higher byte representing the SDK-component (`enum ptxStatus_Comps`) and the lower byte showing the actual return code (`enum ptxStatus_Values`). A return code `0x0000` means the function execution was successful (`ptxStatus_Success`). Any other value represents an error.

After a successful initialization, the NFC polling can be started by the command:

```
st = ptxIoTRd_Initiate_Discovery (&iotRd, &rf_disc_config);
```

Figure 14. NFC Polling Start Command

Finally, the main application loop will be invoked by the command:

```
ptxIoTRdInt_Run_Demo_Loop(&iotRd, &t4tComp);
```

Figure 15. Main Application Loop Command

The heart of the user application is the loop `while (0 == exit_loop){...}` with calling the function `ptxIoTRd_Get_Status_Info()` regularly to retrieve the latest information about the system state and communication/activation status.

```
st = ptxIoTRd_Get_Status_Info (iotRd, StatusType_System, &system_state);
```

Figure 16. Get Status Info Command

Using the parameter `StatusType_System`, it returns the system state that is required for error handling. Should the PTX1xxX get too hot (overtemperature error) or too much current drawn by the antenna (overcurrent error), the chip automatically shuts down after notifying the host and it will not respond to any command. When this happens, the chip needs to be initialized again. When an overtemperature error occurs, it is especially important to use a previously calibrated shutdown temperature value and not doing the calibration again. Otherwise, it will fail or be invalid.

There is a state machine to handle the recognition, activation and subsequent communication cycles with the tag. It starts with Wait for Activation and the value gets updated in each of the states when needed. These states are explained section 8.1.

If an error occurs, the reader gets deactivated, the components will be de-initialized and the application function returns:

```
ptxT4T_DeInit(&t4tComp);
[...]
(void) ptxIoTRd_Reader_Deactivation (&iotRd, PTX_IOTRD_RF_DEACTIVATION_TYPE_IDLE);
[...]
(void)ptxIoTRd_Deinit(&iotRd);
```

Figure 17. Application Function Returns Error Messages

Should a severe error occur, the demo state moves to System Error and the application discontinues the execution (see section 8.5).

8.1 Wait for Activation

This state is handled in the function `ptxIoTRdInt_DemoState_WaitForActivation()` in file `ptxIoTRd_COMMON.c`.

The SDK keeps track of the card activities. To retrieve the actual status, the following function gets called:

```
ptxIoTRd_Get_Status_Info (iotRd, StatusType_Discover, &discover_status);
```

Figure 18. Discover Status

The returned `discover_status` value reveals details about the card activation state:

- **No card**: If there is no card/tag activated and none is in the RF-field, the application does not do anything
- **Card active**: Should a card enter the RF field, it gets activated automatically by the PTX1xxX. The application obtains the card details and moves its current state to Data Exchange.
- **Discover running**: This state is shown during the anti-collision process if more than one card enters the field at once
- **Discover done**: After the anti-collision process finishes, the application can retrieve the information of participating cards so that it can select one for activation in the Select Card state.

8.2 Data Exchange

This state is handled in the function `ptxIoTRdInt_DemoState_DataExchange()` in file `ptxIoTRd_COMMON.c`.

After a successful card-activation, it is possible to read/write data on the card. This state demonstrates the communication with the card using the high-level NDEF API (if the `USE_NDEF` compile macro is defined) and low-level tag specific commands.

The example demonstrates the usage of data exchange functions for the tags. These are discussed in greater detail later in this section.

After the data exchange, the example displays the gained information in the RTT terminal and triggers the deactivation by moving to the Deactivate Reader state.

8.2.1 Using NDEF API

The NDEF API provides functions to communicate with the cards in a high level. The tags can be accessed either by generic NDEF functions that work independent of the type of the participating tag, or via the API unique for each tag type.

Generally, the NDEF component needs to be opened first. Subsequentially, all the other API functions may be called.

8.2.1.1 Generic NDEF API

The application uses the following commands (from `ptxNDEF.h`) to read data:

```
ptxNDEF_Open (ndefComp, ndefInitParams);
ptxNDEF_CheckMessage (ndefComp);
ptxNDEF_ReadMessage (ndefComp, &ndef_msg_buffer[0], &ndef_msg_buffer_len);
```

Figure 19. Generic NDEF API

This way, the extended functionality of a tag is not accessible, but the standard tag functions can be used without knowing or taking care of the underlying tag type.

8.2.1.2 Type-Specific NDEF API

To limit the compatibility to a certain tag type or use special functions provided by a tag type, the dedicated tag API (in files `ptxNDEF_T2TOP.h, ptxNDEF_T3TOP`, etc.) may be used the same way as the Generic NDEF API.

The application reads each card with the API conforms to the reported type during activation, for example:

```
ptxNDEF_T4TOpOpen (t4tOpComp, t4tOpInitParams);
ptxNDEF_T4TOpCheckMessage (t4tOpComp);
ptxNDEF_T4TOpReadMessage (t4tOpComp, &ndef_msg_buffer[0], &ndef_msg_buffer_len);
```

Figure 20. Type-Specific NDEF API

8.2.2 Using Low-Level Commands

Reading a card using low-level commands is generally used to deal with cards that do not support the NDEF or cards where another protocol needs to be used (for example, Chinese Id).

By removing the `-DUSE_NDEF` compiler flag from the project, the application will communicate with the card using this method instead of the NDEF API. The type-specific functionality is covered by the NativeTag API with the following functions used (for T5T card):

```
ptxNativeTag_T5TOpen(t5tComp, t5tInitParams);
ptxNativeTag_T5TSetUID(t5tComp, &cardRegistry->ActiveCard-
>TechParams.CardVParams.UID[0], 8u);
ptxNativeTag_T5TReadSingleBlock (t5tComp, 0, 0, &rx_data[0], (size_t*)&rx_data_length,
app_timeout);
```

Figure 21. Native Tag API (for T5T Card)

The following example also shows the use of sending raw commands to a card independent of type:

```
ptxIoTRd_Data_Exchange(iotRd, &tx_data[0], tx_data_length, &rx_data[0], &rx_data_length,
app_timeout);
```

Figure 22. Sending Raw Commands to a Card Independent of Type

The function is used for all types other than T5T in the demo.

8.3 Select Card

This state is handled in the function `ptxIoTRdInt_DemoState_SelectCard()` in file `ptxIoTRd_COMMON.c`.

After enumeration of the cards in the field has been finished, the example selects the first card from the list and activates it using one of the supported protocols (`Prot_NFCDEP`, `Prot_ISODEP`, `Prot_T2T`, `Prot_T3T`, `Prot_T5T` or `Prot_Undefined`):

```
ptxIoTRd_Activate_Card(iotRd, &cardRegistry->Cards[0], Prot_NFCDEP);
```

Figure 23. Activating the Selected Card

If the activation is successful, a summary of the card's details is printed out to the RTT terminal:

```
ptxIoTRdInt_Get_Card_Details(cardRegistry, cardRegistry->ActiveCard, 1);
```

Figure 24. Activated Card Summary

Finally, the demo continues with Data Exchange unless an error occurs during activation. If an error occurs, then the Deactivate Reader step is invoked.

8.4 Deactivate Reader

This state is handled in the function `ptxIoTRdInt_DemoState_DeactivateReader()` in file `ptxIoTRd_COMMON.c`.

This is the last state of the demo state machine. After reading the card's data or occurs, the example requests a deactivation of the card with restarted polling activity afterwards:

```
ptxIoTRd_Reader_Deactivation(iotRd, PTX_IOTRD_RF_DEACTIVATION_TYPE_DISCOVER);
```

Figure 25. Card Reader Deactivation

Finally, the demo state machine gets reset to Wait for Activation.

8.5 System Error

This state is handled in the function `ptxIoTRdInt_DemoState_SystemError()` in file `ptxIoTRd_COMMON.c`.

Should a severe error happen (for example, overcurrent or overtemperature error), the PTX1xxX chip enters shutdown mode. In this case, the chip needs to be re-initialized, but to close the SDK handles properly, the SDK provides a function to reset the chip and the internal logic:

```
ptxIoTRd_SWReset(iotRd);
```

Figure 26. Resetting the Chip and Internal Logic

Note: The application will enter an endless loop after calling the function.

9. Using an Integrated Development Environment (IDE)

Each MCU vendor encourages their customers to use their respective Integrated Development Environment (IDE) to develop the application. In most cases (also for ST), the suggested IDE is based on the Eclipse IDE. The vendor integrates its own settings and tools to the respective IDE that helps users to create a project for a specific MCU, development board, configure the peripherals and pins graphically, perform compiling and debugging, etc. Using such an IDE provides the user with a firmware development method compatible for the MCUs of the same vendor, however, making the firmware portable and compiling it to an MCU from another vendor is not possible.

To be as vendor-independent as possible, this project uses the `CMake` to build the firmware and Visual Studio Code (VSCode) as IDE. VSCode is a generic IDE and does not provide the user-friendly graphical peripheral, pin and system configuration. It does, however, provide generated configuration files taken from the native IDE as well as a vendor-independent development environment platform with a host of valuable features.

9.1 Required Extensions

VSCode is not an IDE, but rather a versatile text editor with a variety of extensions. It can be customized according to a project's requirements by installing extensions that provide certain functions to make a feature-rich IDE. The following subsections describe the extensions required for this project.

9.1.1 C/C++

The "C" source code extension `ms-vscode.cpptools` is essential as it provides language-service, syntax-highlight, auto-complete and code formatting functionality in the editor window.

9.1.2 CMake Tools

The extension `ms-vscode.cmake-tools` integrates **CMake** functionality into VSCode so that triggering the configuration or build process is accomplished with just one click with the mouse.

VSCode uses "kits" to identify the compiler and compile environments used with the **CMake**, and several kits can be defined in the system. The extension looks for the compilers in the system and creates kits automatically. If there is no compiler found or the right one is not found, the kit needs to be added to the system manually. For more information, see CMake Kits documentation.

9.1.2.1 Build Variants

The extension also offers easy switching among FW variants, for example, compiling the code with different compiler options to produce a FW image with different functionality or for a different MCU. For more information, see the CMake Variants documentation.

The project offers a combination of the following target options as variants:

- Debug Type
  - Can be *Debug* or *Release*. For more information, see the section Build from Command Line.
- Target MCU/Board
  - Options for target platform and MCU. This configuration contains an extended array of compiler options.

The build variants are configured in file `cmake-variants.yaml`.

9.1.3 Cortex Debug

For debugging the firmware, the extension `marus25.cortex-debug` provides a debugger with many suitable features for ARM cortex MCUs and supports most debug probes (ST-Link, J-Link, etc.). For an overview and more information, see the Cortex Debug documentation.

The debugger configuration in VSCode is achieved in the `launch.json` file. This project provides standard debug configuration for the on-board ST-Link debugger, and an additional one for the J-Link.

To debug the target *NUCLEO-L010RB* board, the extension needs to be able to locate the `gdb` client, the stlink-server and ST Cube Programmer for using ST-Link, or the J-Link GDB server for J-Link. If the required applications are not in the PATH, they can be specified directly in the `.vscode/settings.json` file:

```
{
    "cortex-debug.stm32cubeprogrammer.windows":
"C:/Tools/STM32CubeIDE/plugins/com.st.stm32cube.ide.mcu.externaltools.cubeprogrammer.win
32_1.6.0.202101291314/tools/bin",
    "cortex-debug.stlinkPath.windows":
"C:/Tools/STM32CubeIDE/plugins/com.st.stm32cube.ide.mcu.externaltools.stlink-gdb-
server.win32_1.6.0.202101291314/tools/bin/ST-LINK_gdbserver.exe",
    "cortex-debug.JLinkGDBServerPath.windows": "C:/Program
Files/SEGGER/JLink/JLinkGDBServerCL.exe",
    "cortex-debug.gdbPath.windows": "C:/Tools/gcc-arm/gcc-arm-none-eabi-10.3-
2021.10/bin/arm-none-eabi-gdb.exe"
}
```

Figure 27. Cortex Debugging Configuration for ST-Link and J-Link

Using an external J-Link activates a new console window which shows the RTT output without any external or 3rd party application.

If using the ST-Link, an external RTT viewer is needed to display the messages from the application. The project ST-RTT provides such a viewer. After downloading the binaries, the command `strtt -tcp` connects to the debugger (ST-LINK debug server application–already running) and displays the printed messages during debugging as well.

Another version of `strtt` using WebUSB technology that runs directly in a web browser without the need to install any application (other than the ST-LINK drivers) may also be used, but this needs to open the st-link device exclusively. Therefore, no debugging can be performed simultaneously.

Important: The debugger cannot communicate with the MCU when it is in a particular power-saving mode. Therefore, reading out the RTT messages is also not possible.

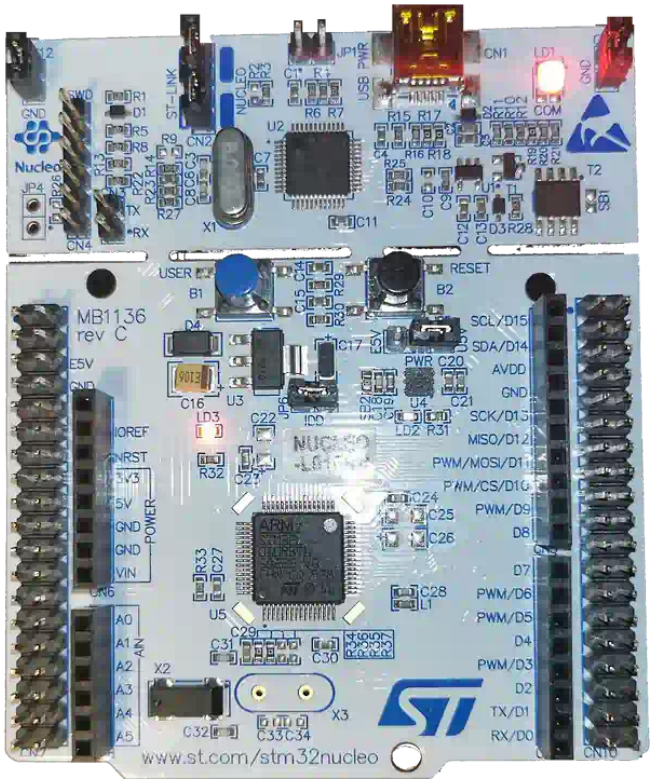
10. Running the Demo

This section describes the steps, processes and hardware required to run the demo application.

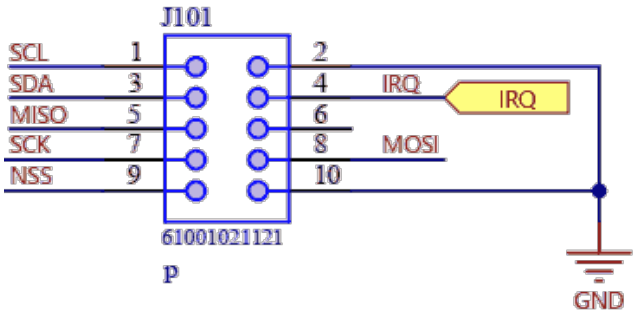
10.1 Hardware Connection

Prior to running the demo, the hardware/boards needs to be prepared and setup. The two participating evaluation boards have the following pinouts:

Nucleo-STM32L010RB Pinout



PTX100R Evaluation Board Connector Pinout



Provide the following connections to enable the I²C interface:

STM32L010RB Pin	Signal Name	PTX100R Evaluation Board Pin
PB8 (SCL/D15)	I²C clock line (SCL)	1
PB9 (SDA/D14)	I²C data line (SDA)	3
PB5 (D4)	IRQ line	4
GND	Ground (GND)	2

The pins SCL and SDA require pull-up resistors. Most MCUs have internal resistors that can be used so no addition external ones are required. Unfortunately, some MCUs implement resistors that are too weak. This can make the bus sensitive to noise from the board or environment, especially during signal change. This situation is also true for the Nucleo-STM32L010RB board, therefore, separate 4.7kΩ resistors need to be added externally to overcome the board's bus sensitivity to noise limitation.

The PTX100R evaluation board has two switches (SIF1 and SIF2) for selecting the communication interface. To configure it to I²C, the switches need to be set to positions 1 and 0 respectively.

As a final step, the boards need to be powered-up by their respective USB-cable. The PTX100R evaluation board requires a USB-C type cable and, for maximal RF output power, the host needs to be able to supply it with a current up to 900mA (this corresponds with the USB 3.0 standard). USB 2.0 hosts may be limited in output current, causing a weaker RF field.

10.2 Build the Firmware

To build the firmware, select the desired compiler and target variant, then click the *Build* button.

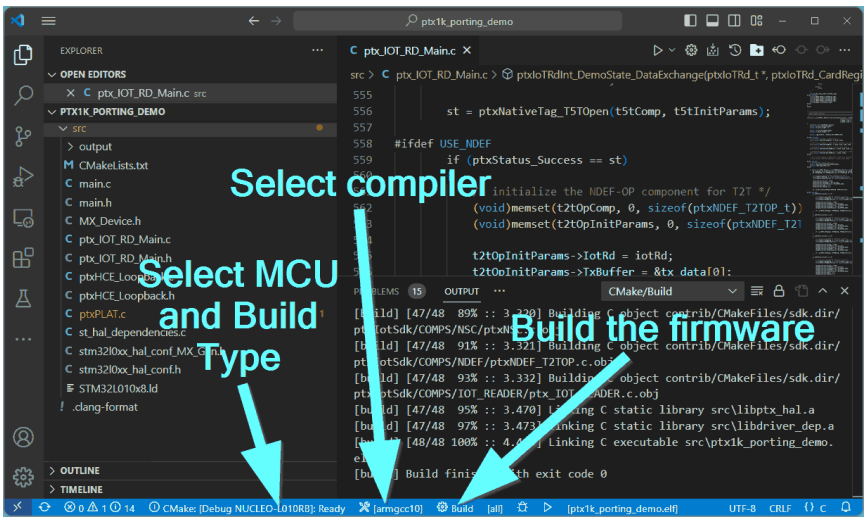


Figure 28. Building the Firmware Window

The build process should finish with the message `Build finished with exit code 0` in the output window, indicating the successful generation of the FW image `ptx1k_porting_demo.elf`.

10.3 Run in Debugger

When the firmware file is present, select the desired debugger and start the debug process. This triggers the application to be flashed into the MCU and started.

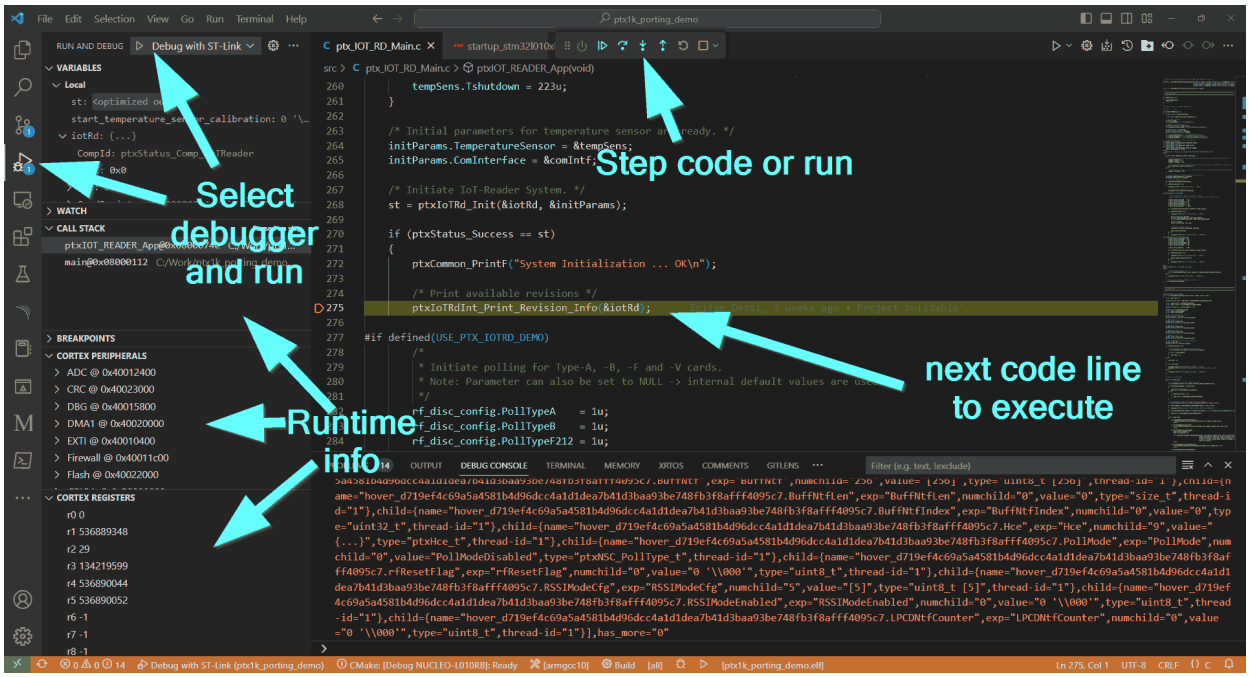


Figure 29. Selecting and Running the Debugger

The debugging will stop at the beginning of the demo's main function. Pressing the *Run* button will continue the execution and allows the FW to start its main activity: initialize the PTX1xxX chip and start polling for a card.

When approaching the antenna by the Field Probe card, the blinking LEDs will show the presence of the RF field with the polling active.

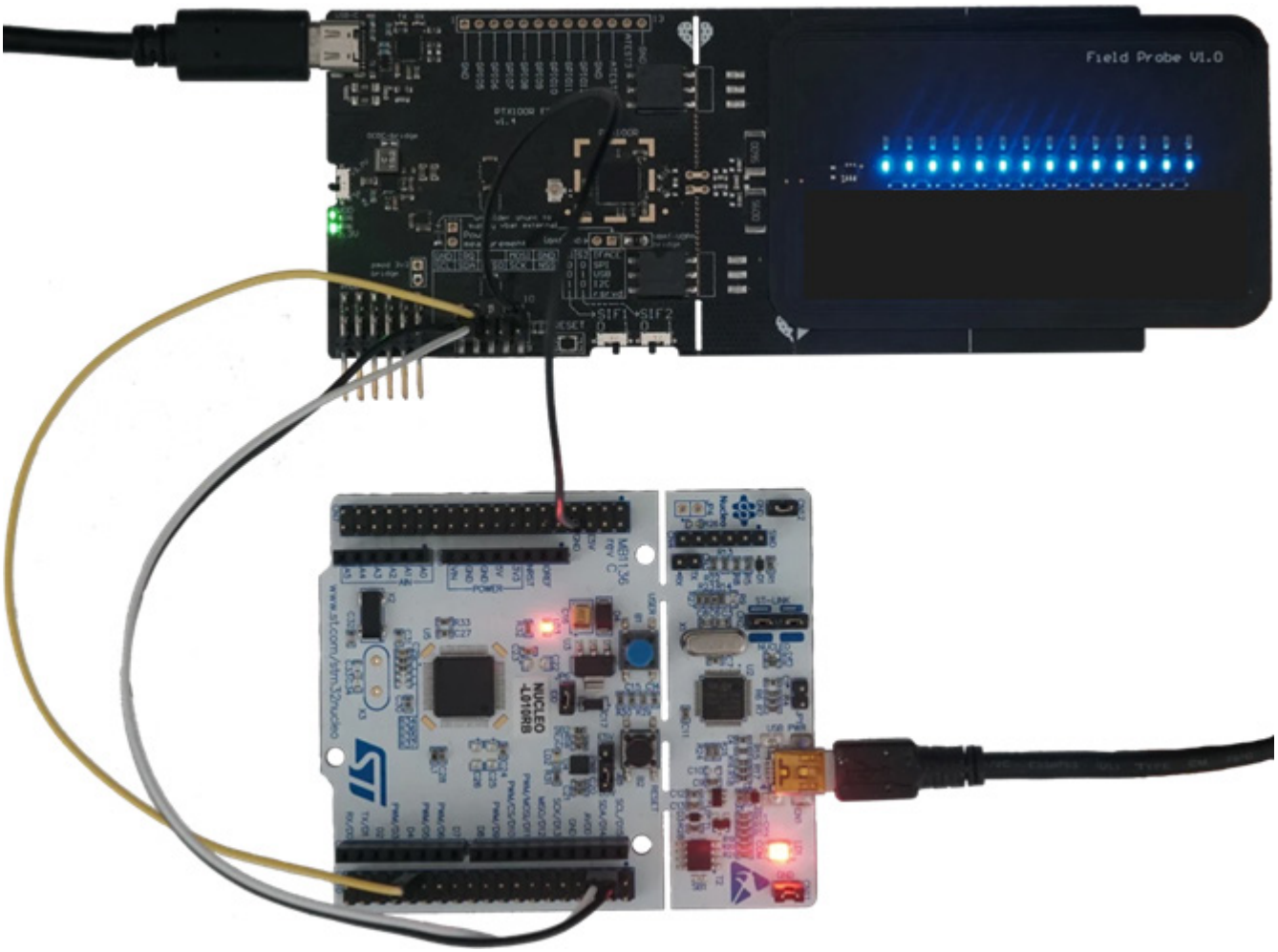


Figure 30. Field Probe Card–Presence of RF Field

10.3.1 Reading the Messages

If the ST-RTT is available in the system, start it in a separate terminal window to display the messages from the firmware.

The following output shown in Figure 31 was captured after a "Type B" followed by a "Type A" card had been held into the field:

```
PS C:\> C:\tools\strtt\strtt.exe -tcp
System Initialization ... OK

Print Revision Information...

C-Stack Revision.......: 0x1234
Local Modifications....: 0x0000
NSC-Code Revision......: 9214
NSC-Toolchain Revision.: 8362
Chip-ID................: 0x21
Product-ID.............: 0x00 (PTX100x)

Print Revision Information...OK
Start of RF-Discovery ... OK
Entering Application-Mode ... OK
Waiting for discovered Cards or external Fields ...
00
TX = 00A404000E325041592E00
TX = 00A404000E325041592E5359532E444446303100
00
TX = 00A404000E325041592E5359532E444446303100
RX = 00B20104009000
00
TX = 00A404000E325041592E5359532E444446303100
RX = 00B20104009000
Restarting RF-Discovery ... OK
Waiting for discovered Cards ...
Card activated ... OK!
01. RF-Technology = Type-B; SENSB_RES: 50C92A8EAC0000000080817100; Protocol....: ISO-
DEP; ATTRIB2: 00; ATTRIB_RES: 00
TX = 00A404000E325041592E5359532E444446303100
A404000E325041592E5359532E444446303100
RX = 00B20104009000
Restarting RF-Discovery ... OK
Waiting for discovered Cards ...
Card activated ... OK!
01. RF-Technology = Type-B; SENSB_RES: 50C92A8EAC0000000080817100; Protocol....: ISO-
DEP; ATTRIB2: 00; ATTRIB_RES: 00
TX = 00A404000E325041592E5359532E444446303100
00A404000E325041592E5359532E444446303100
RX = 00B20104009000
Restarting RF-Discovery ... OK
Waiting for discovered Cards ...
Card activated ... OK!
01. RF-Technology = Type-A; SENS_RES: 0400; NFCID1_LEN: 04; NFCID1: F2CE7633; SEL_RES:
08; Protocol: T2T
TX = Card activated ... OK!
01. RF-Technology = Type-A; SENS_RES: 0400; NFCID1_LEN: 04; NFCID1: F2CE7633; SEL_RES:
08; Protocol: T2T
TX = 3000
RX = 0404
Restarting RF-Discovery ... OK
Waiting for discovered Cards ...
Card activated ... OK!
01. RF-Technology = Type-A; SENS_RES: 0400; NFCID1_LEN: 04; NFCID1: F2CE7633; SEL_RES:
08; Protocol: T2T
TX = 3000
RX = 0404
Restarting RF-Discovery ... OK
Waiting for discovered Cards ...
Card activated ... OK!
```

```
01. RF-Technology = Type-A; SENS_RES: 0400; NFCID1_LEN: 04; NFCID1: F2CE7633; SEL_RES:
08; Protocol: T2T
TX = 30Card activated ... OK!
01. RF-Technology = Type-A; SENS_RES: 0400; NFCID1_LEN: 04; NFCID1: F2CE7633; SEL_RES:
08; Protocol: T2T
TX = 3000
RX = 0404
Restarting RF-Discovery ... OK
Waiting for discovered Cards ...
```

Figure 31. ST-RTT Output

Note: The ST-RTT always needs an active debug session and may be used during debugging.

Once the application is running detached on the MCU, an active debug session is no longer required to display the output of the firmware. Even after restarting the board, it starts polling and displaying the data read from the tag. To better visualize this, launch the ST-RTT WebUSB version.

To view the messages in the web browser, perform the following actions:

1. Flash the firmware and close any existing debug session.
2. Click *Open STLINK* to connect to the debug probe.
3. Reset the NUCLEO-L010RB Evaluation Board by depressing the black pushbutton, **B2** – **Reset**.

4. Click *Start RTT* within 2 seconds, otherwise the web app will not be able to identify the RTT buffer.

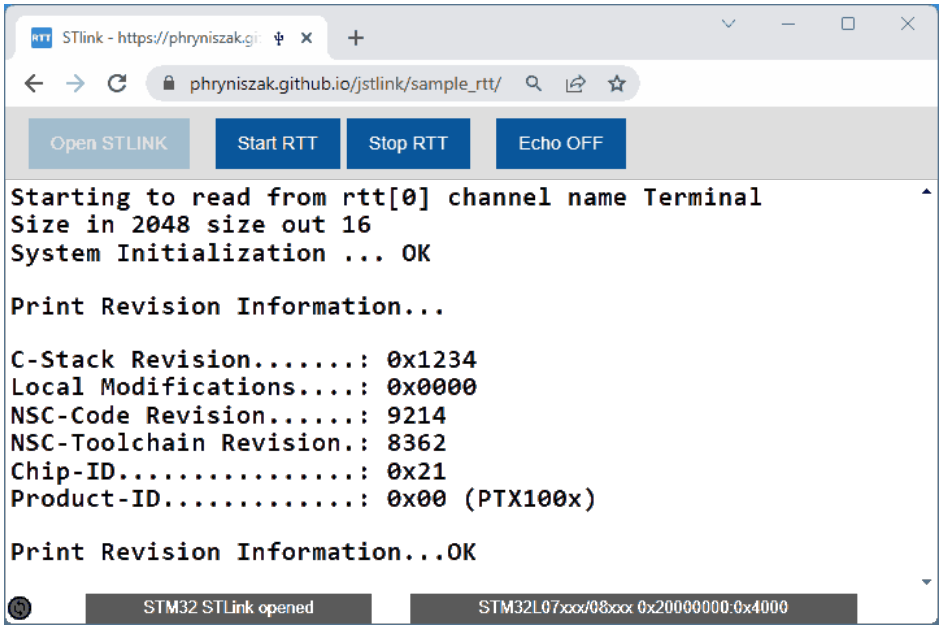


Figure 32. ST-RTT Sample Window

10.3.1.1 Required Changes

As the firmware enters power-saving mode during idle periods, the debugger cannot readout the RTT messages. To enable the correct message readout, DeepSleep mode needs to be disabled by commenting out the responsible statement in the `goToSleep()` in `file ptxPLAT.c`:

```
// SCB->SCR |= SCB_SCR_SLEEPDEEP_Msk; // enables deepSleep to optimize power consumption
```

Figure 33. Disabling DeepSleep Mode

10.3.2 Change the Code

The SDK example code performs shutdown temperature calibration at each start. This is disabled in the actual production code (for more information, see the NFC Reader Demo Application section). This process should be done only once in the chip's life during end-of-line testing in a controlled environment. The resulting raw value representing the shutdown temperature is to be saved to non-volatile memory and accessed from there for subsequent initializations of the SDK.

To disable the automated temperature calibration, set the representative variable in the function `ptxIOT_READER_App()` to 0:

```
uint8_t start_temperature_sensor_calibration = 0;
```

Figure 34. Disabling the Automated Temperature Sensor Calibration

After changing the variable, the default shutdown temperature value (223) will be used without performing a calibration during SDK initialization.

The changes take effect after building the firmware again and restarting the debugger.

11. Final Firmware Image

The final firmware image demonstrated in this project is not difficult to build with cross-platform tools and offers a flexible setup to be used with an arbitrary MCU requiring minimal changes. The following subsections provide some information on the generated FW image (`ptx1k_porting_demo.elf`).

11.1 Memory Footprint

The following table provides a detailed overview of the memory footprint of different software components and can assist in the planning of memory requirements (ROM and RAM) for a custom product.

Module	Code	R/O Data	Init Data	Non-init Data	Total ROM	Total RAM
STDLIB	3694	51	100	44	**3745**	**144**
APP	1578	932	20	0	**2510**	**20**
RTT	3616	58	0	2233	**3674**	**2233**
HAL	1088	0	0	16	**1088**	**16**
DRIVER	19994	85	12	96	**20079**	**108**
SDK	21832	19193	16	1596	**41025**	**1612**
Total:	**51802**	**20319**	**148**	**3985**	**72121**	**4133**

The listed module components are:

- **STDLIB**: The standard library implementing some low-level functions (for example. `printf()`)
- **APP**: Demo application from the SDK with additional code to invoke it and initialize the MCU
- **RTT**: Segger RTT implementation
- **HAL**: Functions implementing the `ptxPLAT.h` functions
- **DRIVER**: CMSIS and low-level ST-HAL implementation to access the peripherals
- **SDK**: PTX SDK implementation

The listed MCU sections are:

- **Code**: Code executed from the ROM memory
- **R/O Data**: ROM memory that stores data read during execution/boot-time
- **Init data**: Global initialized data in readable memory (RAM)
- **Non-init data**: Non-initialized data in RAM

11.2 Stack Usage

The current implementation of the example application requires a stack of 5108 Bytes. This may be optimized further by moving larger structure allocations (for example. `ptxIoTRd_t iotRd`) to the global space.

12. Troubleshooting

Below is a table for troubleshooting issues that may arise in the product, as well as their respective possible causes and solutions.

Issue Description	Possible Cause/Solution
CMake reports error during configuration or build	▪ Command-line parameters are incorrect or missing ▪ Incorrect compiler specified ▪ CMake or the build tool (`ninja`, `make`, etc.) not found ▪ Missing `arm-none-eabi-newlib` installation ▪ CMake version is outdated
No communication possible to the PTX1xxX	▪ PTX100R evaluation board is not powered by USB ▪ **SIF1** and **SIF2** switches are not set in **I²C** position on PTX100R evaluation board ▪ Bus and IRQ wires are not connected to the proper MCU pins ▪ The I²C bus wires are not pulled up or the pull-ups are too weak
Initialization of the PTX1xxX fails	▪ IRQ pin not connected to MCU or misconfigured in HAL ▪ IRQ/Timer service routine or registered callback function gets invoked without proper trigger ▪ Stack size is too small and the firmware crashes due to memory corruption ▪ Data on bus gets corrupted by glitches/crosstalk coupled in from in-circuit or environmental RF-noise
Debug session gets aborted after some time / RTT readout does not work	▪ MCU enters power-saving mode; it needs to be disabled

13. References

Renesas offers several variants of the PTX1xxX chip for different purposes. The PTX105R is one of the variants optimized for IoT applications.

There are two evaluation boards available for this chip and both of them may be used with the firmware presented in this document:

- The PTX105R Pmod™ Board for IoT (PTX105RQC) is fully integrated into the Renesas Quick-Connect IoT ecosystem and provides a full-featured high-end NFC reader solution for easy integration. This enables a user to add an instance of the board to the application in a user-friendly graphical manner using Renesas' e2studio IDE.
- The PTX105R evaluation board (PTX105REK) demonstration kit uses a different chip but is similar to the PTX100R POS/IoT NFC Reader demonstration kit (PTX100REK).

14. Revision History

Revision	Date	Description
1.00	Dec 20, 2023	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.