RF/NFC abstraction layer (RFAL)

Introduction

The RF abstraction layer is a library that provides several functionalities required to perform RF/NFC communications and other common related operations. It integrates different RF ICs (existing and future ST25R devices) into an easy-to-use common interface.

This common interface makes the upper software layers independent from the underlying device. The RFAL with the same APIs is available for all ST25R devices, thus reducing software effort when prototyping and migrating from one device to another.

Table 1. Applicable products

Type	Part number
ST25R3911	ST25R3911B
	ST25R3912
	ST25R3914
	ST25R3915
ST25R3916	ST25R3916
	ST25R3916B
	ST25R3917
	ST25R3917B
	ST25R3918
	ST25R3920
	ST25R3920B
ST25R95	ST25R95
ST25R200	ST25R100
	ST25R200
ST25R500	ST25R300
	ST25R500

1 Acronyms

Table 2. **Acronyms**

Acronym	Description
AC	Analog configuration
ACM	Active communication mode
AFE	Analog front end
AP2P	Active P2P
APDU	Application protocol data unit
API	Application programming interface
CE	Card emulation
CPU	Central processing unit
DEP	Data exchange protocol
DLMA	Dynamic LMA
DPO	Dynamic power output
FAQ	Frequently asked question
FDT	Frame delay time
FWT	Frame wait time
FW	Firmware
FIFO	First in first out
GPIO	General purpose input/output
GT	Guard time
HAL	Hardware abstraction layer
HW	Hardware
IC	Integrated circuit
LMA	Load modulation amplitude
MCU	Microcontroller unit
MPU	Microprocessor unit
NFC	Near field communication
NFCID	NFC identifier
P2P	Peer-to-Peer
PCD	Proximity coupling device
PCM	Passive communication mode
PICC	Proximity inductive coupling card
UID	Unique identifier
WTX	Waiting time extensions
RAM	Random access memory
RFAL	RF abstraction layer
RF HAL	RFAL hardware abstraction layer
RF HL	RFAL higher layer
RW	Reader/writer
SW	Software

Acronym	Description
TC	Test case
TS	Test suite

2 RFAL library

2.1 Features overview

- Complete middleware to build NFC enabled applications using the ST25R high performance NFC readers
- Supports all major NFC technologies and protocols
- Supports all ST25R devices
- Easy portability across multiple platforms (MCUs/RTOSs/OSs) and architectures (8- to 32-bit)
- Compliant with RF/NFC standards: NFC Forum, EMVCo®, ISO4443, ISO15693, ISO18092
- MISRA C:2012 compliant
- Frequently updated
- Several sample application examples available (refer to the software deliverables on the product page)
- Free, user friendly license terms

Table 3. Supported modes

		ST25R3911B	ST25R3916	ST25R95	ST25R200	ST25R500
RW	NFC-A	x	x	x	x	x
	NFC-B	x	x	x	x	x
	NFC-F	x	x	x	-	x
	NFC-V	x	x	x	x	x
CE	NFC-A	-	x	-	-	x
	NFC-F	-	x	-	-	x
P2P		x	x	-	-	x
AP2P	Initiator	x	x	-	-	-
	Target	x	x	-	-	-

2.2 Description

The RFAL provides some functionalities required to perform RF/NFC communications, as well as other common tasks conducted by an NFC device. The different RF ICs (existing and future ST25R devices) are encapsulated into a common interface that provides support for:

- NFC modes:
  - Reader/Writer (Poller)
  - P2P initiator (PCM and ACM)
  - P2P target (PCM and ACM)
  - Card emulation (Listener)
- Technologies:
  - NFC-A (ISO14443-A)
  - NFC-B (ISO14443-B)
  - NFC-F (FeliCa™)
  - NFC-V (ISO15693)
  - P2P (ISO18092)
  - Proprietary technologies (for example, ST25TB, CTS, Kovio™, B', iClass®, Calypso®)
- Protocols:
  - ISO-DEP (ISO data exchange protocol - ISO14443-4)
  - NFC-DEP (NFC data exchange protocol - ISO18092)
- Operating modes:
  - NFC device mode
  - Low power mode
  - Wake-up mode

2.3 Coding rules and conventions

2.3.1 Coding conventions

This section describes the coding rules used in the library:

- All code complies with the C99 standard and compiles without warning under at least its main compiler. Warnings that cannot be eliminated must be documented/commented along the code.
- The library provides both blocking and non blocking interfaces, which can be used separately or in combination, to better suit different project needs.
- Non blocking APIs are designed to last at most 5 ms.
- Non blocking APIs are provided in the form of pairs: `rfalStartOperation()` and `rfalGetOperationStatus()`.
- The library is projected to make use of ANSI standard data types defined in the ANSI C header file `<stdint.h>`. If needed, it is possible to apply user defined types.
- Dynamic memory allocation is not used within the library. Modules that require context allocate it statically, and any buffer of substantial dimension must be provided by the user. Exceptions exist, where some larger buffers are statically allocated in certain optional and convenience features/modules, such as `ST25R_COM_SINGLETXRX`, NFC module.

2.3.2 Run time checking

The library implements run time failure detection by checking the input values of the library functions. If a function is called with an invalid parameter or in inconsistent moment/state, the method returns the corresponding error. Another check at run time is via the `platformAssert()` macro.

Furthermore, in case an unrecoverable error occurs, the library provides an error handler/trap via the `platformErrorHandle()` macro.

2.3.3 MISRA-C 2012 compliance

The RFAL library is fully compliant with the MISRA C:2012 standard, to better enable automotive applications.

MISRA C is a set of guidelines for programming in the C language, developed and published by the motor industry software reliability association (MISRA). These guidelines aim to improve code safety, security, portability and reliability by identifying aspects of the C language that must be avoided due to their ambiguity and susceptibility to common programming mistakes. The MISRA standard is widely used by the automotive industry for embedded software.

For further details refer to the compliance report provided on RFAL's documentation folder enclosed within each release package.

2.3.4 NFC nomenclature

NFC (NFC Forum) is an aggregation of pre-existing RF standards.

As NFC Forum provides a combined definition of those standards in a concise set of specification documents using their own nomenclature, the terms used inside RFAL are aligned with NFC Forum specifications [1], [2], [3]. Along the RFAL references to the other standards may be provided for the sake of completeness and when deemed relevant.

2.4 Hardware

2.4.1 Requirements

Depending on the used device, the hardware requirements in top level to enable RFAL are:

- Serial interface bus (SPI, I²C or UART) capable to drive the ST25R NFC reader (refer to the datasheet)
- GPIOs for the serial bus and other control lines, such as IRQ, bus selection, and reset pins
- External interrupt trigger source (preferred approach, not mandatory)
- An MCU/MPU/system with sufficient RAM and flash memory to accommodate the RFAL library
- Ability to measure time intervals in a simple millisecond tick counter

2.4.2 Supported host devices

As the RFAL is provided in C code, it is supported by the majority of controllers available in the market. Any device featuring a C compiler and with the hardware requirements listed above can potentially support this library and its ST25R device.

STM32 microcontrollers meet all RFAL requirements in a wide variety of controllers and feature sets for different applications.

Less advanced devices with different architectures, such as the STM8 microcontrollers, can also be used. If an embedded operating system is required, the STM32MPU microprocessors can also enable any NFC application using RFAL.

2.5 Software

2.5.1 Dependencies

The RFAL library is based on some common SW abstractions typically available with the C compiler toolchains and the device drivers that are frequently provided by the host device's HAL.

- C toolchain
  - `stdint.h`, `stdbool.h` for type definitions (user defined types are also possible)
  - `limits.h` for parameter checking
  - `math.h` for mathematical operations
- `rfal_utils`
  - Provides a list of predefined errors and macros used by the RFAL to handle and report specific errors/events.
  - A glue layer that provides some convenience macros and makes possible to redirect some typical utilities, such as Min, Max, memcpy.

- HAL
  - Serial interface (SPI, I²C, UART), to transmit and receive data
  - GPIO, to control specific features (such as IRQ, chip select or reset pins)
  - Timer, to control internal mechanisms that must comply with given time requirements

All the dependencies listed above must be provided in a central location/file named `rfal_platform`. Every RFAL modules relies on this configuration file, which allows the user to define/remap all platform specifics in the form of includes, feature switches and macro definitions. More info is available in Section 4.1.

2.5.2 Supported development tools

The RFAL library requires only standard toolchain features, supported by all major tool providers. Suppliers provide a full range of development solutions that deliver start-to-finish control of application development in a single integrated development environment.

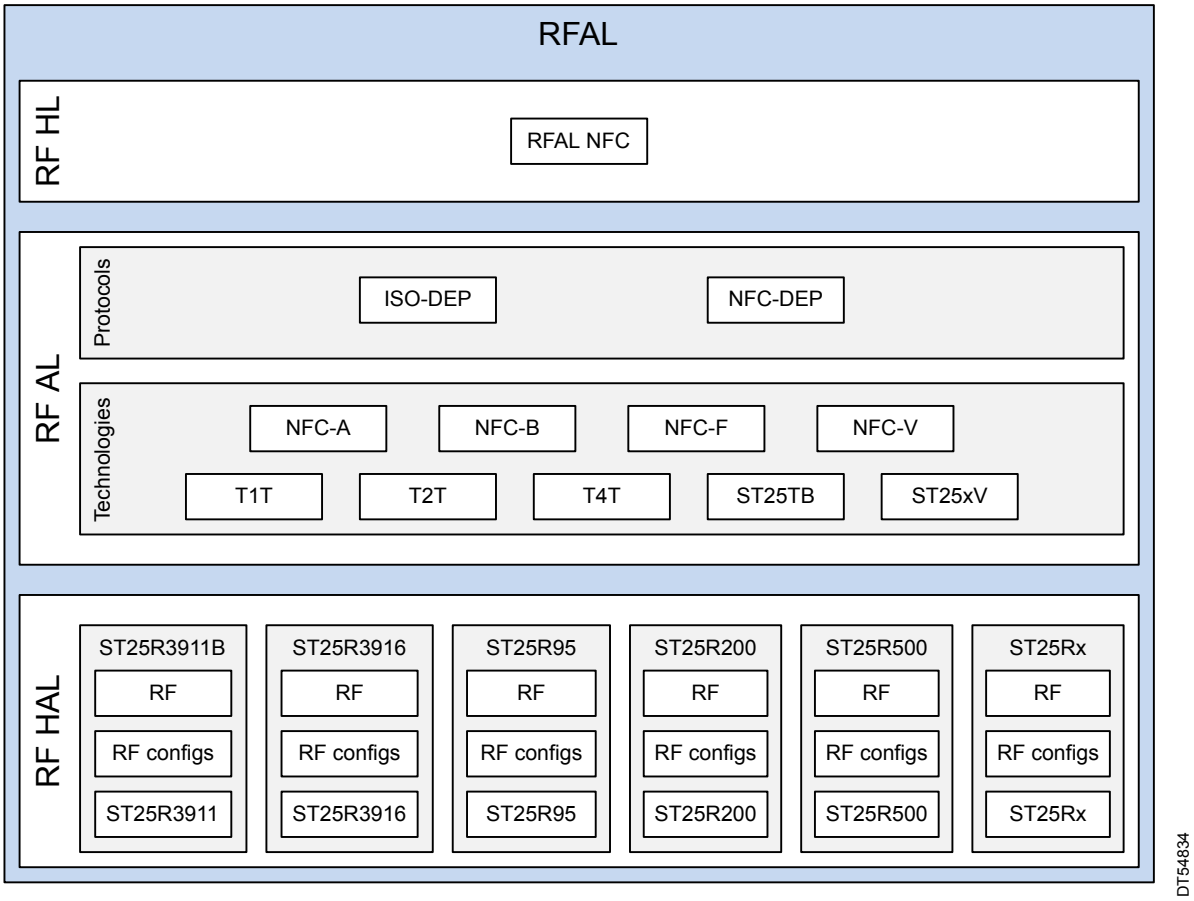
Below a list containing examples of those environments commonly used and where example projects are usually provided:

- STM32CubeIDE
  - Compiler: GCC's C/C++
- ST Visual Develop
  - Compiler: Cosmic's C
- Keil® µVision
  - Compiler: Arm C/C++
- IAR™ embedded workbench
  - Compiler: IAR C/C++

2.6 Architecture

The RFAL library is a multilayer stack with different modules and abstractions for different application needs.

Figure 1. RFAL stack architecture



Note: ST25Rx represents future devices that may be added to the ST25R family.

Internally the RFAL is divided into three sub layers:

- RF HL
- RF AL
- RF HAL

2.6.1 RF HAL

The low level drivers for each NFC IC are provided in this layer.

All encapsulated modules are chip specific/dependent, and are provided for each ST25R device. The modules are:

ST25R

This module contains the low level support of the NFC IC with direct control of the HW by means of registers access and the execution of commands (for futher details refer to the datasheet). These provide an interface to the execution of some HW activities.

Note: Direct access to the APIs is not recommended: APIs are chip specific, and bypass the normal driver flow, making the application not portable. Typical applications use the RF module as the lowest interface.

RF configurations

This module, also called analog configurations, contains table/list of analog settings that are applied by the RFAL at specific moments of the execution.

The table contains a set of RMVs (register, mask, value) that allow for certain configuration (bits) to be loaded at different moments of the driver execution. This allows for the IC settings to be changed if needed depending on the mode the device is currently operating.

Individual sets of RMVs are loaded (if defined) at specific times of the RFAL driver run. These are internally encoded by a 16-bit identifier. Furthermore, specific events are also encoded by this identifier, such as:

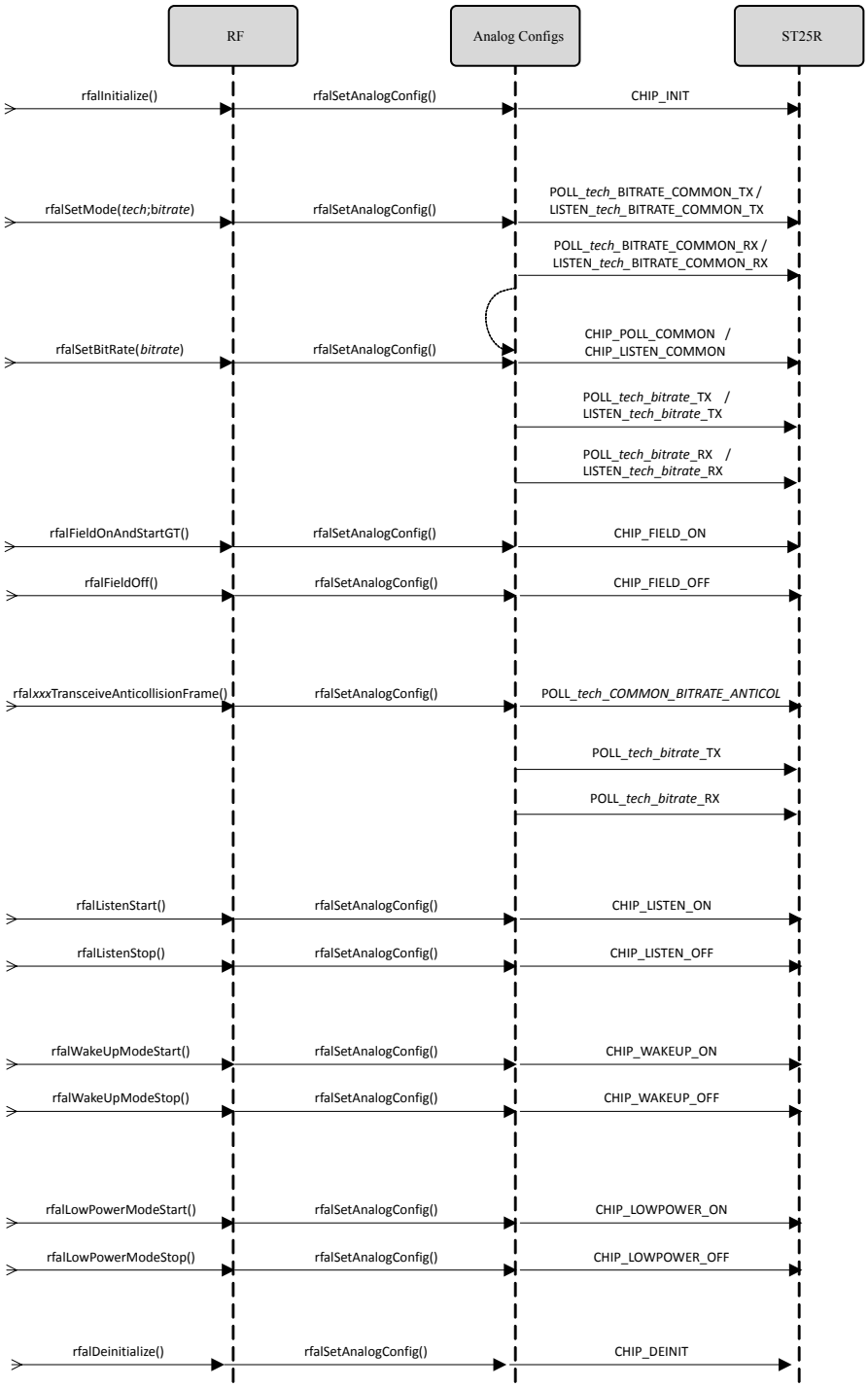
- `Chip Init`: loaded upon IC start up / reset / initialization
- `Chip Deinit`: loaded upon deinitialization of the IC
- `Field On`: loaded before transition to Field On
- `Field Off`: loaded after transition to Field Off
- `Wake-up On`: loaded before entering wake-up mode
- `Wake-up Off`: loaded after exiting wake-up mode
- `Listen On`: loaded before entering Listen mode
- `Listen Off`: loaded after exiting Listen mode
- `Low power mode On`: loaded before entering Low Power mode
- `Low power mode Off`: loaded after exiting low power mode
- `Poll Common`: if in Poll mode, loaded before bitrate specific configurations
- `Listen Common`: if in Listen mode, loaded before bitrate specific configurations
- `Power Level n`: loaded if a specific power level is entered (may be used in conjunction with DPO).

Note: Settings/register bits here concern only analog aspects of the device. The other bits required for proper operation are not manipulated and are handled by the normal driver execution. Great care and a considerable experience level is required when computing/manipulating such analog configuration table.

Note: Some ST25R devices (such as ST25R200 and ST25R500) have registers on different power domains. Settings loaded on registers belonging to RD (Ready) domain are reset when the device is put in PD (Power Down) or WU (Wake-up) mode. Once the device is back in RD mode, the registers contain their default values. Hence, configurations on registers belonging to RD domain must not be placed on Chip Init, which is loaded only at chip initialization, but managed via the applicable analog configuration, and/or restored via Wake-up Off and Low power mode Off.

Figure 2 shows the analog configuration sequence.

Figure 2. Analog configuration sequence



Note: In case of need there is the possibility to instruct the RFAL to use the user defined table, hereafter referred as custom analog configuration.

The RFAL library provides a default analog configuration table, used during testing to verify the release.

The loading of ACs is performed hierarchically. If the same setting/configuration/bit is addressed in multiple AC entries, the last one prevails regardless of the previous.

Note: A common misconception is that the AC entry `POLL_COMMON` or `LISTEN_COMMON` is loaded before all the other mode and bitrate specific entries. To ensure that the `POLL/LISTEN_COMMON` entry is always loaded, this AC is loaded by the `rfalSetBitrate()`, which is executed by the `rfalSetMode()` after loading mode specific the AC entries.

The `rfalSetMode()` usage is particularly complex as it sets both the mode and a bitrate dependent setting. By this, it internally includes the execution of `rfalSetBitrate()`.

1. *`POLL_NFCA_COMMON_TX`*
2. *`POLL_NFCA_COMMON_RX`*
3. *`POLL_ COMMON`*
4. *`POLL_NFCA_106_TX`*
5. *`POLL_NFCA_106_RX`*

If the `rfalSetMode()` API is executed for NFC-A Poller at 106 kbps it is necessary to load the following sequence:

Detailed explanation on the more complex usage (setting a new mode and bitrate)

Custom analog configuration

In this case a user defined custom AC table is required.

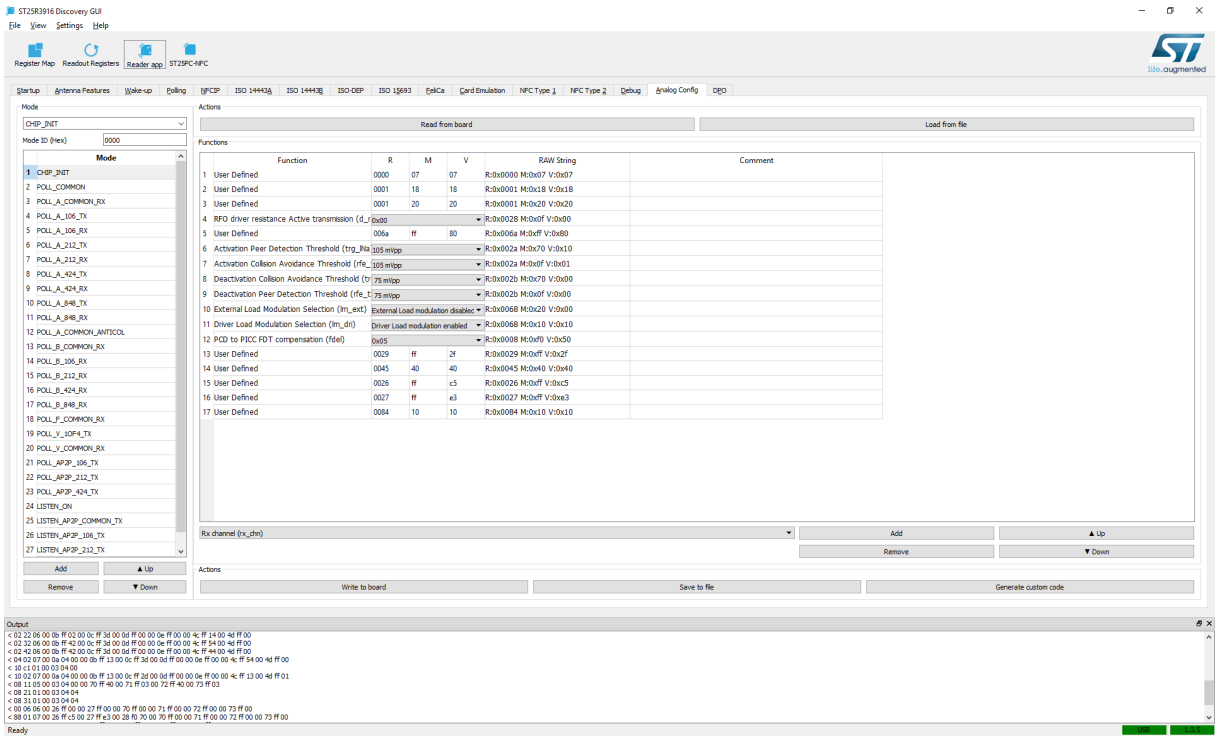
To enable it, the `RFAL_ANALOG_CONFIG_CUSTOM` pre-processor definition must be added together with the following definition variables:

- `rfalAnalogConfigCustomSettings`: the analog configuration table following the same RFAL format with the custom configurations
- `rfalAnalogConfigCustomSettingsLength`: the size in bytes of the `rfalAnalogConfigCustomSettings` table

Examples of such custom AC tables can be found in some demonstration projects (e.g. ST25R3916 NUCLEO).

The process of customizing AC tables is complex, requires great care and a certain level of experience. To facilitate this process, the usage of the DISCO GUI is highly recommended.

Figure 3. **Analog configuration tab**



On the DISCO GUI, a tab dedicated to AC is available (even with no demonstration HW/FW). Within it, users load AC tables, visualize them, modify settings, and generate the custom AC file to include in the projects.

RF

The RF module holds a pivotal part of the whole RFAL library, where all necessary handlings are provided granting support for each individual NFC IC.

This module provides the APIs to initialize the NFC drivers, to set them into the right modes, and to perform the required operations. It is also in charge of several critical activities and the processes, referred as workers, which require the NFC task to be serviced periodically – `rfalWorker()`. Those activities/processes are:

- **Transceive**
  Transceive operations are the most common procedure of the whole library. They are the simplest way to transmit and receive arbitrary data, regardless of protocol selected. During a transceive operation the process/worker controls and monitors the AFE to complete the data exchange as requested. This requires that the process/worker (`rfalWorker()`) is serviced frequently enough to meet some strict time critical operations (reloading of the HW FIFO, EMD handling, among others).
- **Listen mode**
  The Listen mode is used to establish NFC communication in Listen mode, PICC or CE mode. As the Listener is activated in multiple protocols and bitrates, this process enables to configure the supported/allowed modes/protocols, and then it monitors and handles that whole card activation automatically. This mode ends when the first data packet is received from the Poller after the low level activation is completed. In the RF layer the card activation is completed before the protocol activation, as detailed below:
  - NFC-A: first packet received after the UID/NFC ID selection (RATS command in case of a T4T emulation, or an `ATR_REQ` in case of P2P)
  - NFC-F: first packet received after the `SENSF_REQ` (check/update command in case of a T3T emulation, or an `ATR_REQ` in case of P2P)
  - AP2P: first data received `ATR_REQ`

  Once the activation is done, and after a data packet the upper/protocol layers check the correct protocol activation, set the correct mode and make use of transceive operations to exchange data (same as in Poller mode).

- **Wake-up mode**
  In this mode the AFE is put in wake-up mode where the device remains in a low power state and periodically performs specific measurements to assess whether there is some change on its surroundings. The worker needs to run to update.
  An additional option for certain ST25R devices is also available, where the RFAL performs SW Tag detection. In this mode most of the activities are triggered by the host directly (not by the AFE itself) and therefore the process/worker needs to be frequently executed.
  Apart from the common interface provided by this layer in `rfal_rf.h`, access to more chip specific operations can also be achieved via a chip independent interface available via `rfal_chip.h`. This permits for a generic interface to perform certain actions such as read register or execute commands while keeping the SW layers agnostic to the actual AFE itself.

2.6.2 RF AL

The NFC protocol stack, placed on top of the low-level drivers of the RFAL HAL, is provided in this layer. All modules described here are device independent, and their focus is to provide support for individual features, technologies, and protocols.

Individual modules can be disabled/excluded, when not needed, to better fit into a multitude of application needs.

Note: To enable/disable individual features/modules of the RFAL refer to Section 4.2.

Technologies

These modules provide support and convenience methods for several NFC technologies. For each supported technology there are interfaces for common tasks, depending on each technology specific needs (most of them detailed in [1]). Examples of those activities are:

- Initialization
- Technology detection
- Collision resolution
- Activation
- Sleep/Deactivation
- Computing/parsing commands and responses
- Technology/card specific operations

NFC-A

This module provides the functionality required to support NFC-A (ISO14443A) compliant devices. It enables to configure the AFE for this mode covering analog, digital and protocol aspects of this technology. It provides a Poller (ISO14443A PCD) interface as well as some NFC-A Listener (ISO14443A PICC) helpers. Support for performing technology detection, anticollision/collision resolution, card selection/activation and sleep/deactivation are available.

The detection and selection of all NFC-A tag types (T1T, T2T, T4T) is handled by this module, which provides definitions and helper methods up to ISO14443-3 layer. Operations specific to a certain tag type are then provided in a dedicated module.

NFC-B

This module provides the functionality to support NFC-B (ISO14443B) compliant devices. It permits to configure the AFE for this mode covering analog, digital and protocol aspects of this technology as a Poller device (ISO14443B PCD). Support for performing technology detection, anticollision/collision resolution, card activation and sleep/deactivation are available.

The definitions and helper methods provided by this module cover up to ISO14443-three layer (excluding ATTRIB command which is used to enter ISO-DEP / ISO14443-4).

NFC-F

This module provides the interfaces to support NFC-F (FeliCa - JIS X6319-4) compliant devices. It permits to configure the AFE for this mode covering analog, digital and protocol aspects of this technology as a Poller device (FeliCa PCD). Support for performing technology detection, anticollision/collision resolution, card activation are available.

Support for T3T commands to read and write memory blocks is provided (`CHECK/UPDATE`).

NFC-V

This module provides the functionality to support NFC-V (ISO15693) compliant devices. It permits to configure the AFE for this mode covering analog, digital and protocol aspects of this technology as a Poller device (ISO15693 VCD). Support for performing technology detection, anticollision/collision resolution, card selection/activation and sleep/deactivation are available.

Support for typical T5T commands is provided.

Note: The APIs within this module enable accessing the tags in the three different modes as foreseen in ISO15693:

- ***Addressed mode**: after inventory no individual card is selected and each command contains the NFCID/UID of the tag addressed*
- ***Non addressed mode**: no individual card is selected and the commands exchanged do not contain a specific NFCID/UID. This causes all tags in the vicinity to react to the command (broadcast)*
- ***Select mode**: after inventory one individual card is selected using its NFCID/UID and each command thereafter does not contain an NFCID/UID*

These modes are controlled via the flags, and uid parameters on the API, and the usage of `rfalNfcvPollerSelect()`

T1T

This module provides the functionality to support T1T compliant devices. It permits to configure the AFE for this mode covering and exchange tag specific commands. Support for T1T commands to read and write memory blocks are here provided (`RID`, `RALL` and `WRITE_E`).

T2T

This module provides the functionality to support T2T compliant devices and their commands needed to read and write memory blocks (`READ`, `WRITE` and `SECTOR SELECT`).

T4T

This module provides convenience methods and definitions for T4T (ISO7816-4) with an interface to compute and parse T4T APDUs according to NFC Forum and ISO7816-4.

This does not perform any actual data exchange, it simply computes and parses APDUs which shall be then conveyed via the protocol layer ISO-DEP (ISO144443-4).

ST25TB

This module provides the functionality to support ST25TB devices (SRTx, SRIx). It permits to configure the AFE covering analog, digital and protocol aspects of this technology as a Poller device (ISO14443B PCD). Support for performing technology detection, anticollision/collision resolution, card selection/activation and sleep/deactivation are available as well as card specific operations such as memory access.

Protocols

After successful card activation, typically data exchange is performed between the Poller and the Listener (PCD and PICC). This data exchange is sometimes achieved by simple pair of command, response sequence or in the form of more advanced protocols.

These protocols provide a more robust mechanisms to ensure reliable data exchange, allowing data chaining mechanisms to split long frames into several chinks, and define specific handlings in case an error occurs allowing for recovery without breaking the on going communication link. These modules enable the usage of such protocols handling all protocol specifics (framing, timings, error handling, etc).

ISO-DEP

This module provides support for establishing an ISO-DEP (ISO14443-4) protocol link and to perform data exchange. It handles the protocol activation and deactivation in both Poller and Listener mode. Once the protocol is activated, data (typically APDUs) are exchanged by the means of transceive operations.

The transceive interfaces provided are non blocking and can be used in two different manners:

- **Block**: handles the exchange of a single data block (I-Block) which contains a complete or partial APDU. Upon transmission the caller must indicate whether the block is complete or partial (chained) chunk of data (see input `isTxChaining` parameter). Similarly, during reception if a chained packet is received the output parameter `isRxChaining` it set, and a special return code (`ERR_AGAIN`) is sent. This interface handles all protocol specifics such as control packets, error handling, repetitions, and returns either successful or not in case an irrecoverable error has occurred.
- **APDU**: handles the exchange of a complete APDU. In case the data length is bigger than the one supported and announced by one of the devices, it automatically splits the APDU into smaller chunks/ chained packets and handles the chaining, both for transmission and reception, automatically. This interface (which makes use of the block interface internally) requires an additional temporary buffer (of the maximum frame size supported) to cope with protocol specifics.

NFC-DEP

This module provides support for establishing an NFC-DEP (ISO18092, NFCIP1, P2P) protocol link and to perform data exchange. It handles the protocol activation and deactivation in both Poller and Listener mode (Initiator and Target).

This protocol layer are used for passive P2P and active P2P. For the passive P2P is performed a normal device anticollision and selection, for the AP2P an `ATR_REQ` is sent right after initial RF collision avoidance.

Once the protocol is activated, data (typically LLCP packets) can be exchanged by the means of transceive operations.

The transceive interfaces provided are non blocking and are used in two different manners:

- **Block**: it handles the exchange of a single data block (I-PDU) that contains a complete or partial PDU. Upon transmission the caller must indicate whether the block is complete or partial (chained) chunk of data – see input `isTxChaining` parameter. Similarly, during reception if a chained packet is received the output parameter `isRxChaining` is set, and a special return code (`ERR_AGAIN`) is sent. This interface handles all protocol specifics such as control packets, error handling, repetitions, and returns either successful or not in case an irrecoverable error has occurred.
- **PDU**: it handles the exchange of a complete PDU. In case the data length is bigger than the one supported and announced by one of the devices, it automatically splits the PDU into smaller chunks/chained packets and handle the chaining, both for transmission and reception, automatically. This interface (which makes use of the Block interface internally) requires an additional temporary buffer (of the maximum frame size supported) to cope with protocol specifics.

Note: In both protocols (ISO-DEP and NFC-DEP), once a chained block/DEP is received, the error `ERR_AGAIN` is sent. At this point the caller must handle/store the received data immediately. When `ERR_AGAIN` is returned an ACK has already been sent to the other device and the next block might be incoming. If `rfalWorker()` is called frequently it places the next block on the same buffer provided in the transceive parameters.

Note: In both protocols (ISO-DEP and NFC-DEP) a Listener device is only considered active after the first data packet or a protocol parameter selection (PPS or PSL) is received. Therefore, the interface that handles activation in Listen mode holds the first data packet in its buffer passed on activation parameters. Once the Listener activation has completed (`rfalIsoDepListenGetActivationStatus()` / `rfalNfcDepListenGetActivationStatus()` returns `ERR_NONE`) the method `rfalIsoDepGetTransceiveStatus()` / `rfalNfcDepGetTransceiveStatus()` must be called.

Note: If activation has completed due to reception of a data block (not PPS/PSL) the buffer owned by the caller and passed on the listen activation parameters must still contain this data. The first data are processed by `rfalIsoDepGetTransceiveStatus()`, inform the caller and then the next transaction can flow normally starting with `rfalIsoDepStartTransceive()` / `rfalNfcDepStartTransceive()`

2.6.3 RF HL

This layer sits on the top of the device specific drivers (RF HAL) and the necessary NFC technology and protocol stack (RF AL). It is the application/higher layer that makes use of the RFAL functionalities, such as NFC Forum Activities (NFCC), EMVCo, DISCO/NUCLEO demonstration.

This layer provides convenient abstractions for common NFC related tasks, such as NFC Forum compliant Poller and Listener device.

Modules contained in this layer are:

NFC

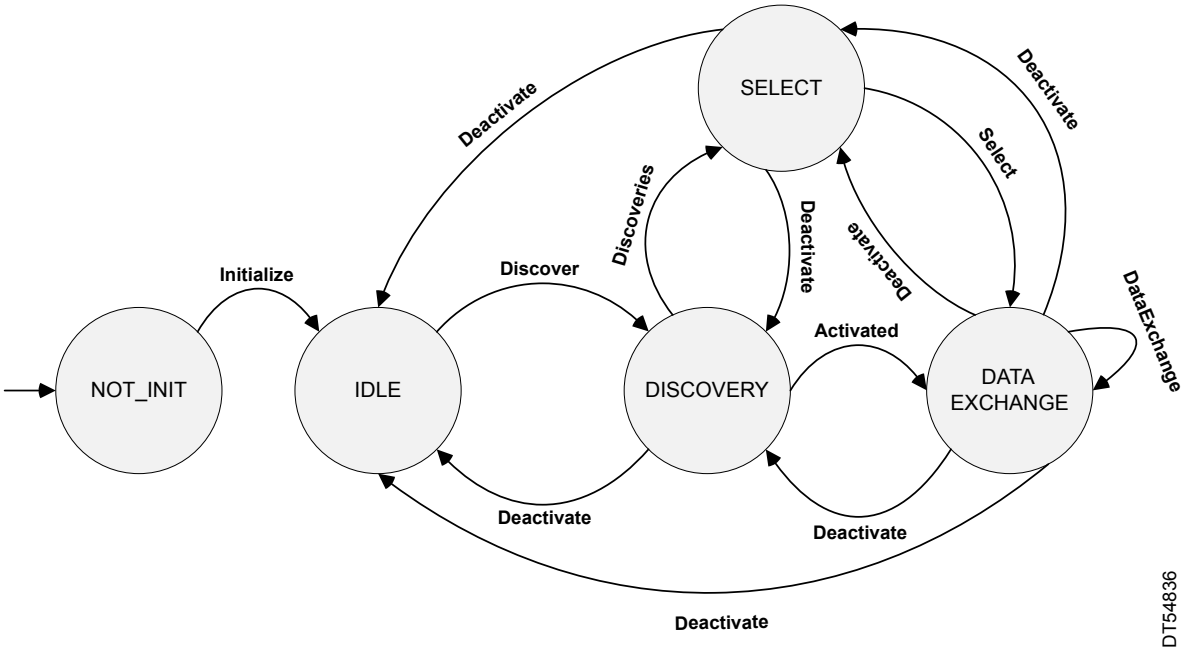
This module provides a simple interface to easily implement an NFC Forum compliant Poller and Listener device. It combines conveniently all NFC technologies and enables the functionality required to perform the multiple NFC activities [1]: technology detection, collision resolution, activation, data exchange, and deactivation.

The NFC module is influenced by (but not fully aligned with) the NFC Forum specifications, in particular Activity [1] and NCI [3].

The aim of this module is to provide support for common NFC implementations, while keeping its usage straightforward. Not all RFAL available parameters and configurations available are exported on this module interface.

Its inner state machine, which mimics NFC Forum NCI recommendation, is shown below. The arrows represent the available APIs where the caller can control this module, and the circles the states.

Figure 4. NFC module state diagram



Each state and transitions represents the following:

- States
  - NOT_INT: start-up state, not yet initialized
  - IDLE: module/RFAL has been initialized and ready
  - DISCOVER: discovery cycle ongoing. Activities depend on the configured modes and technologies. Internally the module transits through the several activities and states (wake up, technology detection, collision resolution, activation) according to the given configuration and devices presented
  - DATA EXCHANGE: represents that data exchange is done, is ongoing or is completed. In this state caller performs data exchange or simply deactivate the device
  - SELECT: in case multiple devices are identified, it notifies the caller and waits for an action to perform. Either activate/select a particular Listener device or deactivate
- Transitions (command / notifications):
  - INITIALIZE: performs RFAL initialization and goes to IDLE state.
  - DISCOVER: places the module in discovery mode according to the given configuration.
  - ACTIVATED: when a device has been activated (Poller or Listener) a notification occurs and the state activated is reached from which data exchange can be performed, or simply deactivate the device.
  - DATA EXCHANGE: data exchange is ongoing or completed.
  - SELECT: in case multiple devices are identified, it notifies the caller and waits for an action to perform. Either activate/select a particular Listener device or deactivate.
  - DEACTIVATE: performs the deactivation sequence, if applicable, and goes to the state requested by the deactivation type. Three deactivation types are available: IDLE, SLEEP, DISCOVERY.

As it contains its own state machine, the NFC module contains its own worker/process `rfalNfcWorker()` to keep control of NFC related activities. Its own worker calls internally the RFAL `rfalNfcWorker()`, alleviating the need for the caller to execute both workers.

This module also allows the user to register a notification callback to keep track of certain asynchronous events (notifications in NCI terminology).

For example, if the configured device limit allows it, multiple tags are found during the discovery state. In this case, the caller is informed that select state has been reached and one of the identified tags must be selected.

Usage of the notification callback (`notifyCb`) is not mandatory and caller opts to poll the state on the NFC task and act accordingly.

After device activation, data exchange is performed in a similar manner as for the other layers: a pair of transceive/data exchange methods are available `rfalNfcDataExchangeStart()` / `rfalNfcDataExchangeGetStatus()`. During device activation a particular interface is selected. The interface dictates the type of activation and deactivation sequence as well as the protocol used during data exchange. Three interfaces are available (analogous to NCI [3]):

- ISO-DEP: ISO-DEP (ISO4443A) protocol is used
- NFC-DEP: NFC-DEP (ISO18092, NFCIP1, P2P) protocol is used
- RF: no specific protocol/framing is used, a complete passthrough during transmission and reception is granted to the caller

Note: For supporting a wider range of technologies and protocols, when the RF interface is used the lengths are in number of bits (not bytes). Therefore both input `txDataLen` and output `rvdLen` of `rfalNfcDataExchangeStart()` / `rfalNfcDataExchangeGetStatus()` refer to bits. When ISO-DEP or NFC-DEP interface is used those parameters are expressed in number of bytes

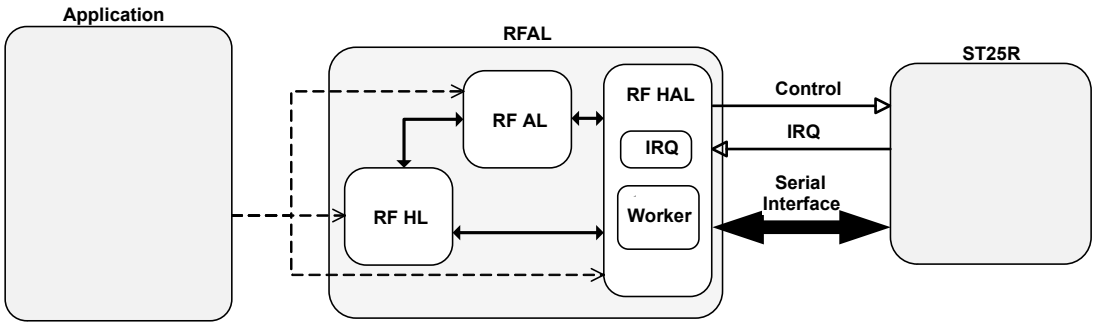
Note: In the protocols section, when in listen mode and activated state has been reached, the `rfalNfcDataExchangeStart()` method must be called initially with `txDataLen` set to zero to retrieve the first data packet on the `rxData` and `rcvLen` locations.

Note: As defined in Section 2.3.1 all buffers of considerable dimensions are not allocated by the library and must be provided by the user. This module is an exception to that rule as it aims for user convenience, and the transceive buffers are statically allocated within. Nevertheless, the size of such memory is user defined via the configurations listed in Section 4.2.

2.6.4 Overview

As shown in Figure 5, the RFAL is composed by several stacked layers and modules, organized according to the functionality they provide and their dependencies.

Figure 5. System overview



The RFAL contains the following components:

- **ST25R**
  `InterfaceHandled` by the low level driver it communicates/controls/monitors the AFE via three interfaces:
  - Control lines (such as RESET): at start up or reinitializations of the device
  - Serial interface (SPI, I²C UART): to access device registers, execute commands, etc
  - IRQ line/signal which the ST25R devices use to indicate a new event/interrupt. Once such occurs the host must retrieve the IRQ information as soon as possible (via ISR on an external interrupt or polling) and react accordingly.
- **RFAL worker**
  The worker/process is a core part of the RFAL library. Once an IRQ is signalled the IRQ status registers are retrieved from the ISR (if so configured) and the control restored to the application context. Several operations/modes require the RFAL worker/process to be executed in order for the new signal(s) from ST25R to be processed and specific handlings to take place. It is of paramount importance to ensure that the RFAL worker is serviced properly in order for the RFAL execute appropriately.

Despite the multiple hierarchal layers the library allows to interact with its higher and lower layers simultaneously. This fine control is granted to enable the different use cases and special needs of each project, but at the same time requires knowledge of the library internal mechanisms (RFAL HL NFC grants the most integrated and easier to use interface).

For example, an application wants to poll in AP2P mode. To achieve this the user has a few modules to interact: the lower RF module for initializations, timings, configurations and RF field control, and the NFC-DEP module for actual protocol polling and handling.

Numerous sequence diagrams are provided in Section Appendix B where the interaction between the different modules is detailed.

2.6.5 Concurrency

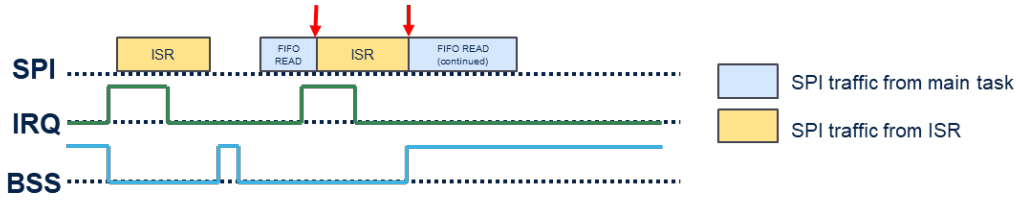
When integrating the library concurrency aspects need to be considered and an orderly access to the HW resources must be ensured. Different complexities are tackled depending on the nature of the systems itself, if bare metal or a multi-thread environment such as an RTOS or embedded Linux.

Serial interface atomicity

The NFC AFE is controlled via the serial interface which is addressed on different contexts or threads. It is important to ensure that each operation is atomic and that an ongoing operation, for example main/user context, is not interrupted by another one (e.g from an interrupt/ISR context).

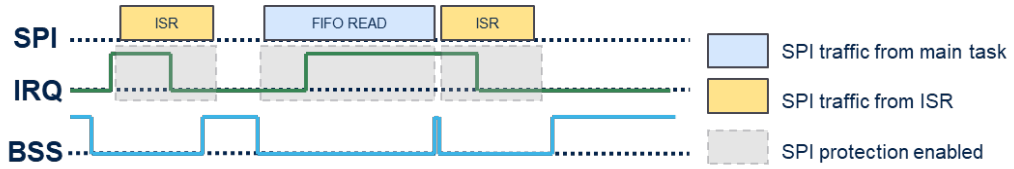
Such scenario is depicted in Figure 6 where a ST25R controlled via the SPI with two interrupts. The driver running on the main/user context then triggers a FIFO read operation to retrieve the received data and another interrupt is flagged by the ST25R which abruptly interrupts a ongoing SPI operation/traffic, resulting in an invalid SPI sequence.

Figure 6. Concurrency without protection



To avoid the scenario above each operation on the serial interface must be protected, blocking/claiming the shared resource preventing concurrent access from multiple contexts.

Figure 7. Concurrency with protection



In Figure 7 if an interrupt occurs (IRQ signal high) during an ongoing operation the handling of this event is deferred to when the serial interface is free again.

The RFAL provides the means of protecting the serial interface by the means of the following APIs: `platformProtectST25RComm()`, `platformUnprotectST25RComm()`, `platformProtectST25RIrqStatus()` and `platformUnprotectST25RIrqStatus()` detailed in Section 4.1. The actual protection implementation takes many forms depending on each platform and the system architecture. Recommendation is to make use of the interrupt enable mechanism if running on an embedded system in bare metal or the use of mutex in a multi-thread environment. Examples on this protection mechanism are available on multiple demonstrations available at st.com, namely X-CUBE-NFC3, X-CUBE-NFC5, X-CUBE-NFC5, STSW-ST25R-LIB as well as in Section Appendix A.

Multi-thread environment

When running in a multi-thread environment such as an RTOS or embedded Linux other constrains need to be considered. Implementations on these environments commonly make use of a thread to take care of the interrupt status and other(s) for user/main context.

In the same manner parallel access from multiple contexts needs to be protected and RFAL allows to deploy further protection mechanisms (commonly by the use of mutexes) by the following APIs: `platformProtectST25RComm()`, `platformUnprotectST25RComm()`, `platformProtectST25RIrqStatus()`, `platformUnprotectST25RIrqStatus()`, `platformProtectWorker()` and `platformUnprotectWorker()` detailed in Section 4.1.

Examples of such protection mechanism are available on several demonstrations at www.st.com. Such as:

- STSW-ST25R009
- STSW-ST25R013 packages for Linux
- STSW-ST25R-LIB for FreeRTOS.

3 Releases

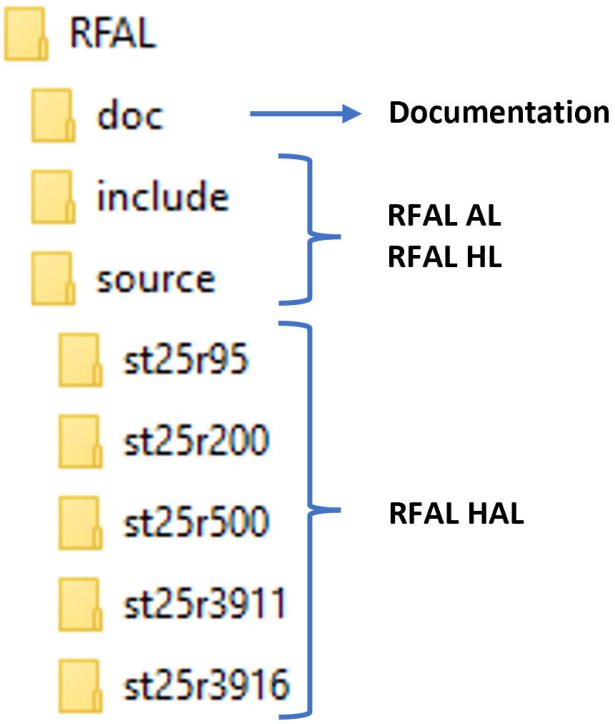
3.1 Package description

The RFAL is available at www.st.com in the form of multiple device specific packages (STSW-ST25RFALxxx). Each package contains the common RFAL AL and RFAL HL layers, as well as the corresponding RFAL HAL layer.

The RFAL HALs for individual ST25R devices coexist in the folder structure, for example on a project that is supported on different ST25R devices.

The package is structured as shown in the following figure.

Figure 8. RFAL package structure



- **doc/**: contains RFAL documentation which include release notes, detailed API documentation (Doxygen format), compliance reports, such as MISRA, for applicable devices
- **include/**: contains exported `.h` (header) files for the common RFAL AL and RFAL HL layers. Projects using the RFAL shall add this location to its include paths
- **source/**: contains `.c` (source) and some internal `.h` (header) files for the common RFAL AL and RFAL HL layers. Projects using the RFAL shall also add this location to its include paths
- **source/st25r.../**: contains `.c` (source) and internal `.h` (header) files for the common RFAL HAL. Projects using the RFAL must add this location to its include paths.

3.2 Quality criteria

To ensure the quality requirements of each RFAL release a series of quality checks are performed as part of the release process. On top of numerous internal tests, official test suites and static code analysis tools are used to ensure the consistency and quality of each release. As each ST25R device has its own feature set, a device specific group of applicable tests are applied. Those include:

- NFC Forum Digital TS
- EMVCo PCD/PICC Digital TS

- ISO IEC 10373-6 PCD Digital TS
- ISO IEC 23917 NFCIP-1 protocol TS
- MISRA-C 2012
- CERT-C

The tools and TSs are regularly updated to the latest available versions to meet the progress of each individual standard. Reports are available within the doc/ folder of the RFAL package and further test reports are provided upon request.

4 How to use the RFAL library

To set up and configure the RFAL a few settings must be defined by the user. These settings must be provided on a file named `rfal_platform`, to whom every RFAL modules relies on. Examples are found at: X-CUBE-NFC3, X-CUBE-NFC5, STSW-ST25R-LIB, as well as in Section Appendix A.

4.1 Peripherals initialization and configuration

As described in Section 2.4.1 the RFAL needs a few components to operate. Initialization of the GPIO, external interrupt source and serial interface must be assured (as per ST25R device requirements) on the Target platform, and HAL interfaces provided/defined to the RFAL.

The RFAL configurations, the platform specific HAL, the included systems and the interface definitions are collected in the `rfal_platform` file, the user has to adapt it to his own needs:

Pin definitions

- `ST25R_SS_PIN / ST25R_SS_PORT`
  Pin and port definitions used for SPI chip select (CS) / slave select (SS) (mandatory). The SPI CS must be controlled by SW, not automatically handled by the SPI peripheral. If HAL driver uses single definition to address GPIOs PORT and PIN, only one definition must be defined and used on GPIO macros.
- `ST25R_INT_PIN / ST25R_INT_PORT`
  Pin and port definitions used for ST25R IRQ line (mandatory). The state of this GPIO is checked on the ISR. If HAL driver uses single definition to address GPIOs PORT and PIN, only one definition must be defined and used on GPIO macros.
- `ST25R_RESET_PIN / ST25R_RESET_PORT`
  Pin and port definitions used for ST25R RESET line (optional). When available and defined, it is used during device initialization and power state management. If HAL driver uses single definition to address GPIOs PORT and PIN, only one must be defined and used on GPIO macros.
- `PLATFORM_LED_RX_PIN / PLATFORM_LED_RX_PORT`
  Pin and port definitions used for reception LED (optional). If defined LED drives this GPIO high during a reception. If optional LED feature is not be used remove these definitions. If HAL driver uses single definition to address GPIOs PORT and PIN only one definition must be defined and used on LED macros.
- `PLATFORM_LED_FIELD_PIN / PLATFORM_LED_FIELD_PORT`
  Pin and port definitions used for field LED (optional). If defined RFAL drives this LED high when the field is turned on. This optional feature is disabled by removing this definition. If HAL driver uses single definition to address GPIOs PORT and PIN, only one definition must be defined and then used on LED macros.

Interfaces to the user specific platform hardware driver (MCU HAL)

- `platformProtectST25RComm()`
  Protects the unique access to ST25R communication channel. IRQ disable on single thread environment (MCU); Mutex lock on a multi thread environment (mandatory).
- `platformUnprotectST25RComm()`
  Unprotects the unique access to ST25R communication channel. IRQ enable on a single thread environment (MCU); Mutex unlock on a multi thread environment (mandatory).
- `platformProtectST25RIrqStatus()`
  Protects the unique access to IRQ status var. IRQ disable on single thread environment (MCU); Mutex lock on a multi thread environment (optional).

- `platformUnprotectST25RIrqStatus()`
  Unprotects the IRQ status var. IRQ enable on a single thread environment (MCU); Mutex unlock on a multi thread environment (optional).
- `platformProtectWorker()`
  Protects/locks the single execution of the RFAL worker (optional). If not required, this macro must be left empty.
- `platformUnprotectWorker()`
  Unprotects/unlocks the unique execution of the RFAL worker (optional). If not required, this macro must be left empty.
- `platformIrqST25RPinInitialize()`
  Initializes the ST25R IRQ pin as input/external interrupt (optional). Upon initializiation, RFAL calls this macro to configure the IRQ pin. If GPIO is already configured by the user, this macro must be left empty.
- `platformIrqST25RSetCallback( cb )`
  Sets the ST25R ISR (optional). Upon initializing, RFAL calls this macro to set the callback (cb) of the IRQ pin. If the interrupt is already configured by the user and `st25r391xIsr()` is called upon an IRQ, this macro must be left empty.
- `platformLedsInitialize()`
  Configures LEDs as outputs. (optional). Upon initializiation, RFAL calls this macro to initialize the required pins as outputs. If the LEDs are not used or already configured by the user, this macro must be left empty.
- `platformLedOff( port, pin )`
  Turns LEDs off (optional). If the LEDs are not used, this macro must be left empty.
- `platformLedOn( port, pin )`
  Turns LEDs on (optional). If the LEDs are not used, this macro must be left empty.
- `platformGpioSet( port, pin )`
  Sets GPIO to logical high state (mandatory).
- `platformGpioClear( port, pin )`
  Sets GPIO to logical low state (mandatory).
- `platformGpioToogle( port, pin )`
  Toogles GPIO logical state (mandatory).
- `platformGpioIsHigh( port, pin )`
  Checks if the GPIO is in logical high state (mandatory).
- `platformGpioIsLow( port, pin )`
  Checks if the GPIO is in logical low state (mandatory).
- `platformTimerCreate( t )`
  Creates a timer that must expire in the given time (t) expressed in milliseconds (mandatory). Calculates a timer/tick/reference to a timeout that must be expired after the time defined, and returns this timer/tick/reference expressed as an unsigned 32-bit variable.
- `platformTimerIsExpired( timer )`
  Checks if the timer previously created is expired (mandatory). Receives the timer/tick/reference returned by `platformTimerCreate()`, and returns true if expired.
- `platformTimerDestroy( timer )`
  Stops and releases the given timer if previously created (optional). If not required, this macro must be left empty.

- `platformDelay( t )`
  Delays/blocks the MCU for the given time (t) in milliseconds (mandatory).
- `platformGetSysTick()`
  Returns the current tick counter in millisecods (optional). If not required, this macro must be left empty.
- `platformAssert()`
  Asserts whether the given expression is true. Logging, error handle or trap otherwise (optional). If not required, this macro must be left empty.
- `platformErrorHandle()`
  Global error handler or trap (optional). If not required, this macro must be left empty.
- `platformSpiSelect()`
  SPI SS/CS: Chip/Slave Select (mandatory).
- `platformSpiDeselect()`
  SPI SS/CS: Chip/Slave Deselect (mandatory).
- `platformSpiTxRx( txBuf, rxBuf, len )`
  SPI transceive (mandatory). Transmits/receives from/to the given buffers for the given amount of bytes (len). The SPI bus must be clocked for the given number of bytes (len). If transmission buffer (txBuff) equals NULL, only reception must be done, leading to 0s on the MOSI line, and the MISO bytes into rxBuff. If reception buffer (rxBuff) equals NULL, only transmission smust be done, leading to transmission on the MOSI line, and data from MISO being ignored/not placed into the rxBuff.
- `platformLog( ... )`
  Logs the given message (optional).

Some definitions and interfaces listed above are mandatory, and some are optional. Specific features/extensions are not required, such as IRQ/GPIO initialization called from RFAL itself, ST25R LED handling, global error handler/trap.

4.2 Library configurations

The RFAL offers a wide range of configurations with great flexibility. Unnecessary modules are disabled, global buffers sizes are tailored and certain specific behaviors are also tweaked. Regarding feature enable/disable there are the following user defined switches are available:

- `RFAL_FEATURE_NFCA` - RFAL support for NFC-A (ISO14443A)
- `RFAL_FEATURE_NFCB` - RFAL support for NFC-B (ISO14443B)
- `RFAL_FEATURE_NFCF` - RFAL support for NFC-F (FeliCa)
- `RFAL_FEATURE_NFCV` - RFAL support for NFC-V (ISO15693)
- `RFAL_FEATURE_T1T` - RFAL support for T1T (Topaz)
- `RFAL_FEATURE_T2T` - RFAL support for T2T (MIFARE ultralight)
- `RFAL_FEATURE_T4T` - RFAL support for T4T
- `RFAL_FEATURE_ST25TB` - RFAL support for ST25TB
- `RFAL_FEATURE_ST25xV` - RFAL support for ST25TV/ST25DV
- `RFAL_FEATURE_ISO_DEP` - RFAL support for ISO-DEP (ISO14443-4)
- `RFAL_FEATURE_ISO_DEP_POLL` - RFAL support for poller mode (PCD) ISO-DEP (ISO14443-4)
- `RFAL_FEATURE_ISO_DEP_LISTEN` - RFAL support for listen mode (PICC) ISO-DEP (ISO14443-4)
- `RFAL_FEATURE_NFC_DEP` - RFAL support for NFC-DEP (NFCIP1/P2P)
- `RFAL_FEATURE_LISTEN_MODE` - RFAL support for listen mode
- `RFAL_FEATURE_WAKEUP_MODE` - RFAL support for the wake-up mode
- `RFAL_FEATURE_DPO` - RFAL support for the DPO
- `RFAL_FEATURE_DLMA` - RFAL support for the Dynamic LMA

Concerning the memory resources it is possible to configure the maximum block size with the following options:

- `RFAL_FEATURE_ISO_DEP_IBLOCK_MAX_LEN` - ISO-DEP I-Block max length. Use values as defined by rfalIsoDepFSx.
- `RFAL_FEATURE_NFC_DEP_BLOCK_MAX_LEN` - NFC-DEP Block/Payload length. Allowed values: 64, 128, 192, 254.
- `RFAL_FEATURE_NFC_RF_BUF_LEN` - RF buffer length used by RFAL NFC layer.

- `RFAL_FEATURE_ISO_DEP_APDU_MAX_LEN` - ISO-DEP APDU max length. An APDU is composed by one or several chained ISO-DEP blocks.
- `RFAL_FEATURE_NFC_DEP_PDU_MAX_LEN` - NFC-DEP PDU maximum length. A PDU is composed by one or several chained NFC-DEP blocks.

Additionally, the user may customize certain RFAL internal behaviors by adding the following pre-processor instructions:

- `ST25R_COM_SINGLETXRX` – ST25R uses a single transceiver operation over the serial interface (SPI/I²C) (available for the ST25R3911 and ST25R3916). It requires an additional buffer (stretching to the maximum payload allowed) to compute the full command that then is used for a unique serial operation reducing its duration.
- `RFAL_COLRES_AGC` - Reduces bit collision recognition capabilities (available for the ST25R3911only). Helpful with certain devices using noisy active load modulation (for example, Samsung Galaxy S8-US (SM-G950U)) that is interpreted as two cards colliding: one weak and one strong.
- `ST25R_SELFTEST` - Enables ST25R. Host communication self-test during initialization. Helpful to diagnose porting issues.
- `ST25R_SELFTEST_TIMER` - Enable timer self-test during initialization. Helpful to diagnose porting issues.

To ensure that the compiled driver meets the user intended device, it is also required to add a pre-processor instruction defining the targeted ST25R device: ST25R3911B, ST25R3916, ST25R3916B, or ST25R95.

4.3 How to run the first example

The RFAL must be set up into an existing FW project. Step by step the instructions to follow:

1. Ensure that the system platform has the required peripherals properly configured and that it provides an HAL suitable for the requirements listed in Section 2.5.1 and Section 2.4.1. In case the host HAL does not provide an one to one interface add an adaption/glue layer on the application to meet the interfaces required and listed in Section 4.1.
2. Copy an existing file `rfal_platform` from a ST25R example into the project folder (available with all ST25R example projects, such as X-CUBE-NFC3, X-CUBE-NFC5, X-CUBE-NFC6, X-CUBE-NFC10, X-CUBE-NFC12, STSW-ST25R-LIB, within the Doxygen generated document as well as in Section Appendix A.
3. Configure the RFAL
   a. In `rfal_platform` define all interfaces required to run the project: GPIO, timer, serial interface
   b. Assess that features and protocols are needed to your specific project/application and buffer sizes. Enable/disable those features accordingly (`RFAL_FEATURE_...`)
4. In case memory optimizations are needed (the default values are typically OK) configure the buffer array sizes: `RFAL_FEATURE_ISO_DEP_IBLOCK_MAX_LEN`, `RFAL_FEATURE_NFC_DEP_BLOCK_MAX_LEN`, `RFAL_FEATURE_NFC_RF_BUF_LEN`, `RFAL_FEATURE_ISO_DEP_APDU_MAX_LEN`, `RFAL_FEATURE_NFC_DEP_PDU_MAX_LEN`
5. Add all `required .c` files to the project compilation from folders:
   a. `rfal/source`
   b. `rfal/source/st25r…`
6. Add the following include paths to the project:
   a. `rfal/include`
   b. `rfal/source`
   c. `rfal/source/st25r…`
7. Add preprocessor directive instructing which ST25R device is intended
8. Make sure that all require peripherals are properly initialized and their interfaces functional
9. Add the ST25R specific ISR method into the IRQ pin ISR handler, e.g `st25r3911Isr()`, `st25r3916Isr()`, etc.
10. Compile the project

11. If compilation was successful start by:
    In case the RFAL initialization does not execute successfully the following checks are performed:
    Verify that SPI is OK by observing the first SPI operation exchanges. A reset/set default command is initially issued followed by a read IC identity register (see the datasheet of the device). If the read out of the IC identity register is not as expected an error is reported.
    Additionally the self-diagnosis feature (`ST25R_SELFTEST`, `ST25R_SELFTEST_TIMER`) is used to understand which aspect of the required functionality is not operational during initialization
    If successful it is possible add the `rfalFieldOnAndStartGT()` after initialization and observe the emitted RF carrier being outputted by the antenna that is a good indication that the setup is OK.

5 FAQs

5.1 Detailed information of each RFAL API

Within every RFAL release, a document containing detailed information of every API is available in the documentation folder `doc/rfal.chm`. This is a Doxygen generated as the source code is commented in such format and provides extended insight of each API.

5.2 Calculating RFAL library footprint

The file `doc/rfal.chm` is available in the RFAL package documentation folder. It comes with the list of all RFAL objects/modules and their requirements, as detailed in the following tables.

Table 4. RFAL objects/modules

File name	text	data	bss	dec	hex
rfal_analogConfig.o	736	0	1036	1772	6EC
rfal_cd.o	1024	0	136	1160	488
rfal_cdHb.o	288	0	0	288	120
rfal_crc.o	48	0	0	48	30
rfal_dpo.o	644	0	30	680	2A8
rfal_iso15693_2.o	970	4	8	982	3D6
rfal_isoDep.o	5890	0	188	6078	17BE
rfal_nfc.o	5336	0	2402	7738	1E3A
rfal_nfcDep.o	4892	0	176	5068	13CC
rfal_nfca.o	2246	0	84	2330	91A
rfal_nfcb.o	1510	0	48	1558	616
rfal_nfcf.o	1380	0	336	1716	6B4
rfal_nfcv.o	2300	0	0	2300	8FC
rfal_st25tb.o	1012	0	0	1012	3F4
rfal_st25xv.o	1828	0	0	1828	724
rfal_t1t.o	336	0	0	336	150
rfal_t2t.o	320	0	0	320	140
rfal_t4t.o	664	0	0	664	298
Total	**31956**	**4**	**4486**	**36414**	**8E3E**

Table 5. RFAL objects/modules - ST25R3911B

File name	text	data	bss	dec	hex
rfal_rfst25r3911.o	9332	0	1072	10404	28A4
st25r3911.o	1900	0	4	1904	770
st25r3911_com.o	1482	0	0	1482	5CA
st25r3911_interrupt.o	832	0	20	852	354
Total	**13546**	**0**	**1096**	**14642**	**3932**

Table 6. **RFAL objects/modules - ST25R3916**

File name	text	data	bss	dec	hex
rfal_rfst25r3916.o	11526	0	1104	12630	3156
st25r3916.o	2010	0	4	2014	7DE
st25r3916_aat.o	798	0	0	798	31E
st25r3916_com.o	1278	0	0	1278	4FE
st25r3916_irq.o	650	0	16	666	29A
st25r3916_led.o	340	0	0	340	154
Total	**16602**	**0**	**1124**	**17726**	**453E**

Table 7. **RFAL objects/modules - ST25R95**

File name	text	data	bss	dec	hex
rfal_rfst25r95.o	4302	0	472	4774	12A6
st25r95.o	302	0	0	302	12E
st25r95_com.o	1178	112	0	1290	50A
st25r95_com_spi.o	1756	16	28	1800	708
Total	**7538**	**128**	**500**	**8166**	**1FE6**

Table 8. **RFAL objects/modules - ST25R200**

File name	text	data	bss	dec	hex
rfal_rfst25r200.o	6670	0	132	6802	1A92
st25r200.o	1560	0	4	1564	61C
st25r200_com.o	1402	0	0	1402	57A
st25r200_irq.o	670	0	36	686	2AE
Total	**10302**	**0**	**152**	**10454**	**28D6**

Table 9. **RFAL objects/modules - ST25R500**

File name	text	data	bss	dec	hex
rfal_rfst25r500.o	9500	0	524	10024	2728
st25r500.o	2304	0	4	2308	904
st25r500_com.o	1318	0	0	1318	526
st25r500_dpocr.o	1188	0	36	1224	4C8
st25r500_irq.o	670	0	16	686	2AE
Total	**14980**	**0**	**580**	**15560**	**3CC8**

The provided data are based on a compilation for STM32L4, with optimizations set at -Oz.

Each module is compiled with all functionalities included. Reductions are observed by disabling specific features of an individual module (detailed in Section 5.1). For example, if Listen mode is not needed, disabling `RFAL_FEATURE_ISO_DEP_LISTEN` and `RFAL_FEATURE_LISTEN_MODE` results in a smaller footprint of the ISO-DEP and RF modules.

Using these tables it is possible to make a rough footprint estimation of the targeted application.

5.3 Using a custom analog configuration table

The RFAL has a default analog configuration table for every supported device.

In case a customized user defined table is required, it is easily deployed.

Refer to Section 2.6.1: RF HAL.

5.4 Changing AM modulation index

Modulation type or index is regarded as an analog configuration and therefore held by on the analog configuration table. To change the modulation index in a particular technology or bitrate the user needs to apply/modify the modulation index at that particular mode. There are common and bitrate specific entries on the AC table and they are applied as described in Section 2.6.1.

To identify which setting to apply refer to the datasheet of the device.

5.5 Changing AM/OOK modulation type

Similarly to Section 5.4, the modulation scheme configuration is held by on the analog configuration table. To change the modulation index in a particular technology or bitrate the user needs to apply/modify the modulation type/index at that particular mode. There are common and bitrate specific entries on the AC table and they are applied as described in Section 2.6.1

To identify the setting to apply, refer to the datasheet.

Devices permit different modulation schemes (resistive and/or regulated) that implies different configurations and techniques to determine the correct settings for a particular antenna/board.

5.6 Usage of user defined data types

Although RFAL makes use of standard C types it does not enforce the usage standard libraries (`stdint`, `stdbool`, `stdlib`, etc) as it does not directly include them. All external library dependencies are gathered at the user defined `rfal_platform` allowing to user specific customizations. In case user defined types are required there is the option to not include C standard libraries and define these types at the application level. An example of such setup is available on the STM8 demonstration (STSW-STM8-NFC5).

5.7 Blocking vs. non-blocking APIs

Along the RFAL several APIs are provided in blocking and non-blocking forms.

The blocking methods are easy to use, result in more readable code, and in many applications are a good fit. However, for some operations blocking other processing can be unacceptable.

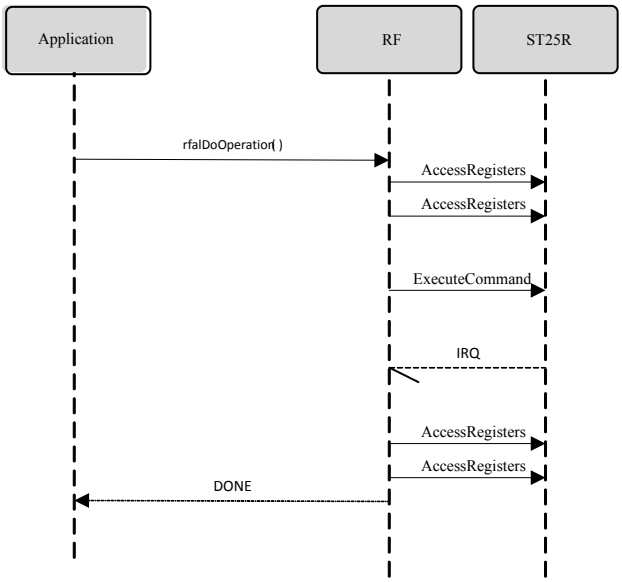
Therefore many APIs also have a non-blocking interface available in the form of `rfalStart`*`Operation`*`()` and `rfalGet`*`Operation`*`Status()`. This requires the caller to maintain context and be able to implement a more complex state machine to control the flow.

As highlighted in Section 2.6.4, there are two important aspects:

- After starting an operation with `rfalStart`*`Operation`*`()` the equivalent `rfalGet`*`Operation`*`Status()` is called/polled until the operation is completed that means its returned value is different than `ERR_BUSY`. Once the operation concluded and its outcome returned, the caller must handle its result immediately. Calling the `rfalGet`*`Operation`*`Status()` thereafter does not ensure that the previous outcome and context remain valid (apart from specific exceptions).
- Non-blocking APIs execute a procedure, they have defined start and end events. It is not allowed to start another RFAL operation while the previous one is ongoing. Doing so leads to unexpected behaviour and inconsistent state.

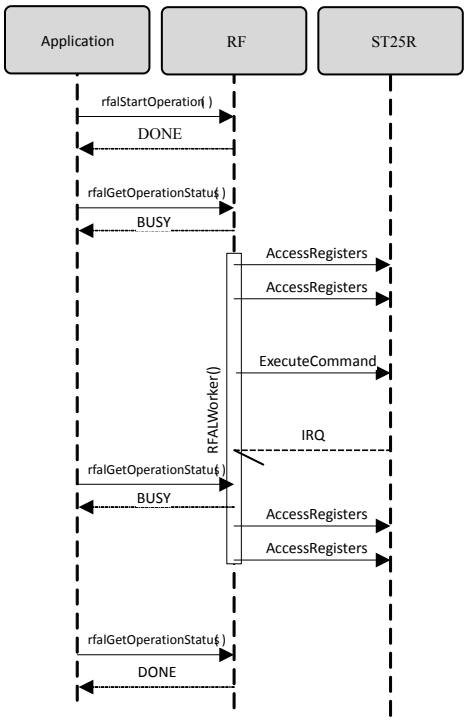
The following diagrams provide a better visualization of these two approaches. When an API is called using its blocking interface, it hands over control to the called RFAL method that executes a determined procedure which takes a certain duration. Internally its duration is affected by external events and configurations, such as serial interface speed. Only after the sequence has completed the control returns to the caller/application. Some blocking APIs may call the `rfalWorker()` internally if this is needed for their operation.

Figure 9. **Blocking API**



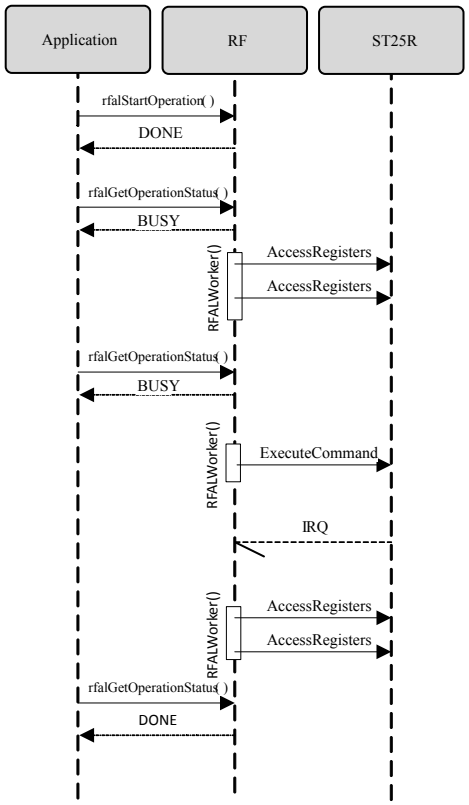
When using non-blocking APIs the flow is quite different. The call that triggers an operation promptly returns control to the caller, which, from this point on, can poll to check if the operation is concluded. The `rfalWorker()` needs to run autonomously performing the requested task.

Figure 10. **Non-blocking API**



In a single thread environment these are actually executed sequentially as the application alternates from executing the `rfalWorker()` and returns to the context that waits the operation completion and polls it. The following figure shows that the activity is executed only when the `rfalWorker()` is run (and of course by the ISR).

Figure 11. Non-blocking API: rfalWorker() detailed



Note: It is recommended to make use of the non-blocking APIs as much as possible while integrating the RFAL into a final project/product. Due to simplicity, code readability, prototyping purposes, and single task application (no parallel/concurrent peripherals) some of the demonstration projects use blocking APIs. Each SW developer must assess if such usage is acceptable for the targeted product and system architecture.

5.8 HW/SW controlled SPI chip select

The RFAL allows for SW and HW controlled SPI chip selection. The recommendation is to make use of SW controlled as it grants more flexibility, alleviates memory management as operations can be split in chunks, and allows to enforce a defined state at initialization.

SW controlled CS/SS may be slower (depending on the platform used) than when handled by the SPI HW block/peripheral itself, and in certain specific applications optimizations are needed. To use HW controlled SPI CS/SS one needs to configure the SPI peripheral accordingly, set `ST25R_COM_SINGLETXRX` configuration and clear the APIs: `platformSpiSelect()` and `platformSpiDeselect()`.

Note: Not all SPI HW blocks/peripherals fully support the required operation when HW controlled chip select as they may automatically select/deselect the CS/SS line after a defined number of bits (e.g. 16-bit). Transmission/reception of an arbitrary number of bytes is required in a single SPI operation for example for write/read of the FIFO.

5.9 Loop within ST25R ISR

ST25R ISRs contains a while loop, which may look unexpected. The reason for such is that not all hosts support level triggered IRQs, only edge-triggered.

The problem arises when an a new IRQ occurs while the ISR from a previous event is ongoing.

Figure 12. Edge-triggered ISR: OK

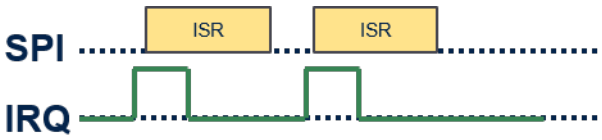
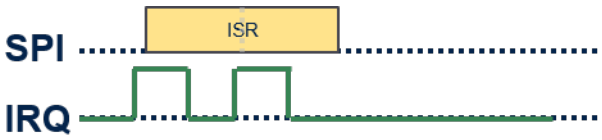


Figure 13. Edge-triggered ISR: fail

In Figure 13 an edge-triggered IRQ is capable of handling multiple interrupts if they are sufficiently apart, so that the ISR has already concluded. This is not always the case: there is the possibility for an IRQ event to occur while the ISR is still running and retrieving the information from IRQ status registers. When this happens, the IRQ goes back to high and remains as the ISR is never triggered again due to the fact that an edge/transition is not observed. This stalls the driver because the new event has not been processed, and upcoming ones are not detected.

To avoid that the IRQ pin is checked at the end of the ISR procedure verifying if a pending event has occurred. If so,the procedure is run again, retrieving the latest information.

Figure 14. Edge-triggered ISR: loop



Such check/loop is not required if the IRQ is configured as level-triggered and can be removed. As such handling causes no harm and for the sake of interoperability the RFAL provides it by default.

5.10 Debugging tools

In case of issues, it is useful to collect a serial interface trace (SPI/I²C/UART) and observe the traffic host AFE. While collecting host AFE traces other relevant signals, such as IRQ, must be included. A great majority of problems are identified and addressed when such traces are available, and the support becomes significantly simpler and faster to provide. To obtain such, it is recommended to use a logic/digital analyzer. An NFC/RF sniffer is also very useful. The downside is that this is a rather specific and somewhat costly tool, not commonly available.

6 Additional references

ID	Reference
[1]	NFC Forum - Activity Technical Specification v2.1
[2]	NFC Forum - Digital Technical Specification v2.2
[3]	NFC Forum – NFC Controller Interface Technical Specification v2.1
[4]	EMV Contactless Interface Specification v3.1

Appendix A rfal_platform.h example

```
/*Example of system platform definitions for a STM32 project*/
#ifndef RFAL_PLATFORM
#define RFAL_PLATFORM
/*
******************************************************************************
* INCLUDES
******************************************************************************
*/
#include <stdint.h>                                             /* Include type definitions
                            */
#include <stdlib.h>
#include <stdbool.h>
#include <limits.h>
#include "stm32l4xx_hal.h"                                      /* Include the platform HAL
                            */
#include "spi.h"                                                /* Include any glue layer need
ed for interfacing with HAL */
#include "timer.h"
#include "main.h"                                               /* Include PIN definitions (ST
M32CubeMx places in main.h) */
/*
******************************************************************************
* GLOBAL DEFINES
******************************************************************************
*/

#define ST25R_COM_SINGLETXRX                                    /*!< Use single Transceive
                         */

#define ST25R_SS_PIN                SPI1_CS_Pin                 /*!< GPIO pin used for ST25R S
PI SS                */
#define ST25R_SS_PORT               SPI1_CS_GPIO_Port           /*!< GPIO port used for ST25R
SPI SS port          */

#define ST25R_INT_PIN               IRQ_391x_Pin                /*!< GPIO pin used for ST25R E
xternal Interrupt    */
#define ST25R_INT_PORT              IRQ_391x_GPIO_Port          /*!< GPIO port used for ST25R
External Interrupt   */

#define ST25R_RESET_PIN             RST_Pin                     /*!< GPIO pin used for ST25R R
eset    */
#define ST25R_INT_PORT              RST_GPIO_Port               /*!< GPIO port used for ST25R
Reset   */

#ifdef LED_FIELD_Pin
#define PLATFORM_LED_FIELD_PIN      LED_FIELD_Pin               /*!< GPIO pin used as field LE
D                        */
#endif

#ifdef LED_FIELD_GPIO_Port
#define PLATFORM_LED_FIELD_PORT     LED_FIELD_GPIO_Port         /*!< GPIO port used as field L
ED                       */
#endif

#ifdef LED_RX_Pin
#define PLATFORM_LED_RX_PIN         LED_RX_Pin                  /*!< GPIO pin used as field LE
D                        */
#endif

#ifdef LED_RX_GPIO_Port
#define PLATFORM_LED_RX_PORT        LED_RX_GPIO_Port            /*!< GPIO port used as field L
ED                       */
#endif


/*
******************************************************************************
```

```
* GLOBAL MACROS
******************************************************************************
*/
#define platformProtectST25RComm()                    do{ globalCommProtectCnt++; __DSB();NVI
C_DisableIRQ(EXTI0_IRQn);__DSB();__ISB();}while(0) /*!< Protect unique access to ST25R commun
ication channel - IRQ disable on single thread environment (MCU) ; Mutex lock on a multi thre
ad environment      */
#define platformUnprotectST25RComm()                  do{ if (--globalCommProtectCnt==0) {NVI
C_EnableIRQ(EXTI0_IRQn);} }while(0)                /*!< Unprotect unique access to ST25R comm
unication channel - IRQ enable on a single thread environment (MCU) ; Mutex unlock on a multi
 thread environment */

#define platformProtectST25RIrqStatus()               platformProtectST25RComm()
                  /*!< Protect unique access to IRQ status var - IRQ disable on single thread
 environment (MCU) ; Mutex lock on a multi thread environment */
#define platformUnprotectST25RIrqStatus()             platformUnprotectST25RComm()
                  /*!< Unprotect the IRQ status var - IRQ enable on a single thread environme
nt (MCU) ; Mutex unlock on a multi thread environment         */


#define platformLedOff( port, pin )                   platformGpioClear((port), (pin))
                  /*!< Turns the given LED Off                    */
#define platformLedOn( port, pin )                    platformGpioSet((port), (pin))
                  /*!< Turns the given LED On                     */
#define platformLedToogle( port, pin )                platformGpioToogle((port), (pin))
                  /*!< Toogle the given LED                       */

#define platformGpioSet( port, pin )                  HAL_GPIO_WritePin(port, pin, GPIO_PIN_S
ET)               /*!< Turns the given GPIO High                  */
#define platformGpioClear( port, pin )                HAL_GPIO_WritePin(port, pin, GPIO_PIN_R
ESET)             /*!< Turns the given GPIO Low                   */
#define platformGpioToogle( port, pin )               HAL_GPIO_TogglePin(port, pin)
                  /*!< Toogles the given GPIO                     */
#define platformGpioIsHigh( port, pin )               (HAL_GPIO_ReadPin(port, pin) == GPIO_PI
N_SET)            /*!< Checks if the given LED is High            */
#define platformGpioIsLow( port, pin )                (!platformGpioIsHigh(port, pin))
                  /*!< Checks if the given LED is Low             */

#define platformTimerCreate( t )                      timerCalculateTimer(t)
                  /*!< Create a timer with the given time (ms)    */
#define platformTimerIsExpired( timer )               timerIsExpired(timer)
                  /*!< Checks if the given timer is expired       */
#define platformDelay( t )                            HAL_Delay( t )
                  /*!< Performs a delay for the given time (ms)   */

#define platformGetSysTick()                          HAL_GetTick()
                  /*!< Get System Tick (1 tick = 1 ms)            */

#define platformAssert( exp )                         assert_param( exp )
                  /*!< Asserts whether the given expression is true*/
#define platformErrorHandle()                         _Error_Handler(__FILE__, __LINE__)
                  /*!< Global error handle\trap                   */

#define platformSpiSelect()                           platformGpioClear( ST25R_SS_PORT, ST25R
_SS_PIN )         /*!< SPI SS\CS: Chip|Slave Select               */
#define platformSpiDeselect()                         platformGpioSet( ST25R_SS_PORT, ST25R_S
S_PIN )           /*!< SPI SS\CS: Chip|Slave Deselect             */
#define platformSpiTxRx( txBuf, rxBuf, len )          spiTxRx(txBuf, rxBuf, len)
                  /*!< SPI transceive                             */


#define platformI2CTx( txBuf, len, last, txOnly )
                  /*!< I2C Transmit                               */
#define platformI2CRx( txBuf, len )
                  /*!< I2C Receive                                */
#define platformI2CStart()
                  /*!< I2C Start condition                        */
#define platformI2CStop()
                  /*!< I2C Stop condition                         */
#define platformI2CRepeatStart()
```

```
                  /*!< I2C Repeat Start                                */
#define platformI2CSlaveAddrWR(add)
                  /*!< I2C Slave address for Write operation       */
#define platformI2CSlaveAddrRD(add)
                  /*!< I2C Slave address for Read operation        */

#define platformLog(...)
                  /*!< Log  method                                 */

/*
******************************************************************************
* GLOBAL VARIABLES
******************************************************************************
*/
extern uint8_t globalCommProtectCnt;                      /* Global Protection Counter provid
ed per platform - instantiated in main.c    */

/*
******************************************************************************
* RFAL FEATURES CONFIGURATION
******************************************************************************
*/

#define RFAL_FEATURE_LISTEN_MODE              true       /*!< Enable/Disable RFAL support fo
r Listen Mode                          */
#define RFAL_FEATURE_WAKEUP_MODE              true       /*!< Enable/Disable RFAL support fo
r the Wake-Up mode                     */
#define RFAL_FEATURE_LOWPOWER_MODE            true       /*!< Enable/Disable RFAL support fo
r the Low Power mode                   */
#define RFAL_FEATURE_NFCA                     true       /*!< Enable/Disable RFAL support fo
r NFC-A (ISO14443A)                    */
#define RFAL_FEATURE_NFCB                     true       /*!< Enable/Disable RFAL support fo
r NFC-B (ISO14443B)                    */
#define RFAL_FEATURE_NFCF                     true       /*!< Enable/Disable RFAL support fo
r NFC-F (FeliCa)                       */
#define RFAL_FEATURE_NFCV                     true       /*!< Enable/Disable RFAL support fo
r NFC-V (ISO15693)                     */
#define RFAL_FEATURE_T1T                      true       /*!< Enable/Disable RFAL support fo
r T1T (Topaz)                          */
#define RFAL_FEATURE_T2T                      true       /*!< Enable/Disable RFAL support fo
r T2T (MIFARE Ultralight)              */
#define RFAL_FEATURE_T4T                      true       /*!< Enable/Disable RFAL support fo
r T4T                                  */
#define RFAL_FEATURE_ST25TB                   true       /*!< Enable/Disable RFAL support fo
r ST25TB                               */
#define RFAL_FEATURE_ST25xV                   true       /*!< Enable/Disable RFAL support fo
r ST25TV/ST25DV                        */
#define RFAL_FEATURE_DYNAMIC_ANALOG_CONFIG    false      /*!< Enable/Disable Analog Configs
to be dynamically updated (RAM)        */
#define RFAL_FEATURE_DPO                      false      /*!< Enable/Disable RFAL Dynamic Po
wer Output upport                      */
#define RFAL_FEATURE_ISO_DEP                  true       /*!< Enable/Disable RFAL support fo
r ISO-DEP (ISO14443-4)                 */
#define RFAL_FEATURE_ISO_DEP_POLL             true       /*!< Enable/Disable RFAL support fo
r Poller mode (PCD) ISO-DEP (ISO14443-4)    */
#define RFAL_FEATURE_ISO_DEP_LISTEN           true       /*!< Enable/Disable RFAL support fo
r Listen mode (PICC) ISO-DEP (ISO14443-4)   */
#define RFAL_FEATURE_NFC_DEP                  true       /*!< Enable/Disable RFAL support fo
r NFC-DEP (NFCIP1/P2P)                 */


#define RFAL_FEATURE_ISO_DEP_IBLOCK_MAX_LEN   256U       /*!< ISO-DEP I-Block max length. Pl
ease use values as defined by rfalIsoDepFSx */
#define RFAL_FEATURE_NFC_DEP_BLOCK_MAX_LEN    254U       /*!< NFC-DEP Block/Payload length.
Allowed values: 64, 128, 192, 254      */
#define RFAL_FEATURE_NFC_RF_BUF_LEN           256U       /*!< RF buffer length used by RFAL
NFC layer                              */

#define RFAL_FEATURE_ISO_DEP_APDU_MAX_LEN     512U       /*!< ISO-DEP APDU max length.
                                       */
```

```
#define RFAL_FEATURE_NFC_DEP_PDU_MAX_LEN       512U       /*!< NFC-DEP PDU max length.
                                                */




/*
 ******************************************************************************
 * RFAL OPTIONAL MACROS           (Do not change)
 ******************************************************************************
 */

#ifndef platformProtectST25RIrqStatus
#define platformProtectST25RIrqStatus()            /*!< Protect unique access to IRQ status v
ar - IRQ disable on single thread environment (MCU) ; Mutex lock on a multi thread environmen
t */
#endif /* platformProtectST25RIrqStatus */

#ifndef platformUnprotectST25RIrqStatus
#define platformUnprotectST25RIrqStatus()          /*!< Unprotect the IRQ status var - IRQ en
able on a single thread environment (MCU) ; Mutex unlock on a multi thread environment
  */
#endif /* platformUnprotectST25RIrqStatus */

#ifndef platformProtectWorker
#define platformProtectWorker()                    /* Protect RFAL Worker/Task/Process from c
oncurrent execution on multi thread platforms   */
#endif /* platformProtectWorker */

#ifndef platformUnprotectWorker
#define platformUnprotectWorker()                  /* Unprotect RFAL Worker/Task/Process from
 concurrent execution on multi thread platforms */
#endif /* platformUnprotectWorker */

#ifndef platformIrqST25RPinInitialize
#define platformIrqST25RPinInitialize()            /*!< Initializes ST25R IRQ pin
        */
#endif /* platformIrqST25RPinInitialize */

#ifndef platformIrqST25RSetCallback
#define platformIrqST25RSetCallback( cb )          /*!< Sets ST25R ISR callback
        */
#endif /* platformIrqST25RSetCallback */

#ifndef platformLedsInitialize
#define platformLedsInitialize()                   /*!< Initializes the pins used as LEDs to
outputs */
#endif /* platformLedsInitialize */

#ifndef platformLedOff
#define platformLedOff( port, pin )                /*!< Turns the given LED Off
        */
#endif /* platformLedOff */


#ifndef platformLedOn

#define platformLedOn( port, pin )                 /*!< Turns the given LED On
        */
#endif /* platformLedOn */


#ifndef platformLedToogle

#define platformLedToggle( port, pin )             /*!< Toggles the given LED
        */
#endif /* platformLedToogle */
```

```
#ifndef platformGetSysTick

#define platformGetSysTick()                        /*!< Get System Tick (1 tick = 1 ms)
        */
#endif /* platformGetSysTick */


#ifndef platformTimerDestroy

#define platformTimerDestroy( timer )               /*!< Stops and released the given timer
        */
#endif /* platformTimerDestroy */

#ifndef platformAssert

#define platformAssert( exp )                       /*!< Asserts whether the given expression
is true */
#endif /* platformAssert */

#ifndef platformErrorHandle

#define platformErrorHandle()                       /*!< Global error handler or trap
        */
#endif /* platformErrorHandle */


#endif /* RFAL_PLATFORM */
```

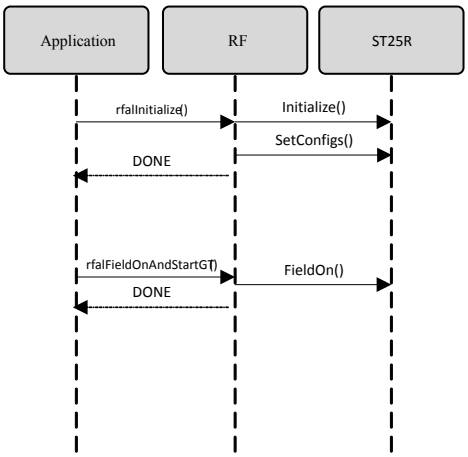
Appendix B Sequence diagrams

This section contains sequence diagrams provided to better visualize how the different modules play a role in specific tasks.

For the sake of simplicity and readability, and due to the fact that certain behaviors are device specific, not all events (such as individual interrupts, registers accesses) are considered, but the action performed is represented. Also, the full names of some APIs are too long, and may be shortened for representation purposes, as an example `rfalNfcaPollerFullCollisionResolution()`…`NfcaFullCollisionResolution()` ; `rfalIsoDepStartApduTransceive()` changes into …`StartApduTransceive()`….

RFAL initialization and RF carrier on

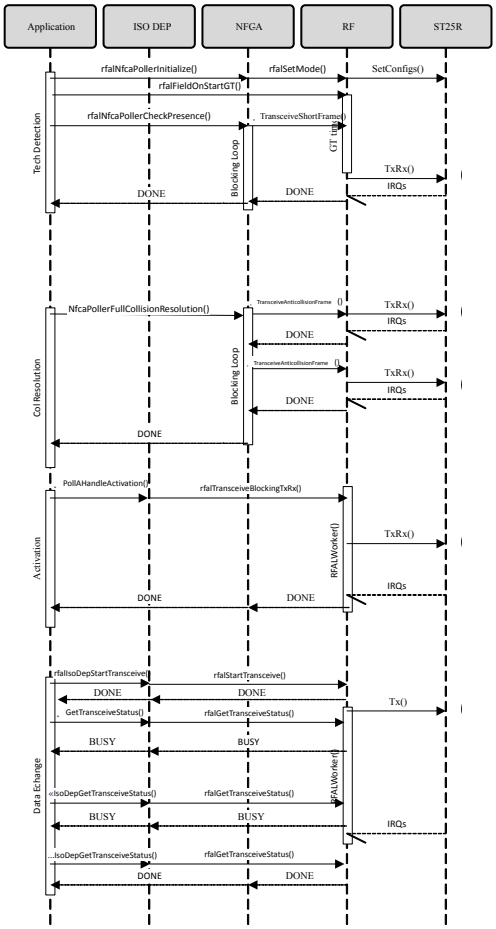
Figure 15. RFAL initialization and RF carrier on



In this simple example, often used for checking the basic functionality of a new HW setup, the application layer initializes the RFAL which in turn initializes the NFC AFE by performing a reset sequence, loading initial configurations and setting it into a defined state. Then field RF carrier is turned on by calling the `rfalFieldOnAndStartGT()`.

NFC-A / ISO14443A blocking activation and data exchange

Figure 16. NFC-A / ISO14443A blocking activation and data exchange



Here the application activates an NFC-A T4T device and exchanges some data. In this example the application makes direct use of the APIs available on NFC-A and ISO-DEP modules, and uses the blocking APIs for the whole card activation process. It starts by initializing for NFC-A communication and enabling the field (RF carrier), which starts the configured GT timer. Once the presence check/technology detection is executed, it waits until the GT requirement is fulfilled (if not yet expired) and then transmit the polling command (`SENS_REQ/REQA`). As a card(s) have been detected, the application goes into collision resolution calling `rfalNfcaPollerFullCollisionResolution()`, a blocking API where some anticollision commands are exchanged (actual number depends on several factors, such as the number of cards present and their NFCID).

When all Listeners/cards have been identified, the application can activate the one it wants to select (in case of multiple, not shown here), and must activate one of the supported protocols (in this example ISO-DEP). To activate the card in ISO14443-4, the API `rfalIsoDepPollAHandleActivation()` is used, it handles the whole layer four activation procedures. This method exchanges one or more commands with the Listener device. It must be highlighted that during this process longer timeouts/FWT are applied (~5 ms), the whole activation may amount to several milliseconds. After activation, data exchange can flow naturally on top of the ISO-DEP protocol. To achieve this the application performs ISO-DEP transceive operations (block or APDU), which internally handle the protocol and trigger the lower level RF transceive process for the actual transmission and reception.

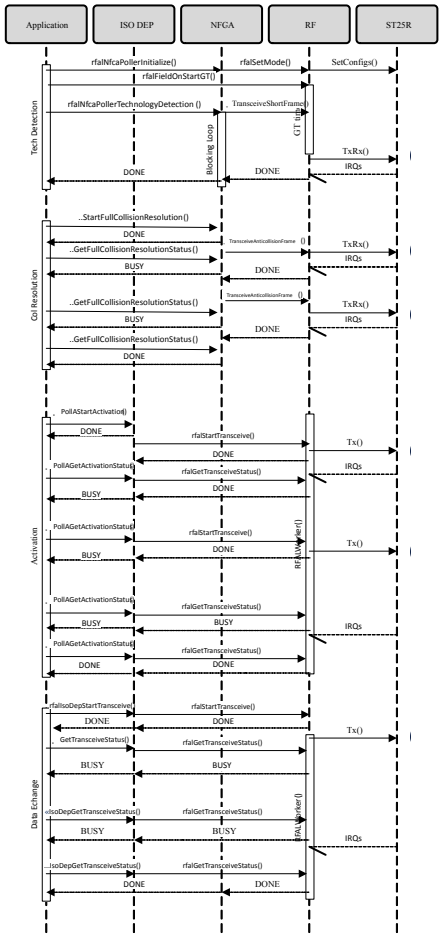
During protocol data exchange (ISO-DEP and NFC-DEP) the FWT can be quite long (up to 5 seconds), and when combined with waiting time extensions and repetitions one single protocol transceive operation can take several seconds. Due to this reason, RFAL does not provide (nor it is recommended) to perform blocking protocol operations unless such long execution does not pose a difficulty to a particular application flow. On the sequence above, after an initial `rfalIsoDepStartTransceive()` the method `rfalIsoDepGetTransceiveStatus()` must be polled until the transceive operation is completed (not `ERR_BUSY`). The actual transmission and reception is handled by the RF module in parallel via the `rfalWorker()`, and, once completed, the outcome is stored until the `rfalIsoDepGetTransceiveStatus()` is called again.

Important:

- *The `rfalGetOperationStatus()` must be called/polled until the operation is completed, which means that its returned value is different from `ERR_BUSY`. Once the operation is concluded and its outcome returned, the caller must handle its result immediately. Calling the `rfalGetOperationStatus()` thereafter does not ensure that the previous outcome and context remain valid (apart from specific exceptions)*
- *Non-blocking APIs execute a certain procedure and have a defined start and end event. It is not allowed to start another RFAL operation while a previous is ongoing. Doing so may lead to unexpected behaviour and inconsistent state.*

NFC-A / ISO14443A activation and data exchange

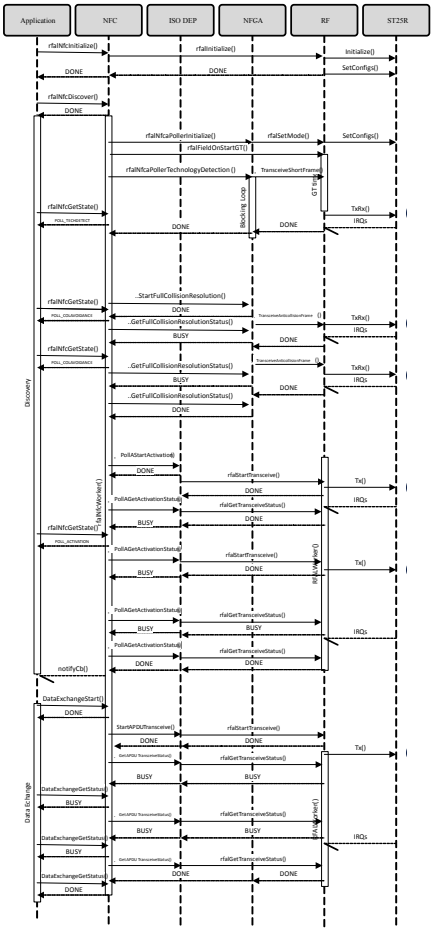
Figure 17. NFC-A / ISO14443A activation and data exchange



Here the same card activation and data exchange shown in the previous section are achieved. The main difference is that non-blocking APIs are used instead of the blocking ones during activation process. As usual the non-blocking APIs initiate a procedure that must be polled until it has been completed, which now are also used collision resolution and protocol activation. Another minor difference from the previous example is that during protocol activation two frames are exchanged (for example RATS and PPS), showing that the protocol activation may trigger one or more RF transceive operations within a single protocol activation operation.

NFC-A / ISO14443A activation and data exchange using RFAL HL

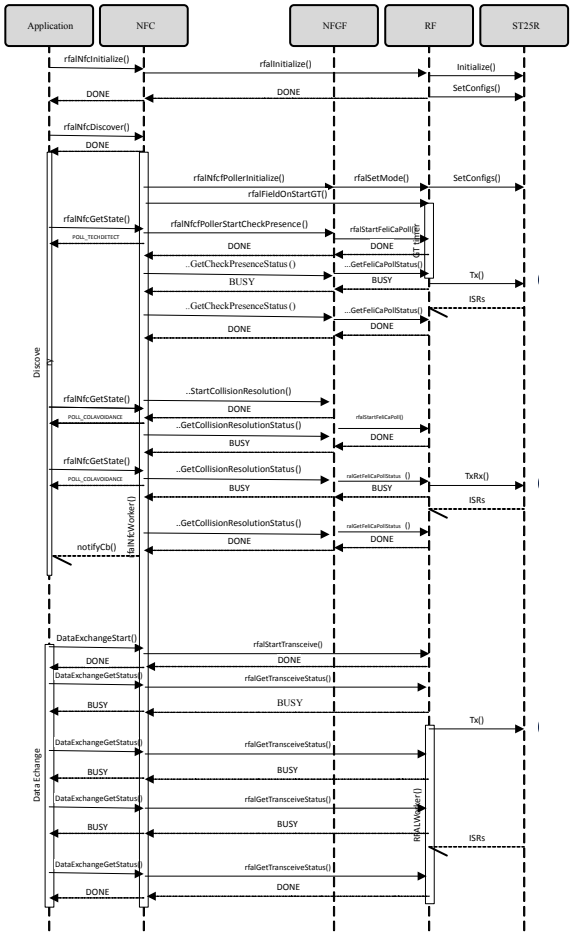
Figure 18. NFC-A / ISO14443A activation and data exchange using RFAL HL



The same card activation and data exchange shown before are achieved. The main difference is that the NFC module on RFAL HL is used. This module conveniently handles all different activities/procedures with minimal user/application interaction. User simply needs to initialize the RFAL, start the discovery process, and exchange the desired data when a device is activated. The whole Listener/card discovery process (technology detection, collision resolution and activation) is handled internally and the application monitors the present state by polling the current state via `rfalNfcGetState()` or by registering with the optional notification callback (notifyCb). The NFC module notifies noteworthy events/states as it evolves. Similarly to the previous examples, once Listener has been activated, data exchange is performed via tranceive operations. In this case using the RFAL HL - NFC module can be achieved by the API pair `rfalNfcDataExchangeStart()` and `GetDataExchangeStatus()`.

NFC-V/FeliCa activation and data exchange using RFAL HL

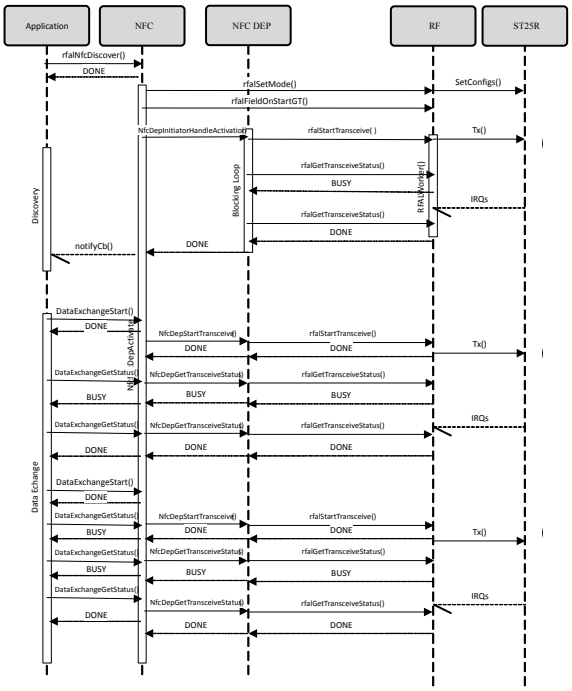
Figure 19. NFC-V/FeliCa activation and data exchange using RFAL HL



In this example a sequence where activation and data exchange is shown again. The main difference in this example is that the Listener is an NFC-F T3T that does not use a data exchange protocol. Therefore no protocol activation is performed, and the RF interface is used for data exchange. An analogous flow takes place when dealing with an NFC-V T5T.

AP2P initiator activation and data exchange using RFAL HL

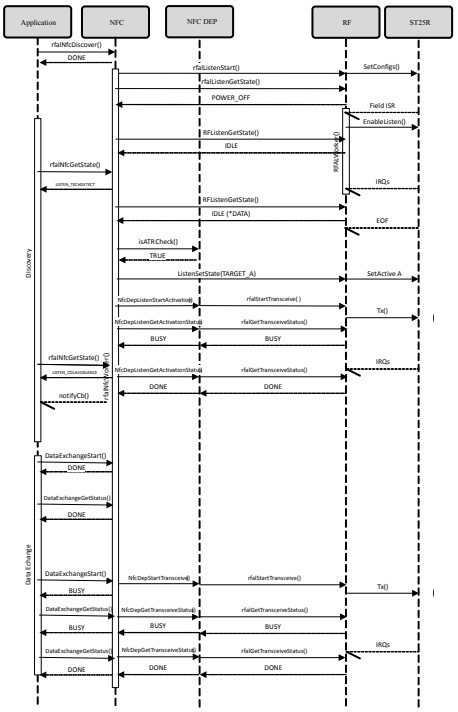
Figure 20. AP2P initiator activation and data exchange using RFAL HL



As in AP2P, there is no collision resolution phase (apart from RF collision avoidance procedure), the Initiator transmits the protocol activation command (ATR_REQ). Once activation is completed, it goes into data exchange and performs a series transceive operations.

AP2P target activation and data exchange using RFAL HL

Figure 21. AP2P target activation and data exchange using RFAL HL

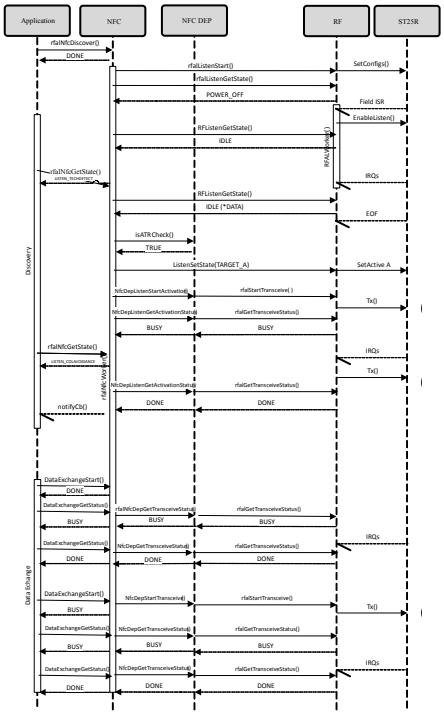


Here we can observe an activation and following data exchange when in listen mode. As before in AP2P there is no collision resolution phase (apart from RF collision avoidance procedure) and the Initiator goes straight to transmitting the protocol activation command (`ATR_REQ`). As a Listener this frame is received while the generic listen mode handler in RF layer which output the payload received. This payload is then verified by the protocol layer (NFC-DEP in this case), and if as expected the protocol activation handler is executed which provides an answer to the `ATR_REQ` received. In case the frame received is already a data packet (no parameter change `PSL_REQ`) the activation is terminated and data exchange is entered.

Important to highlight that in Isten mode if a data packet is received during activation, it is then retrieved by the first time the data exchange / transceive methods are called. In the sequence above one can observe that the first time `rfalNfcDataExchangeStart()` / `rfalNfcDataExchangeGetStatus()` are called no actual exchange takes place. Instead the data received from the Poller/Initiator is immediately handed over to the caller for processing.

AP2P target activation (including PSL_REQ) and data exchange using RFAL HL

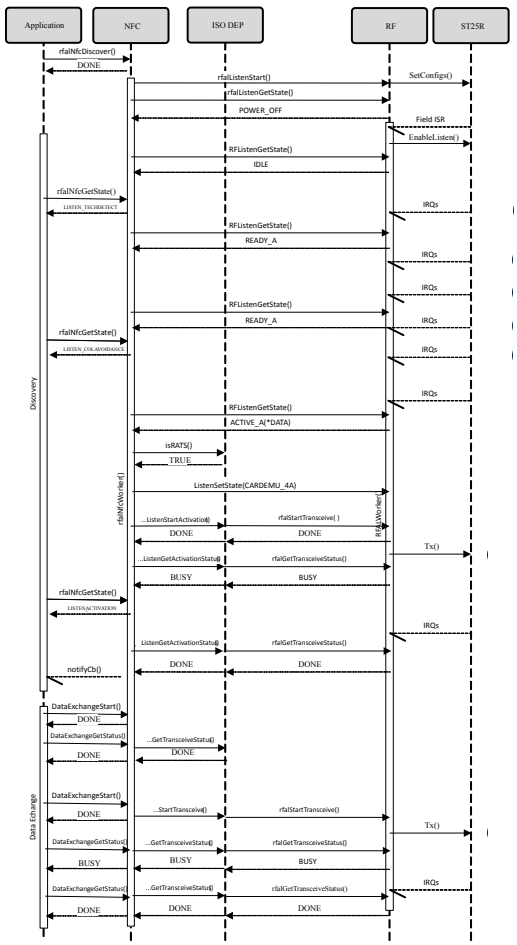
Figure 22. Additional frame (PSL_REQ)



This example is the same as the previous with one additional frame (PSL_REQ) during activation. The PSL allows to modify some parameters of the upcoming communication, and once acknowledged the activation procedure is completed moving into data exchange. This therefore results in a different flow as the data is not readily available on the first time `rfalNfcDataExchangeStart() / rfalNfcDataExchangeGetStatus()` are called.

NFC-A / ISO14443 Listener activation and data exchange using RFAL HL

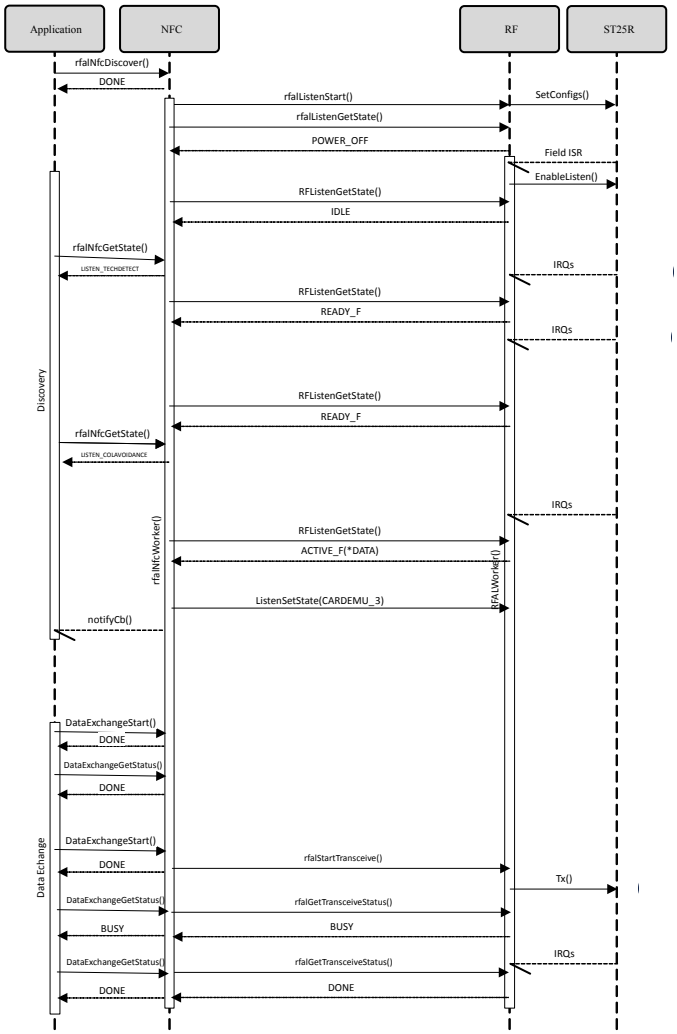
Figure 23. NFC-A / ISO14443 Listener activation and data exchange using RFAL HL



In this example the ST25R is activated as a Listener device in T4T CE mode. Once the listen mode is started, the anticollision sequence is handled, and the device is selected, the `ACTIVE_A` state is reached. Once the following command is received (RATS), the payload is handed over so that the protocol layer (ISO-DEP) checks its validity. The protocol activation is started responding to the RATS. Afterwards, a data packets is received, which concludes the protocol activation (no PPS). As before, received data are passed on the first call of the data exchange to be handled by the application. Then other data exchanges can take place normally.

NFC-F / FeliCa Listener activation and data exchange using RFAL HL

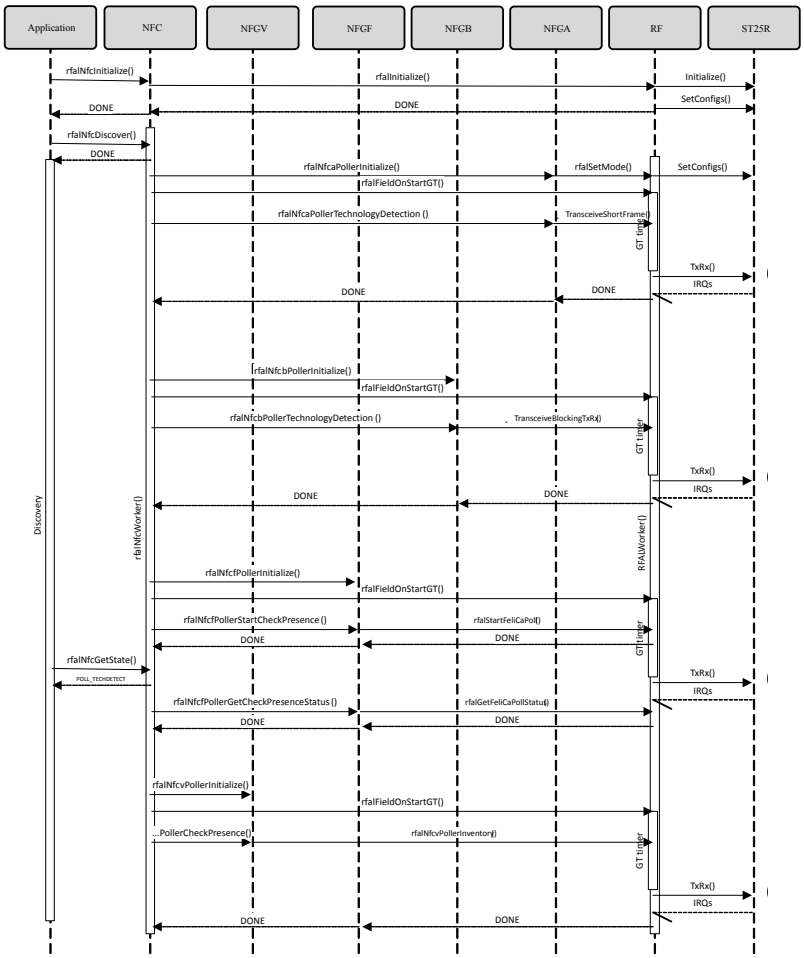
Figure 24. **NFC-F / FeliCa Listener activation and data exchange using RFAL HL**



In this example the ST25R is activated as a Listener device in T3T CE mode. It is the same as the previous flow T4T flow, without the protocol activation, as T3T does not require a particular data exchange protocol.

Polling cycle using RFAL HL

Figure 25. Polling cycle using RFAL HL



Here the NFC module is configured to poll for NFC-A, NFC-B, NFC-F, and NFC-V. In this case no card is presented and the different technologies are polled with timeout error. Once all technologies have been executed, the module waits to complete the request total duration (if not expired) and poll again in the same manner.

Revision history

Table 10. **Document revision history**

Date	Version	Changes
21-Oct-2021	1	Initial release.
25-Jul-2022	2	Updated: • Section 2.1: Features overview • Section 4.2: Library configurations
20-Jun-2024	3	Updated • Section 2.1: Features overview • Section 2.5.1: Dependencies • Figure 1 • Figure 4
20-Aug-2024	4	Updated Table 1. Applicable products. Updated Figure 4. NFC module state diagram. Minor text edits across the whole document.
04-Mar-2025	5	Updated Table 1. Applicable products, Table 2. Acronyms, Table 3. Supported modes, and tables in Section 5.2: Calculating RFAL library footprint. Updated Section 2.6.1: RF HALRF HAL, Section 3.2: Quality criteria, Section 4.3: How to run the first example, Section 4.1: Peripherals initialization and configuration, Section 4.2: Library configurations , Section 5.2: Calculating RFAL library footprint, and Appendix A: rfal_platform example. Updated Figure 1. RFAL stack architecture and Figure 8. RFAL package structure. Minor text edits across the whole document.

Contents

List of tables

List of figures