UFR Series NFC reader's API reference

This document applies to Digital Logic's uFR Series readers only.

For more information, please visit http://www.d-logic.net/nfc-rfid-reader-sdk/

The scope of this document is to give a better insight and provide easy start with uFR Series NFC readers.

uFR Series readers communicate with host via built in FTDI's USB to Serial interface chip.

If you have uFR Series reader with RS232 interface, please refer to "Communication protocol - uFR Series" document at our download section.

We provide dynamic libraries for all major OS: Win x86, Win X86_64, Linux x86, Linux x86_64, Linux ARM (and ARM HF with hardware float) and Mac OS X.

Our dynamic libraries rely on FTDI D2XX direct drivers. Most of them are already built in at today's modern OS. However, we always suggest to perform clean driver installation procedure by downloading and installing drivers from FTDI's download webpage.

Android platform is supported through FTDI's Java D2XX driver. Since this approach introduces new Java class, it shall be a scope of separate document.

Important update:

From library version 4.01 and up, it is possible to establish communication with reader without using FTDI's D2XX driver by calling **ReaderOpenEx** function. Library can talk to reader via COM port (physical or virtual) without implementing FTDI's calls. However, this approach is not fast as with use of D2XX drivers but gives much more flexibility to users who had to use COM protocol only, now they can use whole API set of functions via COM port.

Library naming convention

Dynamic libraries names are built upon following convention:

- Library always have "uFCoder" in its name as mandatory
- Prefix "lib" according to platform demands
- Suffix with architecture description
- Extension according to platform demands

Our standard library pack contains following libraries:

- libuFCoder-arm.so – for Linux on ARM platforms with software float
- libuFCoder-armhf.so - for Linux on ARM platforms with hardware float
- libuFCoder-x86.so – for Linux on Intel 32 bit platforms
- libuFCoder-x86_64.so - for Linux on Intel 64 bit platforms
- uFCoder-x86.dll – for Windows 32 bit
- uFCoder-x86_64.dll – for Windows 64 bit
- libuFCoder.dylib – for all OS X Intel based versions

Update policy: we release updated firmware and libraries frequently, with minor & major updates, bug-fixes, new features etc. All libraries mentioned above are affected with each update. Updates are absolutely free and can be obtained from our download page at "Libraries" section, while firmware updates are available at "Firmware" section by using software tool specially designed for that purpose. Library update package always have the following directory structure:

- "include" - contains "uFCoder.h" header file
- "linux" – contains directories "arm", "armhf", "x86" with appropriate libraries
- "osx" – contains library for OSX
- "windows" – contains libraries for Windows

and appropriate README file with short description of current revision.

Some considerations regarding platform specifics

Because FTDI driver is mandatory, proper installation method must be followed. See appendix for FTDI troubleshooting for details**.**

Reader's firmware and library functions relation

When you call library function, in most cases you are issuing protocol command to reader firmware. Library functions are usually wrapped firmware commands. This approach is very convenient for rapid application development and as time saving feature. Particularly, library function does the following:

- Check if all function parameters are proper
- Send corresponding firmware command to reader with parameters given
- Parses reader's response as "out" parameters and function result

There are exceptions of this rule for certain type of functions. For firmware functions, please refer to "Communication protocol - uFR Series" document at our download section.

Multi reader support

There can be many uFR Series readers connected to a single host. Natively, all library functions are intended for use with “single reader” configuration.

All “single reader” functions have corresponding “multi reader” function. Multi reader functions differs from the “single” functions by following:

Multi-function name always have suffix “M” at the end of function name

First parameter of Multi-function is always “Handle”. For example,

```
SomeFunction(void) => SomeFunctionM(Handle)
```

```
OtherFunction(par1, par2) => OtherFunctionM(Handle, par1, par2)
```

More about Multi-function usage can be found in the Handling with multiple readers.

Function syntax and data types in this document

By default, all functions are shown as their prototypes in C language.

All data types refers C types, except new defined “c_string” data type which representing null terminated char array (also known as “C-String”). Array is always one byte longer (for null character) then string. “c_string” is defined as

```
“typedef const char * c_string”.
```

For quick reference, always consult latest header file “uFCoder.h” at library package. Direct link to “uFCoder.h” can be found on the GIT repository: https://www.d-logic.net/code/nfc-rfid-reader-sdk/ufr-lib/blob/master/include/uFCoder.h

Error codes

All functions always have return result with corresponding status code. Please refer to table ERR_CODES in Appendix: ERROR CODES (DL_STATUS result).

In general you should always get function result = 0x00 if function is finished properly. One exception from this rule is if you get “0x08” – “NO_CARD” result. In a matter of fact, this is not an error, function is executed properly but there is no card present at readers RF field.

All other results indicates that some error occurred.

API set of functions

API set of functions is divided in three categories:

1. Common set
2. Advance set
3. Access control set

Common set of functions is shared among all uFR Series devices.

Advance set contains additional functions for use with uFR Advance and BASE HD uFR devices. It has additional functions for use of Real Time Clock (RTC) and user configurable EEPROM functions.

Access control set contains additional functions for use with BASE HD uFR devices. It has additional functions for use of I/O features like control of door lock, relay contacts and various inputs.

In further reading functions will be marked if they belong to Advance or Access control set.

Library functions

Functions are divided into several groups, based on purpose.

Reader and library related functions

Functions related to reader itself, to obtain some info or set certain device parameters.

Card/tag related commands

Functions used for card (or tag) data manipulation, such as obtaining some info, reading or writing data into card. Can be divided into several groups:

General purpose card related commands

Functions for getting common card data, not specific to card type.

Mifare Classic specific commands

Functions specific to Mifare Classic ® family of cards (Classic 1K and 4K). All functions are dedicated for use with Mifare Classic ® cards. However, some functions can be used with other card types, mostly in cases of direct addressing scheme and those functions will be highlighted in further text.

a) Block manipulation commands – direct and indirect addressing

Functions for manipulating data in blocks of 16 byte according to Mifare Classic ® memory structure organization.

b) Value Block manipulation commands – direct and indirect addressing

Functions for manipulating value blocks byte according to Mifare Classic ® memory structure organization.

c) Linear data manipulation commands

Functions for manipulating data of Mifare Classic ® memory structure as a Linear data space.

NFC – NDEF related commands

Functions for reading and writing common NDEF messages and records into various NFC tags. Currently, only NFC Type 2 Tags are supported, while support for other NFC Tag types will be added in future upgrades.

NTAG related commands

Functions specific to NTAG ® family chips such as NTAG 203, 210, 212, 213, 215, 216. Due to different memory size of various NTAG chips, we implemented functions for handling NTAG chips as generic NFC Type 2 Tag.

UID ASCII mirror support

NTAG 21x family offers specific feature named "UID ASCII mirror function" which is supported by the uFR API using the function **`write_ndef_record_mirroring()`**. For details about "UID ASCII mirror function" refer to http://www.nxp.com/docs/en/data-sheet/NTAG213_215_216.pdf (in Rev. 3.2 from 2. June 2015, page 21)
and http://www.nxp.com/docs/en/data-sheet/NTAG210_212.pdf (in Rev. 3.0 from 14. March 2013, page 16).

NFC counter mirror support

NTAG 213, 215 and 216 devices offers specific feature named "NFC counter mirror function" which is supported by the uFR API using the function **`write_ndef_record_mirroring()`**. For details about "NFC counter mirror function" refer to a document
http://www.nxp.com/docs/en/data-sheet/NTAG213_215_216.pdf (in Rev. 3.2 from 2. June 2015, page 23).

UID and NFC counter mirror support

NTAG 213, 215 and 216 devices offers specific feature named "UID and NFC counter mirror function" which is supported by the uFR API using the function **`write_ndef_record_mirroring()`**. For details about "NFC counter mirror function" refer to a document http://www.nxp.com/docs/en/data-sheet/NTAG213_215_216.pdf (in Rev. 3.2 from 2. June 2015, page 26).

Mifare DESFire specific commands

Functions specific to Mifare DESFire® cards. All uFR Series readers support DESfire set of commands in AES encryption mode according to manufacturer's recommendations. Currently, only Standard Data Files are supported, while other file types shall be supported in future updates.

All readers have hardware built-in AES128 encryption mechanism. That feature provides fast and reliable results with DESFire cards without compromising security keys. Since DESFire EV1/EV2 cards comes in DES mode as factory default setting (due to backward compatibility with older DESfire cards), cards must be turned to AES mode first. There is library built in function for that purpose.

Authentication and password verification protection

Mifare Classic ® family of cards uses authentication mechanism based on 6 bytes keys, which will be explained later in more detail.

NTAG ® 21x family chips and MIFARE Ultralight EV1 uses password verification protection based on PWD and PACK pairs which length is 6 bytes in total. PWD is 4 bytes in length and PACK is contained in 2 bytes. uFR API use this 6 bytes PWD/PACK pair (first goes 4 bytes of the PWD following by the 2 bytes of the PACK) to form PWD/PACK key which is used for password verification with those chip families in the similar manner as the authentication mechanism based on 6 bytes keys.

Selection of the authentication and password verification mechanisms, in the data manipulation functions, is based on the value of the **auth_mode** parameter.

For details about “Password verification protection” refer to following documents: http://www.nxp.com/docs/en/data-sheet/NTAG213_215_216.pdf (in Rev. 3.2 from 2. June 2015, page 30), http://www.nxp.com/docs/en/data-sheet/NTAG210_212.pdf (in Rev. 3.0 from 14. March 2013, page 19) and https://www.nxp.com/docs/en/data-sheet/MF0ULX1.pdf (in Rev. 3.2 from 23. Nov 2017, page 16).

Specific firmware features

There are few firmware features which are specific to uFR Series readers.

Tag Emulation mode

In this mode, reader acts as a Tag. In that mode, not all library functions are available. Reader must be explicitly turned in or out of Tag Emulation mode.

In further reading this topic will be covered in more details.

Combined mode

In combined mode, reader is switching from reader mode to Tag Emulation mode and vice verse few times in seconds. Reader must be explicitly turned in or out of Combined mode.

In further reading this topic will be covered in more details.

Asynchronous UID sending

This feature is turned off by default.

IF turned on, it will send card UID as a row of characters on COM port at defined speed using following format:

```
[Prefix byte] UID_chars [Suffix byte]
```

Where Prefix byte is optional and Suffix byte is mandatory.

In further reading this topic will be covered in more details.

Sleep and Auto Sleep feature

Sleep feature is turned off by default. If turned on, it will put reader into special low power consumption mode to preserve power. In this mode, reader will respond only on function to “wake up”: turn sleep off.

Autosleep feature is different than previous in one major point: it will put reader into sleep after a predefined amount of time and will respond to function calls. Time can be adjusted with dedicated API function.

In further reading this topic will be covered in more details.

Card UID remarks

uFR Series readers support Card Unique IDentifier (Card UID) with various byte length according to defined standards.

4 byte IDs: Non-unique IDs (NUID) are 4 byte long and as the name says, they are Non-Unique, so there is always possibility of existing two or more cards with the same ID (NUID).

7 byte IDs: Card UID are currently 7 byte long with never card types and still provide number range which large enough to provide uniqueness of IDs. These type of UIDs are fully supported at uFR series devices.

10 byte IDs: currently not in use but they are defined by standard for some future use. UFR Series devices are capable of handling this type of IDs when they become available.

Mifare Classic chips overview

One of the most popular and worldwide used contactless card type is NXP's Mifare Classic card, which comes in two memory map layouts: as 1K and 4K card.

Most of mentioned cards comes with 4 byte NUID. Cards with newer production date can be found with 7 byte UID too, especially MF1S70 type.

Mifare Classic 1K (MF1S50) and its derivatives has EEPROM with 1024 bytes storage, where 752 bytes are available for user data.

1 Kbyte EEPROM is organized in 16 sectors with 4 blocks each. A block contains 16 bytes. The last block of each sector is called “trailer”, which contains two secret keys (KeyA and KeyB) and programmable access conditions for each block in this sector.

Keys are encrypted with proprietary algorithm called “Crypto1”.

Figure 1 : MF1S50 memory map

Sector 0	Block 0	Manufacturer Data
	Block 1	DATA
	Block 2	DATA
	Block 3 Trailer	Keys and Access Conditions
Sector 1	Block 0	DATA
	Block 1	DATA
	Block 2	DATA
	Block 3 Trailer	Keys and Access Conditions
…		
Sector 15	Block 0	DATA
	Block 1	DATA
	Block 2	DATA
	Block 3 Trailer	Keys and Access Conditions

Mifare Classic 4K (MF1S70) and its derivatives has EEPROM with 4096 bytes storage, where 3440 bytes are available for user data.

4 Kbyte EEPROM is organized in 40 sectors with 4 blocks each. A block contains 16 bytes. The last block of each sector is called “trailer”, which contains two secret keys (KeyA and KeyB) and programmable access conditions for each block in this sector.

On the contrary of MF1S50, memory is organized in 32 sectors of 4 blocks (sectors 0 -31) and 8 sectors of 16 blocks (sectors 32 - 39).

Keys are encrypted with proprietary algorithm called “Crypto1”.

Figure 2 : MF1S70 memory map

Sector 0	Block 0	Manufacturer Data
	Block 1	DATA
	Block 2	DATA
	Block 3 Trailer	Keys and Access Conditions
Sector 1	Block 0	DATA
	Block 1	DATA
	Block 2	DATA
	Block 3 Trailer	Keys and Access Conditions
…		
Sector 31	Block 0	DATA
	Block 1	DATA
	Block 2	DATA
	Block 3 Trailer	Keys and Access Conditions
Sector 32	Block 0	DATA
	Block 1	DATA
	…	DATA
	Block 15 Trailer	Keys and Access Conditions
…		
Sector 39	Block 0	DATA
	Block 1	DATA
	…	DATA
	Block 15 Trailer	Keys and Access Conditions

Mifare Classic Keys and Access Conditions

Understanding memory map and access conditions of MF1S50 and MF1S70 cards is a must for proper data manipulation with mentioned cards.

Since that subject needs further reading and study, it is out of scope of this document.

Please refer to manufacturer's technical documents for further details. Documents are available at public access on the manufacturer's website.

Further reading of this document is not recommended before one get better insight and understanding of mentioned chip types.

We will try to give brief explanation of access bits and conditions. The next part of the text is taken from manufacturer's documentation "MF1ICS50 – Functional specification" available publicly here.

Access conditions

The access conditions for every data block and sector trailer are defined by 3 bits, which are stored non-inverted and inverted in the sector trailer of the specified sector.

The access bits control the rights of memory access using the secret keys A and B. The access conditions may be altered, provided one knows the relevant key and the current access condition allows this operation.

Remark: With each memory access the internal logic verifies the format of the access conditions. If it detects a format violation the whole sector is irreversible blocked.

Remark: In the following description the access bits are mentioned in the non-inverted mode only.

The internal logic of the MF1ICS50 ensures that the commands are executed only after an authentication procedure or never.

Figure 1 Access conditions

Access Bits	Valid Commands	Block	Description
$C1_3$ $C2_3$ $C3_3$	read, write	3	sector trailer
$C1_2$ $C2_2$ $C3_2$	read, write, increment, decrement, transfer, restore	2	data block

$C1_1$ $C2_1$ $C3_1$	read, write, increment, decrement, transfer, restore	1	data block
$C1_0$ $C2_0$ $C3_0$	read, write, increment, decrement, transfer, restore	0	data block

Figure 2 Organization of Access Bits

Byte number	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	Key A						Access bits				Key B					

Bits	7	6	5	4	3	2	1	0
Byte 6	$C2_3$	$C2_2$	$C2_1$	$C2_0$	$C1_3$	$C1_2$	$C1_1$	$C1_0$
Byte 7	$C1_3$	$C1_2$	$C1_1$	$C1_0$	$C3_3$	$C3_2$	$C3_1$	$C3_0$
Byte 8	$C3_3$	$C3_2$	$C3_1$	$C3_0$	$C2_3$	$C2_2$	$C2_1$	$C2_0$
Byte 9 (GPB)	General Purpose Byte - USER data							

Access conditions for the sector trailer

Depending on the access bits for the sector trailer (block 3) the read/write access to the keys and the access bits is specified as ‘never’, ‘key A’, ‘key B’ or key A|B’ (key A or key B).

On chip delivery the access conditions for the sector trailers and key A are predefined as transport configuration. Since key B may be read in transport configuration, new cards must be authenticated with key A. Since the access bits themselves can also be blocked, special care should be taken during personalization of cards.

Figure 3 Access conditions for the sector trailer

Access bits	Access condition for			Remark
	KEYA	Access bits	KEYB	

C1 3	C2 3	C3 3	read	write	read	write	read	write	
0	0	0	never	key A	key A	never	key A	key A	Key B may be read[1]
0	1	0	never	never	key A	never	key A	never	Key B may be read[1]
1	0	0	never	key B	key A|B	never	never	key B	
1	1	0	never	never	key A|B	never	never	never	
0	0	1	never	key A	key A	key A	key A	key A	Key B may be read, transport configuration[1]
0	1	1	never	key B	key A|B	key B	never	key B	
1	0	1	never	never	key A|B	key B	never	never	
1	1	1	never	never	key A|B	never	never	never	

[1] *Remark: the grey marked lines are access conditions where key B is readable and may be used for data.*

Access conditions for data blocks

Depending on the access bits for data blocks (blocks 0...2) the read/write access is specified as ‘never’, ‘key A’, ‘key B’ or ‘key A|B’ (key A or key B). The setting of the relevant access bits defines the application and the corresponding applicable commands.

- Read/write block: The operations read and write are allowed.
- Value block: Allows the additional value operations increment, decrement, transfer and restore. In one case (‘001’) only read and decrement are possible for a non-rechargeable card. In the other case (‘110’) recharging is possible by using key B.
- Manufacturer block: The read-only condition is not affected by the access bits setting!

Figure 4 Access conditions for data blocks

Access bits			Access condition for				Application
C1	C2	C3	read	write	increment	decrement, transfer, restore	

0	0	0	key A|B[1]	key A|B[1]	key A|B[1]	key A|B[1]	transport configuration
0	1	0	key A|B[1]	never	never	never	read/write block
1	0	0	key A|B[1]	key B[1]	never	never	read/write block
1	1	0	key A|B[1]	key B[1]	key B[1]	key A|B1	value block
0	0	1	key A|B[1]	never	never	key A|B[1]	value block
0	1	1	key B[1]	key B[1]	never	never	read/write block
1	0	1	key B[1]	never	never	never	read/write block
1	1	1	never	never	never	never	read/write block

- Key management: In transport configuration key A must be used for authentication[1]

[1] If Key B may be read in the corresponding Sector Trailer it can't serve for authentication (all grey marked lines in previous table). Consequences: If the RDW tries to authenticate any block of a sector with key B using grey marked access conditions, the card will refuse any subsequent access after authentication.

Reader keys

All uFR Series devices has reserved nonvolatile memory space where following keys are stored:

- 32 Mifare Classic authentication keys, each 6 byte long, indexed [0-31]
- 16 AES keys for use with DESFire cards, each 16 bytes long, indexed [0-15]

All Mifare Classic keys have factory default value as 6 bytes of 0xFF.

All DESfire keys have factory default value as 16 bytes of 0x00.

Important Note: Keys are stored in reader using one way function and protected with password. Keys can be changed with appropriate credentials but can't be read in any circumstances. Please bear this in mind when handling key values.

Mifare Classic authentication modes and usage of keys

There are four possible ways of using Mifare keys when authenticating to card and they are named as follows:

- Reader Keys mode (RK) - default
- Automatic Key Mode 1 (AKM1)
- Automatic Key Mode 2 (AKM2)
- Provided Key mode (PK)

All Mifare Classic related functions have basic function name for default authentication method (RK) and three other variations with appended suffixes AKM1, AKM2 or PK. In further reading we will explain each basic function with variations of key mode usage.

All Mifare keys can be used as “Key A” or “Key B” as defined in Mifare Classic technical document.

For that purpose, each function which use authentication with keys also have parameter “AuthMode” which defines if particular key is used as “Key A” or “Key B”.

In uFR Series API there are two constants defined for this case :

`MIFARE_AUTHENT1A = 0x60` - actual key is used as “Key A”

`MIFARE_AUTHENT1B = 0x61` - actual key is used as “Key B”

Reader Keys mode (RK)

When using this authentication mode, keys stored in reader's memory are used for authentication to Mifare card. Reader Key index [0..31] is passed as function argument.

Example:

Reader keys are all set to default value 6 bytes of 0xFF. We want to use key "`A0 A1 A2 A3 A4 A5h`" as key A to authenticate to card.

First this key must be stored into reader's NVRAM at certain index, for example index=3.

Next, we use "SomeFunction" to do something with card where authentication is must and key is "`A0 A1 A2 A3 A4 A5h`". We will call "SomeFunction" with KeyIndex = 3 and AuthMode =" `MIFARE_AUTHENT1A`".

In this way authentication key is not exposed during communication with host.

Automatic Key Mode 1 (AKM1)

This mode is also using keys stored at reader's memory. Difference between this mode and RK is that keys are used at predefined order.

In this mode, keys indexed from [0..15] are used as "Key A" for each corresponding sector while keys indexed from [16..31] are used as "Key B" for each corresponding sector. That means Key A for Sector 0 is Key indexed as [0] etc.

Brief example:

```
Sector 0  : Key A = Key [0], Key B = Key [16]
Sector 1  : Key A = Key [1], Key B = Key [17]
Sector 2  : Key A = Key [2], Key B = Key [18]
Sector 3  : Key A = Key [3], Key B = Key [19]
…
Sector 15 : Key A = Key [15], Key B = Key [31]
```

Automatic Key Mode 2 (AKM2)

This mode is also using keys stored at reader's memory. Difference is that keys are used at predefined order as even and odd keys.

In this mode, keys indexed with even numbers {0,2,4...30} are used as “Key A” for each corresponding sector while keys indexed with odd numbers {1,3,5...31} are used as “Key B” for each corresponding sector.

Brief example:

```
Sector 0  : Key A = Key [0], Key B = Key [1]
Sector 1  : Key A = Key [2], Key B = Key [3]
Sector 2  : Key A = Key [4], Key B = Key [5]
Sector 3  : Key A = Key [6], Key B = Key [7]
…
Sector 15 : Key A = Key [30], Key B = Key [31]
```

NOTE: In all three above mentioned modes, when using Mifare Classic 4K cards, there are some trade off.

Mifare Classic 4K have 40 sectors instead of 16 as Mifare Classic 1K. In such case, Key A for Sector 0 is the same as Key A for Sector 16 etc. For the last 8 sectors (sectors 32 to 39) the same readers keys are used that correspond to sectors 0 to 7 and 16 to 23.

Example:

```
Sector 16 : Key A, Key B = Sector [0]  keys
Sector 17 : Key A, Key B = Sector [1]  keys
Sector 18 : Key A, Key B = Sector [2]  keys
Sector 31 : Key A, Key B = Sector [15] keys
…
Sector 32 : Key A, Key B = Sector [0]  keys
Sector 33 : Key A, Key B = Sector [1]  keys
…
Sector 39 : Key A, Key B = Sector [7]  keys
```

Provided Key mode (PK)

In this case keys stored into reader are not in use. Key is passed as function parameter as it's real value, like a pointer to array of bytes :"`A0 A1 A2 A3 A4 A5h`".

For example, we will call "SomeFunction" with parameters "Key" and "AuthMode", where "Key" is a pointer to byte array which contains key value bytes.

This method is convenient for testing but we strongly discourage use of this method in real production environments, since keys is exposed on "wire" during communication with host.

Other supported cad/tag types

Currently supported card/tag types in latest firmware revision are:

- Mifare Classic (and derivatives like Fudan FM11RF08)
- Infineon SLE66R35
- Mifare Ultralight (directly supported NFC Type2 Tag)
- Mifare Ultralight C (directly supported NFC Type2 Tag)
- NTAG 203, 210, 212, 213, 215, 216 (directly supported NFC Type2 Tag)
- Mikron MIK640D (directly supported NFC Type2 Tag)
- Other NFC Type2 Tag compatible card are supported as 'T2T generic type', calling **`GetNfcT2tVersion()`** gives more data about tag.
- Mifare Plus (in Mifare Classic compatibility mode)
- Mifare DESFire EV1 (in AES128 mode)
- Mifare DESFire EV2 (in EV1 compatibility mode)

Future firmware and library releases will support additional currently missing features and card types.

API - Programming reference

Scope of this section is to show basic usage scenarios of uFR Series API library functions.

For code snippets and source code examples, please refer to “SDK” section at our download web page.

Most examples are written in various programming languages including C/C++, C#.NET, C++.NET, VB.NET, Java, JavaScript, Python, Lazarus/Delphi.

Dynamic libraries are a part of source code example zip archives. Some libraries may be obsolete due to time of writing of example.

Please be sure to always use the latest library revision from “Libraries” section at our download web page.

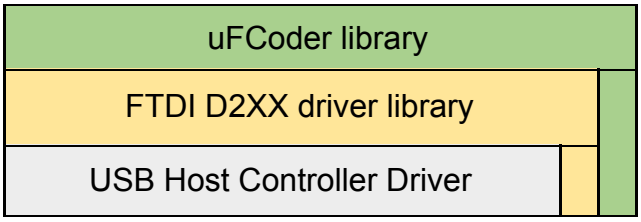
Simply replace obsolete libraries with latest library revision to explore all features mentioned in this document.

Communication and command flow

Communication with uFR Series reader (‘reader” in further text) is established via USB physical communication link.

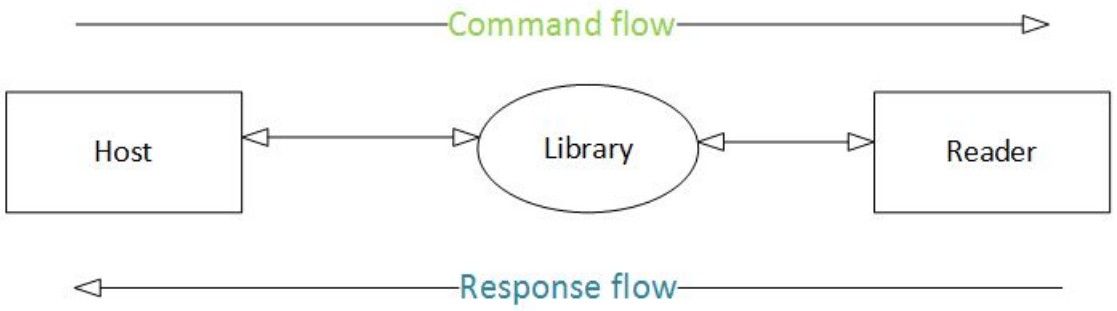
On top physical USB layer is FTDI’s direct access through D2XX drivers library.

uFR Series dynamic library (“uFCoder library” in further reading) is placed above D2XX library.



uFR Series device and host are in master-slave relation, where host represents master and device is a slave.

Command flow is always initiated from master to slave and device is only responding to commands.

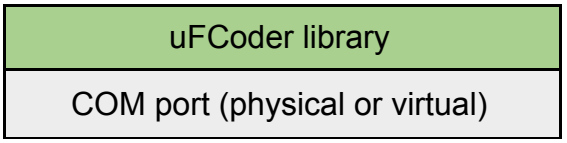


The following sections will describe single reader usage, meaning that only one reader is connected to host.

Connecting several readers to single host is possible and shall be described in separate section.

Important update:

From library version 4.01 and up, it is possible to establish communication with reader without using FTDI's D2XX driver by calling **ReaderOpenEx** function. Library can talk to reader via COM port (physical or virtual) without implementing FTDI's calls. However, this approach is not fast as with use of D2XX drivers but gives much more flexibility to users who had to use COM protocol only, now they can use whole API set of functions via COM port.



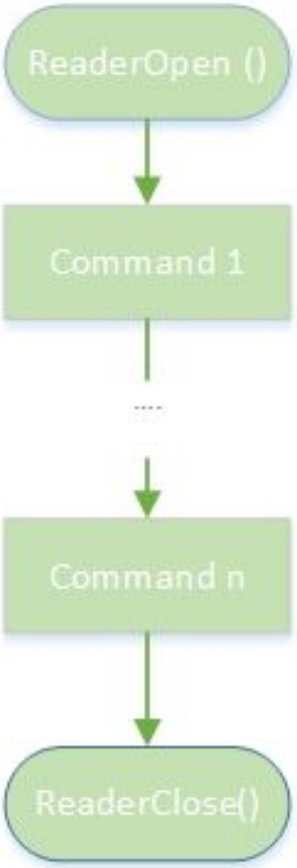
Program flow – basic usage

To establish communication with reader, there must be no other processes to disturbing this communication, which means that only one process or application can have open communication link with reader.

To establish communication link, ReaderOpen () command must be sent.

After successful link opening, all other library functions can be used.

At the end of use, link must be closed by ReaderClose () command, which is usually at application exit or process end.

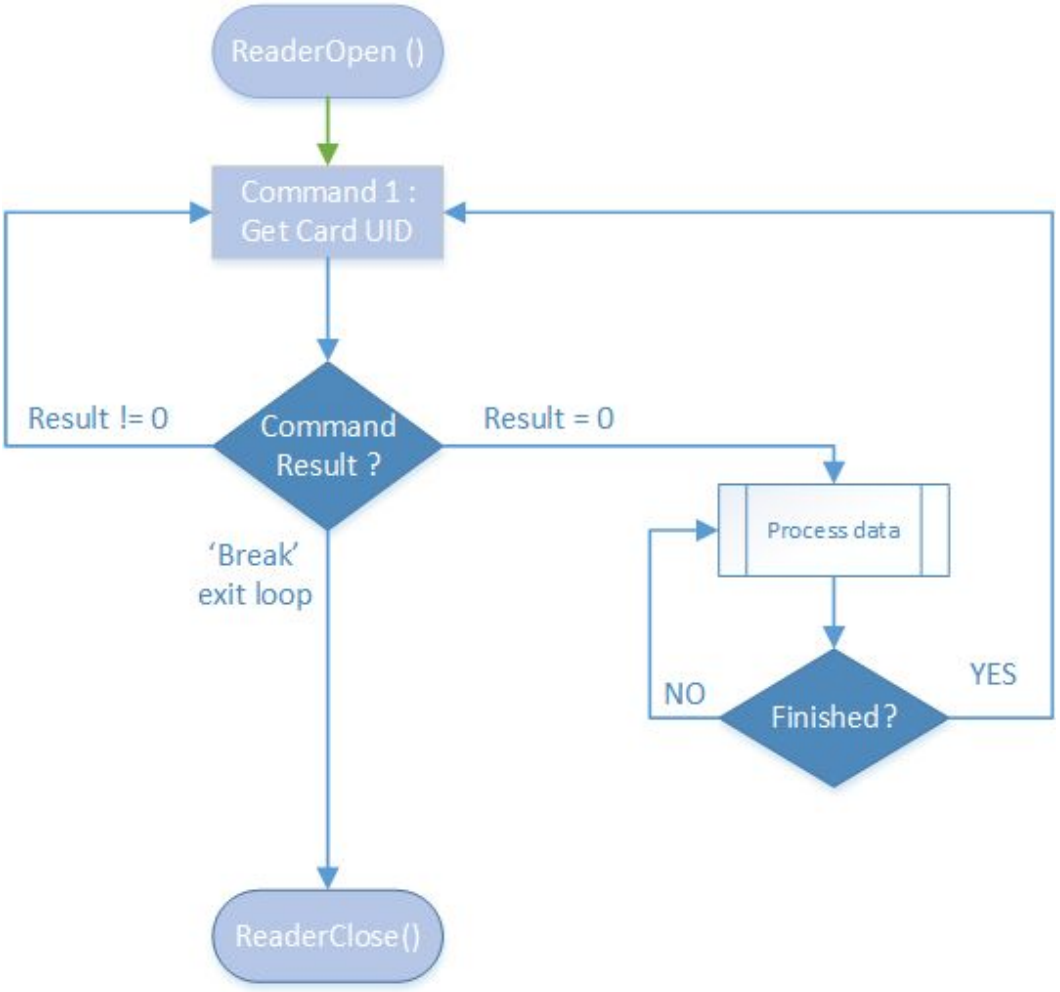


Program flow – polling

In many cases, there is a need to constantly examine some state or check for some events, like for card presence or similar. That is also known as “Polling Loop”.

In polling loop check is performed several times in second and number of check may vary. However, good practice is not to exceed 10 - 15 checks per second.

Almost all uFCoder library functions return Zero value if function call was successful and error code if not.



API - descriptions

Reader and library related functions

As mentioned earlier, uFCoder function call returns (in most cases) integer value as result of function operation. For possible values please refer to table ERR_CODES in Appendix: ERROR CODES (DL_STATUS result).

Exception from this rule are some functions with return parameters “c_string” which is a pointer to array of char (“*typedef const char * c_string*”).

Here is a list of reader and library related functions with return types:

Reader and library functions	
Return Type	Function name
UFR_STATUS	ReaderOpen
UFR_STATUS	ReaderOpenEx
UFR_STATUS	ReaderReset
UFR_STATUS	ReaderClose
UFR_STATUS	ReaderStillConnected
UFR_STATUS	GetReaderType
UFR_STATUS	GetReaderSerialNumber
UFR_STATUS	GetReaderHardwareVersion
UFR_STATUS	GetReaderFirmwareVersion
UFR_STATUS	GetBuildNumber
UFR_STATUS	GetReaderSerialDescription
UFR_STATUS	ChangeReaderPassword
UFR_STATUS	ReaderKeyWrite
UFR_STATUS	ReaderKeysLock
UFR_STATUS	ReaderKeysUnlock
UFR_STATUS	ReadUserData
UFR_STATUS	WriteUserData
UFR_STATUS	UfrEnterSleepMode
UFR_STATUS	UfrLeaveSleepMode
UFR_STATUS	AutoSleepSet
UFR_STATUS	AutoSleepGet
UFR_STATUS	SetSpeedPermanently
UFR_STATUS	GetSpeedParameters
UFR_STATUS	SetAsyncCardIdSendConfig
UFR_STATUS	GetAsyncCardIdSendConfig
UFR_STATUS	ReaderUISignal
UFR_STATUS	UfrRedLightControl
UFR_STATUS	SetDisplayData**

UFR_STATUS	SetDisplayIntensity**
UFR_STATUS	GetDisplayIntensity**
UFR_STATUS	SetSpeakerFrequency
uint32_t	GetDllVersion
c_string	GetDllVersionStr
c_string	UFR_STATUS2String
c_string	GetReaderDescription

** - RFU(reserved for future use)

ReaderOpen

Function description

Open reader communication port.

Function declaration (C language)

```
UFR_STATUS ReaderOpen(void)
```

No parameters required.

ReaderOpenByType

Function description

Opens a port of connected reader using readers family type. Useful for speed up opening for non uFR basic reader type (e.g. BaseHD with uFR support).

Function declaration (C language)

```
UFR_STATUS ReaderOpenByType(uint32_t reader_type);
```

Parameters

0 - auto, same as call ReaderOpen()

1 - uFR type (1 Mbps)

2 - uFR RS232 type (115200 bps)

3 - BASE HD uFR type (250 Kbps)

ReaderOpenEx

Function description

Open reader communication port in several different ways**.** Can be used for establishing communication with COM port too.

Function declaration (C language)

```
UFR_STATUS ReaderOpenEx(uint32_t reader_type,
                        c_string port_name,
                        uint32_t port_interface,
                        void *arg);
```

Parameters

reader_type	0 : auto - same as call ReaderOpen() 1 : uFR type (1 Mbps) 2 : uFR RS232 type (115200 bps) 3 : BASE HD uFR type (250 Kbps)
port_name	is c-string type used to open port by given serial name. If provide NULL or empty string that is AUTO MODE which calls ReaderOpenEx() and test all available ports on the system. serial port name, identifier, like "COM3" on Windows or "/dev/ttyS0" on Linux or "/dev/tty.serial1" on OS X or if you select FTDI, reader serial number like "UN123456", if reader have integrated FTDI interface
port_interface	type of communication interfaces (define interface which we use while connecting to the printer), supported value's: 0 : auto - first try FTDI than serial if port_name is not defined 1 : try serial / virtual COM port / interfaces 2 : try only FTDI communication interfaces 10 : try to open Digital Logic Shields with RS232 uFReader on Raspberry Pi (serial interfaces with GPIO reset)
arg	Reserved for future use, must be NULL.

ReaderReset

Function description

Physical reset of reader communication port.

Function declaration (C language)

```
UFR_STATUS ReaderReset(void)
```

No parameters required.

ReaderClose

Function description

Close reader communication port.

Function declaration (C language)

```
UFR_STATUS ReaderClose(void)
```

No parameters required.

ReaderStillConnected

Function description

Retrieve info if reader is still connected to host.

Function declaration (C language)

```
UFR_STATUS ReaderStillConnected(uint32_t *connected)
```

Parameter

connected	pointer to `connected` variable "`connected`" as result:	
	> 0	Reader is connected on system
	= 0	Reader is not connected on system anymore (or closed)
	< 0	other error
	"`connected`" - Pointer to unsigned int type variable 32 bit long, where the information about readers availability is written. If the reader is connected on system, function store 1 (true) otherwise, on some error, store zero in that variable.	

GetReaderType

Function description

Returns reader type as a pointer to 4 byte value.

Function declaration (C language)

```
UFR_STATUS GetReaderType(uint32_t *lpulReaderType)
```

Parameter

lpulReaderType	pointer to `lpulReaderType` variable. "`lpulReaderType`" as result – please refer to Appendix: DLogic reader type enumeration. E.g. for µFR Nano Classic readers this value is 0xD1180022.

GetReaderSerialNumber

Function description

Returns reader serial number as a pointer to 4 byte value.

Function declaration (C language)

```
UFR_STATUS GetReaderSerialNumber(uint32_t *lpulSerialNumber)
```

Parameter

lpulSerialNumber	pointer to `lpulSerialNumber` variable. "`lpulSerialNumber` " as result holds 4 byte serial number value.

GetReaderHardwareVersion

Function description

Returns reader hardware version as two byte representation of higher and lower byte.

Function declaration (C language)

```
UFR_STATUS GetReaderHardwareVersion(uint8_t *version_major,
                                    uint8_t *version_minor);
```

Parameters

version_major	pointer to version major variable
version_minor	pointer to version minor variable

GetReaderFirmwareVersion

Function description

Returns reader firmware version as two byte representation of higher and lower byte.

Function declaration (C language)

```
UFR_STATUS GetReaderFirmwareVersion(uint8_t *version_major,
                                    uint8_t *version_minor);
```

Parameters

version_major	pointer to version major variable
version_minor	pointer to version minor variable

GetBuildNumber

Function description

Returns reader firmware build version as one byte representation.

Function declaration (C language)

```
UFR_STATUS GetBuildNumber(uint8_t *build)
```

Parameter

build	pointer to `build` variable

GetReaderSerialDescription

Function description

Returns reader's descriptive name as a row of 8 chars.

Function declaration (C language)

```
UFR_STATUS GetReaderSerialDescription(uint8_t pSerialDescription[8])
```

Parameter

pSerialDescription[8]	pointer to pSerialDescription array

ChangeReaderPassword

Function description

This function is used in Common, Advance and Access Control set of functions.

It defines/changes password which I used for:

- Locking/unlocking keys stored into reader
- Setting date/time of RTC

Function declaration (C language)

```
UFR_STATUS ChangeReaderPassword(uint8_t *old_password,
                                uint8_t *new_password)
```

Parameters

old_password	pointer to the 8 bytes array containing current password
new_password	pointer to the 8 bytes array containing new password

ReaderKeyWrite

Function description

Store a new key or change existing key under provided index parameter.The keys are in a special area in EEPROM that can not be read anymore which gains protection.

Function declaration (C language)

```
UFR_STATUS ReaderKeyWrite(const uint8_t *aucKey,
                          uint8_t ucKeyIndex)
```

Parameters

aucKey	Pointer to an array of 6 bytes containing the key. Default key values are always "FF FF FF FF FF FF" hex.
ucKeyIndex	key Index. Possible values are 0 to 31.

ReaderKeysLock

Function description

Lock reader's keys to prevent further changing.

Function declaration (C language)

```
UFR_STATUS ReaderKeysLock(const uint8_t *password);
```

Parameter

password	pointer to the 8 bytes array containing valid password.

ReaderKeysUnlock

Function description

Unlock reader's keys if they are locked with previous function.

The factory setting is that reader keys are unlocked.

Function declaration (C language)

```
UFR_STATUS ReaderKeysUnlock(const uint8_t *password);
```

Parameter

password	pointer to the 8 bytes array containing valid password.

ReaderSoftRestart

Function description

This function is used to restart the reader by software. It sets all readers parameters to default values and close RF field which resets all the cards in the field.

Function declaration (C language)

```
UFR_STATUS ReaderSoftRestart(void);
```

No parameters required.

ReadUserData

Function description

Read user data written in device NV memory.
User data is 16 byte long.

Function declaration (C language)

```
UFR_STATUS ReadUserData(uint8_t *aucData)
```

Parameter

`aucData`	pointer to 16 byte array containing user data.

WriteUserData

Function description

Write user data into device's NV memory. User data is 16 byte long.

Function declaration (C language)

```
UFR_STATUS WriteUserData(uint8_t *aucData)
```

Parameter

`aucData`	pointer to 16 byte array containing user data.

UfrEnterSleepMode

Function description

Turn device into Sleep mode.

Function declaration (C language)

```
UFR_STATUS UfrEnterSleepMode(void)
```

No parameters used.

UfrLeaveSleepMode

Function description

Wake up device from Sleep mode.

Function declaration (C language)

```
UFR_STATUS UfrLeaveSleepMode(void)
```

No parameters used.

AutoSleepSet

Function description

Turn device into Sleep mode after certain amount of time.

Function declaration (C language)

```
UFR_STATUS AutoSleepSet(uint8_t seconds_wait)
```

Parameter

`seconds_wait`	variable holding value of seconds to wait before enter into sleep. If parameter is 0x00, AutoSleep feature is turned off (default state).

AutoSleepGet

Function description

Get status of AutoSleep mode.

Function declaration (C language)

```
UFR_STATUS AutoSleepGet(uint8_t seconds_wait)
```

Parameter

`seconds_wait`	variable holding value of seconds to wait before enter into sleep. If parameter is 0x00, AutoSleep feature is turned off (default state).

SetSpeedPermanently

Function description

This function is used for setting communication speed between reader and ISO144443-4 cards. For other card types, default speed of 106 kbps is in use.

Function declaration (C language)

```
UFR_STATUS SetSpeedPermanently (uint8_t tx_speed,
                                uint8_t rx_speed)
```

Parameters

tx_speed	setup value for transmit speed
rx_speed	setup value for receive speed

Valid speed setup values are:

Const	*Configured speed*
0	106 kbps (default)
1	212 kbps
2	424 kbps

On some reader types maximum **rx_speed** is 212 kbps. If you try to set higher speed than possible, reader will automatically set the maximum possible speed.

GetSpeedParameters

Function description

Returns baud rate configured with previous function.

Function declaration (C language)

```
UFR_STATUS  GetSpeedParameters(uint8_t *tx_speed,
                               uint8_t *rx_speed)
```

Parameters

tx_speed	pointer to variable, returns configured value for transmit speed
rx_speed	pointer to variable, returns configured value for receive speed

SetAsyncCardIdSendConfig

Function description

This function is used for “Asynchronous UID sending” feature. Returned string contains hexadecimal notation of card ID with one mandatory suffix character and one optional prefix character.

Example:

Card ID is 0xA103C256, prefix is 0x58 ('X'), suffix is 0x59 ('Y')

Returned string is “XA103C256Y”

Function sets configuration parameters for this feature.

Function declaration (C language)

```
UFR_STATUS SetAsyncCardIdSendConfig (uint8_t send_enable,
                                     uint8_t prefix_enable,
                                     uint8_t prefix,
                                     uint8_t suffix,
                                     uint8_t send_removed_enable,
                                     uint32_t async_baud_rate);
```

Parameters

send_enable	turn feature on/off (0/1)
prefix_enable	use prefix or not (0/1)
prefix	prefix character
suffix	suffix character
send_removed_enable	Turn feature on/off (0/1). If feature is enabled then Asynchronous UID will also be sent when removing a card from the reader field.
async_baud_rate	baud rate value (e.g. 9600)

GetAsyncCardIdSendConfig

Function description

Returns info about parameters configured with previous function.

Function declaration (C language)

```
UFR_STATUS GetAsyncCardIdSendConfig (uint8_t *send_enable,
                                     uint8_t *prefix_enable,
                                     uint8_t *prefix,
                                     uint8_t *suffix,
                                     uint8_t *send_removed_enable,
                                     uint32_t *async_baud_rate);
```

Parameters

`send_enable`	pointer, if feature is on/off (0/1)
`prefix_enable`	pointer, if prefix is used or not (0/1)
`prefix`	pointer to variable holding prefix character
`suffix`	pointer to variable holding suffix character
`send_removed_enable`	Pointer. If value is 0 then feature is off. Otherwise, feature is on. If feature is enabled then Asynchronous UID is sent when the card is removed from the reader field.
`async_baud_rate`	pointer to variable holding configured baud rate

SetAsyncCardIdSendConfigEx

Function description

Function sets the parameters of card ID sending.

Function declaration (C language)

```
UFR_STATUS SetAsyncCardIdSendConfigEx(
      uint8_t send_enable,
      uint8_t prefix_enable,
      uint8_t prefix,
      uint8_t suffix,
      uint8_t send_removed_enable,
      uint8_t reverse_byte_order,
      uint8_t decimal_representation,
      uint32_t async_baud_rate);
```

Parameters

`send_enable`	turn feature on/off (0/1)
`prefix_enable`	use prefix or not (0/1)
`prefix`	prefix character

suffix	suffix character
send_removed_enable	Turn feature on/off (0/1). If feature is enabled then Asynchronous UID will also be sent when removing a card from the reader field.
reverse_byte_order	Turn feature on/off (0/1). If feature is disabled then the order of bytes (UID) will be as on card. If feature is enabled then the order of bytes will be reversed then the card's order of bytes.
decimal_representation	Turn feature on/off (0/1). If feature is enabled then the UID will be presented as a decimal number. If feature is disabled then the UID will be presented as a hexadecimal number
async_baud_rate	baud rate value (e.g. 9600)

GetAsyncCardIdSendConfigEx

Function description

Function returns the parameters of card ID sending.

Function declaration (C language)

```
UFR_STATUS GetAsyncCardIdSendConfigEx(
      uint8_t *send_enable,
      uint8_t *prefix_enable,
      uint8_t *prefix,
      uint8_t *suffix,
      uint8_t *send_removed_enable,
      uint8_t *reverse_byte_order,
      uint8_t *decimal_representation,
      uint32_t *async_baud_rate);
```

Parameters

send_enable	pointer to the sending enable flag
prefix_enable	pointer to the prefix existing flag
prefix	pointer to prefix character
suffix	pointer to suffix character
send_removed_enable	pointer to flag

reverse_byte_order	pointer to flag
decimal_representation	pointer to flag
async_baud_rate	pointer to baud rate variable

ReaderUISignal

Function description

This function turns sound and light reader signals. Sound signals are performed by reader's buzzer and light signals are performed by reader's LEDs.

There are predefined signal values for sound and light:

light_signal_mode:		beep_signal_mode:	
0	None	0	None
1	Long Green	1	Short
2	Long Red	2	Long
3	Alternation	3	Double Short
4	Flash	4	Triple Short
		5	Triplet Melody

Function declaration (C language)

```
UFR_STATUS ReaderUISignal(uint8_t light_signal_mode,
                          uint8_t beep_signal_mode)
```

Parameters

light_signal_mode	value from table (0 - 4)
beep_signal_mode	value from table (0 - 5)

UfrRedLightControl

Function description

This function turns Red LED only.
If “light_status” value is 1, red light will be constantly turned on until receive “light_status “ value 0.

Function declaration (C language)

```
UFR_STATUS UfrRedLightControl(uint8_t light_status)
```

Parameter

light_status	value 0 or 1

SetSpeakerFrequency

Function description

This function plays constant sound of “frequency” Hertz.

Function declaration (C language)

```
UFR_STATUS SetSpeakerFrequency(uint16_t frequency)
```

Parameter

frequency	frequency in Hz

To stop playing sound, send 0 value for “frequency”.

Handling with multiple readers

If you want to communicate and use multiple readers from an application, you have to follow the initial procedure for enumerating uFR compatible devices and getting theirs handles. First call ReaderList_UpdateAndGetCount() to prepare internal list of connected devices and then call ReaderList_GetInformation() several times to get information of every reader.

Handle is used to identify certain reader when calling multi-functions (with suffix M).

ReaderList_UpdateAndGetCount

Function description

This is the first function in the order for execution for the multi-reader support.

The function prepare the list of connected uF-readers to the system and returns the number of list items - number of connected uFR devices.

ReaderList_UpdateAndGetCount() scan all communication ports for compatible devices, probes opened readers if still connected, if not close and mark their handles for deletion. If some device

is disconnected from system this function should remove its handle.

Function declaration (C language)

```
UFR_STATUS ReaderList_UpdateAndGetCount(int32_t * NumberOfDevices);
```

Parameters

NumberOfDevices	how many compatible devices is connected to the system

Returns: status of execution

ReaderList_GetInformation

Function description

Function for getting all relevant information about connected readers.

You must call the function as many times as there are detected readers. E.g. If you have tree connected readers, detected by ReaderList_UpdateAndGetCount(), you should call this function tree times.

Function declaration (C language)

```
UFR_STATUS ReaderList_GetInformation(
          UFR_HANDLE *DeviceHandle,
          c_string *DeviceSerialNumber,
          int *DeviceType, int *DeviceFWver,
          int *DeviceCommID,int *DeviceCommSpeed,
          c_string *DeviceCommFTDISerial,
          c_string *DeviceCommFTDIDescription,
          int *DeviceIsOpened,
          int *DeviceStatus);
```

Parameters

DeviceHandle	assigned Handle to the uFR reader - pointer for general purpose (void * type in C)
DeviceSerialNumber	device serial number, pointer to static reserved information in library (no need to reserve memory space)
DeviceType	device identification in Digital Logic AIS database
DeviceFWver	version of firmware
DeviceCommID	device identification number (master)
DeviceCommSpeed	communication speed in bps
DeviceCommFTDISerial	FTDI COM port identification, pointer to static reserved information in library (no need to reserve memory space)

DeviceCommFTDIDescription	FTDI COM port description, pointer to static reserved information in library (no need to reserve memory space)
DeviceIsOpened	is Device opened - 0 not opened, other value is opened
DeviceStatus	actual device status

ReaderList_Destroy

Function description

Force handle deletion when you identify that the reader is no longer connected, and want to release the handle immediately. If the handle exists in the list of opened devices, function would try to close communication port and destroy the handle.

When uF-reader is disconnected ReaderList_UpdateAndGetCount() will do that (destroy) automatically in next execution.

Function declaration (C language)

```
UFR_STATUS ReaderList_Destroy(UFR_HANDLE DeviceHandle);
```

Parameter

DeviceHandle	the handle that will be destroyed

Example (in C):

```
int main(void)
{
     puts(GetDllVersionStr());

     UFR_STATUS status;
     int32_t NumberOfDevices;

     status = ReaderList_UpdateAndGetCount(&NumberOfDevices);
     if (status)
     {
          // TODO: check error
          printf("ReaderList_UpdateAndGetCount(): error= %s\n",
                 UFR_Status2String(status));

          return EXIT_SUCCESS;
     }

     printf("ReaderList_UpdateAndGetCount(): NumberOfDevices=
%d\n",
            NumberOfDevices);

     for (int i = 0; i < NumberOfDevices; ++i)
     {
          UFR_HANDLE DeviceHandle;
          c_string DeviceSerialNumber;
          int DeviceType;
          int DeviceFWver;
          int DeviceCommID;
          int DeviceCommSpeed;
          c_string DeviceCommFTDISerial;
          c_string DeviceCommFTDIDescription;
          int DeviceIsOpened;
          int DeviceStatus;

          status = ReaderList_GetInformation(&DeviceHandle,
                   &DeviceSerialNumber, &DeviceType, &DeviceFWver,
                   &DeviceCommID, &DeviceCommSpeed,
                   &DeviceCommFTDISerial,
&DeviceCommFTDIDescription,
                   &DeviceIsOpened, &DeviceStatus);

          printf("{%d/%d} DeviceHandle= %p, DeviceSerialNumber=
%s, "
              "DeviceType= %X, DeviceFWver= %d, "
              "DeviceCommID= %d, DeviceCommSpeed= %d, "
              "\n\t\t"
              "DeviceCommFTDISerial= %s, DeviceCommFTDIDescription=
%s, "
```

```
                "\n\t\t"
                "DeviceIsOpened= %d, DeviceStatus= %d\n", i + 1,
                NumberOfDevices, DeviceHandle, DeviceSerialNumber,
                DeviceType, DeviceFWver, DeviceCommID,
DeviceCommSpeed,
                DeviceCommFTDISerial, DeviceCommFTDIDescription,
                DeviceIsOpened, DeviceStatus);

            puts(GetReaderDescriptionM(DeviceHandle));
        }
        return EXIT_SUCCESS;
}
```

Helper library functions

GetDllVersionStr

Function description

This function returns library version as string.

Function declaration (C language)

```
c_string GetDllVersionStr(void)
```

No parameters used.

GetDllVersion

Function description

This function returns library version as number.

Function declaration (C language)

```
uint32_t GetDllVersion(void);
```

Returns compact version number, in little-endian format

Low Byte: Major version number

High Byte: Minor version number

Upper byte: Build number

Master Byte: reserved -

UFR_STATUS2String

Function description

This is helper library function. Returns DL_STATUS result code as readable descriptive data. Return type is string. For DL_STATUS enumeration, please refer to Appendix: ERROR CODES (DL_STATUS result).

Function declaration (C language)

```
c_string UFR_Status2String(const UFR_STATUS status)
```

GetReaderDescription

Function description

This function returns reader's descriptive name. Return type is string. No parameters required.

Function declaration (C language)

```
c_string GetReaderDescription(void)
```

No parameters used.

Card/tag related commands

General purpose card related commands

Following functions are applicable to all card types.

UFR_STATUS	GetDlogicCardType
UFR_STATUS	GetCardId
UFR_STATUS	GetCardIdEx
UFR_STATUS	GetLastCardIdEx

GetDlogicCardType

Function description

This function returns card type according to DlogicCardType enumeration. For details, please refer to Appendix: DLogic CardType enumeration.

If the card type is not supported, function return the `lpucCardType` value equal to zero :

`TAG_UNKNOWN = 0x00`

Function declaration (C language)

```
UFR_STATUS GetDlogicCardType(uint8_t *lpucCardType)
```

Parameter

lpucCardType	pointer to `lpucCardType` variable. Variable `lpucCardType` holds returned value of actual card type present in RF field.

GetNfcT2TVersion

Function description

This function returns 8 bytes of the T2T version. All modern T2T chips support this functionality and have in common a total of 8 byte long version response. This function is primarily intended to use with NFC_T2T_GENERIC tags (i.e. tags which return 0x0C in the *lpucCardType parameter of the GetDlogicCardType()).

Function declaration (C language)

```
UFR_STATUS GetNfcT2TVersion(uint8_t lpucVersionResponse[8]);
```

Parameter

lpucVersionResponse[8]	array containing 8 bytes which will receive raw T2T version.

NfcT2TSafeConvertVersion

Function description

This is a helper function for converting raw array of 8 bytes received by calling **`GetNfcT2TVersion()`**. All modern T2T chips having same or very similar structure of the T2T version data represented in the uFR API by the structure type **`t2t_version_t`**:

```
typedef struct t2t_version_struct {
     uint8_t header;
     uint8_t vendor_id;
     uint8_t product_type;
     uint8_t product_subtype;
     uint8_t major_product_version;
     uint8_t minor_product_version;
     uint8_t storage_size;
     uint8_t protocol_type;
} t2t_version_t;
```

This function is primarily intended to use with NFC_T2T_GENERIC tags (i.e. tags which return 0x0C in the *lpucCardType parameter of the **`GetDlogicCardType()`**). Conversion done by this

function is "alignment safe".

Function declaration (C language)

```
void NfcT2TSafeConvertVersion(t2t_version_t *version,
                              const uint8_t *version_record);
```

Parameters

version	pointer to the structure of the **t2t_version_t** type which will receive converted T2T version
version_record	pointer to array containing 8 bytes of the raw T2T version acquired using function **GetNfcT2TVersion()**

GetCardId

Function description

Returns card UID as a 4-byte array. This function is deprecated and used only for backward compatibility with older firmware versions (before v2.0). We strongly discourage use of this function. This function can't successfully handle 7 byte UIDS.

Function declaration (C language)

```
UFR_STATUS GetCardId(uint8_t *lpucCardType,
                     uint32_t *lpulCardSerial)
```

Parameters

lpucCardType	returns pointer to variable which holds card type according to SAK
lpulCardSerial	returns pointer to array of card UID bytes, 4 bytes long ONLY

GetCardIdEx

Function description

This function returns UID of card actually present in RF field of reader. It can handle all three known types : 4, 7 and 10 byte long UIDs.

This function is recommended for use instead of GetCardId.

Function declaration (C language)

```
UFR_STATUS GetCardIdEx(uint8_t *lpucSak,
                       uint8_t *aucUid,
                       uint8_t *lpucUidSize);
```

Parameters

lpucSak	returns pointer to variable which holds card type according to SAK
aucUid	returns pointer to array of card UID bytes, variable length
lpucUidSize	returns pointer to variable holding information about UID length

GetLastCardIdEx

Function description

This function returns UID of last card which was present in RF field of reader. It can handle all three known types : 4, 7 and 10 byte long UIDs. Difference with GetCardIdEx is that card does not be in RF field mandatory, UID value is stored in temporary memory area.

Function declaration (C language)

```
UFR_STATUS GetLastCardIdEx(uint8_t *lpucSak,
                           uint8_t *aucUid,
                           uint8_t *lpucUidSize);
```

Parameters :

lpucSak	returns pointer to variable which holds card type according to SAK
aucUid	returns pointer to array of card UID bytes, variable length
lpucUidSize	returns pointer to variable holding information about UID length

Mifare Classic specific functions

Functions specific to Mifare Classic ® family of cards (Classic 1K and 4K). All functions are dedicated for use with Mifare Classic ® cards. However, some functions can be used with other card types, mostly in cases of direct addressing scheme and those functions will be highlighted in further text. There are few types of following functions:

d) Block manipulation functions – direct and indirect addressing

Functions for manipulating data in blocks of 16 byte according to Mifare Classic ® memory structure organization.

e) Value Block manipulation functions – direct and indirect addressing

Functions for manipulating value blocks byte according to Mifare Classic ® memory structure organization.

f) Linear data manipulation functions

Functions for manipulating data of Mifare Classic ® memory structure as a Linear data space.

Function's variations

All listed functions have 4 variations according to key mode, as explained earlier in chapter "Mifare Classic authentication modes and usage of keys". Let's take "BlockRead" function as example:

`BlockRead`	`RK mode`
`BlockRead_AKM1`	`AKM1 mode`
`BlockRead_AKM2`	`AKM2 mode`
`BlockRead_PK`	`PK mode`

Direct or Indirect addressing

In general, when speaking about direct and indirect addressing functions, both function types does the same thing. Main difference is in a way of block addressing.

Direct addressing functions use absolute value for Block address according to Mifare Classic memory map, where real block address (0-63) corresponds to function parameter value.

Indirect addressing functions use Block-In-Sector approach. Each Sector have 4 blocks (or more, for higher Sectors of the Mifare Classic 4K cards), so function always need two parameters: real Sector address and relative Block address in particular sector.

This approach is very useful for loop usage etc. Generally, it is up to user which one of these two function types will use.

Linear Address Data Space

Writing of consecutive data larger than 1 block (16 bytes) can be pretty tricky because of Mifare Classic memory organization map. Each 4th block is so called "Trailer Block" containing keys and access conditions.

For that purpose, uFR Series API use specific set of functions. User can write data even larger than 1 block without concerning about Trailer Blocks. Reader's firmware will take care of Trailer Blocks and arrange data in consecutive order, automatically jumping over Trailer Blocks. Parameters needed for this purpose are starting address in bytes and data length. Linear Address Data Space always begin at first free byte of specific card. In case of Mifare Classic cards, it is Byte 0 of Block 1 in Sector 0.

These type of functions can be used with other card types and Linear Address Data Space may start at different address. For example in case of Mifare Ultralight, Linear Address Data Space start at byte 0 of Page 4, exactly after OTP bytes page.

Following example shows how Linear Address Data Space looks like in case of Mifare Classic card.

Let's write "Data" of 85 bytes, indexed as 0..84 bytes.

Using LinearWrite function, we will send Data, Starting address 0 and DataLength 85.

Reader's firmware will do the rest in following manner:

<table>
<tr><td>Sector 0</td><td>Block 0</td><td>Manufacturer Block</td><td></td><td></td></tr>
<tr><td rowspan="3"></td><td>Block 1</td><td>Bytes 0 -15</td><td rowspan="9">LINEAR SPACE</td><td rowspan="9">Linear Space starts here at Byte 0

Jumping over Trailer

Jumping over Trailer
Rest of Block is not changed (Bytes 5 - 15)</td></tr>
<tr><td>Block 2</td><td>Bytes 16 - 31</td></tr>
<tr><td>Block 3</td><td>Trailer</td></tr>
<tr><td rowspan="3">Sector 1</td><td>Block 0</td><td>Bytes 32 - 47</td></tr>
<tr><td>Block 1</td><td>Bytes 48 - 63</td></tr>
<tr><td>Block 2</td><td>Bytes 64 - 79</td></tr>
<tr><td rowspan="2"></td><td>Block 3</td><td>Trailer</td></tr>
<tr><td>Sector 2</td><td>Block 0</td><td>Bytes 80- 84</td></tr>
</table>

List of Mifare Classic specific functions

UFR_STATUS	BlockRead *1
UFR_STATUS	BlockWrite *1
UFR_STATUS	BlockInSectorRead
UFR_STATUS	BlockInSectorWrite
UFR_STATUS	LinearRead *1
UFR_STATUS	LinearWrite *1
UFR_STATUS	LinRowRead *1
UFR_STATUS	LinearFormatCard
UFR_STATUS	SectorTrailerWrite
UFR_STATUS	SectorTrailerWriteUnsafe
UFR_STATUS	ValueBlockRead
UFR_STATUS	ValueBlockWrite

UFR_STATUS	ValueBlockInSectorRead
UFR_STATUS	ValueBlockInSectorWrite
UFR_STATUS	ValueBlockIncrement
UFR_STATUS	ValueBlockDecrement
UFR_STATUS	ValueBlockInSectorIncrement
UFR_STATUS	ValueBlockInSectorDecrement

"*1" - function can be used with NFC T2T card types (i.e. all varieties of the Mifare Ultralight, NTAG 203, NTAG 21x, Mikron MIK640D and other NFC_T2T_GENERIC tags).

If you want to use the following functions: ValueBlockRead(), ValueBlockWrite(), ValueBlockInSectorRead(), ValueBlockInSectorWrite(), ValueBlockIncrement(), ValueBlockDecrement(), ValueBlockInSectorIncrement() and ValueBlockInSectorDecrement(), then you need to change access bits for data blocks in chosen sector to one of the "value blocks application" access condition. You can do this using uFR API function SectorTrailerWrite().

BlockRead

Function description

Read particular block using absolute Block address.

Function declaration (C language)

```
UFR_STATUS BlockRead(uint8_t *data,
                     uint8_t block_address,
                     uint8_t auth_mode,
                     uint8_t key_index);

UFR_STATUS BlockRead_AKM1(uint8_t *data,
                          uint8_t block_address,
                          uint8_t auth_mode);

UFR_STATUS BlockRead_AKM2(uint8_t *data,
                          uint8_t block_address,
                          uint8_t auth_mode);

UFR_STATUS BlockRead_PK(uint8_t *data,
                        uint8_t block_address,
                        uint8_t auth_mode,
                        const uint8_t *key);
```

Parameters

data	Pointer to array of bytes containing data
block_address	Absolute block address
auth_mode	**For Mifare Classic** tags defines whether to perform authentication with key A or key B: use KeyA - MIFARE_AUTHENT1A = 0x60 or KeyB - MIFARE_AUTHENT1B = 0x61

	For NTAG 21x, Ultralight EV1 and other T2T tags supporting PWD_AUTH value 0x61 means "***use PWD_AUTH***" with BlockRead() or BlockRead_PK() functions. Value 0x60 with BlockRead() or BlockRead_PK() functions means "***without PWD_AUTH***" and in that case *you can send for ucReaderKeyIndex* or *aucProvidedKey parameters anything you want without influence on the result. For NTAG 21x, Ultralight EV1 and other T2T tags supporting PWD_AUTH you can use _AKM1 or _AKM2 function variants only* ***without PWD_AUTH*** *in any case of the valid values (0x60 or 0x61) provided for this parameter.*
key_index	Index of reader's key to be used (RK mode)
key	Pointer to 6 byte array containing key bytes (PK mode)

When using this function with other card types, `auth_mode`, `key_index` and `key` parameters are not relevant but they must take default values.

BlockWrite

Function description

Write particular block using absolute Block address.

Function declaration (C language)

```
UFR_STATUS BlockWrite(uint8_t *data,
                      uint8_t block_address,
                      uint8_t auth_mode,
                      uint8_t key_index);

UFR_STATUS BlockWrite_AKM1(uint8_t *data,
                           uint8_t block_address,
                           uint8_t auth_mode);

UFR_STATUS BlockWrite_AKM2(uint8_t *data,
                           uint8_t block_address,
                           uint8_t auth_mode);

UFR_STATUS BlockWrite_PK(uint8_t *data,
                         uint8_t block_address,
                         uint8_t auth_mode, const uint8_t *key);
```

Parameters

data	Pointer to array of bytes containing data
block_address	Absolute block address
auth_mode	**For Mifare Classic** tags defines whether to perform authentication with key A or key B: use KeyA - MIFARE_AUTHENT1A = 0x60 or KeyB - MIFARE_AUTHENT1B = 0x61 **For NTAG 21x, Ultralight EV1 and other T2T tags supporting PWD_AUTH**

	value 0x61 means "***use PWD_AUTH***" with BlockWrite() or BlockWrite_PK() functions. Value 0x60 with BlockWrite() or BlockWrite_PK() functions means "***without PWD_AUTH***" and in that case *you can send for ucReaderKeyIndex* or *aucProvidedKey parameters anything you want without influence on the result. For NTAG 21x, Ultralight EV1 and other T2T tags supporting PWD_AUTH you can use _AKM1 or _AKM2 function variants only* ***without PWD_AUTH*** *in any case of the valid values (0x60 or 0x61) provided for this parameter.*
key_index	Index of reader's key to be used (RK mode)
key	Pointer to 6 byte array containing key bytes (PK mode)

When using this function with other card types, `auth_mode, key_index` and `key` parameters are not relevant but they must take default values.

BlockInSectorRead

Function description

Read particular block using relative Block in Sector address.

Function declaration (C language)

```
UFR_STATUS BlockInSectorRead(uint8_t *data, uint8_t sector_address,
                             uint8_t block_in_sector_address,
                             uint8_t auth_mode, uint8_t key_index);

UFR_STATUS BlockInSectorRead_AKM1(uint8_t *data, uint8_t
sector_address,
                             uint8_t block_in_sector_address,
                             uint8_t auth_mode);

UFR_STATUS BlockInSectorRead_AKM2(uint8_t *data, uint8_t
sector_address,
                             uint8_t block_in_sector_address,
                             uint8_t auth_mode);

UFR_STATUS BlockInSectorRead_PK(uint8_t *data,uint8_t sector_address,
                             uint8_t block_in_sector_address,
                             uint8_t auth_mode,
                             const uint8_t *key);
```

Parameters

data	Pointer to array of bytes containing data
sector_address	Absolute Sector address
block_in_sector_address	Block address in Sector
auth_mode	**For Mifare Classic** tags defines whether to perform authentication with key A or key B: use KeyA - MIFARE_AUTHENT1A = 0x60 or KeyB - MIFARE_AUTHENT1B = 0x61

	For NTAG 21x, Ultralight EV1 and other T2T tags supporting PWD_AUTH value 0x61 means "***use PWD_AUTH***" with BlockInSectorRead() or BlockInSectorRead_PK() functions. Value 0x60 with BlockInSectorRead() or BlockInSectorRead_PK() functions means "***without PWD_AUTH***" and in that case *you can send for ucReaderKeyIndex* or *aucProvidedKey parameters anything you want without influence on the result. For NTAG 21x, Ultralight EV1 and other T2T tags supporting PWD_AUTH you can use _AKM1 or _AKM2 function variants only* ***without PWD_AUTH*** *in any case of the valid values (0x60 or 0x61) provided for this parameter.*
key_index	Index of reader's key to be used (RK mode)
key	Pointer to 6 byte array containing key bytes (PK mode)

This function can't be used with card types other than Mifare Classic.

BlockInSectorWrite

Function description

Write particular block using relative Block in Sector address.

Function declaration (C language)

```
UFR_STATUS BlockInSectorWrite(uint8_t *data, uint8_t sector_address,
                              uint8_t block_in_sector_address,
                              uint8_t auth_mode, uint8_t key_index);

UFR_STATUS BlockInSectorWrite_AKM1(uint8_t *data,
                              uint8_t sector_address,
                              uint8_t block_in_sector_address,
                              uint8_t auth_mode);

UFR_STATUS BlockInSectorWrite_AKM2(uint8_t *data,
                              uint8_t sector_address,
                              uint8_t block_in_sector_address,
                              uint8_t auth_mode);

UFR_STATUS BlockInSectorWrite_PK(uint8_t *data, uint8_t sector_address,
                              uint8_t block_in_sector_address,
                              uint8_t auth_mode, const uint8_t *key);
```

Parameters

data	Pointer to array of bytes containing data
sector_address	Absolute Sector address
block_in_sector_address	Block address in Sector
auth_mode	**For Mifare Classic** tags defines whether to perform authentication with key A or key B:

	use KeyA - MIFARE_AUTHENT1A = 0x60 or KeyB - MIFARE_AUTHENT1B = 0x61 **For NTAG 21x, Ultralight EV1 and other T2T tags supporting PWD_AUTH** value 0x61 means "***use PWD_AUTH***" with BlockInSectorWrite() or BlockInSectorWrite_PK() functions. Value 0x60 with BlockInSectorWrite() or BlockInSectorWrite_PK() functions means "***without PWD_AUTH***" and in that case *you can send for ucReaderKeyIndex* or *aucProvidedKey parameters anything you want without influence on the result. For NTAG 21x, Ultralight EV1 and other T2T tags supporting PWD_AUTH you can use _AKM1 or _AKM2 function variants only* ***without PWD_AUTH*** *in any case of the valid values (0x60 or 0x61) provided for this parameter.*
key_index	Index of reader's key to be used (RK mode)
key	Pointer to 6 byte array containing key bytes (PK mode)

This function can't be used with card types other than Mifare Classic.

LinearRead

Function description

Group of functions for linear reading in uFR firmware utilise FAST_READ ISO 14443-3 command with NTAG21x and Mifare Ultralight EV1 tags.

Function declaration (C language)

```
UFR_STATUS LinearRead(uint8_t *Data, uint16_t linear_address,
                uint16_t length, uint16_t *bytes_returned,
                uint8_t auth_mode,uint8_t key_index);

UFR_STATUS LinearRead_AKM1(uint8_t *Data, uint16_t linear_address,
                uint16_t length, uint16_t *bytes_returned, uint8_t
           auth_mode);

UFR_STATUS LinearRead_AKM2(uint8_t *Data, uint16_t linear_address,
                uint16_t length, uint16_t *bytes_returned, uint8_t
           auth_mode);

UFR_STATUS LinearRead_PK(uint8_t *Data, uint16_t linear_address,
                uint16_t length, uint16_t *bytes_returned,
                uint8_t auth_mode, const uint8_t *key);
```

Parameters

data	Pointer to array of bytes containing data
linear_address	Address of byte – where to start reading
length	Length of data – how many bytes to read
bytes_returned	Pointer to variable holding how many bytes are returned

auth_mode	**For Mifare Classic** tags defines whether to perform authentication with key A or key B: use KeyA - MIFARE_AUTHENT1A = 0x60 or KeyB - MIFARE_AUTHENT1B = 0x61 **For NTAG 21x, Ultralight EV1 and other T2T tags supporting PWD_AUTH** value 0x61 means "***use PWD_AUTH***" with LinearRead() or LinearRead_PK() functions. Value 0x60 with LinearRead() or LinearRead_PK() functions means "***without PWD_AUTH***" and in that case *you can send for ucReaderKeyIndex* or *aucProvidedKey parameters anything you want without influence on the result. For NTAG 21x, Ultralight EV1 and other T2T tags supporting PWD_AUTH you can use _AKM1 or _AKM2 function variants only* ***without PWD_AUTH*** *in any case of the valid values (0x60 or 0x61) provided for this parameter.*
key_index	Index of reader's key to be used (RK mode)
key	Pointer to 6 byte array containing key bytes (PK mode)

When using this functions with other card types, `auth_mode, key_index` and `key` parameters are not relevant but must take default values.

LinearWrite

Function description

These functions are used for writing data to the card using emulation of the linear address space. The method for proving authenticity is determined by the suffix in the functions names.

Function declaration (C language)

```
UFR_STATUS LinearWrite(uint8_t *Data,
                       uint16_t linear_address,
                       uint16_t length,
                       uint16_t *bytes_returned,
                       uint8_t auth_mode,
                       uint8_t key_index);

UFR_STATUS LinearWrite_AKM1(uint8_t *Data,
                            uint16_t linear_address,
                            uint16_t length,
                            uint16_t *bytes_returned,
                            uint8_t auth_mode);

UFR_STATUS LinearWrite_AKM2(uint8_t *Data,
                            uint16_t linear_address,
                            uint16_t length,
                            uint16_t *bytes_returned,
                            uint8_t auth_mode);

UFR_STATUS LinearWrite_PK(uint8_t *Data,
                          uint16_t linear_address,
                          uint16_t length,
                          uint16_t *bytes_returned,
                          uint8_t auth_mode,
                          const uint8_t *key);
```

Parameters

data	Pointer to array of bytes containing data
linear_address	Address of byte – where to start writing
length	Length of data – how many bytes to write
bytes_returned	Pointer to variable holding how many bytes are returned
auth_mode	**For Mifare Classic** tags defines whether to perform authentication with key A or key B: use KeyA - MIFARE_AUTHENT1A = 0x60 or KeyB - MIFARE_AUTHENT1B = 0x61 **For NTAG 21x, Ultralight EV1 and other T2T tags supporting PWD_AUTH** value 0x61 means "***use PWD_AUTH***" with LinearWrite() or LinearWrite_PK() functions. Value 0x60 with LinearWrite() or LinearWrite_PK() functions means "***without PWD_AUTH***" and in that case *you can send for ucReaderKeyIndex* or *aucProvidedKey parameters anything you want without influence on the result. For NTAG 21x, Ultralight EV1 and other T2T tags supporting PWD_AUTH you can use _AKM1 or _AKM2 function variants only* ***without PWD_AUTH*** *in any case of the valid values (0x60 or 0x61) provided for this parameter.*
key_index	Index of reader's key to be used (RK mode)
key	Pointer to 6 byte array containing key bytes (PK mode)

When using this function with other card types, `auth_mode`, `key_index` and `key` parameters are not relevant but must take default values.

LinRowRead

Function description

Read Linear data Address Space. On the contrary of LinearRead functions, this functions read whole card including trailer blocks and manufacturer block.

This function is useful when making “dump” of whole card.

Group of functions for linear reading in uFR firmware utilise FAST_READ ISO 14443-3 command with NTAG21x and Mifare Ultralight EV1 tags.

Function declaration (C language)

```
UFR_STATUS LinRowRead(uint8_t *Data,
                      uint16_t linRow_address,
                      uint16_t length,
                      uint16_t *bytes_returned,
                      uint8_t auth_mode,
                       uint8_t key_index);

UFR_STATUS LinRowRead_AKM1(uint8_t *Data,
                           uint16_t linRow_address,
                           uint16_t length,
                           uint16_t *bytes_returned,
                           uint8_t auth_mode);

UFR_STATUS LinRowRead_AKM2(uint8_t *Data,
                           uint16_t linRow_address,
                           uint16_t length,
                           uint16_t *bytes_returned,
                           uint8_t auth_mode);

UFR_STATUS LinRowRead_PK(uint8_t *Data,
                         uint16_t linRow_address,
                         uint16_t length,
```

```
                                    uint16_t *bytes_returned,
                                    uint8_t auth_mode,
                                    const uint8_t *key);
```

Parameters

data	Pointer to array of bytes containing data
linear_address	Address of byte – where to start reading
length	Length of data – how many bytes to read
bytes_returned	Pointer to variable holding how many bytes are returned
auth_mode	**For Mifare Classic** tags defines whether to perform authentication with key A or key B: use KeyA - MIFARE_AUTHENT1A = 0x60 or KeyB - MIFARE_AUTHENT1B = 0x61 **For NTAG 21x, Ultralight EV1 and other T2T tags supporting PWD_AUTH** value 0x61 means "***use PWD_AUTH***" with LinRowRead() or LinRowRead_PK() functions. Value 0x60 with LinRowRead() or LinRowRead_PK() functions means "***without PWD_AUTH***" and in that case *you can send for ucReaderKeyIndex* or *aucProvidedKey parameters anything you want without influence on the result. For NTAG 21x, Ultralight EV1 and other T2T tags supporting PWD_AUTH you can use _AKM1 or _AKM2 function variants only* ***without PWD_AUTH*** *in any case of the valid values (0x60 or 0x61) provided for this parameter.*
key_index	Index of reader's key to be used (RK mode)
key	Pointer to 6 byte array containing key bytes (PK mode)

When using this function with other card types, `auth_mode`, `key_index` and `key` parameters are not relevant but they must take default values.

LinearFormatCard

Function description

This function is specific to Mifare Classic cards only. It performs "Format card" operation - write new Sector Trailer values on whole card at once. It writes following data:

KeyA, Block Access Bits, Trailer Access Bits, GeneralPurposeByte(GPB), KeyB, same as construction of Sector Trailer.

Bytes 0 – 5	Bytes 6 - 8	Byte 9	Bytes 10 - 15
KeyA	Block Access & Trailer Access Bits	GPB	KeyB

For more information, please refer to Mifare Classic Keys and Access Conditions in this document.

Function declaration (C language)

```
UFR_STATUS LinearFormatCard(const uint8_t *new_key_A,
                            uint8_t blocks_access_bits,
                            uint8_t sector_trailers_access_bits,
                            uint8_t sector_trailers_byte9,
                            const uint8_t *new_key_B,
                            uint8_t *lpucSectorsFormatted,
                            uint8_t auth_mode,
                            uint8_t key_index);

UFR_STATUS LinearFormatCard_AKM1(const uint8_t *new_key_A,
                                 uint8_t blocks_access_bits,
                                 uint8_t sector_trailers_access_bits,
                                 uint8_t sector_trailers_byte9,
                                 const uint8_t *new_key_B,
                                 uint8_t *lpucSectorsFormatted,
                                 uint8_t auth_mode);

UFR_STATUS LinearFormatCard_AKM2(const uint8_t *new_key_A,
                                 uint8_t blocks_access_bits,
                                 uint8_t sector_trailers_access_bits,
                                 uint8_t sector_trailers_byte9,
                                 const uint8_t *new_key_B,
                                 uint8_t *lpucSectorsFormatted,
                                 uint8_t auth_mode);

UFR_STATUS LinearFormatCard_PK(const uint8_t *new_key_A,
                               uint8_t blocks_access_bits,
                               uint8_t sector_trailers_access_bits,
                               uint8_t sector_trailers_byte9,
                               const uint8_t *new_key_B,
                               uint8_t *lpucSectorsFormatted,
                               uint8_t auth_mode,
                               const uint8_t *key);
```

These functions are used for new keys A and B writing as well as access bits in the trailers of all card sectors. Ninth bit setting is enabled. The same value is set for the entire card. If you need to prove authenticity on the base of previous keys, these functions are suitable to initialize the new card or re-initialize the card with same keys and access rights for all sectors.

Parameters

new_key_A	Pointer on 6 bytes array containing a new KeyA
blocks_access_bits	Block Access permissions bits. Values 0 to 7
sector_trailers_access_bits	Sector Trailer Access permissions bits. Values 0 to 7
sector_trailers_byte9	GPB value
new_key_B	Pointer on 6 bytes array containing a new KeyA
lpucSectorsFormatted	Pointer to variable holding return value how many sectors are successfully formatted

auth_mode	Defines whether to perform authentication with key A or key B: use KeyA - MIFARE_AUTHENT1A = 0x60 or KeyB - MIFARE_AUTHENT1B = 0x61
key_index	Index of reader's key to be used (RK mode)
key	Pointer to 6 byte array containing key bytes (PK mode)

This function can't be used with other card types except Mifare Classic.

GetCardSize

Function description

Function returns size of user data space on the card (LinearSize), and size of total data space on the card (RawSize). The user data space is accessed via functions LinearWrite and LinearRead. Total data space is accessed via functions LinRowWrite and LinRowRead. For example Mifare Classic 1K card have 752 bytes of user data space (sector trailers and block 0 are not included), and 1024 bytes of total data space.

Function declaration (C language)

```
UFR_STATUS GetCardSize(uint32_t *lpulLinearSize,
                       uint32_t *lpulRawSize);
```

Parameters

lpulLinearSize	pointer to variable which contain size of user data space
lpulRawSize	pointer to variable which contain size of total data space

SectorTrailerWrite

Function description

This function is specific to Mifare Classic cards only. It writes new Sector Trailer value at one Sector Trailer. It writes following data:

KeyA, Block Access Bits, Trailer Access Bits, GeneralPurposeByte(GPB), KeyB, same as construction of Sector Trailer.

Function declaration (C language)

```
UFR_STATUS SectorTrailerWrite(uint8_t addressing_mode,
                              uint8_t address,
                              const uint8_t *new_key_A,
                              uint8_t block0_access_bits,
                              uint8_t block1_access_bits,
                              uint8_t block2_access_bits,
                              uint8_t sector_trailers_access_bits,
                              uint8_t sector_trailers_byte9,
                              const uint8_t *new_key_B,
                              uint8_t auth_mode,
                              uint8_t key_index);

UFR_STATUS SectorTrailerWrite_AKM1(uint8_t addressing_mode,
                                   uint8_t address,
                                   const uint8_t *new_key_A,
                                   uint8_t block0_access_bits,
                                   uint8_t block1_access_bits,
                                   uint8_t block2_access_bits,
                                   uint8_t sector_trailers_access_bits,
                                   uint8_t sector_trailers_byte9,
                                   const uint8_t *new_key_B,
                                   uint8_t auth_mode);

UFR_STATUS SectorTrailerWrite_AKM2(uint8_t addressing_mode,
                                   uint8_t address,
                                   const uint8_t *new_key_A,
                                   uint8_t block0_access_bits,
                                   uint8_t block1_access_bits,
                                   uint8_t block2_access_bits,
                                   uint8_t sector_trailers_access_bits,
                                   uint8_t sector_trailers_byte9,
                                   const uint8_t *new_key_B,
                                   uint8_t auth_mode);

UFR_STATUS SectorTrailerWrite_PK(uint8_t addressing_mode,
                                 uint8_t address,
                                 const uint8_t *new_key_A,
                                 uint8_t block0_access_bits,
                                 uint8_t block1_access_bits,
                                 uint8_t block2_access_bits,
                                 uint8_t sector_trailers_access_bits,
                                 uint8_t sector_trailers_byte9,
                                 const uint8_t *new_key_B,
                                 uint8_t auth_mode,
                                 const uint8_t *key);
```

Parameters

addressing_mode	Defines if Absolute (0) or Relative (1) Block Addressing mode is used
address	Address of Trailer according to addressing_mode
new_key_A	Pointer on 6 bytes array containing a new KeyA
block0_access_bits	Access Permissions Bits for Block 0. Values 0 to 7
block1_access_bits	Access Permissions Bits for Block 1. Values 0 to 7
block2_access_bits	Access Permissions Bits for Block 2. Values 0 to 7
sector_trailers_access_bits	Sector Trailer Access permissions bits. Values 0 to 7
sector_trailers_byte9	GPB value
new_key_B	Pointer on 6 bytes array containing a new KeyB
auth_mode	Defines whether to perform authentication with key A or key B: use KeyA - MIFARE_AUTHENT1A = 0x60 or KeyB - MIFARE_AUTHENT1B = 0x61
key_index	Index of reader's key to be used (RK mode)
key	Pointer to 6 byte array containing key bytes (PK mode)

This function can't be used with other card types except Mifare Classic.

For "Block Access Bits" please refer to Mifare Classic Keys and Access Conditions in this document.

For Mifare Classic 4K (MF1S70), in higher addresses range (Sectors 31 - 39), where one sector has 16 blocks, `block0_access_bits` corresponds to blocks 0-4, `block1_access_bits` corresponds to blocks 5-9 and `block2_access_bits` corresponds to blocks 10-15.

SectorTrailerWriteUnsafe

Function description

This function is specific to Mifare Classic cards only. It writes new Sector Trailer value at one Sector Trailer. It writes following data:

KeyA, Block Access Bits, Trailer Access Bits, GeneralPurposeByte(GPB), KeyB, same as construction of Sector Trailer.

Difference between this function and SectorTrailerWrite is :

- `SectorTrailerWrite` will check parameters and "safely" write them into trailer, non valid values will not be written
- `SectorTrailerWriteUnsafe` writes array of 16 bytes as raw binary trailer representation, any value can be written.

USE THIS FUNCTION WITH CAUTION, WRONG VALUES CAN DESTROY CARD!

Function declaration (C language)

```
UFR_STATUS SectorTrailerWriteUnsafe(uint8_t addressing_mode,
                                    uint8_t address,
                                    uint8_t *sector_trailer,
                                    uint8_t auth_mode,
                                    uint8_t key_index);

UFR_STATUS SectorTrailerWriteUnsafe_AKM1(uint8_t addressing_mode,
                                         uint8_t address,
                                         uint8_t *sector_trailer,
                                         uint8_t auth_mode);

UFR_STATUS SectorTrailerWriteUnsafe_AKM2(uint8_t addressing_mode,
                                         uint8_t address,
                                         uint8_t *sector_trailer,
                                         uint8_t auth_mode);

UFR_STATUS SectorTrailerWriteUnsafe_PK(uint8_t addressing_mode,
                                       uint8_t address,
                                       uint8_t *sector_trailer,
                                       uint8_t auth_mode,
                                       const uint8_t *key);
```

Parameters

addressing_mode	Defines if Absolute (0) or Relative (1) Block Addressing mode is used
address	Address of Trailer according to addressing_mode
sector_trailers	Pointer to 16 byte array as binary representation of Sector Trailer
auth_mode	Defines whether to perform authentication with key A or key B: use KeyA - MIFARE_AUTHENT1A = 0x60 or KeyB - MIFARE_AUTHENT1B = 0x61
key_index	Index of reader's key to be used (RK mode)
key	Pointer to 6 byte array containing key bytes (PK mode)

This function can't be used with other card types except Mifare Classic.

ValueBlockRead

Function description

Read particular Value block using absolute Block address. This function uses Mifare Classic specific mechanism of reading value which is stored into whole block. Value blocks have a fixed data format which permits error detection and correction and a backup management. Value is a signed 4-byte value and it is stored three times, twice non-inverted and once inverted. Negative numbers are stored in standard 2's complement format. For more info, please refer to Mifare Classic documentation.

Function declaration (C language)

```
UFR_STATUS ValueBlockRead(int32_t *value,
                          uint8_t *value_addr,
                          uint8_t block_address,
                          uint8_t auth_mode,
                          uint8_t key_index);

UFR_STATUS ValueBlockRead_AKM1(int32_t *value,
                               uint8_t *value_addr,
                               uint8_t block_address,
                               uint8_t auth_mode);

UFR_STATUS ValueBlockRead_AKM2(int32_t *value,
                               uint8_t *value_addr,
                               uint8_t block_address,
                               uint8_t auth_mode);

UFR_STATUS ValueBlockRead_PK(int32_t *value,
                             uint8_t *value_addr,
                             uint8_t block_address,
                             uint8_t auth_mode,
                             const uint8_t *key);
```

Parameters

value	Pointer to variable where retrieved value will be stored
Value_addr	Signifies a 1-byte address, which can be used to save the storage address of a block, when implementing a powerful backup management. For more info, please refer to Mifare Classic documentation.
block_address	Absolute block address
auth_mode	Defines whether to perform authentication with key A or key B: use KeyA - MIFARE_AUTHENT1A = 0x60 or KeyB - MIFARE_AUTHENT1B = 0x61
key_index	Index of reader's key to be used (RK mode)
key	Pointer to 6 byte array containing key bytes (PK mode)

This functions can't be used with other card types except Mifare Classic.

ValueBlockWrite

Function description

Write particular Value block using absolute Block address. This function uses Mifare Classic specific mechanism of writing value which is stored into whole block. Value blocks have a fixed data format which permits error detection and correction and a backup management. Value is a signed 4-byte value and it is stored three times, twice non-inverted and once inverted. Negative numbers are stored in standard 2's complement format. For more info, please refer to Mifare Classic documentation.

Function declaration (C language)

```
UFR_STATUS ValueBlockWrite(int32_t *value,
                           uint8_t *value_addr,
                           uint8_t block_address,
                           uint8_t auth_mode,
                           uint8_t key_index);
UFR_STATUS ValueBlockWrite_AKM1(int32_t *value,
                                uint8_t *value_addr,
                                uint8_t block_address,
                                uint8_t auth_mode);
UFR_STATUS ValueBlockWrite_AKM2(int32_t *value,
                                uint8_t *value_addr,
                                uint8_t block_address,
                                uint8_t auth_mode);
UFR_STATUS ValueBlockWrite_PK(int32_t *value,
                              uint8_t *value_addr,
                              uint8_t block_address,
                              uint8_t auth_mode,
                              const uint8_t *key);
```

Parameters

value	Pointer to value to be stored
Value_addr	Signifies a 1-byte address, which can be used to save the storage address of a block, when implementing a powerful backup management. For more info, please refer to Mifare Classic documentation.
block_address	Absolute block address
auth_mode	Defines whether to perform authentication with key A or key B: use KeyA - MIFARE_AUTHENT1A = 0x60 or KeyB - MIFARE_AUTHENT1B = 0x61
key_index	Index of reader's key to be used (RK mode)
key	Pointer to 6 byte array containing key bytes (PK mode)

This function can't be used with other card types except Mifare Classic.

ValueBlockInSectorRead

Function description

Read particular Value block using absolute Block address. This function uses Mifare Classic specific mechanism of reading value which is stored into whole block. Value blocks have a fixed data format which permits error detection and correction and a backup management. Value is a signed 4-byte value and it is stored three times, twice non-inverted and once inverted. Negative numbers are stored in standard 2's complement format. For more info, please refer to Mifare Classic documentation.

Function declaration (C language)

```
UFR_STATUS ValueBlockInSectorRead(int32_t *value,
                                  uint8_t *value_addr,
                                  uint8_t sector_address,
                                  uint8_t block_in_sector_address,
                                  uint8_t auth_mode,
                                  uint8_t key_index);

UFR_STATUS ValueBlockInSectorRead_AKM1(int32_t *value,
                                  uint8_t *value_addr,
                                  uint8_t sector_address,
                                  uint8_t block_in_sector_address,
                                  uint8_t auth_mode);

UFR_STATUS ValueBlockInSectorRead_AKM2(int32_t *value,
                                  uint8_t *value_addr,
                                  uint8_t sector_address,
                                  uint8_t block_in_sector_address,
                                  uint8_t auth_mode);

UFR_STATUS ValueBlockInSectorRead_PK(int32_t *value,
                                  uint8_t *value_addr,
                                  uint8_t sector_address,
                                  uint8_t block_in_sector_address,
                                  uint8_t auth_mode,
                                  const uint8_t *key);
```

Parameters

value	Pointer to variable where retrieved value will be stored
Value_addr	Signifies a 1-byte address, which can be used to save the storage address of a block, when implementing a powerful backup management. For more info, please refer to Mifare Classic documentation.
sector_address	Absolute Sector address
block_in_sector_address	Block address in Sector
auth_mode	Authentication mode : use `KeyA - MIFARE_AUTHENT1A = 0x60` or `KeyB - MIFARE_AUTHENT1B = 0x61`
key_index	Index of reader's key to be used (RK mode)
key	Pointer to 6 byte array containing key bytes (PK mode)

This function can't be used with other card types except Mifare Classic.

ValueBlockInSectorWrite

Function description

Write particular Value block using absolute Block address. This function uses Mifare Classic specific mechanism of writing value which is stored into whole block. Value blocks have a fixed data format which permits error detection and correction and a backup management. Value is a signed 4-byte value and it is stored three times, twice non-inverted and once inverted. Negative numbers are stored in standard 2's complement format. For more info, please refer to Mifare Classic documentation.

Function declaration (C language)

```
UFR_STATUS ValueBlockInSectorWrite(int32_t value,
                                   uint8_t value_addr,
                                   uint8_t sector_address,
                                   uint8_t block_in_sector_address,
                                   uint8_t auth_mode,
                                   uint8_t key_index);

UFR_STATUS ValueBlockInSectorWrite_AKM1(int32_t value,
                                   uint8_t value_addr,
                                   uint8_t sector_address,
                                   uint8_t block_in_sector_address,
                                   uint8_t auth_mode);

UFR_STATUS ValueBlockInSectorWrite_AKM2(int32_t value,
                                   uint8_t value_addr,
                                   uint8_t sector_address,
                                   uint8_t block_in_sector_address,
                                   uint8_t auth_mode);

UFR_STATUS ValueBlockInSectorWrite_PK(int32_t value,
                                   uint8_t value_addr,
                                   uint8_t sector_address,
                                   uint8_t block_in_sector_address,
                                   uint8_t auth_mode,
                                   const uint8_t *key);
```

Parameters

value	Pointer to value to be stored
Value_addr	Signifies a 1-byte address, which can be used to save the storage address of a block, when implementing a powerful backup management. For more info, please refer to Mifare Classic documentation.
sector_address	Absolute Sector address
block_in_sector_address	Block address in Sector

auth_mode	Authentication mode : use `KeyA - MIFARE_AUTHENT1A = 0x60` or `KeyB - MIFARE_AUTHENT1B = 0x61`
key_index	Index of reader's key to be used (RK mode)
key	Pointer to 6 byte array containing key bytes (PK mode)

This function can't be used with other card types except Mifare Classic.

ValueBlockIncrement

Function description

Increments particular Value block with specified value using absolute Block address.

Function declaration (C language)

```
UFR_STATUS ValueBlockIncrement(int32_t increment_value,
                               uint8_t block_address,
                               uint8_t auth_mode,
                               uint8_t key_index);

UFR_STATUS ValueBlockIncrement_AKM1(int32_t increment_value,
                                    uint8_t block_address,
                                    uint8_t auth_mode;

UFR_STATUS ValueBlockIncrement_AKM2(int32_t increment_value,
                                    uint8_t block_address,
                                    uint8_t auth_mode);

UFR_STATUS ValueBlockIncrement_PK(int32_t increment_value,
                                  uint8_t block_address,
                                  uint8_t auth_mode,
                                  const uint8_t *key);
```

Parameters

increment_value	value showing how much initial block value will be incremented
block_address	Absolute block address
auth_mode	Authentication mode : use `KeyA - MIFARE_AUTHENT1A = 0x60` or `KeyB - MIFARE_AUTHENT1B = 0x61`
key_index	Index of reader's key to be used (RK mode)
key	Pointer to 6 byte array containing key bytes (PK mode)

This function can't be used with other card types except Mifare Classic.

ValueBlockDecrement

Function description

Decrements particular Value block with specified value using absolute Block address.

Function declaration (C language)

```
UFR_STATUS ValueBlockDecrement(int32_t decrement_value,
                               uint8_t block_address,
                               uint8_t auth_mode,
                               uint8_t key_index);

UFR_STATUS ValueBlockDecrement_AKM1(int32_t decrement_value,
                                    uint8_t block_address,
                                    uint8_t auth_mode;

UFR_STATUS ValueBlockDecrement_AKM2(int32_t decrement_value,
                                    uint8_t block_address,
                                    uint8_t auth_mode);

UFR_STATUS ValueBlockDecrement_PK(int32_t decrement_value,
                                  uint8_t block_address,
                                  uint8_t auth_mode,
                                  const uint8_t *key);
```

Parameters

increment_value	value showing how much initial block value will be decremented
block_address	Absolute block address
auth_mode	Authentication mode : use `KeyA - MIFARE_AUTHENT1A = 0x60` or `KeyB - MIFARE_AUTHENT1B = 0x61`
key_index	Index of reader's key to be used (RK mode)
key	Pointer to 6 byte array containing key bytes (PK mode)

This function can't be used with other card types except Mifare Classic.

ValueBlockInSectorIncrement

Function description

Increments particular Value block with specified value using Block in Sector address.

Function declaration (C language)

```
UFR_STATUS
ValueBlockInSectorIncrement(int32_t increment_value,
                            uint8_t sector_address,
                            uint8_t block_in_sector_address,
                            uint8_t auth_mode,
                            uint8_t key_index);

UFR_STATUS
ValueBlockInSectorIncrement_AKM1(int32_t increment_value,
                                 uint8_t sector_address,
                                 uint8_t block_in_sector_address,
                                 uint8_t auth_mode);

UFR_STATUS
ValueBlockInSectorIncrement_AKM2(int32_t increment_value,
                                 uint8_t sector_address,
                                 uint8_t block_in_sector_address,
                                 uint8_t auth_mode);

UFR_STATUS
ValueBlockInSectorIncrement_PK(int32_t increment_value,
                               uint8_t sector_address,
                               uint8_t block_in_sector_address,
                               uint8_t auth_mode,
                               const uint8_t *key);
```

Parameters

increment_value	value showing how much initial block value will be incremented
sector_address	Absolute Sector address
block_in_sector_address	Block address in Sector
auth_mode	Authentication mode : use `KeyA - MIFARE_AUTHENT1A = 0x60` or `KeyB - MIFARE_AUTHENT1B = 0x61`
key_index	Index of reader's key to be used (RK mode)
key	Pointer to 6 byte array containing key bytes (PK mode)

This function can't be used with other card types except Mifare Classic.

ValueBlockInSectorDecrement

Function description

Decrements particular Value block with specified value using Block in Sector address.

Function declaration (C language)

```
UFR_STATUS
ValueBlockInSectorDecrement(int32_t decrement_value,
                            uint8_t sector_address,
                            uint8_t block_in_sector_address,
                            uint8_t auth_mode,
                            uint8_t key_index);

UFR_STATUS
ValueBlockInSectorDecrement_AKM1(int32_t decrement_value,
                                 uint8_t sector_address,
                                 uint8_t block_in_sector_address,
                                 uint8_t auth_mode);

UFR_STATUS
ValueBlockInSectorDecrement_AKM2(int32_t decrement_value,
                                 uint8_t sector_address,
                                 uint8_t block_in_sector_address,
                                 uint8_t auth_mode);

UFR_STATUS
ValueBlockInSectorDecrement_PK(int32_t decrement_value,
                               uint8_t sector_address,
                               uint8_t block_in_sector_address,
                               uint8_t auth_mode,
                               const uint8_t *key);
```

Parameters

decrement_value	value showing how much initial block value will be decremented
sector_address	Absolute Sector address
block_in_sector_address	Block address in Sector
auth_mode	Authentication mode : use `KeyA - MIFARE_AUTHENT1A = 0x60` or `KeyB - MIFARE_AUTHENT1B = 0x61`
key_index	Index of reader's key to be used (RK mode)
key	Pointer to 6 byte array containing key bytes (PK mode)

This function can't be used with other card types except Mifare Classic.

Additional general functions for working with the cards

Functions that support NDEF records

get_ndef_record_count

Function description

Function returns the number of NDEF messages that have been read from the card, and number of NDEF records, number of NDEF empty messages. Also, function returns array of bytes containing number of messages pairs. First byte of pair is message ordinal, and second byte is number of NDEF records in that message. Message ordinal starts from 1.

Function declaration (C language)

```
UFR_STATUS get_ndef_record_count(
               uint8_t *ndef_message_cnt,
               uint8_t *ndef_record_cnt,
               uint8_t *ndef_record_array,
               uint8_t *empty_ndef_message_cnt);
```

Parameters

`ndef_message_cnt`	pointer to the variable containing number of NDEF messages
`ndef_record_cnt`	pointer to the variable containing number of NDEF record
`ndef_record_array`	pointer to the array of bytes containing pairs (message ordinal – number of records)
`empty_ndef_message_cnt`	pointer to the variable containing number of empty messages

read_ndef_record

Function description

Function returns TNF, type of record, ID and payload from the NDEF record. NDEF record shall be elected by the message ordinal and record ordinal in this message.

Function declaration (C language)

```
UFR_STATUS read_ndef_record(uint8_t message_nr,
                uint8_t record_nr,
                uint8_t *tnf,
                uint8_t *type_record,
                uint8_t *type_length,
                uint8_t *id,
                uint8_t *id_length,
                uint8_t *payload,
                uint32_t *payload_length);
```

Parameters

message_nr	NDEF message ordinal (starts from 1)
record_nr	NDEF record ordinal (in message)
tnf	pointer to the variable containing TNF of record
type_record	pointer to array containing type of record
type_length	pointer to the variable containing length of type of record string
id	pointer to array containing ID of record
id_length	pointer to the variable containing length of ID of record string
payload	pointer to array containing payload of record
payload_length	pointer to the variable containing length of payload

write_ndef_record

Function description

Function adds a record to the end of message, if one or more records already exist in this message. If current message is empty, then this empty record will be replaced with the record. Parameters of function are: ordinal of message, TNF, type of record, ID, payload. Function also returns pointer to the variable which reported that the card formatted for NDEF using (card does not have a capability container, for example new Mifare Ultralight, or Mifare Classic card).

Function declaration (C language)

```
UFR_STATUS write_ndef_record(uint8_t message_nr,
                             uint8_t *tnf,
                             uint8_t *type_record,
                             uint8_t *type_length,
                             uint8_t *id,
                             uint8_t *id_length,
                             uint8_t *payload,
                             uint32_t *payload_length,
                             uint8_t *card_formated);
```

Parameters

message_nr	NDEF message ordinal (starts from 1)
tnf	pointer to variable containing TNF of record
type_record	pointer to array containing type of record
type_length	pointer to the variable containing length of type of record string
id	pointer to array containing ID of record
id_length	pointer to the variable containing length of ID of record string
payload	pointer to array containing payload of record
payload_length	pointer to the variable containing length of payload
card_formated	pointer to the variable which shows that the card formatted for NDEF using.

write_ndef_record_mirroring

Function description

This function works the same as the **write_ndef_record()**, with the additional “UID and / or NFC counter mirror” features support. NTAG 21x family of the devices offers these specific features. For details about “ASCII mirror” features refer to http://www.nxp.com/docs/en/data-sheet/NTAG213_215_216.pdf (in Rev. 3.2 from 2. June 2015,

page 20) and http://www.nxp.com/docs/en/data-sheet/NTAG210_212.pdf (in Rev. 3.0 from 14. March 2013, page 16).

Function declaration (C language)

```
UFR_STATUS write_ndef_record_mirroring(uint8_t message_nr,
                                       uint8_t *tnf,
                                       uint8_t *type_record,
                                       uint8_t *type_length,
                                       uint8_t *id,
                                       uint8_t *id_length,
                                       uint8_t *payload,
                                       uint32_t *payload_length,
                                       uint8_t *card_formated,
                                       int use_uid_ascii_mirror,
                                       int use_counter_ascii_mirror,
                                       uint32_t payload_mirroring_pos);
```

Parameters

message_nr	NDEF message ordinal (starts from 1)
tnf	pointer to variable containing TNF of record
type_record	pointer to array containing type of record
type_length	pointer to the variable containing length of type of record string
id	pointer to array containing ID of record
id_length	pointer to the variable containing length of ID of record string
payload	pointer to array containing payload of record
payload_length	pointer to the variable containing length of payload
card_formated	pointer to the variable which shows that the card formatted for NDEF using.
use_uid_ascii_mirror	if `use_uid_ascii_mirror == 1` then “UID ASCII Mirror” feature is in use. if `use_uid_ascii_mirror == 0` then “UID ASCII Mirror” feature is switched off.

use_counter_ascii_mirror	if `use_counter_ascii_mirror == 1` then “NFC counter ASCII Mirror” feature is in use. if `use_counter_ascii_mirror == 0` then “NFC counter ASCII Mirror” feature is switched off.
payload_mirroring_pos	Defines the starting position of the “ASCII Mirror” in to the NDEF record payload.

erase_last_ndef_record

Function description

Function deletes the last record of selected message. If message contains one record, then it will be written empty message.

Function declaration (C language)

```
UFR_STATUS erase_last_ndef_record(uint8_t message_nr);
```

Parameter

message_nr	NDEF message ordinal (starts form 1)

erase_all_ndef_records

Function description

Function deletes all records of message, then writes empty message.

Function declaration (C language)

```
UFR_STATUS erase_all_ndef_records(uint8_t message_nr);
```

Parameter

message_nr	NDEF message ordinal (starts form 1)

ndef_card_initialization

Function description

Function prepares the card for NDEF using. Function writes Capability Container (CC) if necessary, and writes empty message. If card is MIFARE CLASSIC or MIFARE PLUS, then

function writes MAD (MIFARE Application Directory), and default keys and access bits for NDEF using.

Function declaration (C language)

```
UFR_STATUS ndef_card_initialization(void);
```

ERROR CODES OF NDEF FUNCTIONS

```
UFR_WRONG_NDEF_CARD_FORMAT = 0x80
UFR_NDEF_MESSAGE_NOT_FOUND = 0x81
UFR_NDEF_UNSUPPORTED_CARD_TYPE = 0x82
UFR_NDEF_CARD_FORMAT_ERROR = 0x83
UFR_MAD_NOT_ENABLED = 0x84
UFR_MAD_VERSION_NOT_SUPPORTED = 0x85
```

Functions for configuration of asynchronously card ID sending

When the card put on the reader, then the string which contains card ID shall be sent. String contains hexadecimal notation of card ID, after that is one mandatory suffix character. Before the card ID may be one prefix character placed.

Example:

Card ID is 0xA103C256, prefix is 0x58 ('X'), suffix is 0x59 ('Y')

String is “XA103C256Y”

SetAsyncCardIdSendConfig

Function description

Function sets the parameters of card ID sending. Parameters are: prefix existing, prefix character, suffix character, and baud rate for card ID sending.

Function declaration (C language)

```
UFR_STATUS SetAsyncCardIdSendConfig(uint8_t send_enable,
                                    uint8_t prefix_enable,
                                    uint8_t prefix,
                                    uint8_t suffix,
                                    uint32_t async_baud_rate);
```

Parameters

send_enable	sending enable flag (0 – disabled, 1 – enabled)
prefix_enable	prefix existing flag (0 – prefix don't exist, 1 – prefix exist)

prefix	prefix character
suffix	suffix character
async_baud_rate	• baud rate value (e.g. 9600)

GetAsyncCardIdSendConfig

Function description

Function returns the parameters of card ID sending.

Function declaration (C language)

```
UFR_STATUS GetAsyncCardIdSendConfig(uint8_t *send_enable,
                                    uint8_t *prefix_enable,
                                    uint8_t *prefix,
                                    uint8_t *suffix,
                                    uint32_t *async_baud_rate);
```

Parameters

send_enable	pointer to the sending enable flag
prefix_enable	pointer to the prefix existing flag
prefix	pointer to the prefix variable
suffix	pointer to the suffix variable
async_baud_rate	pointer to the baud rate variable

Functions that works with Real Time Clock (RTC)

RTC embedded in uFR Advance device only.

GetReaderTime

Function description

Function returns 6 bytes array of uint8_t that represented current date and time into device's RTC.

- Byte 0 represent year (current year – 2000)
- Byte 1 represent month (1 – 12)

- Byte 2 represent day of the month (1 – 31)
- Byte 3 represent hour (0 – 23)
- Byte 4 represent minute (0 – 59)
- Byte 5 represent second (0 – 59)

Function declaration (C language)

```
UFR_STATUS GetReaderTime(uint8_t *time);
```

Parameter

time	pointer to the array containing current date and time representation

SetReaderTime

Function description

Function sets the date and time into device's RTC. Function requires the 8 bytes password entry to set date and time. Date and time are represent into 6 bytes array in same way as in GetReaderTime function. Factory password is “11111111” (0x31, 0x31, 0x31, 0x31, 0x31, 0x31, 0x31, 0x31).

Function declaration (C language)

```
UFR_STATUS SetReaderTime(uint8_t *password,
                         uint8_t *time);
```

Parameters

password	pointer to the 8 bytes array containing password
time	pointer to the 6 bytes array containing date and time representation

ChangeReaderPassword

Function description

Function changes password for set date and time. Function's parameters are old password and new password.

Function declaration (C language)

```
UFR_STATUS ChangeReaderPassword(uint8_t *old_password,
                                uint8_t *new_password);
```

Parameters

`old_password`	pointer to the 8 bytes array containing current password
`new_password`	pointer to the 8 bytes array containing new password

Functions that works with EEPROM

EEPROM embedded in uFR Advance device only.

Range of user address is from 0 to 32750.

ReaderEepromRead

Function description

Function returns array of data read from EEPROM. Maximal length of array is 128 bytes.

Function declaration (C language)

```
UFR_STATUS ReaderEepromRead(uint8_t *data,
                            uint32_t address,
                            uint32_t size);
```

Parameters

`data`	pointer to array containing data from EEPROM
`address`	address of first data
`size`	length of array

ReaderEepromWrite

Function description

Function writes array of data into EEPROM. Maximal length of array is 128 bytes. Function requires password which length is 8 bytes. Factory password is “11111111” (0x31, 0x31, 0x31, 0x31, 0x31, 0x31, 0x31, 0x31).

Function declaration (C language)

```
UFR_STATUS ReaderEepromWrite(uint8_t *data,
                             uint32_t address,
                             uint32_t size,
                             uint8_t *password);
```

Parameters

data	pointer to array containing data
address	address of first data
size	length of array
password	pointer to array containing password

Functions that works with Mifare Desfire Card (AES encryption in reader)

AES encryption and decryption is performed in the reader. AES keys are stored into reader.

uFR_int_WriteAesKey

Function description

Function writes AES key (16 bytes) into reader.

Function declaration (C language)

```
UFR_STATUS uFR_int_DesfireWriteAesKey(uint8_t aes_key_no,
                                      uint8_t *aes_key);
```

Parameters

aes_key_no	ordinal number of AES key in the reader
aes_key	pointer to 16 byte array containing the AES key

uFR_int_GetDesfireUid

uFR_int_GetDesfireUid_PK

Function description

Mifare Desfire EV1 card can be configured to use Random ID numbers instead Unique ID numbers during anti-collision procedure. In this case card uses single anti-collision loop, and

returns Random Number Tag 0x08 and 3 bytes Random Number (4 bytes Random ID). This function returns Unique ID of card, if the Random ID is used.

Function declaration (C language)

```
UFR_STATUS uFR_int_GetDesfireUid(uint8_t aes_key_nr,
                                 uint32_t aid,
                                 uint8_t aid_key_nr,
                                 uint8_t *card_uid,
                                 uint8_t *card_uid_len,
                                 uint16_t *card_status,
                                 uint16_t *exec_time);
UFR_STATUS uFR_int_GetDesfireUid_PK(uint8_t *aes_key_ext,
                                    uint32_t aid,
                                    uint8_t aid_key_nr,
                                    uint8_t *card_uid,
                                    uint8_t *card_uid_len,
                                    uint16_t *card_status,
                                    uint16_t *exec_time);
```

Parameters

aes_key_nr	ordinal number of AES key in the reader
aes_key_ext	pointer to 16 byte array containing the AES key
aid	ID of application that uses this key (3 bytes long, 0x000000 for card master key)
aid_key_nr	key number into application (0 for card master key or application master key)
card_uid	pointer to array containing card UID
card_uid_len	pointer to card UID length variable
card_status	pointer to card error variable
exec_time	function's execution time

uFR_int_DesfireFreeMem

Function description

Function returns the available bytes on the card.

Function declaration (C language)

```
UFR_STATUS uFR_int_DesfireFreeMem(uint32_t *free_mem_byte,
                                  uint16_t *card_status,
                                  uint16_t *exec_time);
```

Parameters

free_mem_byte	pointer to free memory size variable
card_status	pointer to card error variable
exec_time	function's execution time

uFR_int_DesfireFormatCard

uFR_int_DesfireFormatCard_PK

Function description

Function releases all allocated user memory on the card. All applications will be deleted, also all files within those applications will be deleted. Only the card master key, and card master key settings will not be deleted. This operation requires authentication with the card master key.

Function declaration (C language)

```
UFR_STATUS uFR_int_DesfireFormatCard(uint8_t aes_key_nr,
                                     uint16_t *card_status,
                                     uint16_t *exec_time);
UFR_STATUS uFR_int_DesfireFormatCard_PK(uint8_t *aes_key_ext,
                                        uint16_t *card_status,
                                        uint16_t *exec_time);
```

Parameters

aes_key_nr	ordinal number of card master AES key in the reader
aes_key_ext	pointer to 16 byte array containing the AES key
card_status	pointer to card error variable
exec_time	function's execution time

uFR_int_DesfireSetConfiguration

uFR_int_DesfireSetConfiguration_PK

Function description

Function allows you to activate the Random ID option, and/or Format disable option.
If these options are activated, then they can not be returned to the factory setting (Random ID disabled, Format card enabled). This operation requires authentication with the card master key.

Function declaration (C language)

```
UFR_STATUS uFR_int_DesfireSetConfiguration(uint8_t aes_key_nr,
                                           uint8_t random_uid,
                                           uint8_t format_disable,
                                           uint16_t *card_status,
                                           uint16_t *exec_time);
UFR_STATUS uFR_int_DesfireSetConfiguration_PK(uint8_t *aes_key_ext,
                                              uint8_t random_uid,
                                              uint8_t format_disable,
                                              uint16_t *card_status,
                                              uint16_t *exec_time);
```

Parameters

aes_key_nr	ordinal number of card master AES key in the reader
aes_key_ext	pointer to 16 byte array containing the AES key
random_uid	0 – Random ID disabled, 1 – Random ID enabled
format_disable	0 – Format enabled, 1 – Format disabled
card_status	pointer to card error variable
exec_time	function's execution time

uFR_int_DesfireGetKeySettings

uFR_int_DesfireGetKeySettings_PK

Function description

Function allows to get card master key and application master key configuration settings. In addition it returns the maximum number of keys which can be stored within selected application.

Function declaration (C language)

```
UFR_STATUS uFR_int_DesfireGetKeySettings(uint8_t aes_key_nr,
                                         uint32_t aid,
                                         uint8_t *settings
                                         uint8_t *max_key_no,
                                         uint16_t *card_status,
                                         uint16_t *exec_time);
UFR_STATUS uFR_int_DesfireGetKeySettings_PK(uint8_t *aes_key_ext,
                                            uint32_t aid,
                                            uint8_t *settings
                                            uint8_t *max_key_no,
                                            uint16_t *card_status,
                                            uint16_t *exec_time);
```

Parameters

aes_key_nr	ordinal number of AES key in the reader
aes_key_ext	pointer to 16 byte array containing the AES key
aid	ID of application that uses this key (3 bytes long, 0x000000 for card master key)
settings	pointer to settings variable
max_key_no	maximum number of keys within selected application
card_status	pointer to card error variable
exec_time	function's execution time

uFR_int_DesfireChangeKeySettings

uFR_int_DesfireChangeKeySettings_PK

Function description

Function allows to set card master key, and application master key configuration settings.

Function declaration (C language)

```
UFR_STATUS uFR_int_DesfireChangeKeySettings(uint8_t aes_key_nr,
                                            uint32_t aid,
                                            uint8_t settings,
                                            uint16_t *card_status,
                                            uint16_t *exec_time);
UFR_STATUS uFR_int_DesfireChangeKeySettings_PK(uint8_t *aes_key_ext,
                                               uint32_t aid,
                                               uint8_t settings,
                                               uint16_t *card_status,
                                               uint16_t *exec_time);
```

Parameters

`aes_key_nr`	ordinal number of AES key in the reader
`aes_key_ext`	pointer to 16 byte array containing the AES key
`aid`	ID of application that uses this key (3 bytes long, 0x000000 for card master key)
`settings`	pointer to key settings variable
`card_status`	pointer to card error variable
`exec_time`	function's execution time

uFR_int_DesfireChangeAesKey

uFR_int_DesfireChangeAesKey_PK

uFR_int_DesfireChangeAesKey_A

Function description

Function allow to change any AES key on the card. Changing the card master key require current card master key authentication. Authentication for the application keys changing depend on the application master key settings (which key uses for authentication).

Function declaration (C language)

```
UFR_STATUS uFR_int_DesfireChangeAesKey(uint8_t aes_key_nr,
                                       uint32_t aid,
                                       uint8_t aid_key_nr_auth,
                                       uint8_t new_aes_key[16],
                                       uint8_t aid_key_no,
                                       uint8_t old_aes_key[16],
                                       uint16_t *card_status,
                                       uint16_t *exec_time);
UFR_STATUS uFR_int_DesfireChangeAesKey_PK(uint8_t *aes_key_ext,
                                          uint32_t aid,
                                          uint8_t aid_key_nr_auth,
                                          uint8_t new_aes_key[16],
                                          uint8_t aid_key_no,
                                          uint8_t old_aes_key[16],
                                          uint16_t *card_status,
                                          uint16_t *exec_time);
UFR_STATUS uFR_int_DesfireChangeAesKey_A(uint8_t aes_key_nr,
                                         uint32_t aid,
                                         uint8_t aid_key_no_auth,
                                         uint8_t new_aes_key_nr,
                                         uint8_t aid_key_no,
                                         uint8_t old_aes_key_nr,
                                         uint16_t *card_status,
                                         uint16_t *exec_time);
```

Parameters

aes_key_nr	ordinal number of AES key in the reader
aes_key_ext	pointer to 16 byte array containing the AES key
aid	ID of application that uses this key (3 bytes long, 0x000000 for card master key)
aid_key_nr_auth	key number into application which uses for authentication
new_aes_key[16]	16 bytes array that represent AES key
aid_key_no	key number into application that will be changed
old_aes_key[16]	16 bytes array that represent current AES key that will be changed, if this is not key by which is made authentication

card_status	pointer to card error variable
exec_time	function's execution time

uFR_int_DesfireCreateAesApplication

uFR_int_DesfireCreateAesApplication_PK

uFR_int_DesfireCreateAesApplication_no_auth

Function description

Function allows to create new application on the card. Is the card master key authentication is required, depend on the card master key settings. Maximal number of applications on the card is 28. Each application is linked to set of up 14 different user definable access keys.

Function declaration (C language)

```
UFR_STATUS uFR_int_DesfireCreateAesApplication(uint8_t aes_key_nr,
                                               uint32_t aid_nr,
                                               uint8_t setting,
                                               uint8_t max_key_no,
                                               uint16_t *card_status,
                                               uint16_t *exec_time);
UFR_STATUS uFR_int_DesfireCreateAesApplication_PK(uint8_t *aes_key_ext,
                                                  uint32_t aid_nr,
                                                  uint8_t settings,
                                                  uint8_t max_key_no,
                                                  uint16_t
*card_status,
                                                  uint16_t *exec_time);
UFR_STATUS uFR_int_DesfireCreateAesApplication_no_auth(uint32_t aid_nr,
                                                  uint8_t settings,
                                                  uint8_t max_key_no,
                                                  uint16_t
*card_status,
                                                  uint16_t *exec_time);
```

Parameter

aes_key_nr	ordinal number of card master AES key in the reader
aes_key_ext	pointer to 16 byte array containing the AES key

aid_nr	ID of application that creates (3 bytes long 0x000000 to 0xFFFFFF)
settings	application master key settings
max_key_no	maximal number of keys into application
card_status	pointer to card error variable
exec_time	function's execution time

uFR_int_DesfireDeleteApplication

uFR_int_DesfireDeleteApplication_PK

Function description

Function allows to deactivate application on the card. Is the card master key authentication is required, depend on the card master key settings. AID allocation is removed, but deleted memory blocks can only recovered by using Format card function.

Function declaration (C language)

```
UFR_STATUS uFR_int_DesfireDeleteApplication(uint8_t aes_key_nr,
                                            uint32_t aid_nr,
                                            uint16_t *card_status,
                                            uint16_t *exec_time);
UFR_STATUS uFR_int_DesfireDeleteApplication_PK(uint8_t *aes_key_ext,
                                               uint32_t aid_nr,
                                               uint16_t *card_status,
                                               uint16_t *exec_time);
```

Parameters

aes_key_nr	ordinal number of card master AES key in the reader
aes_key_ext	pointer to 16 byte array containing the AES key
aid_nr	ID of application that deletes (3 bytes long 0x000000 to 0xFFFFFF)
card_status	pointer to card error variable

exec_time	function's execution time

uFR_int_DesfireCreateStdDataFile

uFR_int_DesfireCreateStdDataFile_PK

uFR_int_DesfireCreateStdDataFile_no_auth

Function description

Function allows to create file for the storage unformatted user data within existing application on the card. Maximal number of files into application is 32. The file will be created in the currently selected application. Is the application master key authentication is required, depend on the application master key settings.

Communication settings define communication mode between reader and card. The communication modes are:

- plain communication communication settings value is 0x00
- plain communication secured by MACing communication settings value is 0x01
- fully enciphered communication communication settings value is 0x11

Access rights for read, write, read&write and changing, references certain key within application's keys (0 – 13). If value is 14, this means free access, independent of previous authentication. If value is 15, this means deny access (for example if write access is 15 then the file type is read only).

Function declaration (C language)

```
UFR_STATUS uFR_int_DesfireCreateStdDataFile(
                                    uint8_t aes_key_nr,
                                    uint32_t aid,
                                    uint8_t file_id,
                                    uint32_t file_size,
                                    uint8_t read_key_no,
                                    uint8_t write_key_no,
                                    uint8_t read_write_key_no,
                                    uint8_t change_key_no,
                                    uint8_t communication_settings,
                                    uint16_t *card_status,
                                    uint16_t *exec_time);
UFR_STATUS uFR_int_DesfireCreateStdDataFile_PK(
                                    uint8_t *aes_key_ext,
                                    uint32_t aid,
                                    uint8_t file_id,
                                    uint32_t file_size,
                                    uint8_t read_key_no,
                                    uint8_t write_key_no,
                                    uint8_t read_write_key_no,
                                    uint8_t change_key_no,
                                    uint8_t communication_settings,
                                    uint16_t *card_status,
                                    uint16_t *exec_time);
UFR_STATUS uFR_int_DesfireCreateStdDataFile_no_auth(
                                    uint32_t aid,
                                    uint8_t file_id,
                                    uint32_t file_size,
                                    uint8_t read_key_no,
                                    uint8_t write_key_no,
                                    uint8_t read_write_key_no,
                                    uint8_t change_key_no,
                                    uint8_t communication_settings,
                                    uint16_t *card_status,
                                    uint16_t *exec_time);
```

Parameters

aes_key_nr	ordinal number of AES key in the reader
aes_key_ext	pointer to 16 byte array containing the AES key
aid	ID of application that contains the file

file_id	ID of file that will be created (0 – 31)
file_size	file size in bytes
read_key_no	key for reading
write_key_no	key for writing
read_write_key_no	key for reading and writing
change_key_no	key for changing this setting
communication_settings	variable that contains communication settings
card_status	pointer to card error variable
exec_time	function's execution time

uFR_int_DesfireDeleteFile

uFR_int_DesfireDeleteFile_PK

uFR_int_DesfireDeleteFile_no_auth

Function description

Function deactivates a file within currently selected application. Allocated memory blocks associated with deleted file not set free. Only format card function can delete the memory blocks. Is the application master key authentication is required, depend on the application master key settings.

Function declaration (C language)

```
UFR_STATUS uFR_int_DesfireDeleteFile(uint8_t aes_key_nr,
                                     uint32_t aid,
                                     uint8_t file_id,
                                     uint16_t *card_status,
                                     uint16_t *exec_time);
UFR_STATUS uFR_int_DesfireDeleteFile_PK(uint8_t *aes_key_ext,
                                        uint32_t aid,
                                        uint8_t file_id,
                                        uint16_t *card_status,
                                        uint16_t *exec_time);
UFR_STATUS uFR_int_DesfireDeleteFile_no_auth(uint32_t aid,
                                             uint8_t file_id,
                                             uint16_t *card_status,
                                             uint16_t *exec_time);
```

Parameters

aes_key_nr	ordinal number of AES key in the reader
aes_key_ext	pointer to 16 byte array containing the AES key
aid	ID of application that contains the file
file_id	ID of file that will be deleted (0 – 31)
card_status	pointer to card error variable
exec_time	function's execution time

uFR_int_DesfireReadStdDataFile

uFR_int_DesfireReadStdDataFile_PK

uFR_int_DesfireReadStdDataFile_no_auth

Function description

Function allow to read data from Standard Data File, or from Backup Data File. Read command requires a preceding authentication either with the key specified for Read or Read&Write access.

Function declaration (C language)

```
UFR_STATUS uFR_int_DesfireReadStdDataFile(uint8_t aes_key_nr,
                                          uint32_t aid,
                                          uint8_t aid_key_nr,
                                          uint8_t file_id,
                                          uint16_t offset,
                                          uint16_t data_length,
                                          uint8_t
communication_settings,
                                          uint8_t *data,
                                          uint16_t *card_status,
                                          uint16_t *exec_time);
UFR_STATUS uFR_int_DesfireReadStdDataFile_PK(
                                          uint8_t *aes_key_ext,
                                          uint32_t aid,
                                          uint8_t aid_key_nr,
                                          uint8_t file_id,
                                          uint16_t offset,
                                          uint16_t data_length,
                                          uint8_t
communication_settings,
                                          uint8_t *data,
                                          uint16_t *card_status,
                                          uint16_t *exec_time);
UFR_STATUS uFR_int_DesfireReadStdDataFile_no_auth(
                                          uint32_t aid,
                                          uint8_t aid_key_nr,
                                          uint8_t file_id,
                                          uint16_t offset,
                                          uint16_t data_length,
                                          uint8_t
communication_settings,
                                          uint8_t *data,
                                          uint16_t *card_status,
                                          uint16_t *exec_time);
```

Parameters

aes_key_nr	ordinal number of AES key in the reader
aes_key_ext	pointer to 16 byte array containing the AES key
aid	ID of application that contains the file
aid_key_nr	key number into application

file_id	ID of file (0 – 31)
offset	start position for read operation within file
data_length	number of data to be read
communication_settings	value must be same as in file declaration
data	pointer to data array
card_status	pointer to card error variable
exec_time	function's execution time

uFR_int_DesfireWriteStdDataFile

uFR_int_DesfireWriteStdDataFile_PK

uFR_int_DesfireWriteStdDataFile_no_auth

Function description

Function allow to write data to Standard Data File, or to Backup Data File. Write command requires a preceding authentication either with the key specified for Write or Read&Write access.

Function declaration (C language)

```
UFR_STATUS uFR_int_DesfireWriteStdDataFile(
                                        uint8_t aes_key_nr,
                                        uint32_t aid,
                                        uint8_t aid_key_nr,
                                        uint8_t file_id,
                                        uint16_t offset,
                                        uint16_t data_length,
                                        uint8_t communication_settings,
                                        uint8_t *data,
                                        uint16_t *card_status,
                                        uint16_t *exec_time);
UFR_STATUS uFR_int_DesfireWriteStdDataFile_PK(
                                        uint8_t *aes_key_ext,
                                        uint32_t aid,
                                        uint8_t aid_key_nr,
                                        uint8_t file_id,
                                        uint16_t offset,
                                        uint16_t data_length,
                                        uint8_t communication_settings,
                                        uint8_t *data,
                                        uint16_t *card_status,
                                        uint16_t *exec_time);
UFR_STATUS uFR_int_DesfireWriteStdDataFile_no_auth(
                                        uint32_t aid,
                                        uint8_t aid_key_nr,
                                        uint8_t file_id,
                                        uint16_t offset,
                                        uint16_t data_length,
                                        uint8_t communication_settings,
                                        uint8_t *data,
                                        uint16_t *card_status,
                                        uint16_t *exec_time);
```

Parameters

aes_key_nr	ordinal number of AES key in the reader
aes_key_ext	pointer to 16 byte array containing the AES key
aid	ID of application that contains the file
aid_key_nr	key number into application

file_id	ID of file (0 – 31)
offset	start position for read operation within file
data_length	number of data to be read
communication_settings	value must be same as in file declaration
data	pointer to data array
card_status	pointer to card error variable
exec_time	function's execution time

DES_to_AES_key_type

Function description

Function allow to change the card master key type from DES to AES. Factory setting for DESFIRE card master key is DES key type, and value is 0x0000000000000000. Because the reader uses AES keys, you must change the type key on AES. New AES key is 0x00000000000000000000000000000000.

Function declaration (C language)

```
UFR_STATUS DES_to_AES_key_type(void);
```

AES_to_DES_key_type

Function description

Function allow to change the card master key type from AES to DES. Set master AES key before use this function to 0x00000000000000000000000000000000. New DES key will be 0x0000000000000000 as in factory settings.

Function declaration (C language)

```
UFR_STATUS AES_to_DES_key_type(void);
```

uFR_int_DesfireCreateValueFile

uFR_int_DesfireCreateValueFile_PK

uFR_int_DesfireCreateValueFile_no_auth

Function description

For uFR PLUS devices only.

Function allows to create file for the storage and manipulation of 32 bit signed integer values within existing application on the card. Maximal number of files into application is 32. The file will be created in the currently selected application. Is the application master key authentication is required, depend on the application master key settings.

Communication settings define communication mode between reader and card. The communication modes are:

- plain communication communication settings value is 0x00

- plain communication secured by MACing communication settings value is 0x01

- fully enciphered communication communication settings value is 0x11

Access rights for read, write, read&write and changing, references certain key within application's keys (0 – 13). If value is 14, this means free access, independent of previous authentication. If value is 15, this means deny access (for example if write access is 15 then the file type is read only).

Function declaration (C language)

```
UFR_STATUS uFR_int_DesfireCreateValueFile(
                                        uint8_t aes_key_nr,
                                        uint32_t aid,
                                        uint8_t file_id,
                                        int32_t lower_limit,
                                        int32_t upper_limit,
                                        int32_t value,
                                        uint8_t limited_credit_enabled,
                                        uint8_t read_key_no,
                                        uint8_t write_key_no,
                                        uint8_t read_write_key_no,
                                        uint8_t change_key_no,
                                        uint8_t communication_settings,
                                        uint16_t *card_status,
                                        uint16_t *exec_time);
UFR_STATUS uFR_int_DesfireCreateValueFile_PK(
                                        uint8_t *aes_key_ext,
                                        uint32_t aid,
                                        uint8_t file_id,
                                        uint8_t lower_limit,
                                        int32_t upper_limit,
                                        int32_t value,
                                        uint8_t limited_credit_enabled,
                                        uint8_t read_key_no,
                                        uint8_t write_key_no,
                                        uint8_t read_write_key_no,
                                        uint8_t change_key_no,
                                        uint8_t communication_settings,
                                        uint16_t *card_status,
                                        uint16_t *exec_time);
UFR_STATUS uFR_int_DesfireCreateValueFile_no_auth(
                                        uint32_t aid,
                                        uint8_t file_id,
                                        int32_t lower_limit,
                                        int32_t upper_limit,
                                        int32_t value,
                                        uint8_t limited_credit_enabled,
                                        uint8_t read_key_no,
                                        uint8_t write_key_no,
                                        uint8_t read_write_key_no,
                                        uint8_t change_key_no,
                                        uint8_t communication_settings,
                                        uint16_t *card_status,
                                        uint16_t *exec_time);
```

Parameters

aes_key_nr	ordinal number of AES key in the reader
aes_key_ext	pointer to 16 byte array containing the AES key
aid	ID of application that contains the file
file_id	ID of file that will be created (0 – 31)
lower_limit	lower limit which is valid for this file
upper_limit	upper limit which is valid for this file
value	initial value of the value file
limited_credit_enabled	bit 0 – limited credit enabled (1 – yes, 0 – no) bit 1 – free get value (1 – yes, 0 – no)
read_key_no	key for get and debit value
write_key_no	key for get, debit and limited credit value
read_write_key_no	for get, debit, limited credit and credit value
change_key_no	key for changing this setting
communication_settings	variable that contains communication settings
card_status	pointer to card error variable
exec_time	function's execution time

uFR_int_DesfireReadValueFile

uFR_int_DesfireReadValueFile_PK

uFR_int_DesfireReadValueFile_no_auth

Function description

For uFR PLUS devices only.

Function allow to read value from value files. Read command requires a preceding authentication either with the key specified for Read or Read&Write access.

Function declaration (C language)

```
UFR_STATUS uFR_int_DesfireReadValueFile(
                                        uint8_t aes_key_nr,
                                        uint32_t aid,
                                        uint8_t aid_key_nr,
                                        uint8_t communication_settings,
                                        int32_t *value,
                                        uint16_t *card_status,
                                        uint16_t *exec_time);
UFR_STATUS uFR_int_DesfireReadValueFile_PK(
                                        uint8_t *aes_key_ext,
                                        uint32_t aid,
                                        uint8_t aid_key_nr,
                                        uint8_t communication_settings,
                                        int32_t *value,
                                        uint16_t *card_status,
                                        uint16_t *exec_time);
UFR_STATUS uFR_int_DesfireReadValueFile_no_auth(
                                        uint32_t aid,
                                        uint8_t aid_key_nr,
                                        uint8_t communication_settings,
                                        int32_t *value,
                                        uint16_t *card_status,
                                        uint16_t *exec_time);
```

Parameters

aes_key_nr	ordinal number of AES key in the reader
aes_key_ext	pointer to 16 byte array containing the AES key
aid	ID of application that contains the file

aid_key_nr	key number into application
communication_settings	value must be same as in file declaration
value	pointer to value variable
card_status	pointer to card error variable
exec_time	function's execution time

uFR_int_DesfireIncreaseValueFile

uFR_int_DesfireIncreaseValueFile_PK

uFR_int_DesfireIncreaseValueFile_no_auth

Function description

For uFR PLUS devices only.

Function allows to increase a value stored in a value files. Credit command requires a preceding authentication with the key specified for Read&Write access.

Function declaration (C language)

```
UFR_STATUS uFR_int_DesfireIncreaseValueFile(
                                        uint8_t aes_key_nr,
                                        uint32_t aid,
                                        uint8_t aid_key_nr,
                                        uint8_t communication_settings,
                                        int32_t value,
                                        uint16_t *card_status,
                                        uint16_t *exec_time);
UFR_STATUS uFR_int_DesfireIncreaseValueFile_PK(
                                        uint8_t *aes_key_ext,
                                        uint32_t aid,
                                        uint8_t aid_key_nr,
                                        uint8_t communication_settings,
                                        int32_t value,
                                        uint16_t *card_status,
                                        uint16_t *exec_time);
FR_STATUS uFR_int_DesfireIncreaseValueFile_no_auth(
                                        uint32_t aid,
                                        uint8_t aid_key_nr,
                                        uint8_t communication_settings,
                                        int32_t value,
                                        uint16_t *card_status,
                                        uint16_t *exec_time);
```

Parameters

aes_key_nr	ordinal number of AES key in the reader
aes_key_ext	pointer to 16 byte array containing the AES key
aid	ID of application that contains the file
aid_key_nr	key number into application
communication_settings	value must be same as in file declaration
value	value (must be positive number)
card_status	pointer to card error variable
exec_time	function's execution time

uFR_int_DesfireDecreaseValueFile

uFR_int_DesfireDecreaseValueFile_PK

uFR_int_DesfireDecreaseValueFile_no_auth

Function description

For uFR PLUS devices only

Function allow to decrease value from value files. Debit command requires a preceding authentication with on of the keys specified for Read, Write or Read&Write access.

Function declaration (C language)

```
UFR_STATUS uFR_int_DesfireDecreaseValueFile(
                                        uint8_t aes_key_nr,
                                        uint32_t aid,
                                        uint8_t aid_key_nr,
                                        uint8_t communication_settings,
                                        int32_t value,
                                        uint16_t *card_status,
                                        uint16_t *exec_time);
UFR_STATUS uFR_int_DesfireDecreaseValueFile_PK(
                                        uint8_t *aes_key_ext,
                                        uint32_t aid,
                                        uint8_t aid_key_nr,
                                        uint8_t communication_settings,
                                        int32_t value,
                                        uint16_t *card_status,
                                        uint16_t *exec_time);
UFR_STATUS uFR_int_DesfireDecreaseValueFile_no_auth(
                                        uint32_t aid,
                                        uint8_t aid_key_nr,
                                        uint8_t communication_settings,
                                        int32_t *value,
                                        uint16_t *card_status,
                                        uint16_t *exec_time);
```

Parameters

aes_key_nr	ordinal number of AES key in the reader
aes_key_ext	pointer to 16 byte array containing the AES key

aid	ID of application that contains the file
aid_key_nr	key number into application
communication_settings	value must be same as in file declaration
value	value (must be positive number)
card_status	pointer to card error variable
exec_time	function's execution time

Originality checking

Some card chips supports originality checking mechanism using Elliptic Curve Digital Signature Algorithm (ECDSA). Chip families that support originality checking mechanism are NTAG 21x and Mifare Ultralight EV1. For details on originality checking, you must have an non-disclosure agreement (NDA) with the manufacturer who will provide you with the relevant documentation. In any case, the uFR API provides you with 2 functions that you can use for this purpose:

ReadECCSignature

Function description

This function returns ECC signature of the card chip UID. Card chip UID is signed using EC private key known only to a manufacturer.

Function declaration (C language)

```
#define MAX_UID_LEN       10
#define ECC_SIG_LEN       32
UFR_STATUS ReadECCSignature(uint8_t lpucECCSignature[ECC_SIG_LEN],
                            uint8_t lpucUid[MAX_UID_LEN],
                            uint8_t *lpucUidLen,
                            uint8_t *lpucDlogicCardType);
```

Parameters

lpucECCSignature	pointer to array which (in case of successfully executed operation) will contain 32 bytes long ECDSA signature of the chip UID. Chip UID is signed using EC private key known only to a manufacturer.

lpucUid	pointer to a chip UID (in case of successfully executed operation). Returned here for convenience.
***lpucUidLen**	pointer to variable which will (in case of successfully executed operation) receive true length of the returned UID. (Maximum UID length is 10 bytes but there is three possible UID sizes: 4, 7 and 10).
***lpucDlogicCardType**	pointer to variable which will (in case of successfully executed operation) receive DlogicCardType. Returned here for convenience. For DlogicCardType uFR API uses the same constants as with GetDlogicCardType() function (see Appendix: DLogic CardType enumeration).

OriginalityCheck

Function description

This function depends on OpenSSL crypto library. Since OpenSSL crypto library is dynamically linked during execution, the only prerequisite for a successful call to this function is that the libeay32.dll is in the current folder (valid for Windows) and / or libcrypto.so is in the environment path (e.g. LD_LIBRARY_PATH on Linux / macOS). **OriginalityCheck()** performs the check if the chip on the card / tag is NXP genuine.

Function declaration (C language)

```
UFR_STATUS OriginalityCheck(const uint8_t *signature,
                            const uint8_t *uid,
                            uint8_t uid_len,
                            uint8_t DlogicCardType);
```

Parameters

***signature**	ECCSignature acquired by call to the **ReadECCSignature()** function.
***uid**	Card UID. Best if the card UID is acquired by previous call to the **ReadECCSignature()** function.
uid_len	Card UID length. Best if the card UID length is acquired by previous call to the **ReadECCSignature()** function.
DlogicCardType	Card type. Best if the DlogicCardType is acquired by previous call to the **ReadECCSignature()** function.

UFR_STATUS specific error codes that can be returned by this function:

UFR_NOT_NXP_GENUINE	0x0200	if the chip on the card/tag ISN'T NXP GENUINE
UFR_OPEN_SSL_DYNAMIC_LIB_FAILED	0x0201	in case of OpenSSL library error (e.g. wrong OpenSSL version)
UFR_OPEN_SSL_DYNAMIC_LIB_NOT_FOUND	0x0202	in case there is no OpenSSL library (libeay32.dll on Windows systems, libcrypto.so on Linux and libcrypto.dylib on macOS) in current folder or environment path
UFR_OK	0	if the chip on the card/tag IS NXP GENUINE

NFC Type 2 Tags counters

There are different types of counters implemented in different families of the NFC T2T chips. Ultralight, NTAG 210 and NTAG 212 doesn't have counters.

Ultralight C and NTAG 203 have one 16-bit one-way counter which can be managed using BlockRead and BlockWrite API functions on the appropriate block address (for those two chips, counter page address is 0x29.

Ultralight EV1 variants have three independent 24-bit one-way counters which can be managed using ReadCounter() and IncrementCounter() API functions. Counters are mapped in a separate address space.

NTAG 213, NTAG 215 and NTAG 216 have 24-bit NFC counter which is incremented on every first valid occurrence of the READ or FAST-READ command (ISO 14443-3A proprietary commands) after the tag is powered by an RF field. There is no another way to change value of the 24-bit NFC counter and there is mechanism to enable it or disable it. This counter can be read using ReadNFCCounter() API function if password authentication is not in use. API functions ReadNFCCounterPwdAuth_RK() or ReadNFCCounterPwdAuth_PK() can be used to read NFC counter if it's protected with the password authentication. 24-bit NFC counter have counter address 2 (counter is mapped in a separate address space) so ReadCounter(2, &value) call is equivalent to a ReadNFCCounter(&value) if password authentication isn't in use.

ReadCounter

Function description

This function is used to read one of the three 24-bit one-way counters in Ultralight EV1 chip family. Those counters can't be password protected. In the initial Ultralight EV1 chip state, the counter values are set to 0.

Function declaration (C language)

```
UFR_STATUS ReadCounter(uint8_t counter_address, uint32_t *value);
```

Parameters

`counter_address`	Address of the target counter. Can be in range 0 to 2. Counters are mapped in a separate address space.
`*value`	Pointer to a uint32_t which will contained counter value after successful function execution. Since counters are 24-bit in length, most significant byte of the *value will be always 0.

IncrementCounter

Function description

This function is used to increment one of the three 24-bit one-way counters in Ultralight EV1 chip family. Those counters can't be password protected. If the sum of the addressed counter value and the increment value is higher than 0xFFFFFF, the tag replies with an error and does not update the respective counter.

Function declaration (C language)

```
UFR_STATUS IncrementCounter(uint8_t counter_address, uint32_t
inc_value);
```

Parameters

`counter_address`	Address of the target counter. Can be in range 0 to 2. Counters are mapped in a separate address space.
`inc_value`	Increment value. Only the 3 least significant bytes are relevant.

ReadNFCCounter

Function description

This function is used to read 24-bit NFC counter in NTAG 213, NTAG 215 and NTAG 216 chips without using password authentication. If access to NFC counter is configured to be password protected, this function will return COUNTER_ERROR.

Function declaration (C language)

```
UFR_STATUS ReadNFCCounter(uint32_t *value);
```

Parameter

***value**	Pointer to a uint32_t which will contained counter value after successful function execution. Since counter is 24-bit in length, most significant byte of the *value will be always 0.

ReadNFCCounterPwdAuth_RK

Function description

This function is used to read 24-bit NFC counter in NTAG 213, NTAG 215 and NTAG 216 chips using “reader key password authentication”. If access to NFC counter is configured to be password protected and PWD-PACK pair stored as a 6-byte key in uFR reader disagrees with PWD-PACK pair configured in tag, this function will return UFR_AUTH_ERROR. If access to NFC counter isn’t configured to be password protected, this function will return UFR_AUTH_ERROR.

Function declaration (C language)

```
UFR_STATUS ReadNFCCounterPwdAuth_RK(uint32_t *value,
                                    uint8_t reader_key_index);
```

Parameters

***value**	Pointer to a uint32_t which will contained counter value after successful function execution. Since counter is 24-bit in length, most significant byte of the *value will be always 0.
reader_key_index	Index of the 6-byte key (PWD-PACK pair for this type of NFC tags) stored in the uFR reader. Can be in range 0 to 31.

ReadNFCCounterPwdAuth_PK

Function description

This function is used to read 24-bit NFC counter in NTAG 213, NTAG 215 and NTAG 216 chips using “provided key password authentication”. If access to NFC counter is configured to be password protected and PWD-PACK pair sent as a 6-byte provided key disagrees with PWD-PACK pair configured in tag, this function will return UFR_AUTH_ERROR. If access to NFC counter isn’t configured to be password protected, this function will return UFR_AUTH_ERROR.

Function declaration (C language)

```
UFR_STATUS ReadNFCCounterPwdAuth_PK(uint32_t *value, const uint8_t
*key);
```

Parameters

***value**	Pointer to a uint32_t which will contained counter value after successful function execution. Since counter is 24-bit in length, most significant byte of the *value will be always 0.
***key**	Pointer to an array contains provided 6-byte key (PWD-PACK pair for this type of NFC tags) for password authentication.

Functions for the operating parameters of the reader setting

UfrSetBadSelectCardNrMax

Function description

The function allows you to set the number of unsuccessful card selections before it can be considered that the card is not placed on the reader. Period between two card selections is approximately 10ms. Default value of this parameter is 20 i.e. 200ms. This parameter can be set in the range of 0 to 254.
This is useful for asynchronous card ID transmission, if parameter send_removed_enable in function SetAsyncCardIdSendConfig is set. Then you can set a lower value of the number of unsuccessful card selections, in order to send information to the card removed was faster.
A small value of this parameter may cause a false report that the card is not present, and immediately thereafter true report that the card is present.

Function declaration (C language)

```
UFR_STATUS UfrSetBadSelectCardNrMax(uint8_t bad_select_nr_max);
```

Parameter

bad_select_nr_max	number of unsuccessful card selections

UfrGetBadSelectCardNrMax

Function description

The function returns value of maximal unsuccessful card selections, which is set in reader.

Function declaration (C language)

```
UFR_STATUS UfrGetBadSelectCardNrMax(uint8_t *bad_select_nr_max);
```

Parameter

`bad_select_nr_max`	pointer to number of unsuccessful card selections

Functions for all blocks linear reading

Function description

Functions allow you to quickly read data from the card including the sector trailer blocks. These functions are very similar to the functions for linear reading of users data space.

- ***LinearRowRead***
- ***LinearRowRead_AKM1***
- ***LinearRowRead_AKM2***
- ***LinearRowRead_PK***

Functions declaration (C language):

```
UFR_STATUS LinearRowRead(uint8_t *aucData,
                         uint16_t usLinearAddress,
                         uint16_t usDataLength,
                         uint16_t *lpusBytesReturned,
                         uint8_t ucAuthMode,
                         uint8_t ucReaderKeyIndex);
UFR_STATUS LinearRowRead_AKM1(uint8_t *aucData,
                              uint16_t usLinearAddress,
                              uint16_t usDataLength,
                              uint16_t *lpusBytesReturned,
                              uint8_t ucAuthMode);
UFR_STATUS LinearRowRead_AKM2(uint8_t *aucData,
                              uint16_t usLinearAddress,
                              uint16_t usDataLength,
                              uint16_t *lpusBytesReturned,
                              uint8_t ucAuthMode);
UFR_STATUS LinearRowRead_PK(uint8_t *aucData,
                            uint16_t usLinearAddress,
                            uint16_t usDataLength,
                            uint16_t *lpusBytesReturned,
                            uint8_t ucAuthMode,
                            uint8_t *aucProvidedKey);
```

Parameters

aucData	Pointer to the sequence of bytes where read data will be stored
usLinearAddress	Linear address on the card from which the data want to read
usDataLength	Number of bytes for reading. For aucData a minimum usDataLength bytes must be allocated before calling the function
lpusBytesReturned	Pointer to "uint16_t" type variable, where the number of successfully read bytes from the card is written. If the reading is fully managed this data is equal to the usDataLength parameter. If there is an error reading some of the blocks, the function returns all successfully read data in the aucData before the errors occurrence and the number of successfully read bytes is returned via this parameter
ucAuthMode	This parameter defines whether to perform authentication with key A or key B. It can have two values, namely: AUTHENT1A (0x60) or AUTHENT1B (0x61)
ucReaderKeyIndex	The default method of authentication (when the functions without a suffix is used) performs the authenticity proving by using the selected key index from

	the reader. In the linear address mode, this applies to all sectors that are read
aucProvidedKey	Pointer to the six-byte string containing the key for authenticity proving in the "Provided Key" method. _PK Suffix in the name of the function indicates this method usage

FUNCTIONS FOR READER LOW POWER MODE CONTROL

UfrEnterSleepMode

Function description

Function allows enter to reader low power working mode. Reader is in sleep mode. RF field is turned off. The reader is waiting for the command to return to normal working mode.

Function declaration (C language)

```
UFR_STATUS UfrEnterSleepMode(void);
```

UfrLeaveSleepMode

Function description

Function allows return from low power reader mode to normal working mode.

Function declaration (C language):

```
UFR_STATUS UfrLeaveSleepMode(void);
```

AutoSleepSet

Function description

This function permanently set auto-sleep functionality of the device. Valid seconds_wait range is from 1 to 254. To permanently disable auto-sleep functionality use 0 or 0xFF for the seconds_wait parameter.

Function declaration (C language)

```
unsigned long AutoSleepSet(uint8_t seconds_wait);
```

Parameter

seconds_wait	device inactivity time before entering into sleep mode

AutoSleepGet

Function description

This function uses to get auto-sleep functionality setup from the device. You have to send pointer to already allocated variable of the uint8_t type. If auto-sleep functionality is disabled you will get 0 or 0xFF in the variable pointed by the *seconds_wait parameter.

Function declaration (C language)

```
unsigned long AutoSleepGet(uint8_t *seconds_wait);
```

Parameter

seconds_wait	device inactivity time before entering into sleep mode

Functions for Reader NTAG Emulation Mode

WriteEmulationNdef

Function description

Function store a message record for NTAG emulation mode in to the reader. Parameters of the function are: TNF, type of record, ID, payload.

Function declaration (C language)

```
UFR_STATUS WriteEmulationNdef(uint8_t tnf,
                              uint8_t* type_record,
                              uint8_t type_length,
                              uint8_t* id,
                              uint8_t id_length,
                              uint8_t* payload,
                              uint8_t payload_length);
```

Parameters

tnf	TNF of the record
type_record	pointer to the array containing record type
type_length	length of the record type
id	pointer to the array containing record ID
id_length	length of the record ID

payload	pointer to the array containing record payload
payload_length	length of the record payload

Possible error codes:

```
WRITE_VERIFICATION_ERROR = 0x70
MAX_SIZE_EXCEEDED = 0x10
```

WriteEmulationNdefWithAAR

Function description

This function do the same as WriteEmulationNdef() function with the addition of an AAR embedded in to the NDEF message. AAR stands for “Android Application Record”. AAR is a special type of NDEF record that is used by Google’s Android operating system to signify to an NFC phone that an explicitly defined Android Application which should be used to handle an emulated NFC tag. Android App record will be added as the 2nd NDEF record in the NDEF message.

Function declaration (C language)

```
UFR_STATUS WriteEmulationNdefWithAAR(uint8_t tnf,
                                     uint8_t *type_record,
                                     uint8_t type_length,
                                     uint8_t *id,
                                     uint8_t id_length,
                                     uint8_t *payload,
                                     uint8_t payload_length,
                                     uint8_t *aar,
                                     uint8_t aar_length);
```

Parameters

tnf	TNF of the record
type_record	pointer to the array containing record type
type_length	length of the record type
id	pointer to the array containing record ID
id_length	length of the record ID

payload	pointer to the array containing record payload
payload_length	length of the record payload
aar	pointer to the array containing AAR record
aar_length	length of the AAR record

TagEmulationStart

Function description

Put the reader permanently in a NDEF tag emulation mode. Only way for a reader to exit from this mode is to receive the TAG_EMULATION_STOP command (issued by calling **TagEmulationStop()** function).

In this mode, the reader can only answer to the commands issued by a following library functions:

TagEmulationStart(),

WriteEmulationNdef(),

TagEmulationStop(),

GetReaderSerialNumber(),

GetReaderSerialDescription(),

GetReaderHardwareVersion(),

GetReaderFirmwareVersion(),

GetBuildNumber()

Calls to the other functions in this mode returns following error code:

FORBIDDEN_IN_TAG_EMULATION_MODE = 0x90

Function declaration (C language)

```
UFR_STATUS TagEmulationStart(void);
```

Possible error codes:

WRITE_VERIFICATION_ERROR = 0x70

(command resulting in a direct write to a device non-volatile memory)

TagEmulationStop

Function description

Allows the reader permanent exit from a NDEF tag emulation mode.

Function declaration (C language)

```
UFR_STATUS TagEmulationStop(void);
```

Possible error codes:

```
WRITE_VERIFICATION_ERROR = 0x70
```

(command resulting in a direct write to a device non-volatile memory)

Functions for setting Reader baud rates for ISO 14443 – 4A cards

SetSpeedPermanently

Function declaration (C language)

```
UFR_STATUS SetSpeedPermanently(uint8_t tx_speed, uint8_t rx_speed);
```

Parameters

tx_speed	setup value for transmit speed
rx_speed	setup value for receive speed

Valid speed setup values are:

Const	*Configured speed*
0	106 kbps (default)
1	212 kbps
2	424 kbps

On some reader types maximum rx_speed is 212 kbps. If you try to set higher speed than is allowed, reader firmware will automatically set the maximum possible speed.

Possible error codes:

```
WRITE_VERIFICATION_ERROR = 0x70
```

(command resulting in a direct write to a device non-volatile memory)

GetSpeedParameters

Function declaration (C language)

```
UFR_STATUS GetSpeedParameters(uint8_t* tx_speed, uint8_t* rx_speed);
```

Parameters

`tx_speed`	returns configured value for transmit speed
`rx_speed`	returns configured value for receive speed

FUNCTIONS FOR DISPLAY CONTROL

SetDisplayData

Function description

Function enables sending data to the display. A string of data contains information about the intensity of color in each cell of the display. Each cell has three LED (red, green and blue). For each cell of the three bytes is necessary. The first byte indicates the intensity of the green color, the second byte indicates the intensity of the red color, and the third byte indicates the intensity of blue color. For example, if the display has 16 cells, an array contains 48 bytes. Value of intensity is in range from 0 to 255.

Function declaration (C language)

```
UFR_STATUS SetDisplayData(uint8_t  *display_data,
                          uint8_t data_length);
```

Parameters

`display_data`	pointer to data array
`data_length`	number of data into array

SetSpeakerFrequency

Function description

Function sets the frequency of the speaker. The speaker is working on this frequency until a new frequency setting. To stop the operation set frequency to zero.

Function declaration (C language)

```
UFR_STATUS SetSpeakerFrequency(uint16_t frequency);
```

Parameter

`frequency`	frequency in Hz

FUNCTIONS TO USE THE SHARED RAM INTO DEVICE

Shared RAM is memory space on a device that is used for communication between computer and Android device (phone, tablet) with an NFC reader. PC writes and read data from shared RAM via USB port. Device with Android OS writes and read data from shared RAM via NFC.

EnterShareRamCommMode

Function description

Put reader permanently in the mode that use shared RAM. After execution of this function, must be executed function TagEmulationStart.

Function declaration (C language)

```
UFR_STATUS EnterShareRamCommMode(void);
```

ExitShareRamCommMode

Function description

The permanent exit from mode that use shared RAM. After execution of this function, must be executed function TagEmulationStop.

Function declaration (C language)

```
UFR_STATUS EnterShareRamCommMode(void);
```

WriteShareRam

Function description

Function allows writing data to the shared RAM.

Function declaration (C language)

```
UFR_STATUS WriteShareRam(uint8_t *ram_data,
                         uint8_t addr,
                         uint8_t data_len);
```

Parameters

`ram_data`	pointer to data array

addr	address of first data in an array
data_len	/ength of array. Address + data_len <= 184

ReadShareRam

Function description

Function allows read data from the shared RAM.

Function declaration (C language)

```
UFR_STATUS ReadShareRam(uint8_t *ram_data,
                        uint8_t addr,
                        uint8_t data_len);
```

Functions supporting Ad-Hoc emulation mode

This mode enables user controlled emulation from the user application. There is “nfc-rfid-reader-sdk/ufr-examples-ad_hoc_emulation-c” console example written in C, which demonstrate usage of this functions.

AdHocEmulationStart

Function description

Put uFR in emulation mode with ad-hoc emulation parameters (see. SetAdHocEmulationParams() and GetAdHocEmulationParams() functions). uFR stays in ad-hoc emulation mode until AdHocEmulationStop() is called or reader reset.

Function declaration (C language)

```
UFR_STATUS AdHocEmulationStart(void);
```

AdHocEmulationStop

Function description

Terminate uFR ad-hoc emulation mode.

Function declaration (C language)

```
UFR_STATUS AdHocEmulationStop(void);
```

GetExternalFieldState

Function description

Returns external field state when uFR is in ad-hoc emulation mode.

Function declaration (C language)

```
UFR_STATUS GetExternalFieldState(uint8_t *is_field_present);
```

is_field_present contains 0 if external field isn't present or 1 if field is present.

GetAdHocEmulationParams

Function description

This function returns current ad-hoc emulation parameters. On uFR power on or reset ad-hoc emulation parameters are set back to their default values.

Function declaration (C language)

```
UFR_STATUS GetAdHocEmulationParams(uint8_t *ThresholdMinLevel,
                                   uint8_t *ThresholdCollLevel,
                                   uint8_t *RFLevelAmp,
                                   uint8_t *RxGain,
                                   uint8_t *RFLevel);
```

Parameters

ThresholdMinLevel	default value is 15. Could be in range from 0 to 15
ThresholdCollLevel	default value is 7. Could be in range from 0 to 7
RFLevelAmp	default value is 0. On uFR device should be 0 all the time. (1 for on, 0 for off).
RxGain	Could be in range from 0 to 7.
RFLevel	Could be in range from 0 to 15

SetAdHocEmulationParams

Function description

This command set ad-hoc emulation parameters. On uFR power on or reset ad-hoc emulation parameters are set back to their default values.

Function declaration (C language)

```
UFR_STATUS SetAdHocEmulationParams(uint8_t ThresholdMinLevel,
                                   uint8_t ThresholdCollLevel,
                                   uint8_t RFLevelAmp,
                                   uint8_t RxGain,
                                   uint8_t RFLevel);
```

Parameters

ThresholdMinLevel	default value is 15. Could be in range from 0 to 15
ThresholdCollLevel	default value is 7. Could be in range from 0 to 7
RFLevelAmp	default value is 0. On uFR device should be 0 all the time. (1 for on, 0 for off).
RxGain	Could be in range from 0 to 7.
RFLevel	Could be in range from 0 to 15

CombinedModeEmulationStart

Function description

Puts the uFR reader into a permanently periodical switching from “NDEF tag emulation mode” to “tag reader mode”. Only way for a reader to exit from this mode is to receive the TAG_EMULATION_STOP command (issued by calling the TagEmulationStop() function).

Much better control of the NFC device in a uFR proximity range can be achieved using Ad-Hoc emulation mode, described before.

Function declaration (C language)

```
UFR_STATUS CombinedModeEmulationStart(void);
```

Function takes no parameters.

Support for ISO14443-4 protocol

The protocol defines three fundamental types of blocks:

- I-block used to convey information for use by the application layer.

- R-block used to convey positive or negative acknowledgements. An R-block never contains an INF field. The acknowledgement relates to the last received block.

- S-block used to exchange control information between the PCD and the PICC. Two different types of S-blocks are defined:

1) Waiting time extension containing a 1 byte long INF field and

2) DESELECT containing no INF field.

Function declaration (C language)

```
UFR_STATUS i_block_trans_rcv_chain(uint8_t chaining,
                                   uint8_t timeout,
                                   uint8_t block_length,
                                   uint8_t *snd_data_array,
                                   uint8_t *rcv_length,
                                   uint8_t *rcv_data_array,
                                   uint8_t *rcv_chained,
                                   uint32_t *ufr_status);
```

Parameters

chaining	1 – chaining in use, 0 – no chaining
timeout	timeout for card reply
block_length	inf block length
snd_data_array	pointer to array of data that will be send
rcv_length	length of received data
rcv_data_array	pointer to array of data that will be received
rcv_chained	1 received packet is chained, 0 received packet is not chained
ufr_status	card operation status

Function declaration (C language)

```
UFR_STATUS r_block_transceive(uint8_t ack,
                              uint8_t timeout,
                              uint8_t *rcv_length,
                              uint8_t *rcv_data_array,
                              uint8_t *rcv_chained,
                              uint32_t *ufr_status);
```

Parameters

ack	1 ACK, 0 NOT ACK
timeout	timeout for card reply
rcv_length	length of received data
rcv_data_array	pointer to array of data that will be received
rcv_chained	1 received packet is chained, 0 received packet is not chained
ufr_status	card operation status

Function declaration (C language)

```
UFR_STATUS s_block_deselect(uint8_t timeout);
```

Parameter

timeout	timeout in [ms]

Support for APDU commands in ISO 14443-4 tags

Some ISO 14443-4 tags supports the APDU message structure according to ISO/IEC 7816-4.

For more details you have to check the manual for the tags that you planning to use.

Function declarations used to support APDU message structure:

```
UFR_STATUS SetISO14443_4_Mode(void);

UFR_STATUS uFR_APDU_Transceive(uint8_t cls,
                               uint8_t ins,
                               uint8_t p0,
                               uint8_t p1,
                               uint8_t *data_out,
                               uint8_t data_out_len,
                               uint8_t *data_in,
                               uint32_t max_data_in_len,
                               uint32_t *response_len,
                               uint8_t send_le,
                               uint8_t *apdu_status);

UFR_STATUS s_block_deselect(uint8_t timeout);
```

Parameters

cls	APDU CLA (class byte)
ins	APDU command code (instruction byte)
p0	parameter byte
p1	parameter byte
data_out	APDU command data field. Use NULL if data_out_len is 0
data_out_len	number of bytes in the APDU command data field (Lc field)
data_in	buffer for receiving APDU response. There should be allocated at least (send_le + 2) bytes before function call.
max_data_in_len	size of the receiving buffer. If the APDU response exceeded size of buffer, then function returns error
response_len	value of the Le fied if send_le is not 0. After successful execution

	location pointed by response_len will contain number of bytes in the APDU response.
send_le	if this parameter is 0 then APDU Le field will not be sent. Otherwise Le field will be included in the APDU message. Value response_len pointed to, before function call will be value of the Le field.
apdu_status	APDU error codes SW1 and SW2 in 2 bytes array

To send APDU message you must comply with the following procedure:

1. Call SetISO14443_4_Mode(). ISO 14443-4 tag in a field will be selected and RF field polling will be stopped.
2. Call uFR_APDU_Transceive() as many times as you needed.
3. Call s_block_deselect() to deselect tag and restore RF field polling. This call is mandatory.

Fully uFR firmware support for APDU commands in ISO 14443-4 tags

This group of newly designed functions makes use of the **uFR_APDU_Transceive()** obsolete. However, **uFR_APDU_Transceive()** function is still part of the uFCoder library for backward compatibility.

New functions implemented in the uFCoder library are:

```
UFR_STATUS APDUHexStrTransceive(const char *c_apdu, char **r_apdu);
UFR_STATUS APDUPlainTransceive(const uint8_t *c_apdu,
                               uint32_t c_apdu_len,
                               uint8_t *r_apdu,
                               uint32_t *r_apdu_len);
UFR_STATUS APDUTransceive(uint8_t cls,
                          uint8_t ins,
                          uint8_t p0,
                          uint8_t p1,
                          const uint8_t *data_out,
                          uint32_t Nc,
                          uint8_t *data_in,
                          uint32_t *Ne,
                          uint8_t send_le,
                          uint8_t *apdu_status);
```

These functions are more responsive than obsolete **uFR_APDU_Transceive()**, because most of the work if performed by a uFR firmware.

```
UFR_STATUS APDUHexStrTransceive(const char *c_apdu, char **r_apdu);
```

Using this function, you can send C–APDU in the c_string (zero terminated) containing pairs of the hexadecimal digits. Pairs of the hexadecimal digits can be delimited by any of the punctuation characters or white space.

****r_apdu** returns pointer to the c_string (zero terminated) containing pairs of the hexadecimal digits without delimiters.

```
UFR_STATUS APDUPlainTransceive(const uint8_t *c_apdu,
                               uint32_t c_apdu_len,
                               uint8_t *r_apdu,
                               uint32_t *r_apdu_len);
```

This is binary alternative function to the **APDUHexStrTransceive()**. C-APDU and R-APDU are sent and receive in the form of the byte arrays. There is obvious need for a **c_apdu_len** and ***r_apdu_len** parameters which represents length of the ***c_apdu** and ***r_apdu** byte arrays, respectively.

The memory space on which ***r_apdu** points, have to be allocated before calling of the **APDUPlainTransceive()**. Number of the bytes allocated have to correspond to the N_e bytes, defined by the L_e field in the C–APDU plus 2 bytes for SW1 and SW2.

```
UFR_STATUS APDUTransceive(uint8_t cls,
                          uint8_t ins,
                          uint8_t p0,
                          uint8_t p1,
                          const uint8_t *data_out,
                          uint32_t Nc,
                          uint8_t *data_in,
                          uint32_t *Ne,
                          uint8_t send_le,
                          uint8_t *apdu_status);
```

This is “exploded binary” alternative function intended for support APDU commands in ISO 14443-4A tags. **APDUTransceive()** receives separated parameters which are an integral part of the C–APDU. There is parameters **cls**, **ins**, **p0**, **p1** of the **uint8_t** type.

N_c defines number of bytes in the byte array ***data_out** point to. N_c also defines L_c field in the C–APDU. Maximum value for the N_c *is 255. If* $N_c > 0$ *then* $L_c = N_c$, otherwise L_c *is omitted and* ****data_out*** can be NULL.

send_le and $*N_e$ parameters defines L_c field in the C–APDU. If **send_le** is 1 then L_e field will be included in the C–APDU. If **send_le** is 0 then L_e field will be omitted from the C–APDU.

If $*N_e == 256$ *then* $L_e = 0$, *otherwise* $L_e = *N_e$.

The memory space on which ***data_in**, have to be allocated before calling of the

APDUPlainTransceive(). Number of the bytes allocated have to correspond to the $*N_e$ bytes, defined by the L_e field in the C–APDU.

After successfully executed **APDUTransceive()**, ***data_in** will contain R-APDU data field (body).

***apdu_status** will contain R-APDU trailer (SW1 and SW2 APDU status bytes).

For older uFR firmware / deprecated / library backward compatibility

UFR_STATUS uFR_DESFIRE_Start(void);

UFR_STATUS uFR_DESFIRE_Stop(void);

UFR_STATUS uFR_APDU_Start(void); // Alias for uFR_DESFIRE_Start()

UFR_STATUS uFR_APDU_Stop(void); // Alias for uFR_DESFIRE_Stop()

UFR_STATUS uFR_i_block_transceive(uint8_t chaining, uint8_t timeout,
uint8_t block_length, uint8_t *snd_data_array, size_t *rcv_length,
uint8_t *rcv_data_array, uint32_t *ufr_status);

PKI infrastructure and digital signature support

Fully supported from library version 4.3.8 and firmware version 3.9.55

In our product range, we have special cards called “D-Logic JCApp” (working title), which contains support for PKI infrastructure and digital signing. To invoke API functions that support these features, the following conditions must be met:

1. “D-Logic JCApp” card must be in uFR reader field.
2. NFC tag must be in ISO 14443-4 mode. For entering ISO 14443-4 mode use **SetISO14443_4_Mode()** function.
3. Now you can call any of the API functions with prefix “JCApp” as much as necessary.
4. At the end of JCApp session is necessary to call **s_block_deselect()** to deselect tag and restore RF field polling.

To generate digital signature using “D-Logic JCApp” you need to have at least one of the private keys stored in a card. Further, if your data for signing have more than 255 bytes, you have to split them into the chunks and send them to a card using JCAppSignatureBegin() for the first chunk and JCAppSignatureUpdate() for rest of the chunks. To generate signature, you have to call JCAppSignatureEnd() after you have sent all of the data for signing. At last, to get signature, you have to call JCAppGetSignature().

If your data for signing have 255 bytes or less, it is sufficient to call JCAppGenerateSignature() only once and immediately after that call JCAppGetSignature() to get a signature.

JCAppSelectByAid

Function description

Using this function you can select appropriate application on the card. AID should be “A0 F0 F1 F2 F3 00 01 00 01”. Before calling this function, NFC tag must be in ISO 14443-4 mode. For entering ISO 14443-4 mode use SetISO14443_4_Mode() function.

Function declaration (C language)

```
UFR_STATUS JCAppSelectByAid(const uint8_t *aid,
                            uint8_t aid_len,
                            uint8_t selection_response[16]);
```

Parameters

aid	Pointer to array containing AID (Aplication ID) i.e: “\xA0\xF0\xF1\xF2\xF3\x00\x01\x00\x01”.
aid_len	Length of the AID in bytes.
selection_response	On Application successful selection, card returns 16 bytes. In current version only the first of those bytes (i.e. byte with index 0) is relevant and contains JCApp card type which is 0xA0 for actual revision.

JCAppPutPrivateKey

Function description

In JCApp cards you can put two types of asymmetric crypto keys. Those are RSA and ECDSA private keys, three of each. Before you can use JCApp card for digital signing you have to put appropriate private key in it. There is no way to read out private keys from the card.

Before calling this function, NFC tag must be in ISO 14443-4 mode and JCApp should be selected using JCAppSelectByAid() with AID = “\xA0\xF0\xF1\xF2\xF3\x00\x01\x00\x01”.

Function declaration (C language)

```
UFR_STATUS JCAppPutPrivateKey(uint8_t key_type,
                              uint8_t key_index,
                              const uint8_t *key,
                              uint16_t key_bit_len,
                              const uint8_t *key_param,
                              uint16_t key_parm_len);
```

Parameters

key_type	0 for RSA private key and 1 for ECDSA private key.
key_index	For each of the card types there is 3 different private keys that you can set.

	Their indexes are from 0 to 2.
key	Pointer to array containing key bytes.
key_bit_len	Key **length in bits**.
key_param	Reserved for future use (RFU). Use null for this parameter.
key_parm_len	Reserved for future use (RFU). Use 0 for this parameter.

JCAppSignatureBegin

Function description

Before calling this function, NFC tag must be in ISO 14443-4 mode and JCApp should be selected using JCAppSelectByAid() with AID = “\xA0\xF0\xF1\xF2\xF3\x00\x01\x00\x01”.

Function declaration (C language)

```
UFR_STATUS JCAppSignatureBegin(uint8_t cipher,
                               uint8_t digest,
                               uint8_t padding,
                               uint8_t key_index,
                               const uint8_t *chunk,
                               uint16_t chunk_len,
                               const uint8_t *alg_param,
                               uint16_t alg_parm_len);
```

Parameters

cipher	0 for the RSA private key and 1 for the ECDSA.
digest	0 for none digest (not supported with ECDSA) and 1 for SHA1
padding	0 for none (not supported with RSA) and 1 for pads the digest according to the PKCS#1 (v1.5) scheme.
key_index	For each of the card types there is 3 different private keys that you can set. Their indexes are from 0 to 2.
chunk	Pointer to array containing first chunk of data.
chunk_len	Length of the first chunk of data (max. 255).
alg_param	Reserved for future use (RFU). Use null for this parameter.
alg_parm_len	Reserved for future use (RFU). Use 0 for this parameter.

JCAppSignatureUpdate

Function description

Before calling this function, NFC tag must be in ISO 14443-4 mode and JCApp should be selected using JCAppSelectByAid() with AID = “\xA0\xF0\xF1\xF2\xF3\x00\x01\x00\x01”.

Function declaration (C language)

```
UFR_STATUS JCAppSignatureUpdate(const uint8_t *chunk,
                                uint16_t chunk_len);
```

Parameters

chunk	Pointer to an array containing current one of the remaining chunks of data.
chunk_len	Length of the current one of the remaining chunks of data (max. 255).

JCAppSignatureEnd

Function description

Before calling this function, NFC tag must be in ISO 14443-4 mode and JCApp should be selected using JCAppSelectByAid() with AID = “\xA0\xF0\xF1\xF2\xF3\x00\x01\x00\x01”.

Function declaration (C language)

```
UFR_STATUS JCAppSignatureEnd(uint16_t *sig_len);
```

Parameters

sig_len	Pointer to a 16-bit value in which you will get length of the signature in case of successful executed chain of function calls, described in introduction of this topic.

JCAppGenerateSignature

Function description

This function virtually combines three successive calls of functions JCAppSignatureBegin(), JCAppSignatureUpdate() and JCAppSignatureEnd() and can be used in case your data for signing have 255 bytes or less.

Before calling this function, NFC tag must be in ISO 14443-4 mode and JCApp should be selected using JCAppSelectByAid() with AID = “\xA0\xF0\xF1\xF2\xF3\x00\x01\x00\x01”.

Function declaration (C language)

```
UFR_STATUS JCAppGenerateSignature(uint8_t cipher,
                                  uint8_t digest,
                                  uint8_t padding,
                                  uint8_t key_index,
                                  const uint8_t *plain_data,
                                  uint16_t plain_data_len,
                                  uint16_t *sig_len,
                                  const uint8_t *alg_param,
                                  uint16_t alg_parm_len);
```

Parameters

cipher	0 for the RSA private key and 1 for the ECDSA.
digest	0 for none digest (not supported with ECDSA) and 1 for SHA1
padding	0 for none (not supported with RSA) and 1 for pads the digest according to the PKCS#1 (v1.5) scheme.
key_index	For each of the card types there is 3 different private keys that you can set. Their indexes are from 0 to 2.
plain_data	Pointer to array containing data for signing.
plain_data_len	Length of the data for signing (max. 255).
sig_len	Pointer to a 16-bit value in which you will get length of the signature in case of successful execution.
alg_param	Reserved for future use (RFU). Use null for this parameter.
alg_parm_len	Reserved for future use (RFU). Use 0 for this parameter.

JCAppGetSignature

Function description

At last, to get signature, you have to call JCAppGetSignature().

Before calling this function, NFC tag must be in ISO 14443-4 mode and JCApp should be selected using JCAppSelectByAid() with AID = “\xA0\xF0\xF1\xF2\xF3\x00\x01\x00\x01”.

Function declaration (C language)

```
UFR_STATUS JCAppGetSignature(uint8_t *sig,
                             uint16_t sig_len);
```

Parameters

sig	Pointer to an array of “sig_len” bytes length. Value of the “sig_len” you've got as a parametar of the JCAppSignatureEnd() or JCAppGenerateSignature() functions. You have to allocate those bytes before calling this function.
sig_len	Length of the allocated bytes in a sig array.

JCAppPutObj

Function description

Before calling this function, NFC tag must be in ISO 14443-4 mode and JCApp should be selected using JCAppSelectByAid() with AID = “\xA0\xF0\xF1\xF2\xF3\x00\x01\x00\x01”.

Function declaration (C language)

```
UFR_STATUS JCAppPutObj(uint8_t obj_type,
                       uint8_t obj_index,
                       uint8_t *obj,
                       int16_t obj_size,
                       uint8_t *id,
                       uint8_t id_size);
```

Parameters

obj_type	0 for certificate containing RSA public key, 1 for certificate containing ECDSA public key and 2 for the CA (certificate authority).
obj_index	For each of the certificates containing RSA or ECDSA public keys there is 3 different corresponding private keys that should be set before placing the certificates themselves. Their indexes are from 0 to 2. For CA there is 12 memory slots so there indexes can be from 0 to 11.
obj	Pointer to an array containing object (certificate).
obj_size	Length of the object (certificate).
id	Pointer to an array containing **object id**. Object id is a symbolic value and have to be unique on the card.
id_size	Length of the **object id**. Minimum object id length can be 1 and maximum 253.

JCAppPutObjSubject

Function description

Before calling this function, NFC tag must be in ISO 14443-4 mode and JCApp should be selected using JCAppSelectByAid() with AID = “\xA0\xF0\xF1\xF2\xF3\x00\x01\x00\x01”.

Function declaration (C language)

```
UFR_STATUS JCAppPutObjSubject(uint8_t obj_type,
                              uint8_t obj_index,
                              uint8_t *subject,
                              uint8_t size);
```

Parameters

obj_type	0 for certificate containing RSA public key, 1 for certificate containing ECDSA public key and 2 for the CA (certificate authority).
obj_index	For each of the certificates containing RSA or ECDSA public keys there is 3 different corresponding private keys that should be set before placing the certificates themselves. Their indexes are from 0 to 2. For CA there is 12 memory slots so there indexes can be from 0 to 11.
subject	Pointer to an array containing subject. Subject is a symbolic value linked to a appropriate certificate by the same obj_type and index.
size	Length of the subject. Maximum subject length is 255.

JCAppInvalidateCert

Function description

Using this function you can delete certificate object from a card. This include subjects linked to a certificate.

Before calling this function, NFC tag must be in ISO 14443-4 mode and JCApp should be selected using JCAppSelectByAid() with AID = “\xA0\xF0\xF1\xF2\xF3\x00\x01\x00\x01”.

Function declaration (C language)

```
UFR_STATUS JCAppInvalidateCert(uint8_t obj_type,
                               uint8_t obj_index);
```

Parameters

obj_type	0 for certificate containing RSA public key, 1 for certificate containing ECDSA public key and 2 for the CA (certificate authority).
obj_index	For each of the certificates containing RSA or ECDSA public keys there is 3 different corresponding private keys that should be set before placing the

	certificates themselves. Their indexes are from 0 to 2. For CA there is 12 memory slots so there indexes can be from 0 to 11.

JCAppGetObjId

Function description

This function you always have to call 2 times. Before first call you have to set parameter ***id*** to **null** and you will get ***id_size*** of the obj_type at obj_index. Before second call you have to allocate an array of the returned ***id_size*** bytes and pass that array using parameter ***id***. Before second call, ****id_size*** should be set to a value of the exact bytes allocated.

Before calling this function, NFC tag must be in ISO 14443-4 mode and JCApp should be selected using JCAppSelectByAid() with AID = "\xA0\xF0\xF1\xF2\xF3\x00\x01\x00\x01".

Function declaration (C language)

```
UFR_STATUS JCAppGetObjId(uint8_t obj_type,
                         uint8_t obj_index,
                         uint8_t *id,
                         uint16_t *id_size);
```

Parameters

obj_type	0 for certificate containing RSA public key, 1 for certificate containing ECDSA public key and 2 for the CA (certificate authority).
obj_index	For each of the certificates containing RSA or ECDSA public keys there is 3 different corresponding private keys that should be set before placing the certificates themselves. Their indexes are from 0 to 2. For CA there is 12 memory slots so there indexes can be from 0 to 11.
id	When id == NULL, function returns id_size.
id_size	Before second call, *id_size should be set to a value of the exact bytes allocated.

JCAppGetObjSubject

Function description

This function you always have to call 2 times. Before first call you have to set parameter ***subject*** to **null** and you will get ***size*** of the obj_type at obj_index. Before second call you have to allocate array of returned ***size*** bytes and pass that array using parameter ***subject***. Before second call, ****size*** should be set to a value of the exact bytes allocated.

Before calling this function, NFC tag must be in ISO 14443-4 mode and JCApp should be selected

using JCAppSelectByAid() with AID = “\xA0\xF0\xF1\xF2\xF3\x00\x01\x00\x01”.

Function declaration (C language)

```
UFR_STATUS JCAppGetObjSubject(uint8_t obj_type,
                              uint8_t obj_index,
                              uint8_t *subject,
                              uint16_t *size);
```

Parameters

obj_type	0 for certificate containing RSA public key, 1 for certificate containing ECDSA public key and 2 for the CA (certificate authority).
obj_index	For each of the certificates containing RSA or ECDSA public keys there is 3 different corresponding private keys that should be set before placing the certificates themselves. Their indexes are from 0 to 2. For CA there is 12 memory slots so there indexes can be from 0 to 11.
subject	When subject == NULL, function returns size.
size	Before second call, *size should be set to a value of the exact bytes allocated.

JCAppGetObj

Function description

This function you always have to call 2 times. Before first call you have to set parameter ***obj*** to **null** and you will get ***size*** of the obj_type at obj_index. Before second call you have to allocate array of returned ***size*** bytes and pass that array using parameter ***obj***. Before second call, ****size*** should be set to a value of the exact bytes allocated.

Before calling this function, NFC tag must be in ISO 14443-4 mode and JCApp should be selected using JCAppSelectByAid() with AID = “\xA0\xF0\xF1\xF2\xF3\x00\x01\x00\x01”.

Function declaration (C language)

```
UFR_STATUS JCAppGetObj(uint8_t obj_type,
                       uint8_t obj_index,
                       uint8_t *obj,
                       int16_t size);
```

Parameters

obj_type	0 for certificate containing RSA public key, 1 for certificate containing ECDSA public key and 2 for the CA (certificate authority).
obj_index	For each of the certificates containing RSA or ECDSA public keys there is 3 different corresponding private keys that should be set before placing the certificates themselves. Their indexes are from 0 to 2. For CA there is 12

	memory slots so there indexes can be from 0 to 11.
obj	When obj == NULL, function returns size.
size	Before second call, *size should be set to a value of the exact bytes allocated.

BASE HD UFR SUPPORT FUNCTIONS

UfrXrcLockOn

Function description

Electric strike switches when the function called. Pulse duration determined by function.

Function declaration (C language)

```
UFR_STATUS UfrXrcLockOn(uint8_t pulse_duration);
```

Parameter

pulse_duration	pulse_duration is strike switch on period in ms

UfrXrcRelayState

Function description

Function switches relay.

Function declaration (C language)

```
UFR_STATUS UfrXrcRelayState(uint8_t state);
```

Parameter

state	if the state is 1, then relay is switch on, and if state is 0, then relay is switch off

UfrXrcGetIoState

Function description

Function returns states of 3 IO pins.

Function declaration (C language)

```
UFR_STATUS UfrXrcGetIoState(uint8_t *intercom,
                            uint8_t *door,
                            uint8_t *relay_state);
```

Parameters

intercom	shows that there is voltage at the terminals for intercom connection, or not
door	shows that the door's magnetic switch opened or closed
relay_state	is 1 if relay switch on, and 0 if relay switch off

FUNCTIONS FOR RF ANALOG REGISTERS SETTING

These functions allow you to adjust the value of several registers on PN512. These are registers: RFCfgReg, RxThresholdReg, GsNOnReg, GsNOffReg, CWGsPReg, ModGsPReg. This can be useful if you want to increase the operation distance of card, or when it is necessary to reduce the impact of environmental disturbances.

SetRfAnalogRegistersTypeA

SetRfAnalogRegistersTypeB

SetRfAnalogRegistersISO14443_212

SetRfAnalogRegistersISO14443_424

Function description

Functions allow adjusting values of registers RFCfgReg and RxThresholdReg. Registry setting is applied to the appropriate type of communication with tag. There are ISO14443 Type A, ISO14443 TypeB, and ISO14443-4 on higher communication speeds (211 and 424 Kbps).

Functions declaration (C language):

```
UFR_STATUS SetRfAnalogRegistersTypeA(uint8_t ThresholdMinLevel,
                                     uint8_t ThresholdCollLevel,
                                     uint8_t RFLevelAmp,
                                     uint8_t RxGain,
                                     uint8_t RFLevel);

UFR_STATUS SetRfAnalogRegistersTypeB(uint8_t ThresholdMinLevel,
                                     uint8_t ThresholdCollLevel,
                                     uint8_t RFLevelAmp,
                                     uint8_t RxGain,
                                     uint8_t RFLevel);

UFR_STATUS SetRfAnalogRegistersISO14443_212(
                                     uint8_t ThresholdMinLevel,
                                     uint8_t ThresholdCollLevel,
                                     uint8_t RFLevelAmp,
                                     uint8_t RxGain,
                                     uint8_t RFLevel);

UFR_STATUS SetRfAnalogRegistersISO14443_424(
                                     uint8_t ThresholdMinLevel,
                                     uint8_t ThresholdCollLevel,
                                     uint8_t RFLevelAmp,
                                     uint8_t RxGain,
                                     uint8_t RFLevel);
```

Parameters

ThresholdMinLevel	value in range 0 - 15, part of RxThresholdReg
ThresholdCollLevel	value in range 0 - 7, part of RxThresholdReg
RFLevelAmp	0 or 1, part of RFCfgReg
RxGain	value in range 0 - 7, part of RFCfgReg
RFLevel	value in range 0 - 15, part of RFCfgReg

SetRfAnalogRegistersTypeADefault

SetRfAnalogRegistersTypeBDefault

SetRfAnalogRegistersISO14443_212Default

SetRfAnalogRegistersISO14443_424Default

Function description

The functions set the factory default settings of the registers RFCfgReg and RxThresholdReg.

Functions declaration (C language):

```
UFR_STATUS SetRfAnalogRegistersTypeADefault(void);

UFR_STATUS SetRfAnalogRegistersTypeBDefault(void);

UFR_STATUS SetRfAnalogRegistersISO14443_212Default(void);

UFR_STATUS SetRfAnalogRegistersISO14443_424Default(void);
```

GetRfAnalogRegistersTypeA

GetRfAnalogRegistersTypeB

GetRfAnalogRegistersISO14443_212

GetRfAnalogRegistersISO14443_424

Function description

The functions read the value of the registers RFCfgReg and RxThresholdReg.

Functions declaration (C language):

```
UFR_STATUS GetRfAnalogRegistersTypeA(uint8_t *ThresholdMinLevel,
                                     uint8_t *ThresholdCollLevel,
                                     uint8_t *RFLevelAmp,
                                     uint8_t *RxGain,
                                     uint8_t *RFLevel);

UFR_STATUS GetRfAnalogRegistersTypeB(uint8_t *ThresholdMinLevel,
                                     uint8_t *ThresholdCollLevel,
                                     uint8_t *RFLevelAmp,
                                     uint8_t *RxGain,
                                     uint8_t *RFLevel);

UFR_STATUS GetRfAnalogRegistersISO14443_212(
                                     uint8_t *ThresholdMinLevel,
                                     uint8_t *ThresholdCollLevel,
                                     uint8_t *RFLevelAmp,
                                     uint8_t *RxGain,
                                     uint8_t *RFLevel);

UFR_STATUS GetRfAnalogRegistersISO14443_424(
                                     uint8_t *ThresholdMinLevel,
                                     uint8_t *ThresholdCollLevel,
                                     uint8_t *RFLevelAmp,
                                     uint8_t *RxGain,
                                     uint8_t *RFLevel);
```

Parameters

ThresholdMinLevel	value in range 0 - 15, part of RxThresholdReg
ThresholdCollLevel	value in range 0 - 7, part of RxThresholdReg
RFLevelAmp	0 or 1, part of RFCfgReg
RxGain	value in range 0 - 7, part of RFCfgReg
RFLevel	value in range 0 - 15, part of RFCfgReg

SetRfAnalogRegistersTypeATrans

SetRfAnalogRegistersTypeBTrans

Function description

Functions allow adjusting values of registers RFCfgReg, RxThresholdReg, GsNOnReg, GsNOffReg, CWGsPReg, ModGsPReg. Registry setting is applied to the appropriate type of communication with tag. There are ISO14443 Type A, ISO14443 TypeB, and ISO14443-4 on higher communication speeds (211 and 424 Kbps).

Functions declaration (C language):

```
UFR_STATUS SetRfAnalogRegistersTypeATrans(
                                uint8_t ThresholdMinLevel,
                                uint8_t ThresholdCollLevel,
                                uint8_t RFLevelAmp,
                                uint8_t RxGain,
                                uint8_t RFLevel,
                                uint8_t CWGsNOn,
                                uint8_t ModGsNOn,
                                uint8_t CWGsP,
                                uint8_t CWGsNOff,
                                uint8_t ModGsNOff);

UFR_STATUS SetRfAnalogRegistersTypeBTrans(
                                uint8_t ThresholdMinLevel,
                                uint8_t ThresholdCollLevel,
                                uint8_t RFLevelAmp,
                                uint8_t RxGain,
                                uint8_t RFLevel,
                                uint8_t CWGsNOn,
                                uint8_t ModGsNOn,
                                uint8_t CWGsP,
                                uint8_t ModGsP);
```

Parameters

ThresholdMinLevel	value in range 0 - 15, part of RxThresholdReg
ThresholdCollLevel	value in range 0 - 7, part of RxThresholdReg
RFLevelAmp	0 or 1, part of RFCfgReg
RxGain	value in range 0 - 7, part of RFCfgReg
RFLevel	value in range 0 - 15, part of RFCfgReg

CWGsNOn	value in range 0 - 15, part of GsNOnReg
ModGsNOn	value in range 0 - 15, part of GsNOnReg
CWGsP	value of CWGsPReg (0 - 47)
CWGsNOff	value in range 0 - 15, part of GsNOffReg
ModGsNOff	value in range 0 - 15, part of GsNOffReg
ModGsP	value of ModGsPReg (0 - 47)

GetRfAnalogRegistersTypeATrans

GetRfAnalogRegistersTypeBTrans

Function description

The functions read the value of the registers RFCfgReg, RxThresholdReg, GsNOnReg, GsNOffReg, CWGsPReg, ModGsPReg.

Functions declaration (C language):

```
UFR_STATUS GetRfAnalogRegistersTypeATrans(
                                  uint8_t *ThresholdMinLevel,
                                  uint8_t *ThresholdCollLevel,
                                  uint8_t *RFLevelAmp,
                                  uint8_t *RxGain,
                                  uint8_t *RFLevel,
                                  uint8_t *CWGsNOn,
                                  uint8_t *ModGsNOn,
                                  uint8_t *CWGsP,
                                  uint8_t *CWGsNOff,
                                  uint8_t *ModGsNOff);

UFR_STATUS GetRfAnalogRegistersTypeBTrans(
                                  uint8_t *ThresholdMinLevel,
                                  uint8_t *ThresholdCollLevel,
                                  uint8_t *RFLevelAmp,
                                  uint8_t *RxGain,
                                  uint8_t *RFLevel,
                                  uint8_t *CWGsNOn,
                                  uint8_t *ModGsNOn,
                                  uint8_t *CWGsP,
                                  uint8_t *ModGsP);
```

Parameters

ThresholdMinLevel	value in range 0 - 15, part of RxThresholdReg
ThresholdCollLevel	value in range 0 - 7, part of RxThresholdReg
RFLevelAmp	0 or 1, part of RFCfgReg
RxGain	value in range 0 - 7, part of RFCfgReg
RFLevel	value in range 0 - 15, part of RFCfgReg
CWGsNOn	value in range 0 - 15, part of GsNOnReg
ModGsNOn	value in range 0 - 15, part of GsNOnReg
CWGsP	value of CWGsPReg (0 - 47)
CWGsNOff	value in range 0 - 15, part of GsNOffReg

ModGsNOff	value in range 0 - 15, part of GsNOffReg
ModGsP	value of ModGsPReg (0 - 47)

FUNCTIONS FOR DEVICE SIGNALIZATION SETTINGS

GreenLedBlinkingTurnOn

Function description

The function allows the blinking of the green diode independently of the user's signaling command (default setting).

Function declaration (C language)

```
UFR_STATUS GreenLedBlinkingTurnOn(void);
```

GreenLedBlinkingTurnOff

Function description

The function prohibits the blinking of the green diode independently of the user's signaling command. LED and sound signaling occurs only on the user command.

Function declaration (C language)

```
UFR_STATUS GreenLedBlinkingTurnOff(void);
```

FUNCTIONS FOR DISPLAY CONTROL

SetDisplayData

Function description

This feature working with LED RING 24 display module.
Function enables sending data to the display. A string of data contains information about the intensity of color in each cell of the display. Each cell has three LED (red, green and blue). For each cell of the three bytes is necessary. The first byte indicates the intensity of the green color, the second byte indicates the intensity of the red color, and the third byte indicates the intensity of blue color. For example, if the display has 16 cells, an array contains 48 bytes. Value of intensity is in range from 0 to 255.

Function declaration (C language)

```
UFR_STATUS SetDisplayData(uint8_t *display_data,
                          uint8_t data_length);
```

Parameters

`display_data`	pointer to data array
`data_length`	number of data into array

SetDisplayIntensity

Function description

Function sets the intensity of light on the display. Value of intensity is in range 0 to 100.

Function declaration (C language)

```
UFR_STATUS SetDisplayIntensity(uint8_t intensity);
```

Parameter

`intensity`	value of intensity (0 – 100)

GetDisplayIntensity

Function description

Function gets the intensity of light on the display.

Function declaration (C language)

```
UFR_STATUS GetDisplayIntensity(uint8_t *intensity);
```

Parameter

`intensity`	pointer to intensity

Functions for transceive mode

For uFR PLUS devices only

In this mode, the data is entered via the serial port transmitted through the RF field to the card, and the card response is transmitted to the serial port.

card_transceive_mode_start

Function description

Function sets the parameters for transceive mode. If the hardware CRC option is used, then only command bytes sent to card (hardware will add two bytes of CRC to the end of RF packet). If this option did not use, then command bytes and two bytes of CRC sent to card (i.e. ISO14443 typeA CRC). Timeout for card response in us sets.

Card is selected and waiting for commands.

Function declaration (C language)

```
UFR_STATUS card_transceive_mode_start(uint8_t tx_crc,
                                      uint8_t rx_crc,
                                      uint32_t rf_timeout,
                                      uint32_t uart_timeout);
```

Parameters

tx_crc	hardware RF TX crc using (1 - yes, 0 - no)
rx_crc	hardware RF RX crc using (1 - yes, 0 - no)
rf_timeout	timeout for card response in us
uart_timeout	timeout for UART response in ms

card_transceive_mode_stop

Function description

The function returns the reader to normal mode.

Function declaration (C language)

```
UFR_STATUS DL_API card_transceive_mode_stop(void);
```

uart_transceive

Function description

The function sends data through the serial port to the card.

Function declaration (C language)

```
UFR_STATUS DL_API uart_transceive(uint8_t *send_data,
                                  uint8_t send_len,
                                  uint8_t *rcv_data,
                                  uint32_t bytes_to_receive,
                                  uint32_t *rcv_len);
```

Parameters

send_data	pointer to data array for sending to card
send_len	number of bytes for sending
rcv_data	pointer to data array received from card
bytes_to_receive	expected number of bytes received from card
rcv_len	number of bytes received from card

Appendix: ERROR CODES (DL_STATUS result)

UFR_OK	0x00
UFR_COMMUNICATION_ERROR	0x01
UFR_CHKSUM_ERROR	0x02
UFR_READING_ERROR	0x03
UFR_WRITING_ERROR	0x04
UFR_BUFFER_OVERFLOW	0x05
UFR_MAX_ADDRESS_EXCEEDED	0x06
UFR_MAX_KEY_INDEX_EXCEEDED	0x07
UFR_NO_CARD	0x08
UFR_COMMAND_NOT_SUPPORTED	0x09
UFR_FORBIDEN_DIRECT_WRITE_IN_SECTOR_TRAILER	0x0A
UFR_ADDRESSED_BLOCK_IS_NOT_SECTOR_TRAILER	0x0B
UFR_WRONG_ADDRESS_MODE	0x0C
UFR_WRONG_ACCESS_BITS_VALUES	0x0D
UFR_AUTH_ERROR	0x0E
UFR_PARAMETERS_ERROR	0x0F
UFR_MAX_SIZE_EXCEEDED	0x10
UFR_UNSUPPORTED_CARD_TYPE	0x11
UFR_COUNTER_ERROR	0x12
UFR_WRITE_VERIFICATION_ERROR	0x70
UFR_BUFFER_SIZE_EXCEEDED	0x71
UFR_VALUE_BLOCK_INVALID	0x72
UFR_VALUE_BLOCK_ADDR_INVALID	0x73
UFR_VALUE_BLOCK_MANIPULATION_ERROR	0x74
UFR_WRONG_UI_MODE	0x75
UFR_KEYS_LOCKED	0x76
UFR_KEYS_UNLOCKED	0x77
UFR_WRONG_PASSWORD	0x78
UFR_CAN_NOT_LOCK_DEVICE	0x79
UFR_CAN_NOT_UNLOCK_DEVICE	0x7A
UFR_DEVICE_EEPROM_BUSY	0x7B
UFR_RTC_SET_ERROR	0x7C
UFR_TAG_UNKNOWN	0x7D
UFR_COMMUNICATION_BREAK	0x50
UFR_NO_MEMORY_ERROR	0x51
UFR_CAN_NOT_OPEN_READER	0x52
UFR_READER_NOT_SUPPORTED	0x53
UFR_READER_OPENING_ERROR	0x54
UFR_READER_PORT_NOT_OPENED	0x55
UFR_CANT_CLOSE_READER_PORT	0x56
UFR_TIMEOUT_ERR	0x90
UFR_FT_STATUS_ERROR_1	0xA0
UFR_FT_STATUS_ERROR_2	0xA1

UFR_FT_STATUS_ERROR_3	0xA2
UFR_FT_STATUS_ERROR_4	0xA3
UFR_FT_STATUS_ERROR_5	0xA4
UFR_FT_STATUS_ERROR_6	0xA5
UFR_FT_STATUS_ERROR_7	0xA6
UFR_FT_STATUS_ERROR_8	0xA7
UFR_FT_STATUS_ERROR_9	0xA8
UFR_WRONG_NDEF_CARD_FORMAT	0x80
UFR_NDEF_MESSAGE_NOT_FOUND	0x81
UFR_NDEF_UNSUPPORTED_CARD_TYPE	0x82
UFR_NDEF_CARD_FORMAT_ERROR	0x83
UFR_MAD_NOT_ENABLED	0x84
UFR_MAD_VERSION_NOT_SUPPORTED	0x85
multiple units - return from the functions with ReaderList_ prefix in name	
UFR_DEVICE_WRONG_HANDLE	0x100
UFR_DEVICE_INDEX_OUT_OF_BOUND	0x101
UFR_DEVICE_ALREADY_OPENED	0x102
UFR_DEVICE_ALREADY_CLOSED	0x103
UFR_DEVICE_IS_NOT_CONNECTED	0x104
Originality Check Error Codes	
UFR_NOT_NXP_GENUINE	0x200
UFR_OPEN_SSL_DYNAMIC_LIB_FAILED	0x201
UFR_OPEN_SSL_DYNAMIC_LIB_NOT_FOUND	0x202
UFR_NOT_IMPLEMENTED	0x1000
UFR_COMMAND_FAILED	0x1001
APDU Error Codes	
UFR_APDU_JC_APP_NOT_SELECTED	0x6000
UFR_APDU_JC_APP_BUFF_EMPTY	0x6001
UFR_APDU_WRONG_SELECT_RESPONSE	0x6002
UFR_APDU_WRONG_KEY_TYPE	0x6003
UFR_APDU_WRONG_KEY_SIZE	0x6004
UFR_APDU_WRONG_KEY_PARAMS	0x6005
UFR_APDU_WRONG_ALGORITHM	0x6006
UFR_APDU_PLAIN_TEXT_SIZE_EXCEEDED	0x6007
UFR_APDU_UNSUPPORTED_KEY_SIZE	0x6008
UFR_APDU_UNSUPPORTED_ALGORITHMS	0x6009
UFR_APDU_RECORD_NOT_FOUND	0x600A
UFR_APDU_SW_TAG	0x0A0000

DESFIRE Card Status Error Codes

READER_ERROR	2999
NO_CARD_DETECTED	3000
CARD_OPERATION_OK	3001
WRONG_KEY_TYPE	3002
KEY_AUTH_ERROR	3003
CARD_CRYPTO_ERROR	3004
READER_CARD_COMM_ERROR	3005
PC_READER_COMM_ERROR	3006
COMMIT_TRANSACTION_NO_REPLY	3007
COMMIT_TRANSACTION_ERROR	3008
DESFIRE_CARD_NO_CHANGES	0x0C0C
DESFIRE_CARD_OUT_OF_EEPROM_ERROR	0x0C0E
DESFIRE_CARD_ILLEGAL_COMMAND_CODE	0x0C1C
DESFIRE_CARD_INTEGRITY_ERROR	0x0C1E
DESFIRE_CARD_NO_SUCH_KEY	0x0C40
DESFIRE_CARD_LENGTH_ERROR	0x0C7E
DESFIRE_CARD_PERMISSION_DENIED	0x0C9D
DESFIRE_CARD_PARAMETER_ERROR	0x0C9E
DESFIRE_CARD_APPLICATION_NOT_FOUND	0x0CA0
DESFIRE_CARD_APPL_INTEGRITY_ERROR	0x0CA1
DESFIRE_CARD_AUTHENTICATION_ERROR	0x0CAE
DESFIRE_CARD_ADDITIONAL_FRAME	0x0CAF
DESFIRE_CARD_BOUNDARY_ERROR	0x0CBE
DESFIRE_CARD_PICC_INTEGRITY_ERROR	0x0CC1

DESFIRE_CARD_COMMAND_ABORTED	0x0CCA
DESFIRE_CARD_PICC_DISABLED_ERROR	0x0CCD
DESFIRE_CARD_COUNT_ERROR	0x0CCE
DESFIRE_CARD_DUPLICATE_ERROR	0x0CDE
DESFIRE_CARD_EEPROM_ERROR_DES	0x0CEE
DESFIRE_CARD_FILE_NOT_FOUND	0x0CF0
DESFIRE_CARD_FILE_INTEGRITY_ERROR	0x0CF1

Appendix: DLogic CardType enumeration

TAG_UNKNOWN	0x00
DL_MIFARE_ULTRALIGHT	0x01
DL_MIFARE_ULTRALIGHT_EV1_11	0x02
DL_MIFARE_ULTRALIGHT_EV1_21	0x03
DL_MIFARE_ULTRALIGHT_C	0x04
DL_NTAG_203	0x05
DL_NTAG_210	0x06
DL_NTAG_212	0x07
DL_NTAG_213	0x08
DL_NTAG_215	0x09
DL_NTAG_216	0x0A
DL_MIKRON_MIK640D	0x0B
NFC_T2T_GENERIC	0x0C
DL_MIFARE_MINI	0x20
DL_MIFARE_CLASSIC_1K	0x21
DL_MIFARE_CLASSIC_4K	0x22
DL_MIFARE_PLUS_S_2K	0x23
DL_MIFARE_PLUS_S_4K	0x24
DL_MIFARE_PLUS_X_2K	0x25
DL_MIFARE_PLUS_X_4K	0x26
DL_MIFARE_DESFIRE	0x27
DL_MIFARE_DESFIRE_EV1_2K	0x28
DL_MIFARE_DESFIRE_EV1_4K	0x29
DL_MIFARE_DESFIRE_EV1_8K	0x2A
DL_MIFARE_DESFIRE_EV2_2K	0x2B
DL_MIFARE_DESFIRE_EV2_4K	0x2C
DL_MIFARE_DESFIRE_EV2_8K	0x2D
DL_UNKNOWN_ISO_14443_4	0x40
DL_GENERIC_ISO14443_4	0x40
DL_GENERIC_ISO14443_TYPE_B	0x41
DL_IMEI_UID	0x80

Appendix: DLogic reader type enumeration

Value	Reader name
0xD1150021	µFR Classic
0xD2150021	µFR Advance
0xD3150021	µFR PRO
0xD1180022	µFR Nano Classic
0xD3180022	µFR Nano PRO
0xD1190222	µFR Nano Classic RS232
0xD3190222	µFR Nano PRO RS232
0xD11A0022	µFR Classic Card Size
0xD21A0022	µFR Advance Card Size
0xD31A0022	µFR PRO Card Size
0xD11A0222	µFR Classic Card Size RS232
0xD21A0222	µFR Advance Card Size RS232
0xD31A0222	µFR PRO Card Size RS232
0xD11B0022	µFR Classic Card Size RF-AMP
0xD21B0022	µFR Advance Card Size RF-AMP
0xD31B0022	µFR PRO Card Size RF-AMP
0xD11B0222	µFR Classic Card Size RS232 RF-AMP
0xD21B0222	µFR Advance Card Size RS232 RF-AMP
0xD31B0222	µFR PRO Card Size RS232 RF-AMP

Appendix: FTDI troubleshooting

On Windows systems, it is pretty straightforward with .msi installer executable.

On Linux platforms, few more things must be provided:

- Appropriate user permissions on FTDI and uFCoder libraries

- "`ftdi_sio`" and helper module "`usbserial`" must be removed/unloaded for proper functioning. Each time device is plugged in, Linux kernel loads appropriate module. So, each time device is plugged, you must issue following command in CLI:

```
sudo rmmod ftdi_sio usbserial
```

- This can be painful, so good practice is to blacklist these two modules in "`etc/modprobe.d/`" directory. Create new file called "`ftdi.conf`" and add following line :

```
#disable auto load FTDI modules - D-LOGIC
blacklist ftdi_sio
blacklist usbserial
```

On macOS, it is good enough to follow FTDI's guidelines for proper driver installation.

Update: since Mac OS version 10.11 El Capitan, macOS introduces SIP (System Integration Protection) which does not allow user to write into system directories like 'usr/lib' and similar, which makes a lot of problems in implementation. For that purpose, 'libuFCoder.dylib' library embeds FTDI's library too, so there is no need for installation of FTDI's drivers.

Previous macOS versions works fine with FTDI's D2XX drivers.

D2XX drivers links: http://www.ftdichip.com/Drivers/D2XX.htm

Direct link to current drivers: http://www.ftdichip.com/Drivers/D2XX/MacOSX/D2XX1.2.2.dmg

Install instructions are located in the archive. You need to install/copy needed drivers.

Other kernel extensions problems:

To successfully open the FTDI port, it is necessary to check if another FTDI module (kernel extension) is loaded, and if it is, it needs to be deactivated.

Procedure:

1. plug-in FTDI device (uFReader) and wait a few seconds
2. open console
3. you can check if device is detected:

```
$ sudo dmesg
FTDIUSBSerialDriver:         0  **4036001** start - ok
```

4. check if kernel extension is loaded for FTDI:

```
$ kextstat | grep -i ftdi
```

```
      94  0 0xffffff7f82041000 0x8000   0x8000
**com.FTDI.driver.FTDIUSBSerialDriver** (2.2.18) <70 34 5 4 3  1>
```

5. you need to deactivate it - eject it from memory

```
sudo kextunload /System/Library/Extensions/FTDIUSBSerialDriver.kext
```

Remark - with the system OS X 10.11 (El Capitan)

After the module is removed, it returns again. It is necessary to download the Helper from FTDI site and to run it on the machine, and after that restart is required.

Information from site:

If using a device with standard FTDI vendor and product identifiers, install D2xxHelper to prevent OS X 10.11 (El Capitan) claiming the device as a serial port (locking out D2XX programs).

This is how to load driver on El Capitan:

```
$ kextstat | grep -i ftd  146  0 0xffffff7f82d99000 0x7000    0x7000
com.apple.driver.AppleUSBFTDI (5.0.0) D853EEF2-435D-370E-AFE3-DE49CA29DF47 <123 38 5 4 3
1>
```

```
$ sudo kextunload /System/Library/Extensions/AppleUSBFTDI.kext
```

After this, FTDI devices are ready to work with FTD2XX libraries.

Appendix: Change log

Date	Description	API revision	refers to the lib version / firmware ver.
2018-05-29	PKI infrastructure and digital signature support	2.1	4.3.8 / 3.9.55