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”](http://dld.is.d-logic.net/index.php/documentation-download/latest-docs-ufr/communication-protocol-is21-vcom-3-5-enpdf?format=raw) 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.

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 :

blacklist ftdi\_sio

blacklist usbserial

On OSX, it is good enough to follow FTDI’s guidelines for proper driver installation.

Update: since Mac OS version 10.11 El Capitan, OSX 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 OSX versions works fine with FTDI’s D2XX drivers.

# 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”](http://dld.is.d-logic.net/index.php/documentation-download/latest-docs-ufr/communication-protocol-is21-vcom-3-5-enpdf?format=raw) 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)

SomeFunction(par1, par2) = SomeFunctionM(Handle, par1, par2)

More about Multi-function usage can be found in further reading of this document.

# Function syntax and data types in this document

By default, all function are showed as their prototypes in C++ language. All data types refers C++ types. For quick reference, always consult latest header file “uFCoder.h” at library package.

# Error codes

All functions always have return result with corresponding status code. Please refer to table ERR\_CODES in Table 1. ERR 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 uFR XRC 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 uFR XRC 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. Complete list of functions can be found in Table 2 of Appendix.

## 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.

* 1. Block manipulation commands – direct and indirect addressing

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

* 1. Value Block manipulation commands – direct and indirect addressing

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

* 1. Linear data manipulation commands

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

### NTAG related commands

Functions specific to NTAG ® family chips such as NTAG 203F, 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.

### 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.

### 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.

# 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.

## Asynchonous 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 AutoSleep 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.](http://www.nxp.com/documents/data_sheet/M001053_MF1ICS50_rev5_3.pdf)

## 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 |
| C13 C23 C33 | read, write | 3 | sector trailer |
| C12 C22 C32 | read, write, increment, decrement, transfer, restore | 2 | data block |
| C11 C21 C31 | read, write, increment, decrement, transfer, restore | 1 | data block |
| C10 C20 C30 | 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 | C23 | | C22 | | C21 | | C20 | | C13 | | C12 | | C11 | | C10 | |
| Byte 7 | C13 | | C12 | | C11 | | C10 | | C33 | | C32 | | C31 | | C30 | |
| Byte 8 | C33 | | C32 | | C31 | | C30 | | C23 | | C22 | | C21 | | C20 | |
| 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 | |
| C13 | C23 | C33 | 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!
* Key management: In transport configuration key A must be used for authentication1[[1]](#footnote-1)

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

# 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 cannot 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 tradeoff.

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
* Mifare Ultralight C
* NTAG 203, 210, 212, 213, 215, 216
* Mikron MIK640D
* NFC Type2 Tag compatible card are supported as ‘T2T generic type’, calling GetVersion 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.

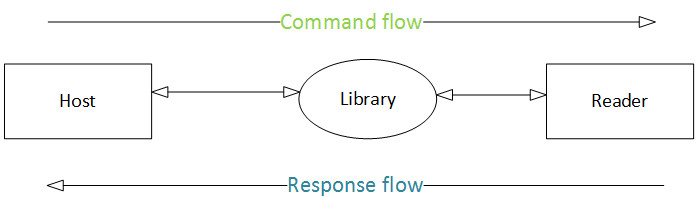
On top physical USB layer is FTDI’s direct access through D2XX drivers library.

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

|  |  |  |
| --- | --- | --- |
| uFCoder library | | |
| FTDI D2XX driver library | |  |
| USB Host Controller Driver |  |  |

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.

|  |
| --- |
| uFCoder library |
| COM port (physical or virtual) |

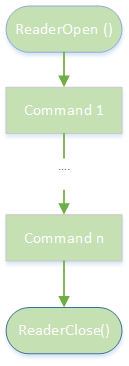
## Program flow – basic usage

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.

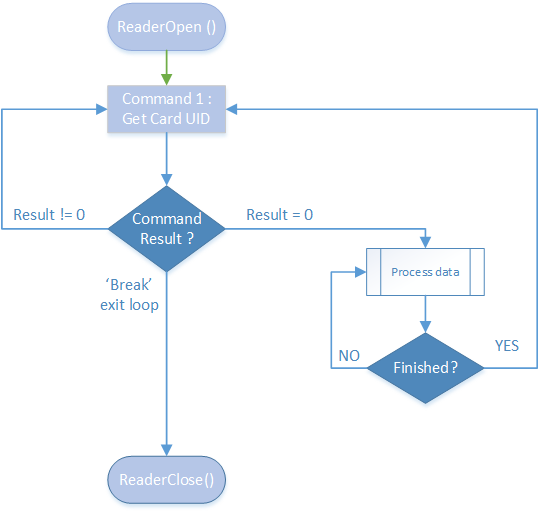


## Program flow – polling

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.



## 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 Table 1. ERR 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 |
| c\_string | GetDllVersionStr |
| c\_string | UFR\_STATUS2String |
| c\_string | GetReaderDescription |

\*\* - RFU(reserved for future use)

### ReaderOpen

Open reader communication port.

Syntax

UFR\_STATUS ReaderOpen(void)

No parameters required.

### ReaderOpenEx

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

Syntax

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 : XRC type (250 Kbps)

port\_name : 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

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

arg : Reserved for future use, must be NULL.

### ReaderReset

Physical reset of reader communication port.

Syntax

UFR\_STATUS ReaderReset(void)

No parameters required.

### ReaderClose

Close reader communication port.

Syntax

UFR\_STATUS ReaderClose(void)

No parameters required.

### ReaderStillConnected

Retrieve info if reader is still connected to host.

Syntax

UFR\_STATUS ReaderStillConnected(uint32\_t \*connected)

Parameter: 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 |

### GetReaderType

Returns reader type as a pointer to 4 byte value.

Syntax UFR\_STATUS GetReaderType(uint32\_t \*lpulReaderType)

Parameter: pointer to lpulReaderType variable.

“lpulReaderType” as result – please refer to Table 3. DLogic reader type enumeration.

### GetReaderSerialNumber

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

Syntax

UFR\_STATUS GetReaderSerialNumber(uint32\_t \*lpulSerialNumber)

Parameter: pointer to lpulSerialNumber variable.

“lpulSerialNumber “ as result holds 4 byte serial number value.

### GetReaderHardwareVersion

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

Syntax

UFR\_STATUS GetReaderHardwareVersion(uint8\_t \*version\_major,

uint8\_t \*version\_minor);

Parameter: pointers to version\_major and version\_minor variables.

“version\_minor”,” version\_major” are holding returned values of hardware version..

### GetReaderFirmwareVersion

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

Syntax

UFR\_STATUS GetReaderFirmwareVersion(uint8\_t \*version\_major,

uint8\_t \*version\_minor);

Parameter: pointers to version\_major and version\_minor variables.

“version\_minor”,” version\_major” are holding returned values of firmware version.

### GetBuildNumber

Returns reader firmware build version as one byte representation.

Syntax

UFR\_STATUS GetBuildNumber(uint8\_t \*build)

Parameter: pointer to build variable.

“build” holds returned value of firmware build version.

### GetReaderSerialDescription

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

Syntax

UFR\_STATUS GetReaderSerialDescription(uint8\_t pSerialDescription[8])

Parameter: pointer to pSerialDescription variable.

“pSerialDescription” holds returned value of descriptive reader’s name.

### ChangeReaderPassword

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

Syntax

UFR\_STATUS ChangeReaderPassword(uint8\_t \*old\_password,

uint8\_t \*new\_password)

Parameters description:

|  |  |
| --- | --- |
| old\_password | pointer to the 8 bytes array containing current password |
| new\_password | pointer to the 8 bytes array containing new password |

### ReaderKeyWrite

Store a new key or change existing key under provided index parameter.

Syntax

UFR\_STATUS ReaderKeyWrite(const unsigned char \*aucKey,   
 unsigned char ucKeyIndex)

|  |  |
| --- | --- |
| 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

Lock reader’s keys to prevent further changing.

Syntax

UFR\_STATUS ReaderKeysLock(const uint8\_t \*password);

“password” – pointer to the 8 bytes array containing valid password.

### ReaderKeysUnlock

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

Syntax

UFR\_STATUS ReaderKeysUnlock(const uint8\_t \*password);

“password” – pointer to the 8 bytes array containing valid password.

### ReadUserData

Read user data written in device NV memory.

User data is 16 byte long.

Syntax

UFR\_STATUS ReadUserData(uint8\_t \*aucData)

“aucData” – pointer to 16 byte array containing user data.

### WriteUserData

Write user data into device’s NV memory.

User data is 16 byte long.

Syntax

UFR\_STATUS WriteUserData(uint8\_t \*aucData)

“aucData” – pointer to 16 byte array containing user data.

### UfrEnterSleepMode

Turn device into Sleep mode.

Syntax

UFR\_STATUS UfrEnterSleepMode(void)

No parameters used.

### UfrLeaveSleepMode

Wake up device from Sleep mode.

Syntax

UFR\_STATUS UfrLeaveSleepMode(void)

No parameters used.

### AutoSleepSet

Turn device into Sleep mode after certain amount of time.

Syntax

UFR\_STATUS AutoSleepSet(uint8\_t seconds\_wait)

**“**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

Get status of AutoSleep mode.

Syntax

UFR\_STATUS AutoSleepGet(uint8\_t seconds\_wait)

“seconds\_wait” – variable returning currently set value of seconds

If parameter is 0x00, AutoSleep feature is turned off (default state).

### SetSpeedPermanently

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.

Syntax:

UFR\_STATUS SetSpeedPermanently (unsigned char tx\_speed,

unsigned char rx\_speed)

**Parameters description:**

* 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

Returns baudrate configured with previous function.

Syntax

UFR\_STATUS GetSpeedParameters ( unsigned char \*tx\_speed,

unsigned char \*rx\_speed)

**Parameters description:**

|  |  |
| --- | --- |
| tx\_speed | pointer to variable, returns configured value for transmit speed |
| rx\_speed | pointer to variable,returns configured value for receive speed |

### SetAsyncCardIdSendConfig

This function is used for “Asynchonous 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.

Syntax

UFR\_STATUS SetAsyncCardIdSendConfig (

unsigned char send\_enable,

unsigned char prefix\_enable,

unsigned char prefix,

unsigned char suffix,

unsigned long async\_baud\_rate);

Parameters description:

|  |  |
| --- | --- |
| send\_enable | turn feature on/off (0/1) |
| prefix\_enable | use prefix or not (0/1) |
| prefix | prefix character |
| suffix | suffix character |
| async\_baud\_rate | baud rate value (e.g. 9600) |

### GetAsyncCardIdSendConfig

Returns info about parameters configured with previous function.

Syntax

UFR\_STATUS GetAsyncCardIdSendConfig (

unsigned char \*send\_enable,

unsigned char \*prefix\_enable,

unsigned char \*prefix,

unsigned char \*suffix,

unsigned long \*async\_baud\_rate);

Parameters description:

|  |  |
| --- | --- |
| 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 |
| async\_baud\_rate | pointer to variable holding configured baud rate |

### ReaderUISignal

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 |

Syntax

UFR\_STATUS ReaderUISignal( uint8\_t light\_signal\_mode,

uint8\_t beep\_signal\_mode)

### UfrRedLightControl

This function turns Red LED only.

If “light\_status” value is 1, red light wil be constantly turned on until receive “light\_status “ value 0.

Syntax

UFR\_STATUS UfrRedLightControl(uint8\_t light\_status)

### SetSpeakerFrequency

This function plays constant sound of “frequency” Herz.

Syntax

UFR\_STATUS SetSpeakerFrequency(uint16\_t frequency)

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

## Helper library functions

NOTE: Three following functions have return type “c\_string”**,** where c\_string is defined as

“typedef const char \* c\_string”.

### GetDllVersionStr

This is helper library function. Returns library version data as a string.

Syntax

c\_string GetDllVersionStr(void)

No parameters used.

### UFR\_STATUS2String

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 [Table 1. ERR CODES (DL\_STATUS result)](#_Error_codes)

Syntax

c\_string UFR\_Status2String(const UFR\_STATUS status)

### GetReaderDescription

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

Syntax

c\_string GetReaderDescription(void)

No parameters used.

**Following functions are reserved for future use:**

**SetDisplayData**

**SetDisplayIntensity**

**GetDisplayIntensity**

## 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

This function returns card type according to DlogicCardType enumeration. For details, please refer to Table 2. DLogicCardType enumeration

If the card type is not supported, function return the lpucCardType value equal to zero : TAG\_UNKNOWN = 0x00

Syntax

UFR\_STATUS GetDlogicCardType(uint8\_t \*lpucCardType)

Parameter: pointer to lpucCardType variable.

Variable lpucCardType holds returned value of actual card type present in RF field.

### GetCardId

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 cannot successfully handle 7 byte UIDS.

Syntax

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

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.

Syntax

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

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.

Syntax

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:

* 1. Block manipulation functions – direct and indirect addressing

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

* 1. Value Block manipulation functions – direct and indirect addressing

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

* 1. 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 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:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Sector 0 | Block 0 | Manufacturer Block |  | |
|  | Block 1 | Bytes 0 -15 | LINEAR SPACE | Linear Space starts here at Byte 0 |
|  | Block 2 | Bytes 16 - 31 |  |
|  | Block 3 | Trailer | Jumping over Trailer |
| Sector 1 | Block 0 | Bytes 32 - 47 |  |
|  | Block 1 | Bytes 48 - 63 |  |
|  | Block 2 | Bytes 64 - 79 |  |
|  | Block 3 | Trailer | Jumping over Trailer |
| Sector 2 | Block 0 | Bytes 80- 84 | Rest of Block is not changed (Bytes 5 - 15) |

*List of Mifare Classic specific functions*

|  |  |
| --- | --- |
| UFR\_STATUS | BlockRead **\*** |
| UFR\_STATUS | BlockWrite **\*** |
| UFR\_STATUS | BlockInSectorRead |
| UFR\_STATUS | BlockInSectorWrite |
| UFR\_STATUS | LinearRead **\*** |
| UFR\_STATUS | LinearWrite **\*** |
| UFR\_STATUS | LinRowRead **\*** |
| 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 |

“**\***” – function can be used with other card types

### BlockRead

Read particular block using absolute Block address.

Syntax

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 | 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) |

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

Write particular block using absolute Block address.

Syntax

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 | 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) |

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

Read particular block using relative Block in Sector address.

Syntax

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 | 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 cannot be used with card types other than Mifare Classic.

### BlockInSectorWrite

Write particular block using relative Block in Sector address.

Syntax

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 | 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 cannot be used with card types other than Mifare Classic.

### LinearRead

Read Linear data as explained in *Linear Address Data Space*.

Syntax

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 | 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) |

When using this function with other card types, auth\_mode, key\_index and key parameters are not relevant but must take default values.

### LinearWrite

Write Linear data as explained in *Linear Address Data Space*.

Syntax

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 reading |
| length | Length of data – how many bytes to read |
| bytes\_returned | Pointer to variable holding how many bytes are returned |
| 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) |

When using this function with other card types, auth\_mode, key\_index and key parameters are not relevant but must take default values.

### LinRowRead

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.

Syntax

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 | 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) |

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

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.

Syntax

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);

**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 | 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) |

### SectorTrailerWrite

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.

Syntax

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 KeyA |
| 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) |

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

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!

Syntax

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);

|  |  |
| --- | --- |
| 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 | 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) |

### ValueBlockRead

Read particular Value block using absolute Block address. This function uses Mifare Clasic 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.

Syntax

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 | 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 cannot be used with other card types except Mifare Clasic.

### ValueBlockWrite

Write particular Value block using absolute Block address. This function uses Mifare Clasic 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.

Syntax

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 | 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 cannot be used with other card types except Mifare Clasic.

### ValueBlockInSectorRead

Read particular Value block using absolute Block address. This function uses Mifare Clasic 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.

Syntax

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 cannot be used with other card types except Mifare Clasic.

### ValueBlockInSectorWrite

Write particular Value block using absolute Block address. This function uses Mifare Clasic 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.

Syntax

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 cannot be used with other card types except Mifare Clasic.

### ValueBlockIncrement

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

Syntax

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 cannot be used with other card types except Mifare Clasic.

### ValueBlockDecrement

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

Syntax

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 cannot be used with other card types except Mifare Clasic.

### ValueBlockInSectorIncrement

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

Syntax

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 cannot be used with other card types except Mifare Clasic.

### ValueBlockInSectorDecrement

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

Syntax

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 cannot be used with other card types except Mifare Clasic.

***---------------------------- dovde --------------------------------***

***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)

**unsigned long get\_ndef\_record\_count(  
 unsigned char \*ndef\_message\_cnt,   
 unsigned char \*ndef\_record\_cnt,   
 unsigned char \*ndef\_record\_array,   
 unsigned char \*empty\_ndef\_message\_cnt);**

Parameters description:

* ndef\_message\_cnt pointer to the variable containing number of NDEF messages
* ndef\_record\_cnt pointer to the variable containing number of NDEF records
* 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)

**unsigned long read\_ndef\_record(unsigned char message\_nr,   
 unsigned char record\_nr,   
 unsigned char \*tnf,   
 unsigned char \*type\_record,   
 unsigned char \*type\_length,   
 unsigned char \*id,   
 unsigned char \*id\_length,  
 unsigned char \*payload,   
 unsigned long \*payload\_length);**

Parameters description:

* message\_nr NDEF message ordinal (starts form 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)

**unsigned long write\_ndef\_record(  
unsigned char message\_nr,   
unsigned char \*tnf,   
unsigned char \*type\_record,   
unsigned char \*type\_length,   
unsigned char \*id,   
unsigned char \*id\_length,  
unsigned char \*payload,   
unsigned long \*payload\_length,   
unsigned char \*card\_formated);**

Parameters description:

* message\_nr NDEF message ordinal (starts form 1)
* 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
* card\_formated pointer to the variable which shows that the card formatted for NDEF using.
* **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)

**unsigned long erase\_last\_ndef\_record(unsigned char message\_nr);**

Parameter description:

* 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)

**unsigned long erase\_all\_ndef\_records(unsigned char message\_nr);**

Parameter description:

* 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)

**unsigned long 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)

**unsigned long SetAsyncCardIdSendConfig(  
unsigned char send\_enable,   
unsigned char prefix\_enable,   
unsigned char prefix,   
unsigned char suffix,   
unsigned long async\_baud\_rate);**

Parameters description:

* 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)

**unsigned long GetAsyncCardIdSendConfig(  
unsigned char \*send\_enable,   
unsigned char \*prefix\_enable,  
unsigned char \*prefix,   
unsigned char \*suffix,   
unsigned long \*async\_baud\_rate);**

Parameters description:

* 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 unsigned char 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)

**unsigned long GetReaderTime(unsigned char \*time);**

Parameter description:

* time is 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)

**unsigned long SetReaderTime(  
unsigned char \*password,  
unsigned char \*time);**

Parameters description:

* 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)

**unsigned long ChangeReaderPassword(  
unsigned char \*old\_password,  
unsigned char \*new\_password);**

Parameters description:

* 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)

**unsigned long ReaderEepromRead(  
unsigned char \*data,   
unsigned long address,   
unsigned long size);**

Parameters description:

* 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)

**unsigned long ReaderEepromWrite(  
unsigned char \*data,   
unsigned long address,   
unsigned long size,   
unsigned char \*password);**

Parameters description:

* 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)

**unsigned long uFR\_int\_WriteAesKey(  
unsigned char** **aes\_key\_nr**,  
**unsigned char \*aes\_key);**

Parameters description:

* aes\_key\_nr ordinal number of AES key in the reader
* aes\_key\_ext 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)

**unsigned long uFR\_int\_GetDesfireUid(  
unsigned char** **aes\_key\_nr**,  
**unsigned long aid**,  
**unsigned char** **aid\_key\_nr**,  
**unsigned char** **\*card\_uid**,  
**unsigned char** **\*card\_uid\_len**,   
**unsigned short** **\*card\_status**,  
**unsigned short** **\*exec\_time);**

**unsigned long uFR\_int\_GetDesfireUid\_PK(  
unsigned char \*aes\_key\_ext,**  
**unsigned long aid**,  
**unsigned char** **aid\_key\_nr**,  
**unsigned char** **\*card\_uid**,  
**unsigned char** **\*card\_uid\_len**,   
**unsigned short** **\*card\_status**,  
**unsigned short** **\*exec\_time);**

Parameters description:

* 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)
* data pointer to array containing card UID
* data\_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)

**unsigned long uFR\_int\_DesfireFreeMem(  
unsigned long \*free\_mem\_byte,   
unsigned short \*card\_status,   
unsigned short \*exec\_time);**

Parameters description:

* 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)

**unsigned long uFR\_int\_DesfireFormatCard(  
unsigned char** **aes\_key\_nr**,  
**unsigned short \*card\_status,   
unsigned short \*exec\_time);**

**unsigned long uFR\_int\_DesfireFormatCard\_PK(  
unsigned char** **\*aes\_key\_ext**,  
**unsigned short \*card\_status,   
unsigned short \*exec\_time);**

Parameters description:

* 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)

**unsigned long uFR\_int\_DesfireSetConfiguration(  
unsigned char** **aes\_key\_nr**,  
**unsigned char random\_uid,  
unsigned char format\_disable,   
unsigned short \*card\_status,  
unsigned short \*exec\_time);**

**unsigned long uFR\_int\_DesfireSetConfiguration\_PK(  
unsigned char** **\*aes\_key\_ext**,  
**unsigned char random\_uid,  
unsigned char format\_disable,   
unsigned short \*card\_status,  
unsigned short \*exec\_time);**

Parameters description:

* 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)

**unsigned long uFR\_int\_DesfireGetKeySettings(  
unsigned char aes\_key\_nr,  
unsigned long aid,  
unsigned char \*settings  
unsigned char \*max\_key\_no,   
unsigned short \*card\_status,   
unsigned short \*exec\_time);**

**unsigned long uFR\_int\_DesfireGetKeySettings\_PK(  
unsigned char \*aes\_key\_ext,  
unsigned long aid,  
unsigned char \*settings   
unsigned char \*max\_key\_no,   
unsigned short \*card\_status,   
unsigned short \*exec\_time);**

Parameters description:

* 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)

**unsigned long uFR\_int\_DesfireChangeKeySettings(  
unsigned char** **aes\_key\_nr**,  
**unsigned long aid**,  
**unsigned char settings,   
unsigned short \*card\_status,   
unsigned short \*exec\_time);**

**unsigned long uFR\_int\_DesfireChangeKeySettings\_PK(  
unsigned char \*aes\_key\_ext,**  
**unsigned long aid**,  
**unsigned char settings,   
unsigned short \*card\_status,   
unsigned short \*exec\_time);**

Parameters description:

* 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**

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)

**unsigned long uFR\_int\_DesfireChangeAesKey(  
unsigned char** **aes\_key\_nr**,  
**unsigned long aid**,  
**unsigned char** **aid\_key\_nr\_auth**,  
**unsigned char new\_aes\_key[16],   
unsigned char aid\_key\_no,  
unsigned char old\_aes\_key[16],  
unsigned short \*card\_status,   
unsigned short \*exec\_time);**

**unsigned long uFR\_int\_DesfireChangeAesKey\_PK(  
unsigned char \*aes\_key\_ext,**  
**unsigned long aid**,  
**unsigned char** **aid\_key\_nr\_auth**,  
**unsigned char new\_aes\_key[16],   
unsigned char aid\_key\_no,  
unsigned char old\_aes\_key[16],  
unsigned short \*card\_status,   
unsigned short \*exec\_time);**

Parameters description:

* 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)

**unsigned long uFR\_int\_DesfireCreateApplication(  
unsigned char** **aes\_key\_nr**,  
**unsigned long aid\_nr,  
unsigned char settings,   
unsigned char max\_key\_no,  
unsigned short \*card\_status,   
unsigned short \*exec\_time);**

**unsigned long uFR\_int\_DesfireCreateApplication\_PK(  
unsigned char \*aes\_key\_ext,  
unsigned long aid\_nr,  
unsigned char settings,   
unsigned char max\_key\_no,  
unsigned short \*card\_status,   
unsigned short \*exec\_time);**

**unsigned long uFR\_int\_DesfireCreateApplication\_no\_auth(  
unsigned long aid\_nr,  
unsigned char settings,   
unsigned char max\_key\_no,  
unsigned short \*card\_status,   
unsigned short \*exec\_time);**

Parameters description:

* 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)

**unsigned long uFR\_int\_DesfireDeleteApplication(  
unsigned char aes\_key\_nr,  
unsigned long aid\_nr,   
unsigned short \*card\_status,   
unsigned short \*exec\_time);**

**unsigned long uFR\_int\_DesfireDeleteApplication\_PK(  
unsigned char \*aes\_key\_ext,  
unsigned long aid\_nr,   
unsigned short \*card\_status,   
unsigned short \*exec\_time);**

Parameters description:

* 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)

**unsigned long uFR\_int\_DesfireCreateStdDataFile(  
unsigned char aes\_key\_nr,  
unsigned long aid,  
unsigned char file\_id,  
unsigned long file\_size,  
unsigned char read\_key\_no,  
unsigned char write\_key\_no,  
unsigned char read\_write\_key\_no,  
unsigned char change\_key\_no,   
unsigned char communication\_settings,  
unsigned short \*card\_status,   
unsigned short \*exec\_time);**

**unsigned long uFR\_int\_DesfireCreateStdDataFile\_PK(  
unsigned char \*aes\_key\_ext,  
unsigned long aid,  
unsigned char file\_id,  
unsigned long file\_size,  
unsigned char read\_key\_no,  
unsigned char write\_key\_no,  
unsigned char read\_write\_key\_no,  
unsigned char change\_key\_no,   
unsigned char communication\_settings,  
unsigned short \*card\_status,   
unsigned short \*exec\_time);**

**unsigned long uFR\_int\_DesfireCreateStdDataFile\_no\_auth(  
unsigned long aid,  
unsigned char file\_id,  
unsigned long file\_size,  
unsigned char read\_key\_no,  
unsigned char write\_key\_no,  
unsigned char read\_write\_key\_no,  
unsigned char change\_key\_no,   
unsigned char communication\_settings,  
unsigned short \*card\_status,   
unsigned short \*exec\_time);**

Parameters description:

* 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)

**unsigned long uFR\_int\_DesfireDeleteFile(  
unsigned char aes\_key\_nr,  
unsigned long aid,  
unsigned char file\_id,   
unsigned short \*card\_status,   
unsigned short \*exec\_time);**

**unsigned long uFR\_int\_DesfireDeleteFile\_PK(  
unsigned char \*aes\_key\_ext,  
unsigned long aid,  
unsigned char file\_id,   
unsigned short \*card\_status,   
unsigned short \*exec\_time);**

**unsigned long uFR\_int\_DesfireDeleteFile\_no\_auth(  
unsigned long aid,  
unsigned char file\_id,   
unsigned short \*card\_status,   
unsigned short \*exec\_time);**

Parameters description:

* internal\_key 1 - AES key stored into reader uses for authentication, 0 – AES key uses for authentication entered as a function's parameter
* 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)
* auth\_req value is 1 if authentication is needed, otherwise it is 0
* 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)

**unsigned long uFR\_int\_DesfireReadStdDataFile(  
unsigned char aes\_key\_nr,  
unsigned long aid,  
unsigned char** **aid\_key\_nr**, **unsigned char file\_id,   
unsigned short offset,   
unsigned short data\_length,  
unsigned char communication\_settings,  
unsigned char \*data,  
unsigned short \*card\_status,   
unsigned short \*exec\_time);**

**unsigned long uFR\_int\_DesfireReadStdDataFile\_PK(  
unsigned char \*aes\_key\_ext,  
unsigned long aid,  
unsigned char** **aid\_key\_nr**, **unsigned char file\_id,   
unsigned short offset,   
unsigned short data\_length,  
unsigned char communication\_settings,  
unsigned char \*data,  
unsigned short \*card\_status,   
unsigned short \*exec\_time);**

**unsigned long uFR\_int\_DesfireReadStdDataFile\_no\_auth(  
unsigned long aid,  
unsigned char** **aid\_key\_nr**, **unsigned char file\_id,   
unsigned short offset,   
unsigned short data\_length,  
unsigned char communication\_settings,  
unsigned char \*data,  
unsigned short \*card\_status,   
unsigned short \*exec\_time);**

Parameters description:

* 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)

**unsigned long uFR\_int\_DesfireWriteStdDataFile(  
unsigned char aes\_key\_nr,  
unsigned long aid,  
unsigned char** **aid\_key\_nr**, **unsigned char file\_id,   
unsigned short offset,   
unsigned short data\_length,  
unsigned char communication\_settings,  
unsigned char \*data,  
unsigned short \*card\_status,   
unsigned short \*exec\_time);**

**unsigned long uFR\_int\_DesfireWriteStdDataFile\_PK(  
unsigned char \*aes\_key\_ext,  
unsigned long aid,  
unsigned char** **aid\_key\_nr**, **unsigned char file\_id,   
unsigned short offset,   
unsigned short data\_length,  
unsigned char communication\_settings,  
unsigned char \*data,  
unsigned short \*card\_status,   
unsigned short \*exec\_time);**

**unsigned long uFR\_int\_DesfireWriteStdDataFile\_no\_auth(  
unsigned long aid,  
unsigned char** **aid\_key\_nr**, **unsigned char file\_id,   
unsigned short offset,   
unsigned short data\_length,  
unsigned char communication\_settings,  
unsigned char \*data,  
unsigned short \*card\_status,   
unsigned short \*exec\_time);**

Parameters description:

* 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 write operation within file
* data\_length number of data to be written
* 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)

**unsigned long DES\_to\_AES\_key\_type(void);**

## Functions for 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)

**unsigned long UfrSetBadSelectCardNrMax(  
unsigned char bad\_select\_nr\_max);**

* **UfrGetBadSelectCardNrMax**

Function description:

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

Function declaration (C language)

**unsigned long UfrGetBadSelectCardNrMax(  
unsigned char \*bad\_select\_nr\_max);**

## Functions for all blocks linear reading

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):

unsigned long LinearRowRead(unsigned char \*aucData,  
 unsigned short usLinearAddress,  
 unsigned short usDataLength,  
 unsigned short \*lpusBytesReturned,  
 unsigned char ucAuthMode,  
 unsigned char ucReaderKeyIndex);

unsigned long LinearRowRead\_AKM1(unsigned char \*aucData,  
 unsigned short usLinearAddress,  
 unsigned short usDataLength,  
 unsigned short \*lpusBytesReturned,  
 unsigned char ucAuthMode);

unsigned long LinearRowRead\_AKM2(unsigned char \*aucData,  
 unsigned short usLinearAddress,  
 unsigned short usDataLength,  
 unsigned short \*lpusBytesReturned,  
 unsigned char ucAuthMode);

**unsigned long LinearRowRead\_PK(unsigned char \*aucData,  
 unsigned short usLinearAddress,  
 unsigned short usDataLength,  
 unsigned short \*lpusBytesReturned,  
 unsigned char ucAuthMode,  
 unsigned char \*aucProvidedKey);**

The method for proving authenticity is determined by the suffix in the functions names:

* *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 "unsigned short" 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 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.

Functions declaration (C language):

**unsigned long UfrEnterSleepMode(void);**

* **UfrLeaveSleepMode**

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

Functions declaration (C language):

**unsigned long UfrLeaveSleepMode(void);**

## 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):**

unsigned long 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 description:**

* 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

### TagEmulationStart

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):**

unsigned long TagEmulationStart(void);

**Possible error codes:**

WRITE\_VERIFICATION\_ERROR = 0x70

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

### TagEmulationStop

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

**Function declaration (C language):**

unsigned long 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 ISO14443 – 4A cards

### SetSpeedPermanently

**Function declaration (C language):**

**unsigned long SetSpeedPermanently(uint8\_t tx\_speed, uint8\_t rx\_speed);**

**Parameters description:**

* 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):**

**unsigned long GetSpeedParameters(uint8\_t\* tx\_speed, uint8\_t\* rx\_speed);**

**Parameters description:**

* tx\_speed – returns configured value for transmit speed
* rx\_speed *–* returns configured value for receive speed

## FUNCTIONS FOR DISPLAY CONTROL

* **SetDisplayData**

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)

**unsigned long SetDisplayData(unsigned char \*display\_data, unsigned char data\_length);**

Parameters description.

* *display\_data* – pointer to data array
* *data\_length* – number of data into array
* **SetSpeakerFrequency**

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)

**unsigned long SetSpeakerFrequency(uint16\_t frequency);**

Parameters description

* *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**

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

Function declaration (C language):

**unsigned long EnterShareRamCommMode(void);**

* **ExitShareRamCommMode**

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

Function declaration (C language):

**unsigned long EnterShareRamCommMode(void);**

* **WriteShareRam**

Function allows writing data to the shared RAM.

Function declaration (C language):

**unsigned long WriteShareRam(uint8\_t \*ram\_data,   
 uint8\_t addr,   
 uint8\_t data\_len);**

Parameters description:

* *ram\_data –* pointer to data array
* *addr –* address of first data in an array
* *data\_len – l*ength of array. Address + data\_len <= 184
* ***ReadShareRam***

Function allows read data from the shared RAM.

Function declaration (C language)

**unsigned long ReadShareRam(uint8\_t \*ram\_data,   
 uint8\_t addr,   
 uint8\_t data\_l**

**Appendix**

# Table 1. ERR CODES (DL\_STATUS result)

|  |  |
| --- | --- |
| **DL\_STATUS ERR CODES** | |
| **Result** | **Code** |
| 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\_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 |

# Table 2. DLogicCardType 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 |

# Table 3. DLogic reader type enumeration

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |

1. If Key B may be read in the corresponding Sector Trailer it cannot 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. [↑](#footnote-ref-1)