MC2002 Hand-Held Smart Card Read/Write Device (RWD) Optional Device Library Manual: MIFARE® Accessing Library

Version 1.2

Mar 2003

REVISION HISTORY

Version Number	Date	Description	Author
1.0	18 Jun, 2002	Initial release	Hwang Toung
1.1	10 Nov, 2002	MFS support added	Hwang Toung
1.2	26 Mar, 2003	Pack exchange support for implementation of ISO 14443-4A	Hwang Toung

Contents

CHAPTER 1 .... **4**

INTRODUCTION **4**

1.1 MIFARE® 5
1.2 SYSTEM REQUIREMENT 5

CHAPTER 2 .... **6**

OVERVIEW **6**

2.1 TRANSACTION FLOW 7
2.2 CARD ACCESSING FLOW CHART 8
2.3 INTERFACE CONTROL 9
2.3.1 Interface initialization 9
2.3.2 Interface initialization 9
2.4 ISO 14443A COMMAND SET 9
2.4.1 Request 9
2.4.2 Anticollision 9
2.4.3 Selection 10
2.4.4 Data exchange for ISO 14443-4A 10
2.4.5 Halt/Card deactivation 10
2.5 MIFARE® PROPRIETARY COMMAND SET 10
2.5.1 Authentication 10
2.5.2 Read 11
2.5.3 Write 11
2.5.4 Increment/Decrement/Restore 11
2.5.5 Transfer 11
2.6 MIFARE® STANDARD (S50) SUBROUTINES 12
2.6.1 Card accessing: Read/Write 12
2.6.2 Card accessing: Value blocks 12
2.6.3 Interface NV Memory initialization 13
2.7 CARD PERSONALIZATION 14

CHAPTER 3 .... **15**

SUBROUTINES **15**

3.1 INTERFACE INITIALIZATION 16
3.2 REQUEST 17
3.3 ANTICOLLISION CASCADED 18
3.4 ANTICOLLISION LEVEL 1 19
3.5 SELECTION CASCADED 20
3.6 EXCHANGE TRANSPARENT DATA 21
3.7 SELECTION LEVEL 1 22

3.8 AUTHENTICATION WITH DIRECT KEY PRESENTATION 23
3.9 STANDARD READ 24
3.10 STANDARD WRITE 25
3.11 VALUE WITH TRANSFER 26
3.12 VALUE WITHOUT TRANSFER 27
3.13 HALT 28
3.14 S50 DATA BLOCK ACCESSING: SINGLE BLOCK READ/WRITE 29
3.15 S50 DATA BLOCK ACCESSING: MULTI BLOCK READ/WRITE 30
3.16 S50 VALUE BLOCK ACCESSING: INCREMENT/DECREMENT 32
3.17 INTERFACE DEACTIVATE 33
3.18 INTERFACE NV MEMORY INITIALIZATION 33
3.19 CARD PERSONALIZATION 33

***CHAPTER 4* 34**

***SAMPLES* 34**

4.1 SOURCE CODE OF CARDACCESSMULTIBLOCKS 35
4.2 SOURCE CODE OF CARDVALUE 37
4.3 READ SECTOR FOR MFI S50 CARD 38
4.4 VALUE BLOCK INCREMENT/DECREMENT FOR MFI S50 CARD 39

Chapter 1
Introduction

Introduction 1

1.1 MIFARE®

MIFARE® system is a contactless smart card system developed by Philips. The communication layer, i.e. MIFARE® RF interface, complies to part 2 and 3 of the ISO/IEC 14443A standard. The security layer of MIFARE® supports CRYTO1 stream cipher for secure data exchange.

1.2 System requirement

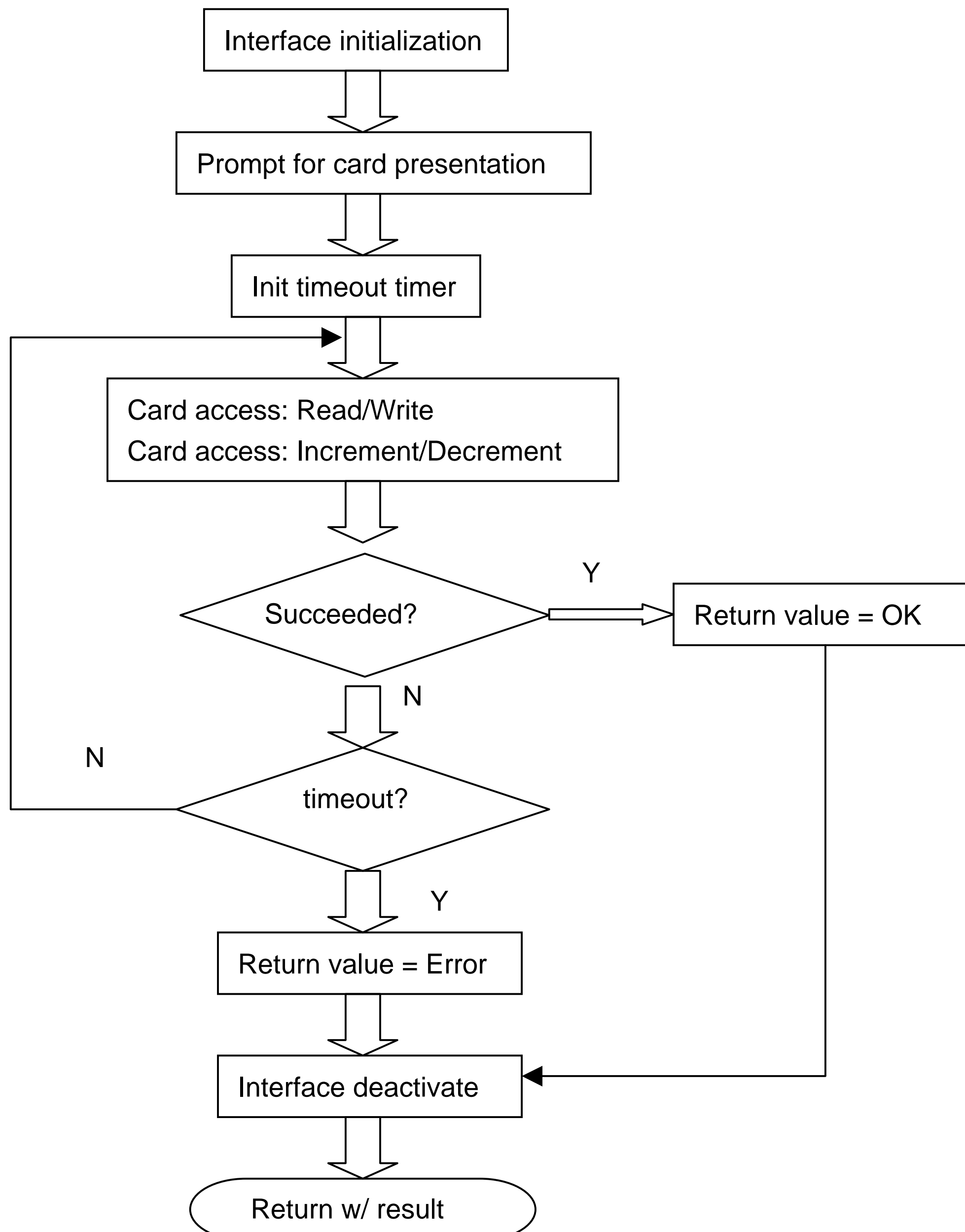
An optional peripheral unit must be used for MC2002 to access MIFARE® cards. This unit features an ISO14443A/B interface chip by Philips, which is capable of accessing both type A & B proximity IC cards.
An additional library named **mifare530.a** should be added to the link script file (.ld) of the project, and a header file **mifare530.h** should also be included in the source code.

Chapter 2
Overview

Overview 2

2.1 Transaction Flow

Since the contactless card-accessing unit is quite power consuming, it’s recommended that it be turned on only when it’s necessary. The transaction flow should be as:



2.2 Card accessing flow chart

Here's a page adapted from *Philips Standard Card IC MF1 IC S50 Functional Specification*:

3.2 Communication Principle

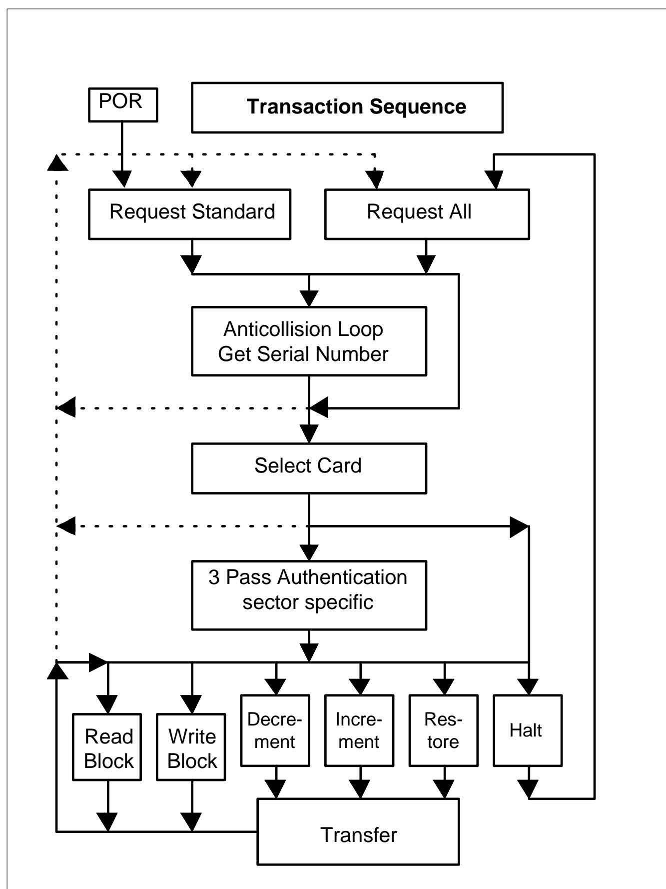
The commands are initiated by the RWD and controlled by the Digital Control Unit of the MF1 IC S50 according to the access conditions valid for the corresponding sector.

3.2.1 REQUEST STANDARD / ALL

After Power On Reset (POR) of a card it can answer toarequestcommand-sentbytheRWDtoall cards in the antenna field - by sending the answer to request code (ATQA according to ISO/IEC 14443A).

3.2.2 ANTICOLLISION LOOP

In the anticollision loop the serial number of a card is read. If there are several cards in the operating range of the RWD, they can be distinguished by their unique serial numbers and one can be selected (select card) for further transactions. The unselected cards return to the standby mode and wait for a new request command.



3.2.3 SELECT CARD

With the select card command the RWD selects one individual card for authentication and memory related operations. The card returns the Answer To Select (ATS) code (= 08h), which determines the type of the selected card. Please refer to the document *MIFARE® Standardised Card Type Identification Procedure* for further details.

3.2.4 3 PASS AUTHENTICATION

After selection of a card the RWD specifies the memory location of the following memory access and uses the corresponding key for the 3 pass authentication procedure. After a successful authentication all memory operations are encrypted.

3.2.5 MEMORY OPERATIONS

After authentication any of the following operations may be performed:

- Read block
- Write block
- Decrement: Decrements the contents of a block and stores the result in a temporary internal data-register
- Increment: Increments the contents of a block and stores the result in the data-register
- Restore: Moves the contents of a block into the data-register
- Transfer: Writes the contents of the temporary internal data-register to a value block

2.3 Interface control

2.3.1 Interface initialization

The contactless card-accessing unit should be initialized before any actual access begins. The interface is powered on, RF interface is initiated, and the current should rise up to about 200mA.

Subroutine used:

```
InitMC530();
```

2.3.2 Interface initialization

Interface deactivation will cut off the power of the entire contactless card interface.

It's strongly recommended that the interface be turned off as soon as it's possible.

Subroutine used:

```
MC530Off();
```

2.4 ISO 14443A command set

The term 'card' is this section always refers to proximity IC card that complies with ISO 14443 type A. C.f. ISO 14443-3 for detailed descriptions of the protocol.

2.4.1 Request

If a card enters the operating field the card is set into its initial state - IDLE. The very first command a card accepts in this initial state is a 'Request' command. Here are 2 most frequently used request codes:

WAKE-UP (52H) is sent by the RWD to put cards which have entered the HALT State back into the READY State. They shall then participate in further anticollision and selection procedures. (defined as PICC_REQALL)

REQA (26H) : only cards that were not set into the halt state before would respond. (defined as PICC_REQIDLE)

The card responds with its card type depending tag-type.

Subroutine used:

```
CardTypeARequest();
```

2.4.2 Anticollision

After a successful request, the card enters READY state, and it will stay there until it's selected by it's UID. Now an 'Anticollision' command should be sent and the card will return its UID (serial number). If there are more than one card within the operating field, an internal procedure is stated, resulting in a serial number of one card.

Subroutine used:

```
CardTypeAAnticoll();
CardTypeAAnticollLevel1();
```

2.4.3 Selection

The card will make a state transition from READY to ACTIVE by a ‘Select’ command with it’s UID. Due to the ‘Select’ command only the card with the specified UID is selected and will continue the communication with the RWD. All other cards are not longer involved during the communication and will stay in READY state.

The card responds to a ‘Select’ command with a card type specific code – SAK, which indicates if the card is ISO 14443-4 compliant

Subroutine used:

```
CardTypeASelect();
CardTypeASelectLevel1();
```

2.4.4 Data exchange for ISO 14443-4A

For ISO 14443-4A compliant card, activation should be performed after successful selection. All the communication are based on a transparent data exchange subroutine with the RWD, including RATS/ATS, PPS, and the following data exchanges. C.f. ISO14443-4A.

Subroutine used:

```
CardTypeAExchangeBlock();
```

2.4.5 Halt/Card deactivation

Card deactivation would set the card with the interface is dealing to halt mode. It’s quite useful when there are several cards present in the operating field. It will prevent a card from answering Request IDLE command, so it would not appear next time as long as it stays in the operating field.

Subroutine used:

```
CardTypeAHalt();
```

2.5 MIFARE® proprietary command set

These commands are not compliant with ISO 14443-4. C.f. MIFARE® Card specification for detailed description of the protocol. They’re used by MF Standard, Light, Ultralight, Pro, Prox cards.

2.5.1 Authentication

To get access to the MIFARE® Classic card's EEPROM and the stored data, the card has to be authenticated. Therefore, an authentication procedure has to be started which is implemented completely inside the RWD. The user has to address the appropriate keys in the key memory. If these keys match to the keys stored in the card's sector trailer further access to the card is allowed. There are two ways to store the master keys for authentication. Keys can be passed directly to the RWD by calling the authentication function

CardMFCAuthKey(); or keys can be stored in the internal non volatile key memory of the RWD used for authentication.

Subroutine used:
CardMFCAuthKey();

2.5.2 Read

The 'Read' command reads out the card EEPROM. For Classic cards, authentication procedure must be accomplished before any accessing to the cards. Various numbers of byte could be read out depending on the card IC. If the content of a sector trailers is read out the keys are masked to zero by the card, because keys could not be read back. In addition to a valid authentication of the Classic card, data can only be read if the card's access condition for that block allows it.

Subroutine used:
CardMFCRead16Bytes();

2.5.3 Write

The 'Write' command writes data to the card EEPROM if the access conditions allows it. For Classic cards, authentication procedure must be accomplished before any accessing to the cards. The number of the bytes written may vary depending on the card IC. In addition to a valid authentication of the Classic card, data can only be read if the card's access condition for that block allows it.

Subroutine used:
CardMFCWrite16Bytes();

2.5.4 Increment/Decrement/Restore

These commands are for cards that support value blocks (Standard, Light, etc.).
The 'Increment' command reads the addressed value block of a card , checks the data structure and increases the content of the value block by the transmitted value and stores the result in the card's internal register; the 'Decrement' command is almost the same except that the value is decreased; the 'Restore' command would just store the content of the value block in the internal register.
No write operation to the EEPROM is actually done for these 3 commands. So these commands are not implemented as single functions, CardMFCValueWithTransfer() and CardMFCValueDebit() includes all value format related functions.

Subroutine used:
CardMFCValueWithTransfer();
CardMFCValueDebit();

2.5.5 Transfer

The 'Transfer' command transfers the content of the card's internal register to the EEPROM

address. This function can only be called after the increment, decrement or restore function. This command is not implemented as a single function, it's included in CardMFCValueWithTransfer().

Subroutine used:

```
CardMFCValueWithTransfer();
```

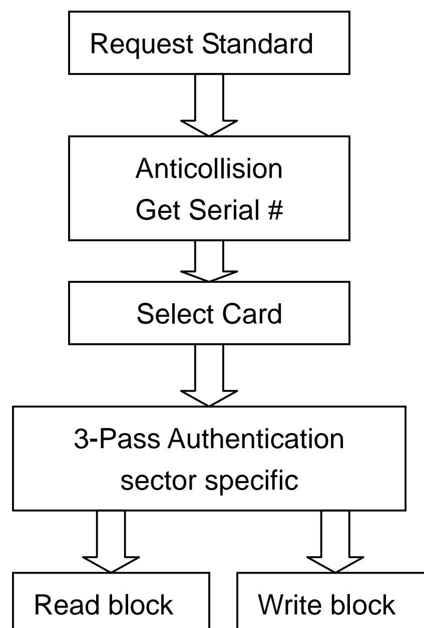
2.6 MIFARE® standard (S50) subroutines

2.6.1 Card accessing: Read/Write

All data blocks on a MIFARE® standard card could always be accessed by reading/writing if certain access conditions (security conditions) are available. Only data blocks could be accessed this way, trailer blocks should be accessed as shown in 2.7.

Authentication mode must be specified to read/write data blocks.

Flowchart of Read/Write access:



Subroutine used:

```
CardAccess();
CardAccessMultiBlocks();
```

2.6.2 Card accessing: Value blocks

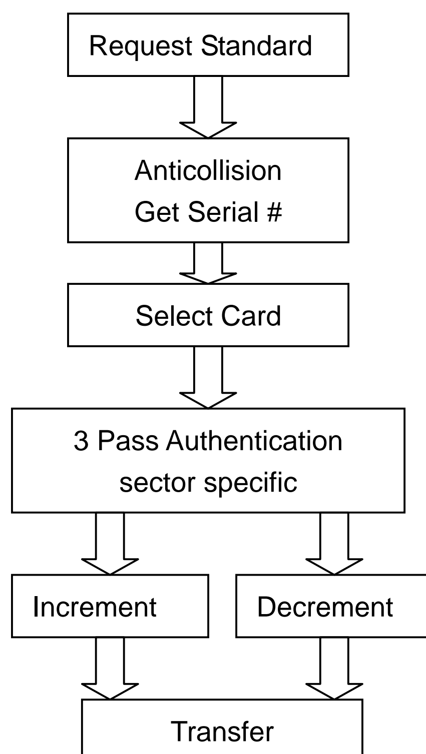
Data blocks on a MIFARE® card could be configured as value blocks, which could be used for electronic purse. (c.f. MIFARE® card S50/S70/L10 specification).

These blocks could be incremented/decremented. Only valid value blocks could be accessed this way, accessing other blocks would only return with error.

Transfer is performed automatically after increment/decrement, so the value would be updated. (c.f. MIFARE® card specification).

Authentication mode must be specified to increment/decrement value blocks.

Flowchart of value block access:



Subroutine used:

`CardValue();`

2.6.3 Interface NV Memory initialization

The keys used by the sector specific authentication are stored in the EEPROM of the interface unit. There are 2 keys for each sector, KEYA and KEYB; up to 16 key pairs could be stored in the EEPROM, corresponding to the 16 sectors of MIFARE® card.

The keys must be stored in the EEPROM prior to any card access.

It's recommended that this initialization be performed in secure environment by authorized person/program, because the keys are NOT encrypted. Do not include keys in application program for end user.

There might be 2 ways to do that:

1) Download a program to initialize all the keys, and then download the application program to overwrite the key writer.
2) Obtain keys by other means (input from keypad, get it from another computer via UART, read it from a contact smartcard and decrypt it by a SAM card), destroy the key storage in RAM as soon as the keys are written into EEPROM.

Subroutine used:

```
Mf500KeyWriteToEeprom();
```

2.7 Card personalization

Although it is possible to personalize a MIFARE® card with a handheld, it's not recommended for security reasons.

The keys and access conditions are stored in the trailer block of each sector on a MIFARE® card, so writing the trailer block would get this sector ready for utilization phase.

Please refer to the user manual of MIFARE® card for detailed information on the trailer block.

Chapter 3
Subroutines

Subroutines 3

3.1 Interface Initialization

`InitMC530()` is the contactless card interface unit initialization subroutine. The interface would be powered on after invoking it.

```
char InitMC530(
    short card_type,
) ;
```

Parameters:

card_type

0 for MIFARE® cards

Return Values

MI_OK:	interface initialized
Other values:	error occurred, hardware error mostly.

3.2 Request

CardTypeARequest() sends Request command for ISO 14443 A Command set.

```
char CardTypeARequest(
    unsigned char req_code,
    unsigned char * atq,
) ;
```

Parameters:

req_code

The request code value. C.f. ISO 14443-3 for specification. 2 predefined value:
PICC_REQALL : WAKE_UP command (52H)
PICC_REQIDL : REQA command (26H)

atq

Pointer to a 2 byte buffer storing the answer to request. Data is stored with LSB 1st.

Return Values

MI_OK:	succeeded
Other values:	error occurred, hardware error mostly.

3.3 Anticollision cascaded

`CardTypeAAnticoll()` sends Anticollision command for ISO 14443 A Command set. According to ISO 14443-3 A, this subroutine handles extended UID with 3 possible level codes. The function transmits a select code and all cards in READY state are responding. After this subroutine returns, only one UID set is selected and returned. This subroutine may be called several times on cascaded levels to select a card with extended UID (twice for 7-byte UIDs, 3 times for 10-byte UIDs).

```
char CardTypeAAnticoll (
    unsigned char level,
    unsigned char * serial_number,
) ;
```

Parameters:

level

Anticollision level code. C.f. ISO 14443-3 for specification. 3 predefined value:
PICC_ANTICOLL1 : level 1 (93H)
PICC_ANTICOLL2 : level 1 (95H)
PICC_ANTICOLL3 : level 1 (97H)

serial_number

Pointer to a 4 byte buffer storing the UID or UID section found.

Return Values

MI_OK:	succeeded
Other values:	error occurred, hardware error mostly.

3.4 Anticollision Level 1

`CardTypeAAnticollLevel1()` is a macro that performs level 1 anticollision. It's designed for those cards with standard 4-byte UIDs.

```
char CardTypeAAnticollLevel1 (
    unsigned char * serial_number,
) ;
```

Parameters:

serial_number

Pointer to a 4 byte buffer storing the UID.

Return Values

MI_OK:	succeeded
Other values:	error occurred, hardware error mostly.

3.5 Selection cascaded

`CardTypeASelect()` sends Select command for ISO 14443A Command set. It selects a UID level according the level code, and returns the result ATS.
According to ISO 14443-3 A, this subroutine handles extended UID with 3 possible level codes. The function transmits a select code and all cards in READY state are responding. After this subroutine returns, only one UID set is selected and returned. This subroutine may be called several times on cascaded levels to select a card with extended UID (twice for 7-byte UIDs, 3 times for 10-byte UIDs).

```
char CardTypeASelect (
    unsigned char level,
    unsigned char * serial_number,
    unsigned char * ATS
) ;
```

Parameters:

level

Selection level code. C.f. ISO 14443-3 for specification. 3 predefined value:
PICC_ANTICOLL1 : level 1 (93H)
PICC_ANTICOLL2 : level 1 (95H)
PICC_ANTICOLL3 : level 1 (97H)

serial_number

Pointer to a 4 byte buffer storing the UID or UID section found.

ATS

Pointer to a byte buffer storing the ATS code (aka SAK, select acknowledge in ISO 14443A). The value depends on the card type. According to ISO 14443-3, Valid combinations are:

XX1XX0XX	UID complete, PICC compliant with ISO/IEC 14443-4
XX0XX0XX	UID complete, PICC not compliant with ISO/IEC 14443-4
XXXXX1XX	UID not complete

Return Values

MI_OK:	succeeded
Other values:	error occurred, hardware error mostly.

3.6 Exchange transparent data

CardTypeAExchangeBlock() is a basic transparent data exchange subroutine for any type of ISO14443-3 compliant card. All ISO14443-4A commands and data exchange could be implemented with it; even other non ISO14443-4 compliant cards (e.g. Mifare® UltraLite) can also be dealt with via this subroutine.

```
char CardTypeAExchangeBlock(
    unsigned char *send_data,
    unsigned short send_bytelen,
    unsigned char *rec_data,
    unsigned short *rec_bytelen,
    unsigned char append_crc,
    unsigned long timeout
) ;
```

Parameters:

send_data

Pointer to a buffer storing the data to send for the exchange.

send_bytelen

The length in byte for the buffer of *send_data*.

rec_data

Pointer to a buffer to store the data to receive for the exchange.

rec_bytelen

Pointer to a buffer storing the length in byte of data received.

append_crc

Specifies if CRC should be calculated and verified for the exchange.

possible values are:

1	yes, CRC used
0	no CRC

timeout

The maximum time that the RWD should wait for a valid response from the card. The time is in 1/13560000 second. The upper limit of this parameter is 8388607, any value larger than this will be automatically trimmed to it.

Return Values

MI_OK:	succeeded
Other values:	error occurred, hardware error mostly.

3.7 Selection Level 1

`CardTypeASelectLevel1()` is a macro that performs level 1 selection. It's designed for those cards with standard 4-byte UIDs.

```
char CardTypeASelectLevel1 (
    unsigned char * serial_number,
    unsigned char * ATS
) ;
```

Parameters:

serial_number

Pointer to a 4 byte buffer storing the UID.

ATS

Pointer to a byte buffer storing the ATS code (aka SAK, select acknowledge in ISO 14443A). The value depends on the card type. According to ISO 14443-3, Valid combinations are:

XX1XX0XX	PICC compliant with ISO/IEC 14443-4
XX0XX0XX	PICC not compliant with ISO/IEC 14443-4

Return Values

MI_OK:	succeeded
Other values:	error occurred, hardware error mostly.

3.8 Authentication with direct key presentation

CardMFCAuthKey() performs Authentication with direct key loading from the CPU to the RWD.

```
char CardMFCAuthKey (
    unsigned char auth_mode,
    unsigned char * serial_number,
    unsigned char * keys,
    unsigned char  block_number
) ;
```

Parameters:

auth_mode

Selects master key A or master key B. Possible values:

PICC_AUTHENT1A: use master key A
PICC_AUTHENT1B: use master key B

serial_number

Pointer to a 4-byte buffer storing the UID of the card to be authenticated.

keys

Pointer to a 6-byte buffer storing the clear form master key for the authentication.

block_number

Indicates the card's block address, which shall be authenticated. For MIFARE® Standard cards, block can range from 0 to 63 for S50, 0 to 255 for S70. For other card types please refer to the according product description.

Return Values

MI_OK:	succeeded
Other values:	error occurred, hardware error mostly.

3.9 Standard Read

CardMFCRead16Bytes() reads out a 16 byte block from the specified card's block address.

```
char CardMFCRead16Bytes (
    unsigned char  block_address
    unsigned char * pdata,
) ;
```

Parameters:

block_address

Indicates the card's block address of the block to be read. For MIFARE® Standard cards, block can range from 0 to 63 for S50, 0 to 255 for S70. For other card types please refer to the according product description.

pdata

Pointer to a 16-byte buffer storing the data block read from the card.

Return Values

MI_OK:	succeeded
MI_BYTECOUNTERR:	succeeded, but the number of bytes returned is not 16. The data is still stored in * *pdata*.
MI_NOTAUTHERR:	not authenticated.
Other values:	error occurred, hardware error mostly.

3.10 Standard Write

CardMFCWrite16Bytes() writes a 16 byte block to the specified card's block address.

```
char CardMFCWrite16Bytes (
    unsigned char  block_address
    unsigned char * pdata,
) ;
```

Parameters:

block_address

Indicates the card's block address of the block to be written. For MIFARE® Standard cards, block can range from 0 to 63 for S50, 0 to 255 for S70. For other card types please refer to the according product description.

pdata

Pointer to a 16-byte buffer storing the data to be written to the block on the card.

Return Values

MI_OK:	succeeded
MI_NOTAUTHERR:	not authenticated.
Other values:	error occurred, hardware error mostly.

3.11 Value with transfer

`CardMFCValueWithTransfer()` accesses a value block using the INCREMENT, DECREMENT or RESTORE command. To be able to perform a value format related operation the data block has to be formatted as value block. First, the data is read from the block address *from_block*. For INCREMENT and DECREMENT, the transfer buffer is loaded with the increased value, which could be transferred to any authenticated block by the TRANFER command; for RESTORE command, the transfer buffer is loaded with the value stored at data block address *from_block*, while the given value is only a dummy value, which only have to be in valid range. After the transfer buffer is loaded, a TRANSFER is automatically performed to the block address *to_block*. A TRANSFER command is only possible directly after a successful RESTORE, INCREMENT or DECREMENT command.

```
char CardMFCValueWithTransfer(
    unsigned char dd_mode,
    unsigned char from_block,
    unsigned char to_block,
    unsigned long lvalue
)
```

Parameters:

dd_mode

Selects the value format related operation. Possible values:

PICC_INCREMENT:	increment
PICC_DECREMENT:	decrement
PICC_RESTORE:	restore

from_block

Indicates the card's block address of the value block from which the value is taken. For MIFARE® Standard cards, block can range from 0 to 63 for S50, 0 to 255 for S70. For other card types please refer to the according product description.

to_block

Indicates the card's block address of the value block to which the value calculated is written. For MIFARE® Standard cards, block can range from 0 to 63 for S50, 0 to 255 for S70. For other card types please refer to the according product description.

lvalue

A positive value used for increment/decrement.

Return Values

MI_OK:	succeeded
Other values:	failed

3.12 Value without transfer

`CardMFCValueDebit()`is almost the same as **`CardMFCValueWithTransfer()`** except for the transfer part. It's mainly desingned for those cards with automatic transfer(MIFARE light, MIFARE PLUS, MIFARE PRO, MIFARE PROX…).

```
char CardMFCValueDebit(
    unsigned char dd_mode,
    unsigned char block_addr,
    unsigned long lvalue
)
```

Parameters:

dd_mode

Selects the value format related operation. Possible values:

```
PICC_INCREMENT:    increment
PICC_DECREMENT:    decrement
PICC_RESTORE:      restore
```

block_addr

Indicates the card's block address of the value block to be accessed. For MIFARE® Standard cards, block can range from 0 to 63 for S50, 0 to 255 for S70. For other card types please refer to the according product description.

lvalue

A positive value used for increment/decrement.

Return Values

MI_OK:	succeeded
Other values:	failed

3.13 Halt

`CardTypeAHalt()` is used to deactivate the card it's talking with.

```
char CardTypeAHalt (
    void
) ;
```

Return Values

MI_OK:	card halted
Other values:	error occurred.

3.14 S50 Data Block Accessing: Single block Read/Write

CardAccess() accesses a single block

```
char CardAccess(
    unsigned char auth_mode,
    unsigned char WR_mode,
    unsigned char sector,
    unsigned char Block,
    unsigned char* rdata,
    unsigned char* wdata
)
```

Parameters:

auth_mode

Selects master key A or master key B. Possible values:

    PICC_AUTHENT1A: use master key A
    PICC_AUTHENT1B: use master key B

WR_mode

Selects access of the block. Possible values:

    PICC_ReadBlock: block reading
    PICC_WriteBlock: block writing

Sector

Sector number of the block to be accessed.

Range: from 0 to 15

Block

The block to be accessed. Range:

from 0 to 3 for reading

from 0 to 2 for writing (because block #3 is the trailer block)

wdata

Pointer to a 16 bytes data buffer that contains data to be written to the card

rdata

Pointer to a 16 bytes data buffer to store the data read from the card

Return Values

MI_OK: succeeded

Other values: failed

3.15 S50 Data Block Accessing: Multi block Read/Write

CardAccessMultiBlocks() accesses several blocks.

```
char CardAccessMultiBlocks (
    unsigned char auth_mode,
    unsigned char WR_mode,
    unsigned char sector,
    unsigned char Block,
    unsigned char BlockCnt,
    unsigned char* rdata,
    unsigned char* wdata
)
```

Parameters:

auth_mode

Selects master key A or master key B. Possible values:

```
PICC_AUTHENT1A:    use master key A
PICC_AUTHENT1B:    use master key B
```

WR_mode

Selects access of the block. Possible values:

```
PICC_ReadBlock:    block reading
PICC_WriteBlock:   block writing
```

Sector

Sector number of the block to be accessed.

Range: from 0 to 15

Block

The block to be accessed. Range:

from 0 to 3 for reading

from 0 to 2 for writing (because block #3 is the trailer block)

BlockCnt

Total number of the blocks to be accessed. The block reference specified by *Block* and *BlockCnt* should not exceed the valid block range.

wdata

Pointer to a data buffer that contains data to be written to the card

rdata

Pointer to a data buffer to store the data read from the card. The buffer should be

allocated large enough to hold all the data read.

Return Values

MI_OK: succeeded

Other values: failed

Note:

Trailer blocks are skipped during the access. So, if you are reading 5 blocks from block #2 of sector #3, the blocks actually read are:

Block #2 of sector #3
Block #0 of sector #4
Block #1 of sector #4
Block #2 of sector #4
Block #0 of sector #5

3.16 S50 Value Block Accessing: Increment/Decrement

CardValue() accesses a value block.

```
char CardValue(
    unsigned char auth_mode,
    unsigned char dd_mode,
    unsigned char sector,
    unsigned char Block,
    unsigned long value
)
```

Parameters:

auth_mode

Selects master key A or master key B. Possible values:

```
PICC_AUTHENT1A:    use master key A
PICC_AUTHENT1B:    use master key B
```

dd_mode

Selects the value format related operation. Possible values:

```
PICC_INCREMENT:    increment
PICC_DECREMENT:    decrement
```

Sector

Sector number of the value block to be accessed.

Range: from 0 to 15

Block

The value block to be accessed. Range:

from 0 to 2 (because block #3 is the trailer block)

value

A positive value used for increment/decrement. After this subroutine has returned successfully, the value in the value block is incremented/decremented by this value.

Return Values

MI_OK:	succeeded
Other values:	failed

3.17 Interface Deactivate

`MC530Off()` is the contact less card interface unit deactivation subroutine. The interface would be powered off after invoking it.

```
void MC530Off(
    void
) ;
```

3.18 Interface NV Memory initialization

`Mf500KeyWriteToEeprom()` writes a key related to a designated sector to the EEPROM of the interface unit.

```
char Mf500KeyWriteToEeprom (
    unsigned char key_type,
    unsigned char sector,
    unsigned char * uncoded_keys
)
```

Parameters:

key_type

Selects master key A or master key B to be written. Possible values:

PICC_AUTHENT1A: write master key A

PICC_AUTHENT1B: write master key B

Sector

Sector number of the value block to be accessed.

Range: from 0 to 15

uncoded_keys

Pointer to a 6 bytes data buffer that contains unencrypted key data to be written to the interface unit.

Return Values

MI_OK:	card halted
Other values:	error occurred.

3.19 Card personalization

N/A

Chapter 4
Samples

4.1 Source code of CardAccessMultiBlocks

This is the source code of CardAccessMultiBlocks with a modified key strategy. This subroutine will use the same key (dummy_key) for all blocks.

```
// just a dummy key for testing only
const unsigned char dummy_key[6] = "\xFF\xFF\xFF\xFF\xFF\xFF";

char MyCardAccessMultiBlocks(
     unsigned char auth_mode,
     unsigned char WR_mode,
     unsigned char sector,
     unsigned char Block,
     unsigned char BlockCnt,
     unsigned char* rdata,
     unsigned char* wdata)
{
  char TempStatus;
  unsigned char ATQ[3];       // answer to request. (2bytes)
  unsigned char PICCsnr[5];   // The CARD Serial Number.(4bytes)

  // Aswer to Select Code. for mifare1 IC CARD it has to be 0x08.(1bytes)
  unsigned char ATS[2];

  unsigned char i=0;

  // request standard
  if( (TempStatus = CardTypeARequest(PICC_REQIDL,ATQ)) != MI_OK)
    return(TempStatus);
  // anticollision and get serial number back to PICCsnr
  if( (TempStatus = CardTypeAAnticollLevel1 (PICCsnr)) != MI_OK)
    return(TempStatus);
  // select the card found, get SAK back to ATS
  if( (TempStatus = CardTypeASelectLevel1(PICCsnr,ATS)) != MI_OK)
    return(TempStatus);
  // check if it is a MF Standard card?
  if(ATS[0] != 0x08) {
    TempStatus = MI_ERROR_CARD_TYPE;
    return(TempStatus);
  }

// check if the parameters are out of range
  if ( (sector > 15) ||
       (Block > 2) ||
       ((sector +(BlockCnt +Block -1)/3) > 15) ){
        TempStatus = MI_WRONG_PARAMETER_VALUE;
        return(TempStatus);
  }
  while (BlockCnt --) {
// authenticate this block
    TempStatus=CardMFCAuthKey(auth_mode,PICCsnr,dummy_key,(4*sector+Block));
    if( TempStatus != MI_OK)
       return(TempStatus);
```

```
        if(WR_mode) {
        // write one block
          TempStatus = CardMFCWrite16Bytes(4*sector+Block,wdata));
          if( TempStatus != MI_OK) {
              TempStatus = MI_WRITE_BLOCK_ERROR;
              return(TempStatus);
          }
          else { // write success
        // read the same sector and check if the data is correct
        // could be omitted if you want it to be faster.
            if((TempStatus = CardMFCRead16Bytes(4*sector+Block,rdata)) != MI_OK) {
              TempStatus = MI_WRITE_BLOCK_ERROR;
              return(TempStatus);
            }
            if(memcmp(wdata,rdata,16)!=0) {
              TempStatus = MI_WRITE_BLOCK_ERROR;
              return(TempStatus);
            }
          }
        }
        else {          // read one block
          if((TempStatus = CardMFCRead16Bytes(4*sector+Block,rdata)) != MI_OK)
             return(TempStatus);
        }
        rdata += 16;
        wdata += 16;
        if ((++Block) > 2) {
          sector++;
          Block = 0;
        }
      }
      TempStatus = MI_OK;
      return(TempStatus);
    }
```

4.2 Source code of CardValue

This is the source code of CardValue with a modified key strategy. This subroutine will use the same key (dummy_key) for all blocks.

```
// just a dummy key for testing only
const unsigned char dummy_key[6] = "\xFF\xFF\xFF\xFF\xFF\xFF";

char MyCardValue( unsigned char auth_mode,
    unsigned char dd_mode,
    unsigned char sector,
    unsigned char Block,
    unsigned long lvalue)
{
  char TempStatus;
  unsigned char ATQ[3];      // answer to request. (2bytes)
  unsigned char PICCsnr[5];   // The CARD Serial Number.(4bytes)

  // Aswer to Select Code. for mifare1 IC CARD it has to be 0x08.(1bytes)
  unsigned char ATS[2];

  unsigned char i=0;

  // check if the parameters are out of range
  if(sector >15 || Block >= 3) {
    TempStatus = MI_WRONG_PARAMETER_VALUE;  // sth. weird
    return(TempStatus);
  }
  // request standard
  if( (TempStatus = CardTypeARequest(PICC_REQIDL,ATQ)) != MI_OK)
    return(TempStatus);
  // anticollision and get serial number back to PICCsnr
  if( (TempStatus = CardTypeAAnticollLevel1 (PICCsnr)) != MI_OK)
    return(TempStatus);
  // select the card found, get SAK back to ATS
  if( (TempStatus = CardTypeASelectLevel1(PICCsnr,ATS)) != MI_OK)
    return(TempStatus);
  // check if it is a MF Standard card?
  if(ATS[0] != 0x08) {
    TempStatus = MI_ERROR_CARD_TYPE;
    return(TempStatus);
  }
  // authenticate this block
  TempStatus=CardMFCAuthKey(auth_mode,PICCsnr,dummy_key,(4*sector+Block));
  if(TempStatus != MI_OK)
    return(TempStatus);
  // value it and transfer it back to the original block
  TempStatus = CardMFCValueWithTransfer(
                 dd_mode,
                 (4*sector+Block),
                 (4*sector+Block),
                 lvalue);
  if( TempStatus != MI_OK)
    return(TempStatus);
  TempStatus = MI_OK;
  return(TempStatus);
}
```

4.3 Read Sector for MFI S50 card

This is a sample subroutine of reading a sector using master key A.

```
int read_sector( unsigned char sec_num, unsigned char * rdata){
  unsigned char cntr, card_read_ok;
  unsigned char TempStatus;

  set_console_english();
  clear_console();
  move_cursor(0,0);
// initialize the interface unit
  if( InitMC530() != MI_OK) {
     puts("Card Interface Error ");
     goto read_card_error;
   }

   move_cursor(0,0);
   puts("please present card");
   move_cursor(0,1);
   puts("in 8 seconds");
   SPT_set(DELAY_1S);

   for(card_read_ok = 0, cntr = 8; cntr && !card_read_ok;) {
      if (!SPT_read()) {
         move_cursor(4,1);
   // Display at the position of '8' to show the remaining waiting time
         printf("%d",cntr-1);
         SPT_set(DELAY_1S);
         cntr --;
      }
   // read 3 blocks of sector #sec_num, data stored in rdata
      TempStatus = CardAccessMultiBlocks(
         PICC_AUTHENT1A, PICC_ReadBlock, sec_num, 0, 3, rdata, NULL
      );
      if(TempStatus == MI_OK )
         card_read_ok = 1;
   }
   if (!card_read_ok) {
      move_cursor(0,3);
      puts("No valid card detected");
      goto read_card_error;
   }
   else {
   // Ok, we turn it off and return.
      MC530Off();
      return 0;
   }
read_card_error:
   // turn it off anyway
   MC530Off();
   move_cursor(0,6);
   puts("press anykey to return");
   getch();
   return 1;
}
```

4.4 Value block Increment/Decrement for MFI S50 card

This is a sample subroutine of increment/decrement a value sector using master key A.

```
int value_sector( unsigned char sec_num, unsigned char block_num,signed long value)
{
  unsigned char cntr, card_read_ok;
  unsigned char TempStatus;

  set_console_english();
  clear_console();
  move_cursor(0,0);
// initialize the interface unit
  if( InitMC530() != MI_OK) {
     puts("Card Interface Error ");
     goto read_card_error;
   }

   move_cursor(0,0);
   puts("please present card");
   move_cursor(0,1);
   puts("in 8 seconds");
   SPT_set(DELAY_1S);

   for(card_read_ok = 0, cntr = 8; cntr && !card_read_ok;) {
      if (!SPT_read()) {
         move_cursor(4,1);
   // Display at the position of '8' to show the remaining waiting time
         printf("%d",cntr-1);
         SPT_set(DELAY_1S);
         cntr --;
      }
   // block #block_num of sector #sec_num
      TempStatus = CardValue(
         PICC_AUTHENT1A,
         val > 0? PICC_INCREMENT:PICC_DECREMENT,
         sec_num,
         block_num,
         val > 0? val:-val
      );
      if(TempStatus == MI_OK )
         card_read_ok = 1;
   }
   if (!card_read_ok) {
      move_cursor(0,3);
      puts("No valid card detected");
      goto read_card_error;
   }
   else {
      MC530Off();
      return 0;
   }
read_card_error:
   MC530Off();
   move_cursor(0,6);
   puts("press anykey to return");
   getch();
   return 1;
}
```