MIFARE® Read/Write Device
C Library CCTDAPL for MIFARE®
Read/Write Device with 3964R Interface

Programming Guide

Release 02.98
Order no. 6GT2 397-6AA00-0DA2

Release: February 1998
Notes on safety

This manual contains notes which must be adhered to for your personal safety and avoidance of property damage. The notes are indicated with a warning triangle showing the degree of danger.

Danger

Death, severe injury or substantial property damage will result if appropriate precautions are not taken.

Warning

Death, severe injury or substantial property damage may result if appropriate precautions are not taken.

Caution

Minor injury or property damage may occur if appropriate precautions are not taken.

Note

Important information on the product, its handling or a certain part of the documentation requiring special attention.

Documentation history

The following releases of this documentation have been published to date. The letter in the "Remarks" column indicates the status of the release.

A  First edition
B  Rewritten version with new release status

<table>
<thead>
<tr>
<th>Release</th>
<th>Order No.</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>09.96</td>
<td>6GT2 397-6AA00-0DA2</td>
<td>A</td>
</tr>
<tr>
<td>02.98</td>
<td>6GT2 397-6AA00-0DA2</td>
<td>B</td>
</tr>
</tbody>
</table>

Editing: A&D SE V1
Publisher: A&D SE V1

Disclaimer of liability

Although we have checked the contents of this publication for agreement with the actual hardware and software described, no responsibility can be accepted for complete agreement. The information in this publication is checked at regular intervals and necessary corrections included in the next printing. Your suggestions are always welcome.

Subject to change without prior notice

Copyright

Copyright © Siemens AG 1996, 1998 All Rights Reserved

Passing on or reproduction of this documentation, and utilization and revelation of its contents is prohibited unless express permission is obtained. Violators shall be subject to legal action. All rights reserved, particularly when a patent is granted or a utility model registered.
# Table of Contents

1 Introduction .......................................................................................................................... 2  
   1.1 Product Overview............................................................................................................. 2  
   1.2 System Requirements...................................................................................................... 3  
2 Installation.......................................................................................................................... 5  
3 MIFARE® Card .................................................................................................................. 6  
   3.1 Data Structure of the MIFARE® Card ............................................................................. 6  
   3.2 Card Functions.............................................................................................................. 7  
      3.2.1 Read Card Data ...................................................................................................... 7  
      3.2.2 Write Card Data .................................................................................................... 11  
      3.2.3 Value Block Functions ......................................................................................... 12  
      3.2.4 Multiple Card Detection and Card Selection ....................................................... 13  
      3.2.5 Card Keys A and B with Individual Access Conditions ....................................... 15  
4 CCTDAPix.LIB Interface ...................................................................................................... 16  
   4.1 General ....................................................................................................................... 16  
      4.2 General Functions..................................................................................................... 20  
      4.2.1 Initialize Device Management - CCT_INIT Function .......................................... 20  
      4.2.2 Poll Version Number - CCT_VERSION Function ................................................ 20  
      4.2.3 Poll Error State - CCT_GET_ERR_STATE Function .............................................. 21  
      4.2.4 Poll Card in Field - CCT_GET_EVENT Function .................................................. 25  
      4.2.5 Poll Card in Field - CCT_CHECK_EVENT Function ............................................. 26  
      4.2.6 Cancel Wait for Event - CCT_FREE_EVENT Function ......................................... 27  
      4.2.7 Cancel Wait for Event - CCT_FREE_EVENT2 Function ....................................... 27  
   4.3 Device-Related Functions ........................................................................................... 28  
      4.3.1 Open Device - CCT_OPEN Function .................................................................... 28  
      4.3.2 Close Device - CCT_CLOSE Function .................................................................. 29  
      4.3.3 Read Card Data - CCT_READ_BLOCK Function ................................................. 30  
      4.3.4 Read Card Data - CCT_READ_N_BLOCK Function ............................................ 33  
      4.3.5 Write Card Data - CCT_WRITE_BLOCK Function ............................................... 36  
      4.3.6 Write Card Data - CCT_WRITE_N_BLOCK Function ........................................... 38  
      4.3.7 Set Up Value Block - CCT_INIT_VALUE Function .............................................. 40  
      4.3.8 Read Value Block - CCT_GET_VALUE Function ................................................... 41  
      4.3.9 Add Value to Value Block - CCT_ADD_VALUE Function ..................................... 42  
      4.3.10 Subtract Value from Value Block - CCT_SUB_VALUE Function ......................... 43  
      4.3.11 Select Card - CCT_SELECT_CARD Function ................................................... 44  
      4.3.12 Release Card - CCT_RELEASE_CARD Function ............................................... 45  
      4.3.13 Read Digital Inputs - CCT_READ_SIGNAL Function ......................................... 46  
      4.3.14 Address Digital Output - CCT_WRITE_SIGNAL Function ................................ 47  
      4.3.15 Parameterize Card Read/Write Interface - CCT_SETKEY Function .................. 49  
      4.3.16 Change Temporary Key Set - CCT_SET_KEY_TEMP Function .......................... 50  
      4.3.17 Set Temporary Keys Permanently - CCT_SET_KEY_PERM Function .................. 51  
      4.3.18 Poll Current Card Indices - CCT_GET_FIELD_STATE Function ......................... 52  
      4.3.19 Activate Event Control - CCT_ENABLE Function .............................................. 53  
      4.3.20 Deactivate Event Control - CCT_DISABLE Function ......................................... 54  
      4.3.21 Switch Off Antenna Field - CCT_OFF Function ............................................... 55  
      4.3.22 Set Field Operating Mode of the Antenna Field - CCT_MODE Function ............ 56  
      4.3.23 Reset Firmware - CCT_RESET Function ........................................................... 58  
      4.3.24 Poll Firmware Version - CCT_GET_FW_STATE Function ................................... 59
5 EC-COM Error States .......................................................................................................... 60
6 MIFARE® Read/Write Device Application........................................................................... 61
   6.1 Executable Sample Application................................................................................ 62
      6.1.1 Installing the EC-COM Driver ........................................................................ 62
      6.1.2 Parameterizing the EC-COM Driver ................................................................ 63
      6.1.3 Installing the Sample Application .................................................................... 64
      6.1.4 Using the Sample Application ......................................................................... 64
6.2 Sample Application in Source Code .............................................................................. 65
      6.2.1 Example of a Generation File XMPL_DOS.MAK........................................... 66
7 Appendix ....................................................................................................................... 68
   7.1 Terms/Abbreviations .............................................................................................. 68
1 Introduction

1.1 Product Overview

In addition to ES2-MIFARE® readers, the MOBY E SIM/OEM module is available for communication with MIFARE® proximity cards. This product can be used to customize MIFARE® read/write devices. Primary areas of application of the MOBY E SIM/OEM module include security, access monitoring, acquisition of plant data and attendance recording, trade and transportation, warehousing, logistics and assembly lines. The MIFARE® demonstrator kit/development kit is available for acquisition support, development and pilot installation. The MOBY E SIM/OEM module and the MIFARE® demonstrator kit/development kit are referred to in this manual as the MIFARE® read/write device.

The MIFARE® read/write device is connected via a serial interface of the PC (e.g., Siemens or SNI PC, or compatible models with at least an 80386SX processor) and driven with the 3964R protocol. EC-COM driver software is available for the 3964R protocol. Since communication with MIFARE® read/write devices at the 3964R protocol level would make quick and easy implementation of user applications difficult, a CCTDAPI C library containing functions has been created. Based on the EC-COM driver software, this library simplifies implementation by modularizing the applications.

This CCTDAPI C library makes it possible to work with MIFARE® read/write devices using applications under MS-DOS.

C library CCTDAPI

This product consists of four components.

- **CCTDAPIx.LIB** Import library in various memory models
  - x specifies the memory model version.

- **CCTDAPI.H and ECCOMD.H** Header files for user programming

- **Demonstrator software package** Sample application as executable program and as source code

In addition to the C library, a demonstrator software package is available. Its uses are listed below.

- Check the MIFARE® read/write device using a sample application after installation of the EC-COM 3964R driver
- Use the source code of the sample application as a basis on which to build a user application
1.2 System Requirements

Use of the C library under MS-DOS requires that the following system prerequisites be provided.

- **Personal computer** Siemens or SNI PC or compatible model with 80386SX processor or better
- **Operating system** Microsoft DOS, version 6.2
- **Serial interface** EC-COM driver software, version 1.1 for the 3964R protocol
- **Programming conventions**
  - The import library CCTDAPix.LIB was prepared with C++ and is compatible with Microsoft Visual C++ Compiler, version 1.50.01.
  - Conversion conventions
    - x = L: Large memory model
    - x = C: Compact memory model
    - x = H: Huge memory model
    - x = S: Small memory model
    - x = T: Tiny memory model
    - x = M: Medium memory model
- **Compiler options used**
  - The following compiler options were used for the CCTDAPix.LIB C with the Large memory model (x = L).
    - Project type: DOS EXE Release Mode
    - Customize Build Options
    - Option string (all options are coded here)
      /nologo/Gs/G2/AL/0x/D"NDEBUG"/FR
    - Code Generation
      CPU 80386
      Calling Convention C/C++
      Floating-Point Calls use Emulator
      Struct Member Byte Alignment 2byte
      Code Generator Auto Select
      Disable Stack Checking
    - Custom Options C++
      Presentation Method Best case always
      General-Purpose Representation
      Point to any class
    - Memory Model
      Model Large
      Segment Setup SS==DS
    - Optimization
      Customize Full optimization
      Inline Expansion of Functions Disable
      Inlined Function Size 4-Size Grade
      Preprocessor NDEBUG
      Windows Prolog/Epilog None

The included EC-COM driver software can be used for utilization of the sample application and testing a user application. When one or more applications are to be created, utilized or marketed, the EC-COM driver software must be ordered separately with a single or multiple license.
2 Installation

To create the user applications, the CCTDAPILIB import library and the CCTDAPILIB and ECCOMD.H header files must be loaded from floppy disk to the appropriate working directory of the MS-DOS operating system.

The data carrier contains the CCTDAPI C library with the EC-COM driver software stored under three directories:

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\lib</td>
<td>Interface library and header files for user applications</td>
</tr>
<tr>
<td>\sample</td>
<td>Demonstrator software package containing sample application as runnable program and in source code</td>
</tr>
<tr>
<td>\eccom</td>
<td>EC-COM driver software</td>
</tr>
</tbody>
</table>

To create the user applications, the CCTDAPIx.LIB interface library and the CCTDAPILIB and ECCOMD.H header files must be loaded from the data carrier to the appropriate working directory of the MS-DOS operating system.

The EC-COM driver software must be installed and parameterized (see chapters 6.1.1 and 6.1.2) before the sample application can be used or the user application tested.

Installation of the sample application is described in chapter 6.
3 MIFARE® Card

3.1 Data Structure of the MIFARE® Card

The MIFARE® card has an 8-kbit EEPROM memory which is divided into 16 sectors (i.e., sectors 0 to 15). Each sector consists of four blocks (i.e., blocks 0 to 3). The block is 16 bytes in length and is the smallest addressable memory area on the MIFARE® card. With one exception, each sector consists of three blocks of user data (i.e., blocks 0 to 2) and a sector trailer (i.e., block 3). Sector 0 is the exception. It contains the card manufacturer data in block 0 and can only be read-accessed. Access to the MIFARE® card memory is protected by two keys (i.e., A and B) per sector with individual access conditions for each block of the sector. These keys and the individual access conditions must be specified and stored in the appropriate sector trailer during so-called personalization of the card.

Data structure of the MIFARE® card

<table>
<thead>
<tr>
<th>8192-bit EEPROM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sector 0</td>
</tr>
<tr>
<td>Block 0</td>
</tr>
<tr>
<td>Block 1</td>
</tr>
<tr>
<td>Block 2</td>
</tr>
<tr>
<td>Block 3</td>
</tr>
<tr>
<td>Manufacturer data</td>
</tr>
<tr>
<td>User data</td>
</tr>
<tr>
<td>Key A (0)</td>
</tr>
<tr>
<td>Access conditions (0)</td>
</tr>
<tr>
<td>Key B (0)</td>
</tr>
<tr>
<td>Sector trailer 0</td>
</tr>
<tr>
<td>Sector 1</td>
</tr>
<tr>
<td>Block 0</td>
</tr>
<tr>
<td>Block 1</td>
</tr>
<tr>
<td>Block 2</td>
</tr>
<tr>
<td>Block 3</td>
</tr>
<tr>
<td>User data</td>
</tr>
<tr>
<td>Key A (1)</td>
</tr>
<tr>
<td>Access conditions (1)</td>
</tr>
<tr>
<td>Key B (1)</td>
</tr>
<tr>
<td>Sector trailer 1</td>
</tr>
<tr>
<td>Sector 15</td>
</tr>
<tr>
<td>Block 0</td>
</tr>
<tr>
<td>Block 1</td>
</tr>
<tr>
<td>Block 2</td>
</tr>
<tr>
<td>Block 3</td>
</tr>
<tr>
<td>User data</td>
</tr>
<tr>
<td>Key A (15)</td>
</tr>
<tr>
<td>Access conditions (15)</td>
</tr>
<tr>
<td>Key B (15)</td>
</tr>
<tr>
<td>Sector trailer 15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>16 sectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 blocks per sector = 64 blocks</td>
</tr>
<tr>
<td>16 bytes per block = 1024 bytes</td>
</tr>
<tr>
<td>47 user data blocks = 752 bytes</td>
</tr>
<tr>
<td>Keys A and B = 6 bytes each</td>
</tr>
<tr>
<td>Access conditions = 4 bytes</td>
</tr>
</tbody>
</table>
3.2 Card Functions

The MIFARE® read/write device offers the following card functions.

- Read card data
- Write card data
- Value block functions
- Multiple card detection and card selection

3.2.1 Read Card Data

The following can be read from the MIFARE® card:
- User data blocks (data blocks 0 to 2 of each sector)
- Sector trailer (block 3 of each sector)

Data block 0 of sector 0 is treated like a user data block during read accessing. (See section on reading sector trailers for restrictions when several cards are read.) See chapter 3.1.

- Read user data blocks
  The read function can be used to read data from one or more blocks of user data. Permissible data lengths range from 1 byte to 192 bytes. This provides up to 12 consecutive user data blocks (i.e., 1/4 of total available user data). The start address of the data to be read is specified by a block number from 0 to 62 (i.e., read-accesses must always start at the beginning of a block). The block number is generated consecutively over all sectors and not generated separately for each sector as described in chapter 3.1. A sector trailer cannot be used as the start address. When data lengths greater than 16 bytes are involved (i.e., more than one user data block) and one or more sector trailers are located between the first user data block to be read and the last, the sector trailer or sector trailers are skipped. Only once block of user data can be read at a time with the MIFARE® demonstrator kit/development kit.

- Read sector trailer or data block 0 of sector 0
  The read function can only read the data from one sector trailer at a time. Permissible data lengths range from 1 byte to 16 bytes. The start address of the data to be read is specified by block number 3, 7, 11, ..., 63. The block number is generated consecutively over all sectors and not separately for each sector as described in chapter 3.1.

Read-accessing requires that the necessary keys be loaded in the firmware of the MIFARE® read/write device and the card fulfill the required access conditions. If the required keys for the user data block or sector trailer to be read are not set and/or access is not permissible, the call is rejected with an error message and no data are transferred.

The MIFARE® read/write device offers two methods of read-access.

- "Unspecified" read command (READ)
- "Specified" read command (SREAD)

READ determines the series number of a card in the field and reads the desired data. The series number of the card is only stored intermediately and is not sent to the user program in the host computer.

SREAD uses the series number of the last read or write job to read-access the desired data. The intermediately stored series number is sent to the card. Use of the same series number for consecutive card accesses ensures that the same card is always addressed. The series number is not sent to the user program in the host computer.
The read job can be performed in one of two operating modes.
- Event-controlled
- Polled (i.e., not event-controlled)

Event-controlled operating mode (without multiple cards)

```
  Activate event control
  CCT_ENABLE function
  Return value "OK"

  Read call (READ)
  CCT_READ_N_BLOCK or
  CCT_READ_BLOCK function
  Return value "OK"

  Poll for "card in the field"
  CCT_CHECK_EVENT or
  CCT_GET_EVENT function

  Card in the field
  YES Return value "DEVICE_EVENT" or "OK"
  NO Return value "NOEVENT"

  Read card data from buffer

  Read or write call
  (SREAD or SWRITE)

  Deactivate event control?
  YES

  Deactivate event control
  CCT_DISABLE function
  Return value "OK"
```

Possible errors are not included in this flowchart.
Possible correct processing sequences with event control activated. See chapter 4.3.19.

A) A card is located in the field at the time of the read call.

1. Read job (READ)  
   (Acknowledgment of the firmware): Return value "OK"

2. Poll for "card in field"  
   Return value "DEVICE-EVENT" or "OK"
   Transfer buffer contains card data.

3. Accept card data which were read

4. If necessary, read and/or write additional data on the same card with SREAD and SWRITE

B) No card is located in the field at the time of the read call.

1. Read job (READ)  
   (Acknowledgment of the firmware): Return value "OK"

2. Poll for "card in field"  
   Return value "NOEVENT"

3. Poll for "card in field"  
   Return value "DEVICE-EVENT" or "OK"
   Transfer buffer contains card data.

4. Accept card data which were read

5. If necessary, read and/or write additional data on the same card with SREAD and SWRITE.
Polling operating mode (without multiple cards)

Possible correct processing sequences with event control deactivated. See chapter 4.3.20.

A) A card is located in the field at the time of the read call.
   1. Read job (READ) (Acknowledgment of the firmware): Return value "OK"
      Transfer buffer contains card data.
   2. Accept card data which were read
   3. If necessary, read and/or write additional data on the same card with SREAD and SWRITE

B) No card is located in the field at the time of the read call.
   1. Read job (READ) (Acknowledgment of the firmware): Return value "NO_CARD"

Behavior when two or more cards are located in the field is described in chapter 3.2.4.
3.2.2 Write Card Data

With the exception of sector 0 of block 0, data blocks 0 to 2 of each sector can be write-accessed on the MIFARE® card. See chapter 3.1. These data blocks are user data blocks. Block 3 (i.e., sector trailer) of each sector cannot be write-accessed.

The write function can be used to write data in one or more user data blocks. Permissible data lengths range from 1 byte to 192 bytes. This provides up to 12 consecutive user data blocks (i.e., 1/4 of total available user data). The start address of the data to be written is specified by a block number from 1 to 62 (i.e., write-accesses must always start at the beginning of a block). The block is generated consecutively over all sectors and not separately for each sector as described in chapter 3.1. A sector trailer cannot be used as the start address. When a data length greater than 16 bytes is involved (i.e., more than one block of user data) and one or more sector trailers are located between the first user data block to be written and the last, these are skipped. If the keys required for write-accessing the user data block are not set and/or access is not permissible, the call is rejected with an error message and no data are transferred.

Only one block of user data can be written at a time with the MIFARE® demonstrator kit/development kit.

Like the read command, two methods of write-access are available.

- "Unspecified" write command (WRITE)
- "Specified" write command (SWRITE)

WRITE determines the series number of a card in the field and writes the desired data. The series number of the card is not sent to the user program in the host computer.

SWRITE uses the series number of the last read or write job to write the desired data. The intermediately stored series number is sent to the card. Use of the same series number for consecutive card accesses ensures that the same card is always addressed. The series number is not sent to the user program in the host computer.

The two methods of write-access are specified with the "nOption" parameter in the function call.

In contrast to the read procedure, the write procedure cannot be event-controlled. A card must always be located in the field when a write procedure occurs. A verify read (SREAD) should be used to ensure that the data have actually been written on the card. See chapter 4.3.3 or 4.3.4.

Behavior when two or more cards are located in the field is described in chapter 3.2.4.
3.2.3 Value Block Functions

Amount values can be entered on the card in value blocks when MIFARE® cards are used in automatic units, for example. Value blocks are handled by special read and write mechanisms on the MIFARE® card. They can only be used with the MOBY E SIM/OEM module.

A value block consists of a block containing the original value and a backup block with a copy of the original value. Changes in amount are always performed on the backup block first and then repeated in the original block.

At the beginning of a value block processing procedure, the original block and the backup block are checked for internal consistency and equality. If one of the two blocks is inconsistent, it is generated from the one which is consistent. If both are inconsistent, the card is reported as defective. If both blocks are consistent but not equal, the backup block is generated from the original block.

The original block and the backup block must be located in one sector. The blocks in this sector which are still free should be read and write-disabled when the card is personalized.

Various functions are available for value block processing.

• Set up value block
  This function sets up a value block initialized with zero.

• Read value block
  This function reads the stored value.

• Add value to the value block
  This function adds the specified value to the value stored in the value block.

• Subtract value from the value block
  This function subtracts the specified value from the value stored in the value block.

Behavior when two or more cards are located in the field is described in chapter 3.2.4.
3.2.4 Multiple Card Detection and Card Selection

A MIFARE® card does not necessarily have to be taken out of a coin purse or wallet for a transaction (e.g., read data). It is possible that several MIFARE® cards may be needed for different applications, and the appropriate card must be selected for the transaction to be performed. The MIFARE® read/write device provides appropriate functions for this. These can only be used with the MOBY E SIM/OEM module.

The MIFARE® read/write device recognizes when more than one card is located in the field. It identifies each card by its series number, assigns a card index to the card and enters the card in an internal list. Eight can be stored theoretically but in reality only 3 are physically possible (if the read-access distance is not exceeded or no foreign bodies (e.g., coins) are located in the field). A selection function (see chapter 4.3.11) can be used by the application to select a certain card and then process this card synchronously.

The following card states are available.

- **One card in the field**
  1.) **Status**: No card selected
      The card can be processed synchronously. Selection is not necessary.
      ⇒ When another card enters the field, the card is entered in the internal list with the series number and its card index. When possible, an action started with the first card is concluded correctly. Another call is rejected with a "double card" message. The card must first be selected for further call(s). This is the first card index (i.e., index 0).
  2.) **Status**: The card in the field is selected.
      The selected card can be processed synchronously.
      ⇒ When another card enters the field, the card is entered in the internal list with the series number and its card index.
  3.) **Status**: A card which was previously located in the field is still selected.
      The card in the field cannot be processed synchronously until the disabled card which is no longer in the field is released. The card receives the lowest free index. Otherwise, see 1).
• **More than one card in the field**
  
  4.) **Status: No card selected**
  A card can be selected.
  ⇒ When an unselected card exits the field, the series number and its card index are removed from the internal list.
  ⇒ When another card enters the field, it is entered in the internal list with the series number and its card index.

  5.) **Status: A card in the field is selected.**
  The selected card can be processed synchronously.
  ⇒ When an unselected card exits the field, the series number and its card index are removed from the internal list.
  ⇒ When another card enters the field, it is entered in the internal list with the series number and its card index.
  ⇒ When the selected card exits the field and other cards are still located in the field, the card index remains reserved for this card until it is released by the application. See chapter 4.3.12. The old index cannot be used for a new card until this occurs. If the card index is not released by the application and the same card exits and enters the field again, it can be addressed by its old card index.
  ⇒ When the selected card has been released but still remains in the field, the card index is not assigned again until the card has exited the field.

  6.) **Status: A card which was previously in the field is still selected.**
  A card in the field cannot be processed synchronously until the disabled card which is no longer in the field is released and the card to be processed is selected. Continue with 4).

• **No card in the field**

  7.) **Status: A selected card was not released.**
  The card index for this card remains reserved until the card is released by the application. See chapter 4.3.12. If the same card returns to the field, it can be addressed by its old card index.

  8.) **Status: No card selected**
  The internal index list is blank.

There are several methods by which the application can determine whether more than one card is located in the field.

• Via a transaction call as DOUBLE_CARD return value
  - After read card data (See chapter 4.3.3 or 4.3.4.)
  - After write card data (See chapter 4.3.5 or 4.3.6.)
  - After value block functions (See chapters 4.3.7, 4.3.8, 4.3.9, and 4.3.10.)

• Via a status scan with the CCT_GET_FIELD_STATE poll current card indices function of the cards in the field. See chapter 4.3.18.
### 3.2.5 Card Keys A and B with Individual Access Conditions

Access to the MIFARE® card memory is protected by two keys (i.e., A and B) for each sector with individual access conditions for each block of the sector. See chapter 3.1. These keys and the individual access conditions must be specified during card personalization.

When the following card functions are used, the access conditions for the user data blocks must be set as shown in the table below.

- Read card data
- Write card data
- Value block functions (i.e., set up value block, read value block, add value to value block, and subtract value from value block)

<table>
<thead>
<tr>
<th>Card Function Used</th>
<th>Card Access Permitted (Access Condition)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read card data</td>
<td>Read</td>
</tr>
<tr>
<td>Write card data</td>
<td>Write</td>
</tr>
<tr>
<td>Set up value block</td>
<td>Read and write</td>
</tr>
<tr>
<td>Read value block</td>
<td>Read, restore and transfer</td>
</tr>
<tr>
<td>Add value to value block</td>
<td>Read, increment, restore and transfer</td>
</tr>
<tr>
<td>Subtract value from value block</td>
<td>Read, decrement, restore and transfer</td>
</tr>
</tbody>
</table>
4 CCTDAPlx.LIB Interface

4.1 General

The MIFARE® read/write unit consists of several logical devices. These logical units are controlled by the transmission of structured telegrams over the serial interface with the 3964R protocol. These telegrams must be generated or received with library functions. These library functions are based on the EC-COM driver (i.e., the 3964R protocol).

<table>
<thead>
<tr>
<th>Application with CCTDAPlx.LIB</th>
</tr>
</thead>
<tbody>
<tr>
<td>EC-COM 3964R driver</td>
</tr>
<tr>
<td>Hardware with COM interface</td>
</tr>
<tr>
<td>MIFARE® read/write device with 3964R interface</td>
</tr>
</tbody>
</table>

The function calls are stored in import library CCTDAPlx.LIB.

The functions are divided into two classes.
- General
- Device-related

General functions are not directly related to the MIFARE® read/write device. They do not initiate communication with the MIFARE® read/write device. The sole task of the general functions is to provide data/states managed by the library.
General functions:

<table>
<thead>
<tr>
<th>Function Call</th>
<th>Short Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCT_INIT</td>
<td>Initialize internal library device management</td>
</tr>
<tr>
<td>CCT_VERSION</td>
<td>Poll version number of the function library</td>
</tr>
<tr>
<td>CCT_GET_ERR_STATE</td>
<td>Poll the error status of the card read/write unit if an error was transferred as the return value for a function</td>
</tr>
<tr>
<td>CCT_GET_EVENT</td>
<td>Poll &quot;card in the field&quot;</td>
</tr>
<tr>
<td>CCT_CHECK_EVENT</td>
<td>Poll &quot;card in the field&quot;</td>
</tr>
<tr>
<td>CCT_FREE_EVENT</td>
<td>Wait for &quot;card in the field&quot; to be concluded</td>
</tr>
<tr>
<td>CCT_FREE_EVENT2</td>
<td>Wait for &quot;card in the field&quot; to be concluded</td>
</tr>
</tbody>
</table>

The device-related functions initiate communication with the MIFARE® read/write device. They refer to the three logical devices of the MIFARE® read/write device.

- MIF - card read/write unit (i.e., MIFARE® read/write device)
- DOP - digital inputs/outputs (i.e., Digital Input/Output Port)
- CTL - terminal controller (Card Terminal)

The device-related functions are divided into two function classes.

- Main functions which a logical device executes
- Monitoring functions with which the behavior of the card read/write unit is determined or polled
### Device-related main functions:

<table>
<thead>
<tr>
<th>Function Call</th>
<th>Short Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCT_OPEN</td>
<td>Open logical device</td>
</tr>
<tr>
<td>CCT_CLOSE</td>
<td>Close logical device</td>
</tr>
<tr>
<td>CCT_READ_BLOCK</td>
<td>Read card data (1 data block)</td>
</tr>
<tr>
<td>CCT_READ_N_BLOCK</td>
<td>Read card data (n data blocks)</td>
</tr>
<tr>
<td>CCT_WRITE_BLOCK</td>
<td>Write card data (1 data block)</td>
</tr>
<tr>
<td>CCT_WRITE_N_BLOCK</td>
<td>Write card data (n data blocks)</td>
</tr>
<tr>
<td>CCT_INIT_VALUE</td>
<td>Set up value block (i.e., initialize)</td>
</tr>
<tr>
<td>CCT_GET_VALUE</td>
<td>Read value block</td>
</tr>
<tr>
<td>CCT_ADD_VALUE</td>
<td>Add value to value block</td>
</tr>
<tr>
<td>CCT_SUB_VALUE</td>
<td>Subtract value from value block</td>
</tr>
<tr>
<td>CCT_SELECT_CARD</td>
<td>Select card</td>
</tr>
<tr>
<td>CCT_RELEASE_CARD</td>
<td>Release card (i.e., deselect)</td>
</tr>
<tr>
<td>CCT_READ_SIGNAL</td>
<td>Read digital inputs</td>
</tr>
<tr>
<td>CCT_WRITE_SIGNAL</td>
<td>Address digital output</td>
</tr>
</tbody>
</table>

### Device-related monitoring functions:

<table>
<thead>
<tr>
<th>Function Call</th>
<th>Short Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCT_SETKEY</td>
<td>Parameterize card read/write interface</td>
</tr>
<tr>
<td>CCT_SET_KEY_TEMP</td>
<td>Change temporary keys</td>
</tr>
<tr>
<td>CCT_SET_KEY_PERM</td>
<td>Set temporary keys permanently</td>
</tr>
<tr>
<td>CCT_GET_FIELD_STATE</td>
<td>Poll current card indices</td>
</tr>
<tr>
<td>CCT_ENABLE</td>
<td>Activate event control</td>
</tr>
<tr>
<td>CCT_DISABLE</td>
<td>Deactivate event control</td>
</tr>
<tr>
<td>CCT_OFF</td>
<td>Switch off antenna field</td>
</tr>
<tr>
<td>CCT_MODE</td>
<td>Set field operating mode of the antenna field</td>
</tr>
<tr>
<td>CCT_RESET</td>
<td>Reset firmware</td>
</tr>
<tr>
<td>CCT_GET_FW_STATE</td>
<td>Poll firmware version</td>
</tr>
</tbody>
</table>
Table of functions and their allocation to the logical devices:

<table>
<thead>
<tr>
<th>Function</th>
<th>MIF</th>
<th>DOP</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCT_OPEN</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>CCT_CLOSE</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>CCT_READ_BLOCK</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CCT_READ_N_BLOCK</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CCT_WRITE_BLOCK</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CCT_WRITE_N_BLOCK</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CCT_INIT_VALUE</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CCT_GET_VALUE</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CCT_ADD_VALUE</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CCT_SUB_VALUE</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CCT_SELECT_CARD</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CCT_RELEASE_CARD</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CCT_READ_SIGNAL</td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>CCT_WRITE_SIGNAL</td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>CCT_SETKEY</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CCT_SET_KEY_TEMP</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CCT_SET_KEY_PERM</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CCT_GET_FIELD_STATE</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CCT_ENABLE</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CCT_DISABLE</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CCT_OFF</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CCT_MODE</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CCT_RESET</td>
<td>x(^1)</td>
<td>x(^1)</td>
<td>x(^1)</td>
</tr>
<tr>
<td>CCT_GET_FW_STATE</td>
<td>x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1) This function is used to reset the firmware of the MIFARE® read/write device. All logical devices are then closed.

Each logical device (i.e., MIF, DOP, CTL) can be manipulated by selecting telegrams (i.e., function calls). Each telegram consists of a request and the appropriate response. This requires that one read and one write call (EC-COM driver function) be executed over the interface. The application is only informed of read and write calls of the EC-COM driver when errors occur. When an error occurs, an appropriate error status is returned by the EC-COM. Possible error status values of the EC-COM driver are listed in chapter 4.

**Note**

The EC-COM interface is opened automatically by the CCT_OPEN function and closed by the CCT_CLOSE function.

The individual functions and their parameters and return values are described in detail in the following sections.
4.2 General Functions

4.2.1 Initialize Device Management - CCT_INIT Function

The CCT_INIT() function initializes internal library management. All previously opened and managed devices are deleted. Execution of this command is implicit with a CCT_RESET.

Function header:

```c
int CCT_INIT (void)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Return value:

OK

The constants (e.g., OK) are declared in the CCTDAPI.H header file.

4.2.2 Poll Version Number - CCT_VERSION Function

The CCT_VERSION() function supplies the pointer to the version number of the function library and returns the integer value of the version number of the function library.

Function header:

```c
int CCT_VERSION (byte *szVersion)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>szVersion</td>
<td>* byte</td>
<td>Five-position version number interpreted as Version.<em>.</em>._.</td>
</tr>
</tbody>
</table>

Return value:

Version number // Integer value

// Example: 103 corresponds to version 1.03.

Example:

```c
byte cc[5];
uint c;
c = CCT_VERSION(cc);
printf(" Version as string %s",cc);
printf(" Version as integer %d",c);
..```

Note

The parameter types "byte" and "uint" are type definitions and stand for "unsigned char" and "unsigned int". These types are used with this function and sometimes with other functions.
4.2.3 Poll Error State - CCT_GET_ERR_STATE Function

The CCT_GET_ERR_STATE() function is used to poll the error states of the card read/write unit when one of the following errors is returned as a return value for a function.

- WRONG_PARAM
- WRONG_REQ
- DEVICE_ERROR
- DOUBLE_CARD

This error status is sent as the third byte after the return value and the device identifier.

When the return value is DOUBLE_CARD, the third byte contains the number of cards in the field and the card indices (from the fourth byte onwards) instead of an error status.

**Function header:**

```c
int CCT_GET_ERR_STATE (StructErrFrame *ErrState)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ErrState</td>
<td>* StructErrFrame</td>
<td>Pointer to a user structure in which the error status of the card read/write unit is to be stored</td>
</tr>
</tbody>
</table>

**Return value:**

- OK       // Function executed
- NOT_AVAILABLE // The card read/write unit is okay. No error status // present.

Error status information after the return value:

- DEVICE_ERROR
- WRONG_REQ
- WRONG_PARAM

<table>
<thead>
<tr>
<th>err_structure.response_id</th>
<th>Response identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>err_structure.device_id</td>
<td>Device identifier</td>
</tr>
<tr>
<td>err_structure.data[0]</td>
<td>Error parameter</td>
</tr>
</tbody>
</table>

err_structure is of the type StructErrFrame.
Possible error parameter values after CT_DEVICE_ERROR:

<table>
<thead>
<tr>
<th>Define Instruction</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>MI_FLASH_WR_ERR</td>
<td>0x01</td>
<td>Flash error</td>
</tr>
<tr>
<td>MI_CONTROL_ERR</td>
<td>0x02</td>
<td>Control error</td>
</tr>
<tr>
<td>MI_READ_ERR</td>
<td>0x03</td>
<td>Read error during READ function</td>
</tr>
<tr>
<td>MI_SREAD_ERR</td>
<td>0x04</td>
<td>Read error during SREAD function</td>
</tr>
<tr>
<td>MI_REQU_ALL_FAIL</td>
<td>0x70</td>
<td>Core module error during introduction of card polling</td>
</tr>
<tr>
<td>MI_NO_CARD</td>
<td>0x71</td>
<td>No card in the antenna field</td>
</tr>
<tr>
<td>MI_DOUBLE_CARD</td>
<td>0x73</td>
<td>More than one card in the antenna field</td>
</tr>
<tr>
<td>MI_READERR</td>
<td>0x74</td>
<td>Trouble during the read card function</td>
</tr>
<tr>
<td>MI_WRITEERR</td>
<td>0x75</td>
<td>Trouble during the write card function</td>
</tr>
<tr>
<td>MI_SWRITEERR</td>
<td>0x76</td>
<td>Write error during the SWRITE function</td>
</tr>
<tr>
<td>MI_NOTAGERR</td>
<td>-1</td>
<td>No card in the antenna field</td>
</tr>
<tr>
<td>MI_CRCERR</td>
<td>-2</td>
<td>CRC error reported by the card</td>
</tr>
<tr>
<td>MI_EMPTY</td>
<td>-3</td>
<td>Value overflow</td>
</tr>
<tr>
<td>MI_AUTHERR</td>
<td>-4</td>
<td>Authentication of the card not possible</td>
</tr>
<tr>
<td>MI_TESTERR</td>
<td>-5</td>
<td>Test error</td>
</tr>
<tr>
<td>MI_CODEERR</td>
<td>-6</td>
<td>Communication problem</td>
</tr>
<tr>
<td>MI_QUITERR</td>
<td>-7</td>
<td>Function error</td>
</tr>
<tr>
<td>MI_SERNRERR</td>
<td>-8</td>
<td>Wrong series number read during anti-collision</td>
</tr>
<tr>
<td>MI_SERVICEERR</td>
<td>-9</td>
<td>Function error</td>
</tr>
<tr>
<td>MI_NOTAUTHERR</td>
<td>-10</td>
<td>Card is not authenticated.</td>
</tr>
<tr>
<td>MI_VALUEERR</td>
<td>-11</td>
<td>Function error</td>
</tr>
<tr>
<td>MI_KEYLOADERR</td>
<td>-12</td>
<td>XRAM error</td>
</tr>
<tr>
<td>MI_IDLE</td>
<td>-13</td>
<td>Wait for DV-Bit = 1</td>
</tr>
<tr>
<td>MI_TRANSERR</td>
<td>-14</td>
<td>Trouble during transfer card function</td>
</tr>
<tr>
<td>MI_INCRERR</td>
<td>-16</td>
<td>Trouble during increment card function</td>
</tr>
<tr>
<td>MI_DECRERR</td>
<td>-17</td>
<td>Trouble during decrement card function</td>
</tr>
<tr>
<td>MI_POLLING</td>
<td>-20</td>
<td>Message after initialization of polling mode</td>
</tr>
<tr>
<td>MI_NY_IMPLEMENTIERT</td>
<td>-100</td>
<td>Function not implemented</td>
</tr>
<tr>
<td>MI_SELECT_FAIL</td>
<td>0xB0</td>
<td>Card no longer accessible during access (selection)</td>
</tr>
<tr>
<td>MI_LOAD_KEY_FAIL</td>
<td>0xB1</td>
<td>Error while loading the sector keys. Firmware reset recommended.</td>
</tr>
<tr>
<td>MI_BOTH_INCONSIST</td>
<td>0xB2</td>
<td>Original and backup value block are inconsistent.</td>
</tr>
</tbody>
</table>
Define Instruction | Value | Meaning
--- | --- | ---
MI_ORIG_INCONSIST | 0xB3 | Original value block is inconsistent.
MI_BACK_INCONSIST | 0xB4 | Backup value block is inconsistent.
MI_ABORT | 0xB5 | Function aborted
MI_NO_RETRY | 0xB6 | Do not try again.
MI_VALUE_LOW | 0xB7 | Value low -> not yet used
MI_CMODUL_CONF_FAIL | 0xB8 | Firmware error during selection of key AB. Firmware reset recommended.
MI_WRONG_KEY_AB | 0xB9 | Firmware error during selection of key. The key is not specified (i.e., not equal to key A or B). Firmware reset recommended.
MI_SECURITY_BLOCK | 0xBA | Write-access in security block
MI_DEVICE_ERROR | 0xBB | MIFARE error
MI_WRONG_KEY | 0xBC | Wrong key
MI_VBACK_INCONSISTENT | 0xBD | Initialize value block
MI_VORG_INCONSISTENT | 0xBE | Initialize value block

Possible error parameter values after CT_WRONG_REQ:

Define Instruction | Value | Meaning
--- | --- | ---
MI_INVALID_ID | 0x11 | Illegal request identifier
Possible error parameter values after CT_WRONG_PARAM:

<table>
<thead>
<tr>
<th>Define Instruction</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>MI_WRONG_SECTOR</td>
<td>0x03</td>
<td>Illegal sector</td>
</tr>
<tr>
<td>MI_WRONG_KEYAB</td>
<td>0x06</td>
<td>KEY_AB &gt; 1 (setkey function).</td>
</tr>
<tr>
<td>MI_WRONG_KEY_AB</td>
<td>0x02</td>
<td>KEY_AB &gt; 1 putkey (temporary)</td>
</tr>
<tr>
<td>MI_WRONG_KEYSET</td>
<td>0x05</td>
<td>KEYSET &gt; 3</td>
</tr>
<tr>
<td>MI_WRONG_MODE</td>
<td>0x02</td>
<td>Illegal mode specified</td>
</tr>
<tr>
<td>MI_WRONG_BACK_ADR</td>
<td>0x02</td>
<td>Illegal backup value block address</td>
</tr>
<tr>
<td>DOP_ADDRESS_ERR</td>
<td>0x01</td>
<td>Illegal DOP address specified</td>
</tr>
<tr>
<td>MI_WRONG_ADR</td>
<td>0x01</td>
<td>Block address not in permissible range</td>
</tr>
<tr>
<td>MI_WRONG_LENGTH</td>
<td>0x02</td>
<td>Block length not in permissible range</td>
</tr>
<tr>
<td>MI_WRONG_VERSION</td>
<td>0x01</td>
<td>FW-STATE telegram: Illegal object ID</td>
</tr>
<tr>
<td>MI_WRONG_INDEX</td>
<td>0x01</td>
<td>Index too high</td>
</tr>
<tr>
<td>MI_DIFF_SECTOR_SPEC</td>
<td>0xFF</td>
<td>Addresses of the original and backup block are not located in the same sector.</td>
</tr>
<tr>
<td>MI_WRONG_CONTROL_ID</td>
<td>0x01</td>
<td>Control ID too high</td>
</tr>
<tr>
<td>MI_WRONG_BACK_BLOCK</td>
<td>0x02</td>
<td>Value function error</td>
</tr>
<tr>
<td>MI_WRONG_ORG_BLOCK</td>
<td>0x01</td>
<td>Value function error</td>
</tr>
</tbody>
</table>

The value of the error parameter specifies the position of the function parameter on the firmware level, and not the C library level at which the error occurred. The first parameter of a function has the value 1.

Error status information after the return value

- DOUBLE_CARD

<table>
<thead>
<tr>
<th>err_structure.response_id</th>
<th>Response identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>err_structure.device_id</td>
<td>Device identifier</td>
</tr>
<tr>
<td>err_structure.data[0]</td>
<td>Number of cards in bytes. Value 0x02 to 0x08</td>
</tr>
<tr>
<td>err_structure.data[1]</td>
<td>1st index assigned</td>
</tr>
<tr>
<td>err_structure.data[n]</td>
<td>nth index assigned</td>
</tr>
</tbody>
</table>

err_structure is of the type StructErrFrame.
4.2.4 Poll Card in Field - CCT_GET_EVENT Function

The CCT_GET_EVENT() function polls the status of the interface driver operating in the background and, if necessary, transfers the receive data in the buffer supplied.

Function header:

```
int CCT_GET_EVENT (byte *cReceiveData)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cReceiveData</td>
<td>* byte</td>
<td>Pointer to a data area in which the read data are to be stored</td>
</tr>
</tbody>
</table>

Return value:

- DEVICE_EVENT  // Function executed with data transfer
- NODATA        // Function executed without data transfer
- NOEVENT       // Error (receive length < 4 bytes)
- NOEVENT       // No EVENT recorded. Continue with
- DOUBLE_CARD   // CCT_GET_EVENT or CCT_CHECK_EVENT
- WAIT_FOR_EVENT // polling.
- DOUBLE_CARD   // More than one card
- WAIT_FOR_EVENT // Wait for an event

Note

The MIFARE® demonstrator kit/development kit uses "OK" instead of "DEVICE_EVENT."
4.2.5 Poll Card in Field - CCT_CHECK_EVENT Function

The CCT_CHECK_EVENT() function polls the status of the interface driver operating in the background and, if necessary, transfers the receive data to the supplied buffer.

In contrast to the CCT_GET_EVENT function, an interface is specified here (i.e., waiting for an event or aborting event mode can be handled for each interface individually). This permits several MIFARE® read/write devices to be processed simultaneously in event mode.

Function header:

```c
int CCT_CHECK_EVENT (byte cComNum, byte *cReceive_Data)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cComNum</td>
<td>byte</td>
<td>Number of the COM interface</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = COM1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 = COM2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>..</td>
</tr>
<tr>
<td>cReceive_Data</td>
<td>* byte</td>
<td>Pointer to a data area in which the read data are to be stored</td>
</tr>
</tbody>
</table>

Return value:

- **DEVICE_EVENT**  // Function executed with data transfer
- **NODATA**        // Function executed without data transfer
  // Error (received length < 4 bytes)
- **NOEVENT**        // No event recorded. Continue polling with
  // CCT_GET_EVENT or CCT_CHECK_EVENT.
- **DOUBLE_CARD**    // More than one card
- **WAIT_FOR_EVENT** // Wait for an event

**Note**

The MIFARE® demonstrator kit/development kit uses "OK" instead of "DEVICE_EVENT."
4.2.6 Cancel Wait for Event - CCT_FREE_EVENT Function

The CCT_FREE_EVENT() function cancels the wait procedure for an event which is blocking all other commands (except CCT_GET_EVENT and CCT_RESET). The EC-COM interface is briefly closed. Event mode can then be disabled with the CCT_DISABLE function.

Function header:

```c
int CCT_FREE_EVENT (void)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Return value:

- **OK** // Function executed
- **Error status of EC-COM** // Value < 0. For meaning, see chapter 4.

4.2.7 Cancel Wait for Event - CCT_FREE_EVENT2 Function

The CCT_FREE_EVENT2() function cancels the wait procedure for an event which is blocking all other commands (except CCT_GET_EVENT and CCT_RESET). The EC-COM interface is briefly closed. Event mode can then be disabled with the CCT_DISABLE function.

In contrast to the CCT_FREE_EVENT function, an interface is specified here (i.e., waiting for an event or aborting event mode can be handled for each interface individually). This permits several MIFARE® read/write devices to be processed simultaneously in event mode.

Function header:

```c
int CCT_FREE_EVENT2 (byte cComNum)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cComNum</td>
<td>byte</td>
<td>Number of the COM interface</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = COM1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 = COM2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>..</td>
</tr>
</tbody>
</table>

Return value:

- **OK** // Function executed
- **Error status of EC-COM** // Value < 0. For meaning, see chapter 4.
4.3 Device-Related Functions

4.3.1 Open Device - CCT_OPEN Function

The CCT_OPEN() function opens the specified logical device (i.e., MIF, DOP and CTL) for further accesses.

The sending and receiving buffers are set up. The MIFARE-OPEN data record is generated with the necessary function identifier (Request_Id) and device type. If necessary, the COM interface is opened. Acknowledgments from the function calls (i.e., driver functions) are evaluated and the device identifier is administered for future MIFARE commands.

Function header:

\[ \text{int } \text{CCT\_OPEN (byte cComNum, byte device\_type[5])} \]

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cComNum</td>
<td>byte</td>
<td>Number of the COM interface</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = COM1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 = COM2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>..</td>
</tr>
<tr>
<td>device_type</td>
<td>byte</td>
<td>Device type:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;MIF:&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;DOP:&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;CTL:&quot;</td>
</tr>
</tbody>
</table>

Return value:

Return value \( \geq 0 \)  // Return value of the logical device

OK  // No error

WRONG\_REQ  // Wrong request identifier

WRONG\_PARAM  // Wrong parameter value range

DEVICE\_ALREADY\_OPEN  // Device already open

WAIT\_FOR\_EVENT  // Wait for an event

Return value \(< 0\)  // Error status of EC-COM. For meaning, see chapter 4.

Example:

\[ \text{uint com\_n = 1; \quad // COM1} \]
\[ \text{open\_status = CCT\_OPEN(com\_n, "MIF:");} \]
\[ \text{if (open\_status ! = 0)} \quad // \text{MIFARE failure} \quad > 0 \]
\[ \text{\quad // EC-COM failure} \quad < 0 \]
\[ \{ \text{Output\_Message("MIF - open failed"); } \} \]
### 4.3.2 Close Device - CCT_CLOSE Function

The CCT_CLOSE() function closes the logical device specified. If no other logical device is present on the COM interface, this interface is closed.

**Function header:**

`int CCT_CLOSE (byte cComNum, byte device_type[5])`

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cComNum</td>
<td>byte</td>
<td>Number of the COM interface</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = COM1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 = COM2</td>
</tr>
<tr>
<td>device_type</td>
<td>byte</td>
<td>Device type:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;MIF:&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;DOP:&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;CTL :&quot;</td>
</tr>
</tbody>
</table>

**Return value:**

- Return value $\geq 0$  
  - OK  
  - WRONG_REQ  
  - WRONG_PARAM  
  - DEVICE_NOT_OPEN  
  - DEVICE_UNKNOWN  
  - WAIT_FOR_EVENT
  
- Return value $< 0$  
  - Error status of EC-COM. For meaning, see chapter 4.

**Example:**

```c
uint com_n = 1;  // COM1
byte device[5] = "MIF:";
...
close_status = CCT_CLOSE (com_n, device);
if (close_status ==0)
{
    Output_Message("MIF - closed "); }
```
4.3.3 Read Card Data - CCT_READ_BLOCK Function

The CCT_READ_BLOCK() function reads a block of data from the card. An entire block must be read.

The MIFARE™ read/write device offers two methods of read-access.

- "Unspecified" read command (READ)
- "Specified" read command (SREAD)

READ determines the series number of a card in the field and reads the desired data block. The series number of the card is only stored intermediately and is not sent to the application program in the host computer.

SREAD uses the series number of the last read or write job to read the desired data block. The series number stored intermediately is also sent to the card. Using the same series number for consecutive card accesses ensures that the same card is always addressed. The series number is not sent to the application program in the host computer.

These two types of read-access are specified with the "nOption" parameter in the function call.

Performance of the read job depends on the operating mode set.

- Event control activated
- Event control deactivated

Possible processing sequences with event control activated

A) A card is located in the field at the time of the read call.

1. Read job (READ)  
   (Acknowledgment of the firmware): Return value "OK"

2. Poll for "card in field."  
   Return value "DEVICE_EVENT" or "OK"  
   Transfer buffer contains card data.

3. Accept card data which were read.

4. If necessary, read additional data on the same card with SREAD and/or write with SWRITE.
B) No card is located in the field at the time of the read call.

1. Read job (READ)  
(Acknowledgment of the firmware):  
Return value "OK"
2. Poll for "card in field."  
Return value "NOEVENT"
3. Poll for "card in field."  
Return value "DEVICE_EVENT" or "OK"  
Transfer buffer contains card data.
4. Accept card data which were read.
5. If necessary, read additional data on the same card with SREAD and/or write with SWRITE.

The read call is acknowledged positively (i.e., return value "OK"). Not until a card moves into the field and is read successfully is an event message issued (CTDEVICE_EVENT as device ID) and the desired data block in the read job (READ) is transferred to the specified buffer. Additional data blocks can then be read from and/or written to the same card.

---

**Note**

After a READ call, additional read or write calls cannot be made until an event message is issued. This state is exited when a card event has occurred or event control is disabled.

---

**Possible processing sequences with event control deactivated**

A) A card is located in the field at the time of the read call.

1. Read job (READ)  
(Acknowledgment of the firmware):  
Return value "OK"  
Transfer buffer contains card data.
2. Accept card data which were read

If necessary, additional data can be read from the same card with SREAD and/or written with SWRITE.

B) No card is located in the field at the time of the read call.

1. Read job (READ)  
(Acknowledgment of the firmware):  
Return value "NO_CARD"

If two or more cards are located in the field, the read job is rejected with the error message "DOUBLE_CARD".
**Function header:**

```
int CCT_READ_BLOCK(byte hComNum, uint nAddress, byte *cData, uint nOption)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cComNum</td>
<td>byte</td>
<td>Number of the COM interface</td>
</tr>
<tr>
<td>nAddress</td>
<td>uint</td>
<td>0 to 63 = Block number</td>
</tr>
<tr>
<td>cData</td>
<td>* byte</td>
<td>Pointer to a read buffer from which the read data can then be fetched</td>
</tr>
<tr>
<td>nOption</td>
<td>uint</td>
<td>Type of read</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0    = READ command</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1    = SREAD command</td>
</tr>
</tbody>
</table>

**Return value:**

```
Return value ≥ 0      // Return value of the logical device
OK                      // No error
WRONG_REQ                // Wrong request identifier
WRONG_PARAM              // Wrong parameter value range
DEVICE_NOT_OPEN          // Device not open
DEVICE_UNKNOWN           // Device unknown
NO_CARD                  // No card
DEVICE_BUSY              // Device busy
DOUBLE_CARD              // More than one card
DEVICE_ERROR             // Device error
AUTHENT_ERR              // Authentification error
WAIT_FOR_EVENT           // Wait for an event
```

Return value < 0        // Error status of EC-COM- For meaning, see chapter 4.

**Example:**

```
uint com_n = 1;       // COM1
uint block_nAdress = 34; // Start block
byte R_cData[16];     // 16 bytes to read:Bytes #34 to #49
...
if (event_mode == 0)  // Event mode off
{   Read_Status = CCT_READ_BLOCK (com_n, block_nAdress, R_cData, 0);
   if (Read_Status == 0) // MIFARE_OK = 0
       // EC-COM failure < 0
{.."READ-Command successful, go ahead"..}
}
```
4.3.4 Read Card Data - CCT_READ_N_BLOCK Function

The CCT_READ_N_BLOCK() function reads a number of bytes from one or more blocks of user data or a sector trailer. See chapter 3.2.1.

The MIFARE\textsuperscript{®} read/write device offers two methods of read-access.

- "Unspecified" read command (READ)
- "Specified" read command (SREAD).

READ determines the series number of a card in the field and reads the desired data. The series number of the card is only stored intermediately and is not sent to the application program in the host computer.

SREAD uses the series number of the last read or write job to read the desired data. The series number stored intermediately is also sent to the card. Using the same series number for consecutive card accesses ensures that the same card is always addressed. The series number is not sent to the application program in the host computer.

Both methods of read-access are specified by the "nOption" parameter in the function call.

The read job can be performed in one of two operating modes.

- Event-controlled
- Polling mode (i.e., not event-controlled)

Possible correct processing sequences with event control activated (See chapter 4.3.19.)

A) A card is located in the field at the time of the read call.

1. Read job (READ) (Acknowledgment of the firmware): Return value "OK"
2. Poll for "card in field" Return value "DEVICE_EVENT" or "OK" Transfer buffer contains card data.
3. Accept card data which were read
4. If necessary, read additional data from the same card with SREAD and/or write with SWRITE.

B) No card is located in the field at the time of the read call.

1. Read job (READ) (Acknowledgment of the firmware): Return value "OK"
2. Poll for "card in field" Return value "NOEVENT"
3. Poll for "card in field" Return value "DEVICE_EVENT" or "OK" Transfer buffer contains card data.
4. Accept card data which were read
5. If necessary, read additional data from the same card with SREAD and/or write with SWRITE.
Possible correct processing sequences with event control deactivated (See chapter 4.3.20.)

A) A card is located in the field at the time of the read call.
   1. Read job (READ)
      (Acknowledgment of the firmware): Return value “OK”
      Transfer buffer contains card data.
   2. Accept card data which were read
   3. If necessary, read additional data from the same card with SREAD and/or write with SWRITE.

B) No card is located in the field at the time of the read call.
   1. Read job (READ)
      (Acknowledgment of the firmware): Return value “NO_CARD”

See chapter 3.2.4 for a description of behavior when two or more cards are located in the field.
Function header:
int **CCT_READ_N_BLOCK**(byte cComNum, byte *tr_data, int nBlock_adr, int nlen, int nOption)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cComNum</td>
<td>byte</td>
<td>Number of the COM interface</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = COM1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 = COM2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>..</td>
</tr>
<tr>
<td>tr_data</td>
<td>* byte</td>
<td>Pointer to a read buffer from which the read data can then be fetched</td>
</tr>
<tr>
<td>nBlock_adr</td>
<td>uint</td>
<td>0 to 63 = Block number</td>
</tr>
<tr>
<td>nlen</td>
<td>uint</td>
<td>Amount of data to be read (1 to max. of 192 bytes)</td>
</tr>
<tr>
<td>nOption</td>
<td>uint</td>
<td>Type of read</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = READ command</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = SREAD command</td>
</tr>
</tbody>
</table>

Return value:

Return value ≥ 0  // Return value of the logical device

OK  // No error
WRONG_REQ  // Wrong request identifier
WRONG_PARAM  // Wrong parameter value range
DEVICE_NOT_OPEN  // Device not open
DEVICE_UNKNOWN  // Device unknown
NO_CARD  // No card in the field
DEVICE_BUSY  // Device busy
DOUBLE_CARD  // More than one card
DEVICE_ERROR  // Device error
CARD_NOT_ACCESSIBLE  // Card no longer in the field
AUTHENT_ERR  // Authentification error
WAIT_FOR_EVENT  // Wait for an event

Return value < 0  // Error status of EC-COM. For meaning, see chapter 4.
4.3.5 Write Card Data - CCT_WRITE_BLOCK Function

The CCT_WRITE_BLOCK() function writes a block of data on a card. An entire data block must be written.

Similar to the read command, there are two methods of write-access.

- "Unspecified" write command (WRITE)
- "Specified" write command (SWRITE)

WRITE determines the series number of a card in the field and writes the desired data block. The series number of the card is not sent to the application program in the host computer.

SWRITE uses the series number of the last read or write job to write the desired data block. The series number stored intermediately is also sent to the card. Using the same series number for consecutive card accesses ensures that the same card is always addressed. The series number is not sent to the application program in the host computer.

The two methods of write-access are specified by the "nOption" parameter in the function call.

In contrast to the read procedure, the write procedure is not event-controlled. A card must always be located in the field during the write procedure. Verified read-access (SREAD) should be used to ensure that the card actually contains the written data.

Function header:

```c
int CCT_WRITE_BLOCK(byte cComNum, uint nAddress, byte *cData, uint nOption)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cComNum</td>
<td>byte</td>
<td>Number of the COM interface</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = COM1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 = COM2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>..</td>
</tr>
<tr>
<td>nAddress</td>
<td>uint</td>
<td>1 to 62 = Block number with the exception of all sector trailers</td>
</tr>
<tr>
<td>cData</td>
<td>* byte</td>
<td>Pointer to a write buffer in which the data to be sent are stored</td>
</tr>
<tr>
<td>nOption</td>
<td>uint</td>
<td>Type of write</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = WRITE command</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = SWRITE command</td>
</tr>
</tbody>
</table>
Return value:

Return value $> 0$ // Return value of the logical device
  OK // No error
  WRONG_REQ // Wrong request identifier
  WRONG_PARAM // Wrong parameter value range
  DEVICE_NOT_OPEN // Device not open
  DEVICE_UNKNOWN // Device unknown
  NO_CARD // No card
  DEVICE_BUSY // Device busy
  DOUBLE_CARD // More than one card
  DEVICE_ERROR // Device error
  AUTHENT_ERR // Authentification error
  WAIT_FOR_EVENT // Wait for an event

Return value $< 0$ // Error status of EC-COM. For meaning, see chapter 4.

Example:

uint com_n = 1; // COM1
uint block_nAddress = 34; // Start block

Write_Status = CCT_WRITE_BLOCK (com_n, block_nAddress, w_cData, 0);
.."something failed within the CCT_WRITE-Command"..
{
  switch (Write_Status)
  {case 0: output_err.message ("OK"); break;
  case 2: output_err.message ("WRONG_REQ"); break;
  case 3: output_err.message ("WRONG_PARAM"); break;
  case 4: output_err.message ("DEVICE_NOT_OPEN"); break;
  case 6: output_err.message ("DEVICE_UNKNOWN"); break;
  case 7: output_err.message ("DEVICE_ERROR"); break;
  case 8: output_err.message ("DEVICE_BUSY"); break;
  case 9: output_err.message ("DOUBLE_CARD"); break;
  case 10: output_err.message ("NO_CARD"); break;
  default: output_err.message ("COM-Failure")
  }
}
4.3.6 Write Card Data - CCT_WRITE_N_BLOCK Function

The CCT_WRITE_N_BLOCK() function writes data in one or more user data blocks. Data lengths of 1 byte to 192 bytes are permitted (i.e., a maximum of 12 consecutive blocks of user data). The start address of the data to be written is specified by a block number from 1 to 62 (i.e., writing can only start at the beginning of a block). The block number is generated consecutively over all sectors and not generated separately for each sector as described in chapter 3.1. Sector trailers cannot be used as the start address. When data lengths greater than 16 bytes are involved (i.e., more than one user data block) and one or more sector trailers are located between the first user data block to be written and the last, the sector trailer or sector trailers are skipped. When the required key for the user data block to be written is not set and/or access is not permitted, the call is rejected with an error message and no data are accepted.

Similar to the read command, there are two methods of write-access.

- "Unspecified" write command (WRITE)
- "Specified" write command (WRITE)

WRITE determines the series number of a card in the field and writes the desired data. The series number of the card is not sent to the application program in the host computer.

SWRITE uses the series number of the last read or write job to write the desired data. The series number stored intermediately is also sent to the card. Using the same series number for consecutive card accesses ensures that the same card is always addressed. The series number is not sent to the application program in the host computer.

The two methods of write-access are specified by the "nOption" parameter in the function call.

In contrast to the read procedure, the write procedure is not event-controlled. A card must always be located in the field during the write procedure. Verified read-access (SREAD) should be used to ensure that the card actually contains the written data. See chapter 4.3.4.

See chapter 3.2.4 for a description of behavior when two or more cards are located in the field.
Function header:
int **CCT_WRITE_N_BLOCK**(byte cComNum, byte *tr_data, int nBlock_adr, int nlen, int nOption)

Parameter | Type | Description
--- | --- | ---
cComNum | byte | Number of the COM interface
1 | COM1
2 | COM2
.. | ..
tr_data | * byte | Pointer to the write buffer in which the data to be sent are stored
nBlock_adr | uint | 1 to 62 = Block number with the exception of all sector trailers
nlen | uint | Amount of data to be written (1 byte to 192 bytes)
nOption | uint | Type of write
0 | WRITE command
1 | SWRITE command

Return value:
Return value > 0 // Return value of the logical device
OK // No error
WRONG_REQ // Wrong request identifier
WRONG_PARAM // Wrong parameter value range
DEVICE_NOT_OPEN // Device not open
DEVICE_UNKNOWN // Device unknown
NO_CARD // No card in the field
DEVICE_BUSY // Device busy
DOUBLE_CARD // More than one card
DEVICE_ERROR // Device error
CARD_NOT_ACCESSIBLE // Card no longer in the field
AUTHENT_ERR // Authentification error
WAIT_FOR_EVENT // Wait for an event
Return value < 0 // Error status of EC-COM. For meaning, see chapter 4.
4.3.7 Set Up Value Block - CCT_INIT_VALUE Function

The CCT_INIT_VALUE() function sets up a value block with the value 0. 
See chapter 3.2.4 for a description of behavior when two or more cards are located in the field.

Function header:

```c
int CCT_INIT_VALUE (byte cComNum, uint nOrigin_Adr, uint nBackup_Adr)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cComNum</td>
<td>byte</td>
<td>Number of the COM interface</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = COM1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 = COM2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>..</td>
</tr>
<tr>
<td>nOrigin_Adr</td>
<td>uint</td>
<td>Block number of the original block</td>
</tr>
<tr>
<td>nBackup_Adr</td>
<td>uint</td>
<td>Block number of the backup block</td>
</tr>
</tbody>
</table>

Return value:

- Return value > 0  // Return value of the logical device
- OK                // No error
- WRONG_REQ         // Wrong request identifier
- WRONG_PARAM       // Wrong parameter value range
- DEVICE_NOT_OPEN   // Device not open
- DEVICE_UNKNOWN    // Device unknown
- NO_CARD           // No card in the field
- DOUBLE_CARD       // More than one card
- CARD_DEFECT       // Card defective
- CARD_NOT_ACCESSIBLE // Card no longer in the field
- AUTHENT_ERR       // Authentification error
- WAIT_FOR_EVENT    // Wait for an event

Return value < 0  // Error status of EC-COM. For meaning, see chapter 4.
4.3.8 Read Value Block - CCT_GET_VALUE Function

The CCT_GET_VALUE() function reads a value block.
See chapter 3.2.4 for a description of behavior when two or more cards are located in the field.

Function header:

```c
int CCT_GET_VALUE(byte cComNum, uint nOrigin_Adr, uint nBackup_Adr,
                   unsigned long int *Valueptr)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cComNum</td>
<td>byte</td>
<td>Number of the COM interface</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = COM1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 = COM2</td>
</tr>
<tr>
<td>nOrigin_Adr</td>
<td>uint</td>
<td>Block number of the original block</td>
</tr>
<tr>
<td>nBackup_Adr</td>
<td>uint</td>
<td>Block number of the backup block</td>
</tr>
<tr>
<td>*Valueptr</td>
<td>unsigned long int</td>
<td>Address to which the contents of the block which was read are transferred</td>
</tr>
</tbody>
</table>

Return value:

- Return value ≥ 0 // Return value of the logical device
  - OK // No error
  - WRONG_REQ // Wrong request identifier
  - WRONG_PARAM // Wrong parameter value range
  - DEVICE_NOT_OPEN // Device not open
  - DEVICE_UNKNOWN // Device unknown
  - NO_CARD // No card in the field
  - DOUBLE_CARD // More than one card
  - TRANSFER_ERROR // Fatal transfer error during value block handling
  - CARD_DEFECT // Card defective
  - CARD_NOT_ACCESSIBLE // Card no longer in the field
  - AUTHENT_ERR // Authentication error
  - WAIT_FOR_EVENT // Wait for an event

- Return value < 0 // Error status of EC-COM. For meaning, see chapter 4.
4.3.9 Add Value to Value Block - CCT_ADD_VALUE Function

The CCT_ADD_VALUE() function adds the specified value to a value block.
See chapter 3.2.4 for a description of behavior when two or more cards are located in the field.

Function header:

```c
int CCT_ADD_VALUE(byte cComNum, uint nOrigin_Adr, uint nBackup_Adr,
                   unsigned long int value)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cComNum</td>
<td>byte</td>
<td>Number of the COM interface</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = COM1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 = COM2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>..</td>
</tr>
<tr>
<td>nOrigin_Adr</td>
<td>uint</td>
<td>Block number of the original block</td>
</tr>
<tr>
<td>nBackup_Adr</td>
<td>uint</td>
<td>Block number of the backup block</td>
</tr>
<tr>
<td>value</td>
<td>unsigned long int</td>
<td>Value to be added</td>
</tr>
</tbody>
</table>

Return value:

- Return value $\geq 0$ // Return value of the logical device
  - OK // No error
  - WRONG_REQ // Wrong request identifier
  - WRONG_PARAM // Wrong parameter value range
  - DEVICE_NOT_OPEN // Device not open
  - DEVICE_UNKNOWN // Device unknown
  - NO_CARD // No card in the field
  - DOUBLE_CARD // More than one card
  - TRANSFER_ABORTED // Value block change aborted
  - TRANSFER_ERROR // Fatal transfer error during value handling
  - CARD_DEFECT // Card defective
  - VALUE_OVERFLOW // Value no longer fits in data type.
  - CARD_NOT_ACCESSIBLE // Card no longer in the field
  - AUTHENT_ERR // Authentication error
  - WAIT_FOR_EVENT // Wait for an event

- Return value $< 0$ // Error status of EC-COM. For meaning, see chapter 4.
4.3.10 Subtract Value from Value Block - CCT_SUB_VALUE Function

The CCT_SUB_VALUE() function subtracts the specified value from a value block. See chapter 3.2.4 for a description of behavior when two or more cards are located in the field.

Function header:

```c
int CCT_SUB_VALUE(byte cComNum, uint nOrigin_Adr, uint nBackup_Adr,
                   unsigned long int value)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cComNum</td>
<td>byte</td>
<td>Number of the COM interface</td>
</tr>
<tr>
<td>nOrigin_Adr</td>
<td>uint</td>
<td>Block number of the original block</td>
</tr>
<tr>
<td>nBackup_Adr</td>
<td>uint</td>
<td>Block number of the backup block</td>
</tr>
<tr>
<td>value</td>
<td>unsigned long int</td>
<td>Value to be subtracted</td>
</tr>
</tbody>
</table>

Return value:

- Return value ≥ 0  // Return value of the logical device
  - OK  // No error
  - WRONG_REQ  // Wrong request identifier
  - WRONG_PARAM  // Wrong parameter value range
  - DEVICE_NOT_OPEN  // Device not open
  - DEVICE_UNKNOWN  // Device unknown
  - NO_CARD  // No card in the field
  - DOUBLE_CARD  // More than one card
  - TRANSFER_ABORTED  // Value block change aborted
  - TRANSFER_ERROR  // Fatal transfer error during value handling
  - CARD_DEFECT  // Card defective
  - VALUE_OVERFLOW  // Value no longer fits in data type.
  - CARD_NOT_ACCESSIBLE  // Card no longer in the field
  - AUTHENT_ERR  // Authentification error
  - WAIT_FOR_EVENT  // Wait for an event

- Return value < 0  // Error status of EC-COM. For meaning, see chapter 4.
4.3.11 Select Card - CCT_SELECT_CARD Function

The CCT_SELECT_CARD() function selects a card based on its index. This card can then be accessed even though several cards are located in the read field. A valid index can be obtained from the DOUBLE_CARD response or the CCT_GET_FIELD_STATE call.

Function header:

```c
int CCT_SELECT_CARD(byte cComNum, byte cCard_Index);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cComNum</td>
<td>byte</td>
<td>Number of the COM interface</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = COM1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 = COM2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>..</td>
</tr>
<tr>
<td>cCard_Index</td>
<td>byte</td>
<td>Index: 0 to 7</td>
</tr>
</tbody>
</table>

Return value:

- `Return value > 0` // Return value of the logical device
- `OK` // No error
- `WRONG_REQ` // Wrong request identifier
- `WRONG_PARAM` // Wrong parameter value range
- `DEVICE_NOT_OPEN` // Device not open
- `DEVICE_UNKNOWN` // Device unknown
- `NO_CARD` // No card in the field
- `CARD_ALREADY_SELECTED` // This card or another card is already selected.
- `WAIT_FOR_EVENT` // Wait for an event
- `Return value < 0` // Error status of EC-COM. For meaning, see chapter 4.
4.3.12 Release Card - CCT_RELEASE_CARD Function

The CCT_RELEASE_CARD() function releases a selected card. This card or any other card can then be selected again.

Function header:

```c
int CCT_RELEASE_CARD(byte cComNum, byte cCard_Index);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cComNum</td>
<td>byte</td>
<td>Number of the COM interface</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = COM1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 = COM2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>..</td>
</tr>
<tr>
<td>cCard_Index</td>
<td>byte</td>
<td>Index: 0 to 7</td>
</tr>
</tbody>
</table>

Return value:

- Return value > 0 // Return value of the logical device
  - OK // No error
  - WRONG_REQ // Wrong request identifier
  - WRONG_PARAM // Wrong parameter value range
  - DEVICE_NOT_OPEN // Device not open
  - DEVICE_UNKNOWN // Device unknown
  - NO_CARD // No card in the field
  - CARD_NOT_SELECTED // Attempt to release a card which is not selected
  - CARD_NOT_ACCESSIBLE // Card no longer in the field
  - WAIT_FOR_EVENT // Wait for an event
- Return value < 0 // Error status of EC-COM. For meaning, see chapter 4.
4.3.13 Read Digital Inputs - CCT_READ_SIGNAL Function

The CCT_READ_SIGNAL() function reads all available digital inputs.

Function header:

```c
int CCT_READ_SIGNAL (byte cComNum, byte* cInput, int nSignal_id)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cComNum</td>
<td>byte</td>
<td>Number of the COM interface</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = COM1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 = COM2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>..</td>
</tr>
<tr>
<td>cInput</td>
<td>*byte</td>
<td>Pointer to a read buffer from which the read data can then be fetched</td>
</tr>
<tr>
<td>nSignal_Id</td>
<td>int</td>
<td>Logical address of the input port = 0</td>
</tr>
</tbody>
</table>

Return value:

- Return value ≥ 0 // Return value of the logical device
  - OK // No error
  - WRONG_REQ // Wrong request identifier
  - WRONG_PARAM // Wrong parameter value range
  - DEVICE_NOT_OPEN // Device not open
  - DEVICE_UNKNOWN // Device unknown
  - WAIT_FOR_EVENT // Wait for an event

- Return value < 0 // Error status of EC-COM. For meaning, see chapter 4.

After successful conclusion (i.e., OK), one byte of the specified buffer (i.e., cBuffer) contains the signal states of the digital inputs. Signal state of the inputs: Bit = 1 \(\rightarrow\) HIGH level.

<table>
<thead>
<tr>
<th>Bit</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DI0</td>
</tr>
<tr>
<td>1</td>
<td>DI1</td>
</tr>
<tr>
<td>2</td>
<td>DI2</td>
</tr>
<tr>
<td>3</td>
<td>DI3</td>
</tr>
<tr>
<td>4</td>
<td>Not used</td>
</tr>
<tr>
<td>5</td>
<td>Not used</td>
</tr>
<tr>
<td>6</td>
<td>Not used</td>
</tr>
<tr>
<td>7</td>
<td>Not used</td>
</tr>
</tbody>
</table>
4.3.14 Address Digital Output - CCT_WRITE_SIGNAL Function

The CCT_WRITE_SIGNAL() function addresses the available digital outputs. The digital outputs are addressed by a signal number. The signal status is specified by the signal duration. If an unused output is addressed, the call is acknowledged with the return value “OK” and no further action is taken.

Function header:

```c
int CCT_WRITE_SIGNAL(byte cComNum, uint nSignal_id, uint nDuration)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cComNum</td>
<td>byte</td>
<td>Number of the COM interface</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = COM1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 = COM2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>..</td>
</tr>
<tr>
<td>nSignal_id</td>
<td>uint</td>
<td>Signal number (meaning for the MIFARE® demonstrator kit/development kit):</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = Buzzer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = Not used</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 = Green LED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 = Red LED</td>
</tr>
<tr>
<td>nDuration</td>
<td>uint</td>
<td>Signal duration:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0xFFFF (hex) = Continuous signal on</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0xnnnn (hex) = Duration t1 = Value · 20 msec</td>
</tr>
<tr>
<td></td>
<td></td>
<td>≥ 0001 (hex) value &lt; 8000 (hex)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Pulsing signal in scan ratio 1:1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Pulse period t2 = (Value - 8000 (hex) · 40 msec)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>≥ 8000 (hex) value &lt; FFFE (hex)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x0000 (hex) = Continuous or pulsing signal off</td>
</tr>
</tbody>
</table>

Return value:

```c
Return value ≥ 0 // Return value of the logical device
OK // No error
WRONG_REQ // Wrong request identifier
WRONG_PARAM // Wrong parameter value range
DEVICE_NOT_OPEN // Device not open
DEVICE_UNKNOWN // Device unknown
WAIT_FOR_EVENT // Wait for an event
```

Return value < 0 // Error status of EC-COM. For meaning, see chapter 4.

Example:

```c
byte com_n = 1; // COM1
uint signal = 2; // Green LED
uint dure = 50; // Signal duration: 1 sec
...
D_WRITE_Status = CCT_WRITE_SIGNAL (com_n, signal, dure);
...```
4.3.15 Parameterize Card Read/Write Interface - CCT_SETKEY Function

The CCT_SETKEY() function is a control command with which the operating behavior of the card read/write device can be set. It sets key sets and the A/B key mode of the MIFARE® read/write device addressed.

Note

Key specifications depend on the individual application. They must be discussed and agreed to by the system operator and the card manufacturer. Since the card keys for a system are permanently integrated in the firmware for security reasons, these must be generated individually for the specific system.

Function header:

int CCT_SETKEY (byte cComNum, byte res[4], byte key_set, byte Key-AB)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cComNum</td>
<td>byte</td>
<td>Number of the COM interface</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = COM1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 = COM2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>..</td>
</tr>
<tr>
<td>res [0..3]</td>
<td>byte</td>
<td>Reserved parameters. Must be set to 0.</td>
</tr>
<tr>
<td>key_set</td>
<td>byte</td>
<td>Key set = 0</td>
</tr>
<tr>
<td>key_AB</td>
<td>byte</td>
<td>Key A or B ∈ {0,1}</td>
</tr>
</tbody>
</table>

Return value:

Return value > 0 // Return value of the logical device

OK // No error

WRONG_REQ // Wrong request identifier

WRONG_PARAM // Wrong parameter value range

DEVICE_NOT_OPEN // Device not open

DEVICE_UNKNOWN // Device unknown

WAIT_FOR_EVENT // Wait for an event

Return value < 0 // Error status of EC-COM. For meaning, see chapter 4.

Example:

```
#define key_a 0

uint com_n = 1;
byte key-set = 0;
...
set_status = CCT_SETKEY(com_n, key-set, key_a)
...
```
4.3.16 Change Temporary Key Set - CCT_SET_KEY_TEMP Function

There are three keys for each sector of a MIFARE® card.
- Core module key
- Key A
- Key B

The core module key is permanently stored in the firmware of the MIFARE® read/write unit and cannot be changed.

The A and B keys for each sector can be temporarily or permanently changed in the firmware of the MIFARE® read/write unit. See chapter 3.1. Temporary means that the changed A and B keys will be lost when the power is turned off. Permanent means that the key settings will be also retained after return of power.

The "parameterize card read/write interface" function (CCT_SETKEY) can be used to set a set of keys and the type of key. This setting is binding for all subsequent read and write operations of the logical MIF device.

The CCT_SET_KEY_TEMP() function loads the A or B key of a sector in a volatile memory. The A and B keys each contain 6 bytes and must be transferred without encryption.

Function header:

```c
int CCT_SET_KEY_TEMP (byte cComNum, set_key_temp_struct *sSetKeyTemp)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cComNum</td>
<td>byte</td>
<td>Number of the COM interface</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = COM1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 = COM2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>..</td>
</tr>
<tr>
<td>sSetKeyTemp</td>
<td>* set_key_temp_struct</td>
<td>Structure</td>
</tr>
<tr>
<td></td>
<td></td>
<td>byte key_id,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// 0 = Key A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// 1 = Key B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>byte sector_id,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// 0 to 15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>byte key_old[6],</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// Old key</td>
</tr>
<tr>
<td></td>
<td></td>
<td>byte key_new[6]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// New key</td>
</tr>
</tbody>
</table>

Return value:

<table>
<thead>
<tr>
<th>Return value ≥ 0</th>
<th>// Return value of the logical device</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>// No error</td>
</tr>
<tr>
<td>WRONG_REQ</td>
<td>// Wrong request identifier</td>
</tr>
<tr>
<td>WRONG_PARAM</td>
<td>// Wrong parameter value range</td>
</tr>
<tr>
<td>DEVICE_NOT_OPEN</td>
<td>// Device not open</td>
</tr>
<tr>
<td>DEVICE_UNKNOWN</td>
<td>// Device unknown</td>
</tr>
<tr>
<td>WRONG_KEY</td>
<td>// Old key wrong</td>
</tr>
<tr>
<td>WAIT_FOR_EVENT</td>
<td>// Wait for an event</td>
</tr>
</tbody>
</table>

Return value < 0

// Error status of EC-COM. For meaning, see chapter 4.
4.3.17 Set Temporary Keys Permanently - CCT_SET_KEY_PERM Function

The CCT_SET_KEY_TEMP() function sets all previously temporary keys permanently. The temporary keys are stored in non-volatile memory.

Function header:

```c
int CCT_SET_KEY_PERM (byte cComNum)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cComNum</td>
<td>byte</td>
<td>Number of the COM interface</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = COM1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 = COM2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>..</td>
</tr>
</tbody>
</table>

Return value:

- Return value ≥ 0 // Return value of the logical device
- OK // No error
- WRONG_REQ // Wrong request identifier
- WRONG_PARAM // Wrong parameter value range
- DEVICE_NOT_OPEN // Device not open
- DEVICE_UNKNOWN // Device unknown
- WAIT_FOR_EVENT // Wait for an event

Return value < 0 // Error status of EC-COM. For meaning, see chapter 4.
4.3.18 Poll Current Card Indices - CCT_GET_FIELD_STATE Function

The CCT_GET_FIELD_STATE() function returns the currently specified indices of the cards located in the field. The field is searched before for existing cards.

Function header:
int CCT_GET_FIELD_STATE(byte cComNum, byte *cBuffer)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cComNum</td>
<td>byte</td>
<td>Number of the COM interface</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = COM1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 = COM2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>..</td>
</tr>
<tr>
<td>cBuffer</td>
<td>* byte</td>
<td>Buffer in which the number of n existing cards and n existing indices is to be stored</td>
</tr>
</tbody>
</table>

Return value:
- Return value ≥ 0       // Return value of the logical device
  - OK                   // No error
  - WRONG_REQ            // Wrong request identifier
  - WRONG_PARAM          // Wrong parameter value range
  - DEVICE_NOT_OPEN      // Device not open
  - DEVICE_UNKNOWN       // Device unknown
  - DEVICE_ERROR         // Device error
  - WAIT_FOR_EVENT       // Wait for an event

- Return value < 0       // Error status of EC-COM. For meaning, see chapter 4.

Example:
After successful conclusion (i.e., "OK"), the specified buffer (i.e., cBuffer) contains the following byte values.
- n: Number of cards in the field (0 to 8)
- n: Indices (0 to 7)
4.3.19 Activate Event Control - CCT_ENABLE Function

The CCT_ENABLE() function changes the mode of the read request (READ) of the addressed reader. When no card is located in the antenna field during a READ command, the READ command is acknowledged with "OK." As soon as a card moves into the antenna field, the requested data are transferred to the receiving buffer. The CCT_GET_EVENT command can be used to determine whether a card has been read. All library functions are disabled while waiting for an event. The wait procedure can only be canceled by the CCT_FREE_EVENT and CCT_RESET commands.

Function header:
int CCT_ENABLE(byte cComNum)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cComNum</td>
<td>byte</td>
<td>Number of the COM interface</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = COM1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 = COM2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>..</td>
</tr>
</tbody>
</table>

Return value:

Return value ≥ 0 // Return value of the logical device
OK // No error
WRONG_REQ // Wrong request identifier
WRONG_PARAM // Wrong parameter value range
DEVICE_NOT_OPEN // Device not open
DEVICE_UNKNOWN // Device unknown
MIF_IS_ENABLED // Logical MIF device is already activated.
WAIT_FOR_EVENT // Wait for an event

Return value < 0 // Error status of EC-COM. For meaning, see chapter 4.

Example:
uint com_n = 1; //COM1

uint set_status;
...
set_status = CCT_ENABLE (com_n);
4.3.20 Deactivate Event Control - CCT_DISABLE Function

The CCT_DISABLE() function deactivates event control mode. If a read job was issued before, a CCT_FREE_EVENT or CCT_FREE_EVENT2 function must be called first. Otherwise this read job will be rejected with an error message.

Function header:
int CCT_DISABLE (byte cComNum)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cComNum</td>
<td>byte</td>
<td>Number of the COM interface</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = COM1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 = COM2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>..</td>
</tr>
</tbody>
</table>

Return value:

Return value ≥ 0  // Return value of the logical device

OK                   // No error
WRONG_REQ            // Wrong request identifier
WRONG_PARAM          // Wrong parameter value range
DEVICE_NOT_OPEN      // Device not open
DEVICE_UNKNOWN       // Device unknown
MIF_IS_NOT_ENABLED   // Logical MIF device is not activated.
WAIT_FOR_EVENT       // Wait for an event

Return value < 0  // Error status of EC-COM. For meaning, see chapter 4.

Example:

```c
uint com_n = 1;       //COM1
uint set_status;
..                     
set_status = CCT_DISABLE (com_n);
```
4.3.21 Switch Off Antenna Field - CCT_OFF Function

The CCT_OFF function turns off the antenna field of the addressed read/write device.

Function header:

```c
int CCT_OFF (byte cComNum)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cComNum</td>
<td>byte</td>
<td>Number of the COM interface</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = COM1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 = COM2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>..</td>
</tr>
</tbody>
</table>

Return value:

- Return value ≥ 0 // Return value of the logical device
  - OK // No error
  - WRONG_REQ // Wrong request identifier
  - WRONG_PARAM // Wrong parameter value range
  - DEVICE_NOT_OPEN // Device not open
  - DEVICE_UNKNOWN // Device unknown
  - WAIT_FOR_EVENT // Wait for an event
- Return value < 0 // Error status of EC-COM. For meaning, see chapter 4.

Example:

```c
uint com_n = 1; // COM1
...
set_status = CCT_OFF(com_n);
...```

4.3.22 Set Field Operating Mode of the Antenna Field - CCT_MODE Function

The CCT_MODE() function specifies the field operating mode of the antenna field. One of two operating modes can be selected.

- Cyclic operating mode
- Continuous operating mode

Cyclic operating mode is the standard operating mode after the device is turned on.

Switch on and switch off cycles:

- Switch on cycle: 220 to 240 msec
- Switch off cycle: 840 to 860 msec

When a read/write call occurs during off time, the response is delayed until the card may be processed. This increases the time spent by the card in the field proportionally.

During continuous operation mode, the antenna field is switched on for card accesses and remains on until the read/write call has been concluded.

Function header:

```c
int CCT_MODE (byte cComNum, byte cMode)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cComNum</td>
<td>byte</td>
<td>Number of the COM interface</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = COM1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 = COM2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>..</td>
</tr>
<tr>
<td>cMode</td>
<td>byte</td>
<td>Mode:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = Cyclic operating mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = Continuous operating mode</td>
</tr>
</tbody>
</table>

Return value:

- Return value ≥ 0 // Return value of the logical device
  - OK // No error
  - WRONG_REQ // Wrong request identifier
  - WRONG_PARAM // Wrong parameter value range
  - DEVICE_NOT_OPEN // Device not open
  - DEVICE_UNKNOWN // Device unknown
  - WAIT_FOR_EVENT // Wait for an event

- Return value < 0 // Error status of EC-COM. For meaning, see chapter 4.
Example:

```c
#define cyclic_operation 0
#define continuous_operation 1

uint com_n = 1;  // COM1
...
set_status = CCT_MODE(com_n, continuous_operation);
...
```

---

**Warning**

When the design of the MIFARE™ read/write device does not take into account the effects of the induction field of the antenna, the induction field of the antenna can seriously affect the functionality of medical implants in the human body (e.g., pace makers) if the person comes too close to this field. For this reason, adherence to the following regulations is mandatory when setting the operating modes of the antenna field.

- **Cyclic operating mode**
  
  Cyclic operating mode permits persons with pace makers to approach the antenna surface of the device without maintaining a safety distance.

- **Continuous operating mode**
  
  Continuous operating mode may only be set when it is ensured that persons with pace makers cannot come closer to the antenna surface of the device than the maximum safety distance permitted.

  This safety interval depends on the output power set and the effective surface of the antenna. When standard settings and standard antennas are used (i.e., antenna surface of 15 cm x 15 cm), safety distance is approximately 15 cm vertical to the antenna surface. It decreases with reduced output power and smaller antennas. A lesser safety distance must also be maintained on the sides of the antenna surface.

  The safety distance can be ensured by providing suitable protective guards or device design adapted to the requirements.

  Further information on the design of read/write devices and required safety distances is available in the data sheet of your device or from your local sales office.
4.3.23 Reset Firmware - CCT_RESET Function

The CCT_RESET() function sends a reset identifier to the card read/write device. This causes the firmware to be reset. The logical MIF, DOP or CTL device (i.e., the entire device) must not have been opened with CCT_OPEN.

Function header:

```c
int CCT_RESET (byte cComNum)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cComNum</td>
<td>byte</td>
<td>Number of the COM interface</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = COM1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 = COM2</td>
</tr>
</tbody>
</table>

Return value:

- Return value > 0 // Return value of the logical device
- FW_INIT // FW initialized
- WRONG_REQ // Wrong request identifier
- WRONG_PARAM // Wrong parameter value range
- DEVICE_UNKNOWN // Device unknown
- Return value < 0 // Error status of EC-COM. For meaning, see chapter 4.

Note

In the MIFARE® demonstrator kit/development kit, an acknowledgment is only provided when an error occurs. No acknowledgment is provided when the command was executed successfully.

The reset identifier can be polled with the "poll error status" function CCT_GET_ERR_STATE. See chapter 4.2.3.
4.3.24 Poll Firmware Version - CCT_GET_FW_STATE Function

The CCT_GET_FW_STATE function returns the kernel and application version of the firmware. Information is also provided on whether a hardware reset was performed. Before the function is executed, the logical device "CTL" must have been opened.

The kernel version provides information on the existing version of the interface driver and firmware loader. The application version provides information on the existing version of the reloadable firmware application.

Function header:

```c
int CCT_FW_STATE (byte cComNum, uint cObject_id, StructStateFrame* sUserState)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cComNum</td>
<td>byte</td>
<td>Number of the COM interface</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = COM1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 = COM2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>..</td>
</tr>
<tr>
<td>cObject_id</td>
<td>uint</td>
<td>0</td>
</tr>
<tr>
<td>sUserState</td>
<td>*StructStateFrame</td>
<td>Pointer to a predefined structure (header definition)</td>
</tr>
</tbody>
</table>

Return value:

- `Return value ≥ 0` // Return value of the logical device
- `OK` // No error
- `WRONG_REQ` // Wrong request identifier
- `WRONG_PARAM` // Wrong parameter value range
- `DEVICE_NOT_OPEN` // Device not open
- `DEVICE_UNKNOWN` // Device unknown
- `DEVICE_ERROR` // Device error
- `WAIT_FOR_EVENT` // Wait for an event
- `Return value < 0` // Error status of EC-COM. For meaning, see chapter 4.

After successful conclusion (i.e., "OK"), the specified sUserState structure contains the following values.

- **Reset ID (byte)**
  - 0 = Hardware reset
  - 255 = No reset
- **Kernel version (uint)**
  - Ex: 100 = Version 1.00
- **Application version (uint)**
  - 0 = No application exists.
  - Ex: 100 = Version 1.00
5 EC-COM Error States

An error of the EC-COM driver can be transferred as the return value for the functions of the CCTDAPI library. The error states of EC-COM version 1.1 are listed below and explained. Before the constants listed in the table can be used by an application, the header file ECCOMD.H must be included. See chapter 2. The error numbers are indicated in hexadecimal format.

Remark: A change in EC-COM version may also change the error states.

The error states may indicate more than one error (e.g., "no acknowledgment or negative acknowledgment" (0x8002) and "timeout during connection establishment" (0x8020)). The error states for which several errors may occur are OR-linked and output as an error number (e.g., 0x8002 and 0x8020 for 0x8022).

Table of the EC-COM errors:

<table>
<thead>
<tr>
<th>Value</th>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8000</td>
<td>COM_ST_ERROR</td>
<td>Job canceled incorrectly</td>
</tr>
<tr>
<td>0x8001</td>
<td>COM_ST_2MANY</td>
<td>Job rejected (too many jobs)</td>
</tr>
<tr>
<td>0x8002</td>
<td>COM_ST_NO_CON</td>
<td>nth unsuccessful attempt to establish connection, no acknowledgment or negative acknowledgment</td>
</tr>
<tr>
<td>0x8004</td>
<td>COM_ST_NO_TRA</td>
<td>nth unsuccessful attempt to transfer telegram</td>
</tr>
<tr>
<td>0x8008</td>
<td>COM_ST_2SMALL</td>
<td>Receiving buffer too small during receipt</td>
</tr>
<tr>
<td>0x8010</td>
<td>COM_ST_BCCERR</td>
<td>Block check error</td>
</tr>
<tr>
<td>0x8020</td>
<td>COM_ST_TIMCON</td>
<td>Timeout during connection establishment</td>
</tr>
<tr>
<td>0x8040</td>
<td>COM_ST_TIMTRA</td>
<td>Timeout during data transfer</td>
</tr>
<tr>
<td>0x8080</td>
<td>COM_ST_TIMQUI</td>
<td>Timeout during acknowledgment</td>
</tr>
<tr>
<td>0x8100</td>
<td>COM_ST_SCC_BR</td>
<td>Break recognized</td>
</tr>
<tr>
<td>0x8200</td>
<td>COM_ST_SCC_PY</td>
<td>Parity error</td>
</tr>
<tr>
<td>0x8400</td>
<td>COM_ST_SCC_FR</td>
<td>Framing error</td>
</tr>
<tr>
<td>0x8800</td>
<td>COM_ST_SCC_OR</td>
<td>Overrun</td>
</tr>
<tr>
<td>0x9000</td>
<td>COM_ST_SNDRCV</td>
<td>Duplex conflict</td>
</tr>
<tr>
<td>0xC000</td>
<td>COM_ST_SYSERR</td>
<td>Fatal system error =&gt; Call hotline.</td>
</tr>
</tbody>
</table>

The error state "job rejected (too many jobs)" (0x8001) always occurs when a new job is requested before the old one has been concluded (e.g., a ComWrite() command is to be executed before the last write job has been concluded).
6 MIFARE® Read/Write Device Application

The CCTDAPILx.LIB import library and the CCTDAPI.H and ECCOMD.H header files are required to create a user application. The CCTDAPI.H header file must be specified as "#include" file in each C module with function call(s) from the CCTDAPILx.LIB import library. The CCTDAPI.H header file contains definitions for constants, data and functions. The ECCOMD.H header file must be specified as "#include" file for each C module for polling of the error states of the EC-COM driver with the constants listed in chapter 4.

A demonstrator software package is available for getting starting quickly with the creation of a MIFARE® read/write device application.

- Runnable sample application for a preliminary test of the MIFARE® read/write device after the EC-COM 3964R driver has been installed
- The source code of this sample application is also provided so that it can be adjusted to an individual application.
### 6.1 Executable Sample Application

Sample application xmpl_dos.exe is an executable program for a preliminary test of the MIFARE® read/write device after the EC-COM 3964R driver has been installed. The functions of the sample application represent only a selection of the many library functions available.

### 6.1.1 Installing the EC-COM Driver

Entries in the CONFIG.SYS and AUTOEXEC.BAT files are required for the linking and communication of the EC-COM 3964R driver.

Additions to the CONFIG.SYS file:

```
STACKS=20,128
DEVICE=[path]ECCOM.SYS [/1] [/2]
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>path</td>
<td>Path name under which the driver software is to be copied</td>
</tr>
<tr>
<td></td>
<td>=&gt; C:\MIFARE\</td>
</tr>
<tr>
<td>/1</td>
<td>COM1 interface for EC-COM</td>
</tr>
<tr>
<td></td>
<td>=&gt; Device name COM1</td>
</tr>
<tr>
<td>/2</td>
<td>COM2 interface for EC-COM</td>
</tr>
<tr>
<td></td>
<td>=&gt; Device name COM2</td>
</tr>
</tbody>
</table>

The parameters shown in square brackets [..] are optional and can be omitted.

The addition to the AUTOEXEC.BAT file is described in chapter 6.3.2.

The driver is then automatically loaded when a system restart is performed. If no parameters are specified, the driver will be loaded but no COM interface installed (i.e., the driver cannot be addressed).

The EC-COM driver software consisting of

- ECCOM.SYS  EC-COM 3964R driver
- ECPAR.EXE  Parameterization program for the interface parameters of EC-COM
- ECCOM.PAR  Parameterization file for the COM interface and the 3964R protocol

must then be copied in the directory (path) C:\MIFARE\.

The ECCOM.PAR parameterization file contains default values. It can be viewed and some of the values changed with the ECPAR.EXE parameterization program.

---

**Note**

The entry DEVICE=ANSI.SYS must be present for ECPAR.EXE in the CONFIG.SYS file.
6.1.2 Parameterizing the EC-COM Driver

The ECPAR parameterization program can be used to select the COM interface, set the EC-COM communication driver, and store this information in a so-called parameter file. COM1 or COM2 can be selected as the COM interface.

The parameters are divided into interface parameters and protocol parameters. The interface parameters specify the character framework for the transmission to the MIFARE® read/write device. The following interface parameters have been preset for the EC-COM driver and the MIFARE® read/write device in the included ECCOM.PAR file. These preset values may not be changed.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default Setting</th>
<th>EC-COM and MIFARE® Read/Write Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transmission speed</td>
<td>9600 bits/sec</td>
<td>9600 bits/sec</td>
</tr>
<tr>
<td>Bits per character</td>
<td>8 data bits</td>
<td>8 data bits</td>
</tr>
<tr>
<td>Parity</td>
<td>Odd</td>
<td>Odd</td>
</tr>
<tr>
<td>Stop bits</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The procedure parameters specify the individual characteristics of the 3964R protocol used. As with the interface parameters in the ECCOM.PAR file, these parameters have been preset. Some of them can be changed.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default Setting</th>
<th>EC-COM Value Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transmission mode</td>
<td>Transparent</td>
<td>Set permanently</td>
</tr>
<tr>
<td>Behavior during initiation conflict</td>
<td>Master</td>
<td>Set permanently</td>
</tr>
<tr>
<td>Block length for blocking</td>
<td>No blocking</td>
<td>Set permanently</td>
</tr>
<tr>
<td>Maximum telegram length</td>
<td>256 bytes</td>
<td>Set permanently</td>
</tr>
<tr>
<td>Acknowledgment delay time, q</td>
<td>5 x 100 msec</td>
<td>5 x 100 msec</td>
</tr>
<tr>
<td>Character monitoring time, z</td>
<td>2 x 100 msec</td>
<td>2 x 100 msec</td>
</tr>
<tr>
<td>Max. no. of sends during connection establishment, count c</td>
<td>4</td>
<td>1 to n1</td>
</tr>
<tr>
<td>Max. no. of sends during data transfer, count t</td>
<td>4</td>
<td>1 to n2</td>
</tr>
</tbody>
</table>

The repetition counters c and t should always have the value 4.

The parameter settings must have been transferred to the EC-COM driver before it is addressed for the first time. The parameters can be loaded via command to the EC-COM driver or transferred automatically to the EC-COM driver when the PC boots. Automatic transfer requires that the following entry be added to the AUTOEXEC.BAT file.

```
[path]ECPAR -l [path]ECCOM.PAR
```

path Path name under which the parameterization program and the parameter file are stored
Example: C:\MIFARE\
ECPAR Parameterization program
ECCOM.PAR Parameter file

The parameters shown in square brackets [..] are optional and can be omitted.
6.1.3 Installing the Sample Application

After installation of the EC-COM driver software, the xmlpl_dos.exe sample application must be loaded under the directory (usually C:\MIFARE) from which the sample application is to be started on the PC.

6.1.4 Using the Sample Application

After the xmlpl_dos.exe sample application has been started, sample application functions can be selected in the DOS window. The EC-COM driver software must have been installed and parameterized beforehand.

- Sample application functions:
  - The following functions can be called from the device-related main and control functions. See chapter 4.1.
  - Open DOP and MIF logical devices o: Open devices
  - Parameterize card read/write interface s: SETKEY
  - Read card data r: READ
  - Reset firmware R: RESET
  - Address digital output b: BUZZER
  - Close DPS and MIF logical devices c: Close devices

After a function has been selected, a message appears indicating whether the function was executed.

- Select COM interface 1 or 2 S: Set COM
- Conclude sample application q: QUIT MENU
6.2 Sample Application in Source Code

The sample application consists of the following components.
- C module (main module): xmpl_dos.c
- Header files as Include files: cctdapi.h, eccomd.h

Include files cctdapi.h and eccomd.h are called in the C module xmpl_dos.c.

The xmpl_dos.mak file controls generation of the sample application. The generation file contains the instructions for the compiler and link run with Microsoft Visual C++. The project path is specified as floppy disk driver "A:".
6.2.1 Example of a Generation File XMPL_DOS.MAK

# Microsoft Visual C++ generated build script - Do not modify

PROJ = XMPL_DOS
DEBUG = 0
PROGTYPE = 6
CALLER =
ARGS =
DLLS =
D_RCEDEFINES = -d_DEBUG
R_RCEDFINES = -DNDEBUG
ORIGIN = MSVC
ORIGIN_VER = 1.00
PROJPATH = S:\CLIB\VERSION\DOS\V200\DISK\SAMPLE\
USEMFC = 1
CC = cl
CPP = cl
CXX = cl
CCREATEPCHFLAG =
CPPCREATEPCHFLAG =
CUSEPCHFLAG =
CPPUSEPCHFLAG =
FIRSTC = XMPL_DOS.C
FIRSTCPP =
RC = rc
         /Fd"XMPL_DOS.PDB"
CFLAGS_R_DEXE = /nologo /Gs /G2 /W3 /AL /Ox /D "NDEBUG" /D "_DOS" /FR
LFLAGS_D_DEXE = /NOLOGO /ONERROR:NOEXE /NOI /CO /STACK:5120
LFLAGS_R_DEXE = /NOLOGO /ONERROR:NOEXE /NOI /STACK:5120
LIBS_D_DEXE = lafxcrd oldnames llibce
LIBS_R_DEXE = lafxcr oldnames llibce
RCFLAGS = /nologo
RESFLAGS = /nologo
RUNFLAGS =
OBJ_EXT =
LIBS_EXT = ECCOM.LIB ..\..\..\..\..\USER\MO\JOB\CCTDAPI.LIB
!if "$(DEBUG)" == "1"
CFLAGS = $(CFLAGS_D_DEXE)
LFLAGS = $(LFLAGS_D_DEXE)
LIBS = $(LIBS_D_DEXE)
MAPFILE = nul
RCDFINES = $(D_RCEDEFINES)
lsez
CFLAGS = $(CFLAGS_R_DEXE)
LFLAGS = $(LFLAGS_R_DEXE)
LIBS = $(LIBS_R_DEXE)
MAPFILE = nul
RCDFINES = $(R_RCEDFINES)
!endif
!if [if exist MSVC.BND del MSVC.BND]
!endif
SBRS = XMPL_DOS.SBR
ECCOM_DEP =
XMPL_DOS_DEP = s:\clib\user\mo\src\cctdapi.h \
    s:\clib\version\dos\v200\disk\sample\eccomd.h

CCTDAPI_DEP =

all:   $(PROJ).EXE $(PROJ).BSC

XMPL_DOS.OBJ:   XMPL_DOS.C $(XMPL_DOS_DEP)
    $(CC) $(CFLAGS) $(CCREATEPCHFLAG) /c XMPL_DOS.C

$(PROJ).EXE:: XMPL_DOS.OBJ $(OBJS_EXT) $(DEFFILE)
    echo >NUL @<<$(PROJ).CRF
    XMPL_DOS.OBJ +
    $(OBJS_EXT)
    $(PROJ).EXE
    $(MAPFILE)
    X:\MSVC15\LIB+
    X:\MSVC15\MFC\LIB+
    s:\clib\user\ho\job+
    ECCOM.LIB+
    ..\..\..\..\..\..\USER\MO\JOB\CCTDAPI.LIB+
    $(LIBS)
    $(DEFFILE);
    <<
    link $(LFLAGS) @$($(PROJ).CRF

run: $(PROJ).EXE
    $(PROJ) $(RUNFLAGS)

$(PROJ).BSC: $(SBRS)
    bscmake @<<
    /o$@ $(SBRS)
    <<
7 Appendix

7.1 Terms/Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3964R</td>
<td>Interface protocol for serial interfaces with software handshake</td>
</tr>
<tr>
<td>API</td>
<td>Application Interface</td>
</tr>
<tr>
<td>CCT</td>
<td>Chip Card Terminal</td>
</tr>
<tr>
<td>CCTDAPIM</td>
<td>MS-DOS-C library as application interface for chipcard terminal</td>
</tr>
<tr>
<td>CTL</td>
<td>Card Terminal as Logical device</td>
</tr>
<tr>
<td>Device</td>
<td>Logical device (i.e., function group of the card read/write device which can be addressed separately)</td>
</tr>
<tr>
<td>DOP</td>
<td>Digital Output Port</td>
</tr>
<tr>
<td>EC-COM</td>
<td>Communications driver for 3964R</td>
</tr>
<tr>
<td>OEM</td>
<td>Original Equipment Manufacturer</td>
</tr>
<tr>
<td>SIM</td>
<td>Serial Interface Module</td>
</tr>
</tbody>
</table>