Using a MFRC522 reader to read and write MIFARE RFID cards on ARDUINO through the MFRC522 library BY COOQROBOT.

Mario Capurso (m.capurso@libero.it)

MIFARE is the NXP Semiconductors-owned trademark of a series of chips widely used in contactless smart cards and proximity cards. According to the producers, billions of smart card chips and many millions of reader modules have been sold. [1]

The technology is owned by NXP Semiconductors (spin off from Philips Electronics in 2006) with headquarters in Eindhoven, Netherlands, and main business sites in Nijmegen, Netherlands, and Hamburg, Germany.

The MIFARE name covers proprietary technologies based upon various levels of the ISO/IEC 14443 Type A 13.56 MHz contactless smart card standard.
The MIFARE name (derived from the term MIkron FARE Collection System) covers seven different kinds of contactless cards.

MIFARE Classic employs a proprietary protocol compliant to parts (but not all) of ISO/IEC 14443-3 Type A, with an NXP proprietary security protocol for authentication and ciphering.

The MIFARE Classic card is fundamentally just a memory storage device, where the memory is divided into segments and blocks with simple security mechanisms for access control. They are ASIC-based and have limited computational power. Thanks to their reliability and low cost, those cards are widely used for electronic wallet, access control, corporate ID cards, transportation or stadium ticketing.

The MIFARE Classic 1K offers 1024 bytes of data storage, split into 16 sectors; each sector is protected by two different keys, called A and B. Each key can be programmed to allow operations such as reading, writing, increasing valueblocks, etc. MIFARE Classic 4K offers 4096 bytes split into forty sectors, of which 32 are same size as in the 1K with eight more that are quadruple size sectors. MIFARE Classic mini offers 320 bytes split into five sectors.

For each of these card types, 16 bytes per sector are reserved for the keys and access conditions and can not normally be used for user data. Also, the very first 16 bytes contain the serial number of the card and certain other manufacturer data and are read only. That brings the net storage capacity of these cards down to 752 bytes for MIFARE Classic 1k, 3440 bytes for MIFARE Classic 4k, and 224 bytes for Mini. It uses an NXP proprietary security protocol (Crypto-1) for authentication and ciphering.

The encryption used by the MIFARE Classic card uses a 48 bit key. [18]
A presentation by Henryk Plötz and Karsten Nohl[19] at the Chaos Communication Congress in December 2007 described a partial reverse-engineering of the algorithm used in the MIFARE Classic chip. Abstract and slides[20] are available online. A paper that describes the process of reverse engineering this chip was published at the August 2008 USENIX security conference.[21]

In March 2008 the Digital Security[22] research group of the Radboud University Nijmegen made public that they performed a complete reverse-engineering and were able to clone and manipulate the contents of an OV-Chipkaart which is a MIFARE Classic card.[23] For demonstration they used the Proxmark device, a 125 kHz / 13.56 MHz research instrument.[24] The schematics and software are released under the free GNU General Public License by Jonathan Westhues in 2007. They demonstrate it is even possible to perform card-only attacks using just an ordinary stock-commercial NFC reader in combination with the libnfc library.

In April 2009 new and better card-only attack on MIFARE Classic has been found. It was first announced at the Rump session of Eurocrypt 2009.[35] This attack was presented at SECRYPT 2009.[36] The full description of this latest and fastest attack to date can also be found in the IACR preprint archive.[37] The new attack improves by a factor of more than 10 all previous card-only attacks on MIFARE Classic, has instant running time, and it does not require a costly precomputation.

The new attack allows to recover the secret key of any sector of MIFARE Classic card via wireless interaction, within about 300 queries to the card. It can then be combined with the nested authentication attack in the Nijmegen Oakland paper to recover subsequent keys almost instantly. Both attacks combined and with the right hardware equipment such as Proxmark3, one should be able to clone any MIFARE Classic card in not more than 10 seconds. This is much faster than previously thought.

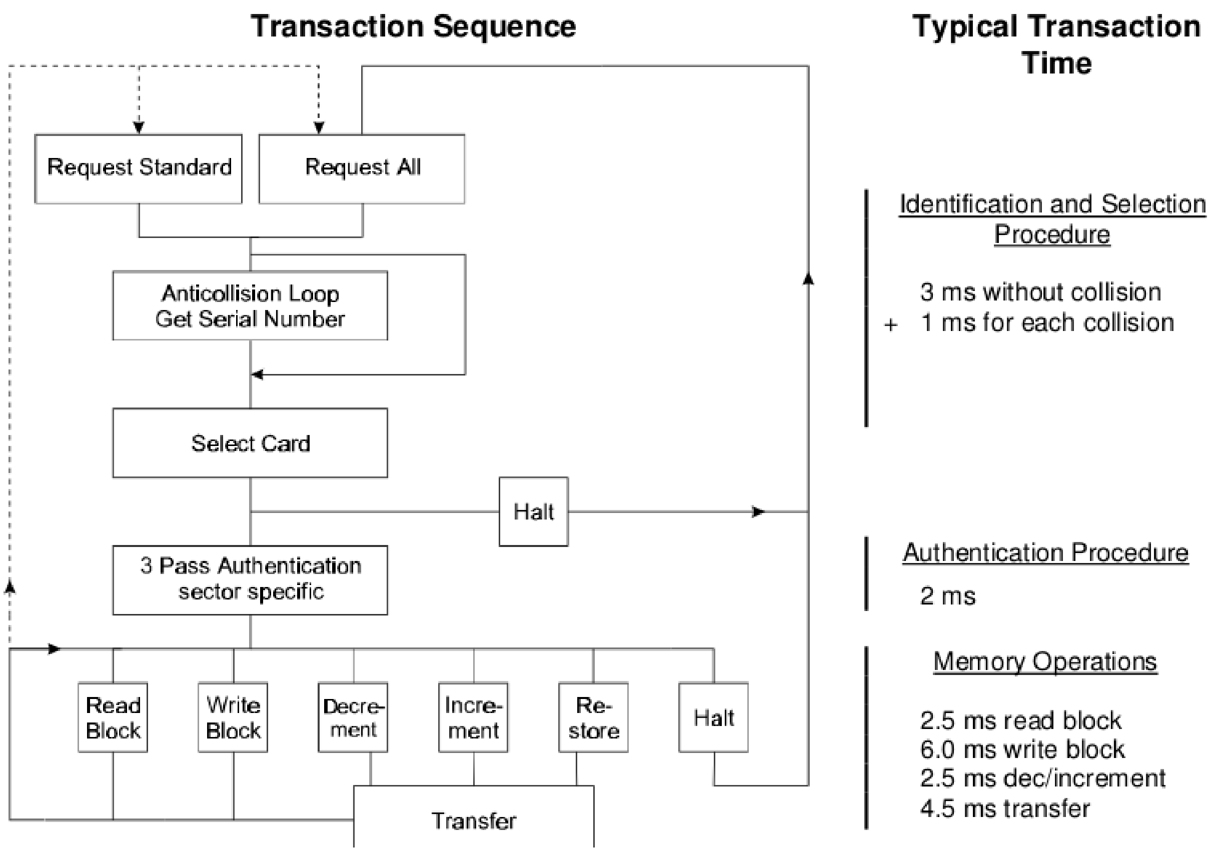
The MFRC522 is a highly integrated reader/writer IC for contactless communication at 13.56 MHz. The MFRC522 reader supports ISO/IEC 14443 A/MIFARE mode.
The MFRC522’s internal transmitter is able to drive a reader/writer antenna designed to communicate with ISO/IEC 14443 A/MIFARE cards and transponders without additional active circuitry. The receiver module provides a robust and efficient implementation for demodulating and decoding signals from ISO/IEC 14443 A/MIFARE compatible cards and transponders. The digital module manages the complete ISO/IEC 14443 A framing and error detection (parity and CRC) functionality.

The MFRC522 supports all variants of the MIFARE Mini, MIFARE 1K, MIFARE 4K, MIFARE Ultralight, MIFARE DESFire EV1 and MIFARE Plus RF identification protocols. To aid readability throughout this data sheet, the MIFARE Mini,

MIFARE 1K, MIFARE 4K, MIFARE Ultralight, MIFARE DESFire EV1 and MIFARE Plus products and protocols have the generic name MIFARE.

The following host interfaces are provided:

- Serial Peripheral Interface (SPI)
- Serial UART (similar to RS232 with voltage levels dependant on pin voltage supply)
- I2C-bus interface



Answer to Request :

With the Answer to Request sequence the MIFARE® RWD (Read Write Device) requests all MIFARE® cards in the antenna field. When a card is in the operating range of a RWD, the RWD continues communication with the appropriate protocol.

Anticollision loop
In the Anticollision loop the serial number of the card is read. If there are several cards in the operating range of a RWD they can be distinguished by their different serial numbers and one can be selected (Select card) for further transactions. The unselected cards return to the standby mode and wait for a new Answer to Request and Anticollision loop.

Select Card
With the Select Card command the RWD selects one individual card for further authentication and memory related operations. The card returns the Answer to Select (ATS) code, which determines the individual type of the selected card.

Access Specification
After identification and selection of one card the RWD specifies the memory location of the following access.

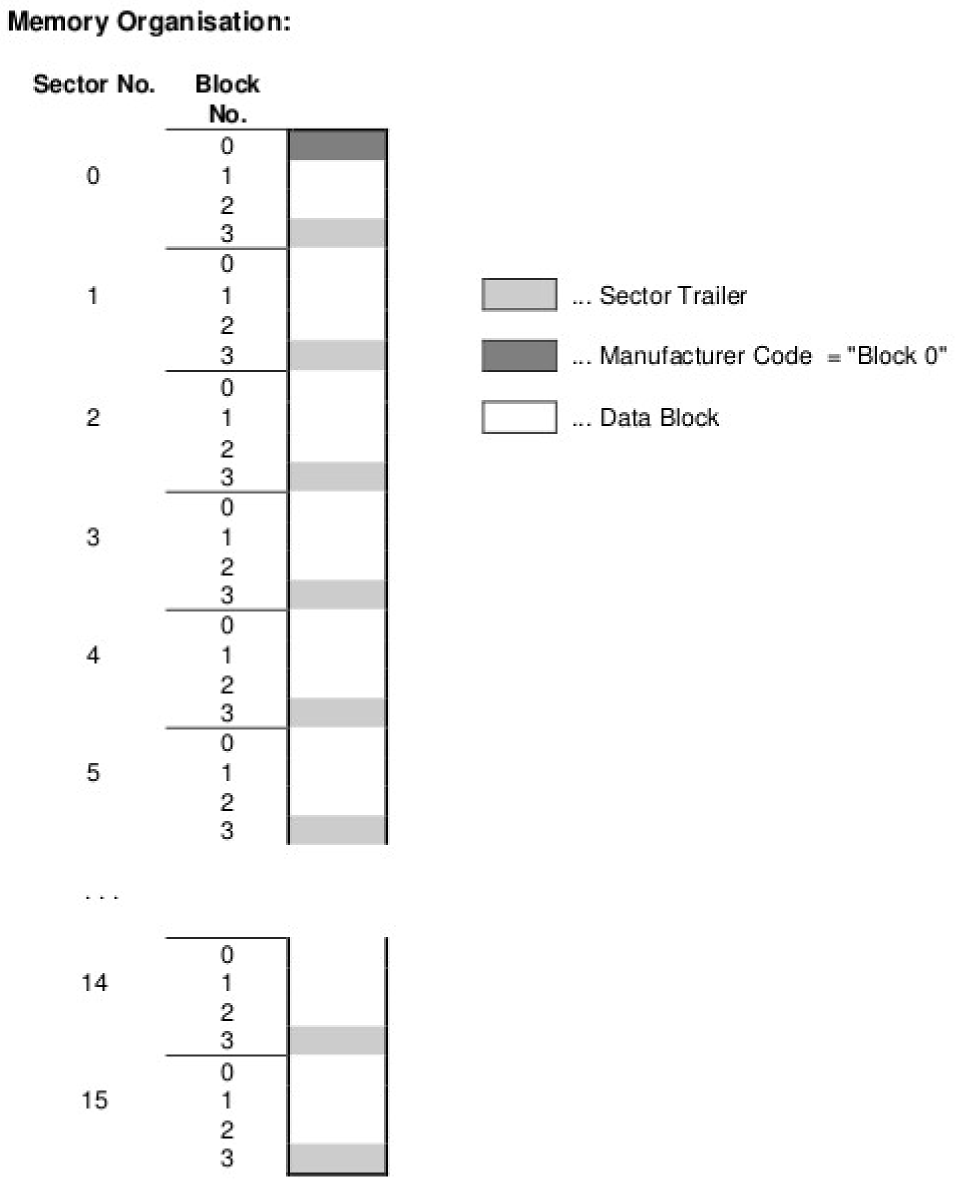
Three Pass Authentication
The appropriate access key for the previously specified access is used for 3 Pass Authentication. Any communication after authentication is automatically encrypted at the sender and decrypted by the receiver.

Read/Write

After authentication any of the following operations may be performed:

READ	reads one block
WRITE	writes one block
DECREMENT	decrements the contents of one block and stores the result in the data-register
INCREMENT	increments the contents of one block and stores the result in the data-register
TRANSFER	writes the contents of the data-register to one block
RESTORE	stores the contents of one block in the data-register

The MF1ICS50 IC of a Mifare Classic has integrated a 8192 Bit EEPROM which is split into 16 sectors with 4 blocks. One block consists of 16 bytes (1 Byte = 8 Bit).



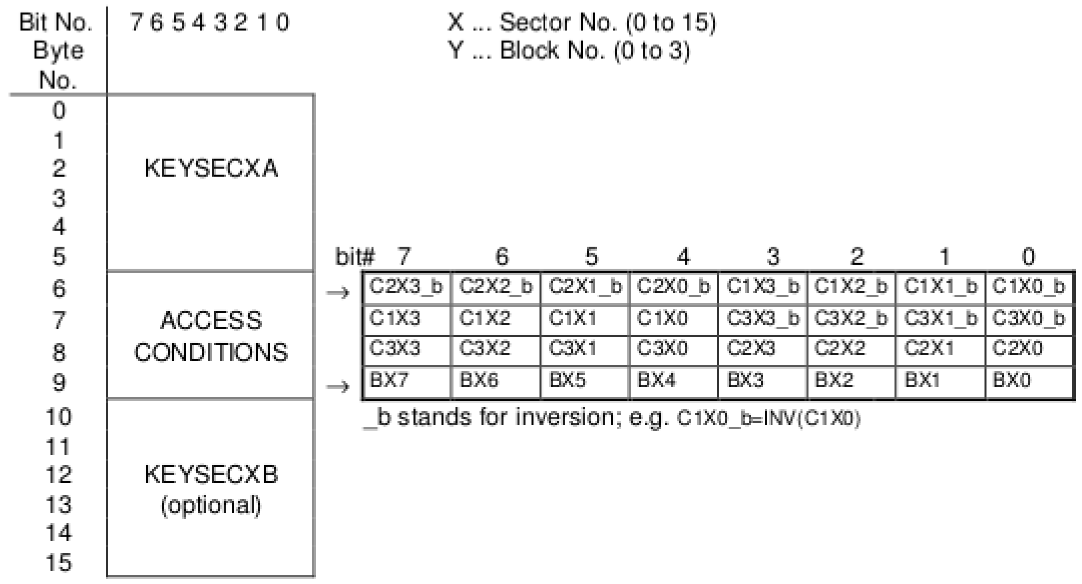
Manufacturer Code (Block 0 of Sector 0)

The first block of the memory is reserved for manufacturer data like 32 bit serial number. This is a read only block. In many documents it is named "Block 0".

Data Block (Block 0 to 3 except "Block 0")

Access conditions for the Data Blocks are defined in the Sector Trailers. According to these conditions data can be read, written, incremented, decremented, transferred or restored either with Key A, Key B or never.

2.6.1 Sector Trailer (Block 3):



bit# 7	6	5	4	3	2	1	0
C2X3_b	C2X2_b	C2X1_b	C2X0_b	C1X3_b	C1X2_b	C1X1_b	C1X0_b
C1X3	C1X2	C1X1	C1X0	C3X3_b	C3X2_b	C3X1_b	C3X0_b
C3X3	C3X2	C3X1	C3X0	C2X3	C2X2	C2X1	C2X0
BX7	BX6	BX5	BX4	BX3	BX2	BX1	BX0

_b stands for inversion; e.g. C1X0_b=INV(C1X0)

The fourth block of any sector is the Sector Trailer. The Sector Trailer contains access Key A (KEYSECXA) an optional Key B (KEYSECXB) and the access conditions for the four blocks of that sector. If Key B is not needed, the last 6 Bytes of block 3 can be used as data bytes. The corresponding access condition settings are marked grey below.
C1XY to C3XY which are stored twice for safety reasons define the access condition independently for the sector's four blocks. The last byte of the access conditions may be used to store some specific application data (e.g. location of the write backup block).

- Access condition for the Sector Trailer (Y = 3)

			KEYSECXA		ACCESS COND.		KEYSECXB	
C1X3	C2X3	C3X3	read	write	read	write	read	write
0	0	0	never	key A	key A	never	key A	key A
0	1	0	never	never	key A	never	key A	never
1	0	0	never	key B	key A|B	never	never	key B
1	1	0	never	never	key A|B	never	never	never
0	0	1	never	key A	key A	key A	key A	key A
0	1	1	never	key B	key A|B	key B	never	key B
1	0	1	never	never	key A|B	key B	never	never
1	1	1	never	never	key A|B	never	never	never

incr, decr, transfer, restore : never

NOTE: Key A|B means key A <u>or</u> key B;
If key B may be read (all grey marked lines) the memory space for Key B is used for data storage and it shall not be used for authentication because all further memory access operations will fail.

Since the transport access conditions (after chip manufacturing) equal to <u>001</u>, new cards must not be authenticated with Key B !

- Access condition for Data Blocks (Y = 0 to 2)

C1XY	C2XY	C3XY	read	write	incr	decr, transfer, restore
0	0	0	keyA|B[1]	key A|B[1]	key A|B[1]	key A|B[1]
0	1	0	keyA|B[1]	never	never	never
1	0	0	keyA|B[1]	key B[1]	never	never
1	1	0	keyA|B[1]	key B[1]	key B[1]	key A|B[1]
0	0	1	keyA|B[1]	never	never	key A|B[1]
0	1	1	key B[1]	key B[1]	never	never
1	0	1	key B[1]	never	never	never
1	1	1	never	never	never	never

The process of decrement and increment of a block's data is performed and controlled by the Card-IC.

- Transport code

For transportation, KEYSECXA and the access conditions are predefined by the manufacturer as follows:

C1X0, C2X0, C3X0 = 0 0 0	block 0 (data block)
C1X1, C2X1, C3X1 = 0 0 0	block 1 (data block)
C1X2, C2X2, C3X2 = 0 0 0	block 2 (data block)
C1X3, C2X3, C3X3 = 0 0 1	block 3 (Sector Trailer)

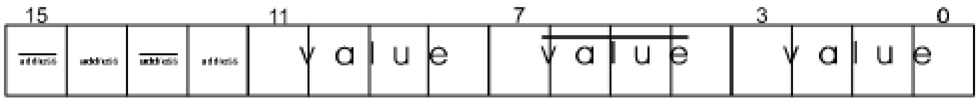
KEYSECXA .secret key, known only by
the manufacturer and system integrator

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

In the MF1ICS50 IC two types of Data Blocks are used:

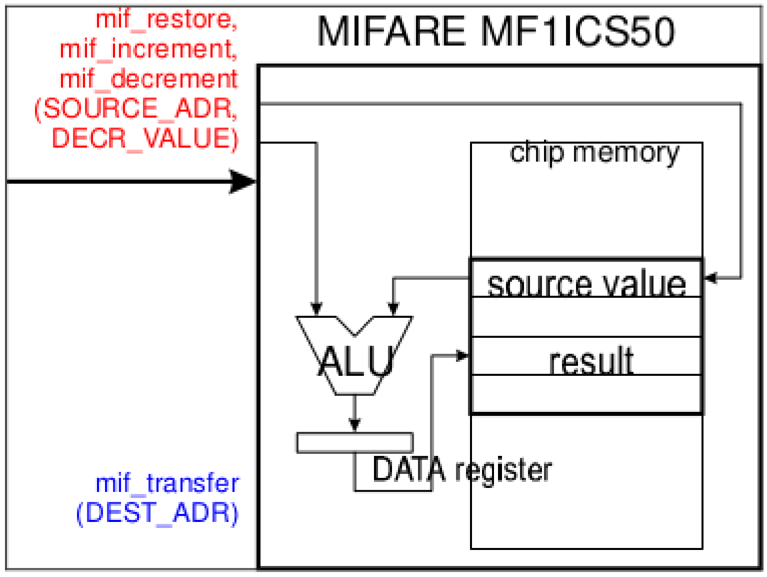
a) read/write blocks
are used to read and write general 16 bytes of data.

b) value blocks
are used for electronic purse functions (read, increment, decrement, transfer, restore). The maximum size of a value is 4 byte including sign bit, even when a complete 16 byte block has to be reserved. To provide error detection and correction capability, any value is stored 3 times into one value block. The remaining 4 bytes are reserved to some extent for check bits.



value:	32 bit signed 2th complement format stored 3 times (the consistency of the 3 occurrences of the value is internally checked before the chip can perform any calculation)
address:	8 bit arbitrary address byte stored 4 times (this byte is not internally interpreted)

A value blocks is first time generated by a WRITE instruction to the desired address. The value may be used for subsequent DECREMENT / INCREMENT / RESTORE instructions.



The result of a calculation instruction is temporally stored in a buffer register. For updating the memory with the calculation result the TRANSFER instruction has to be issued. The chip refuses calculations if any error in the block format could be detected.

The described memory organization makes it possible to appoint different sectors to different applications and to prevent data corruption by using application specific secret keys. Keys can only be altered by a RWD which has stored the actual Key A or Key B if this is allowed according to access conditions. Otherwise the actual key cannot be changed anymore.

Before the execution of a command the correct format of the Access Conditions is checked by the Card-IC. Thus, when programming the Sector Trailer the card needs to be fixed within the operating range of a RWD's antenna to prevent interruption of the write operation because any unsuccessful write operation may lead to blocking the whole sector.

2.7 Memory contents after IC test

2.7.1 Block 0 (manufacturer block):

byte byte

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Serial number				CB	manufacturer data										

CB: “serial number check byte“ CB = byte 0 ^ byte 1 ^ byte 2 ^ byte 3 (^ ... XOR)

2.7.2 Data Blocks:

Data blocks: contain variable data.
(blocks 1,2 / 4,5,6 / 8,9,10 / 12,13,14 / 16,17,18 / 20,21,22 / 24,25,26 / 28,29,30 / 32,33,34 / 36,37,38 / 40,41,42 / 44,45,46 / 48,49,50 / 52,53,54 / 56,57,58 / 60,61,62)

2.7.3 Sector Trailers:

Note: The initial state of sector trailers after IC test can be modified depending on the personalisation done e.g. at the card manufacturer.

default coding:
byte byte

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
transport key A						FF	07	80	xx	transport key B					

(blocks 3 / 7 / 11 / 15 / 19 / 23 / 27 / 31 / 35 / 39 / 43 / 47 / 51 / 55 / 59 / 63)
Byte 9 of all sector trailers is not defined. Its memory contents after IC test can vary.

MFRC522.h – A Library to use ARDUINO RFID MODULE KIT 13.56 MHZ BY COOQROBOT.

There are three hardware components involved:

1) The micro controller: An Arduino
2) The PCD (Proximity Coupling Device): NXP MFRC522 Contactless Reader IC
3) The PICC (short for Proximity Integrated Circuit Card): A card or tag using the ISO 14443A interface, eg Mifare or NTAG203.

MIFARE Classic 1K (MF1S503x):

Has 16 sectors * 4 blocks/sector * 16 bytes/block = 1024 bytes. The blocks are numbered 0-63.

```
 Block 3 in each sector is the Sector Trailer.
 *                      Bytes 0-5:   Key A
 *                      Bytes 6-8:   Access Bits
 *                      Bytes 9:     User data
 *                      Bytes 10-15: Key B (or user data)
```

Block 0 is read only manufacturer data.

To access a block, an authentication using a key from the block's sector must be performed first.

Example: To read from block 10, first authenticate using a key from sector 3 (blocks 8-11).

All keys are set to FFFFFFFFFFFFh at chip delivery.

Warning: Please read section 8.7 "Memory Access". It includes this text: if the PICC detects a format violation the whole sector is irreversibly blocked.

To use a block in "value block" mode (for Increment/Decrement operations) you need to change the sector trailer. Use PICC_SetAccessBits() to calculate the bit patterns.

Commands sent to the PICC.

The commands used by the PCD to manage communication with several PICCs (ISO 14443-3, Type A, section 6.4)

PICC_CMD_REQA = 0x26,
REQuest command, Type A. Invites PICCs in state IDLE to go to READY and prepare for anticollision or selection. 7 bit frame.
PICC_CMD_WUPA = 0x52,
Wake-UP command, Type A. Invites PICCs in state IDLE and HALT to go to READY(*) and prepare for anticollision or selection. 7 bit frame.
PICC_CMD_CT = 0x88,
Cascade Tag. Not really a command, but used during anti collision.
PICC_CMD_SEL_CL1 = 0x93,
Anti collision/Select, Cascade Level 1
PICC_CMD_SEL_CL2 = 0x95,
Anti collision/Select, Cascade Level 2
PICC_CMD_SEL_CL3 = 0x97,
Anti collision/Select, Cascade Level 3
PICC_CMD_HLTA = 0x50,
HaLT command, Type A. Instructs an ACTIVE PICC to go to state HALT.

The commands used for MIFARE Classic
Use PCD_MFAuthent to authenticate access to a sector, then use these commands to read/write/modify the blocks on the sector.
The read/write commands can also be used for MIFARE Ultralight.

```
PICC_CMD_MF_AUTH_KEY_A  = 0x60,
Perform authentication with Key A
PICC_CMD_MF_AUTH_KEY_B  = 0x61,
Perform authentication with Key B
PICC_CMD_MF_READ        = 0x30,
Reads one 16 byte block from the authenticated sector of the PICC. Also used for MIFARE Ultralight.
PICC_CMD_MF_WRITE       = 0xA0,
Writes one 16 byte block to the authenticated sector of the PICC. Called "COMPATIBILITY WRITE" for MIFARE Ultralight.
PICC_CMD_MF_DECREMENT   = 0xC0,
Decrements the contents of a block and stores the result in the internal data register.
PICC_CMD_MF_INCREMENT   = 0xC1,
Increments the contents of a block and stores the result in the internal data register.
PICC_CMD_MF_RESTORE           = 0xC2,
Reads the contents of a block into the internal data register.
PICC_CMD_MF_TRANSFER    = 0xB0,
Writes the contents of the internal data register to a block.
```

List of the functions in the library

Functions for setting up the Arduino

```
MFRC522(byte chipSelectPin, byte resetPowerDownPin);
void setSPIConfig();
```

Basic interface functions for communicating with the MFRC522

```
void PCD_WriteRegister(byte reg, byte value);
void PCD_WriteRegister(byte reg, byte count, byte *values);
byte PCD_ReadRegister(byte reg);
void PCD_ReadRegister(byte reg, byte count, byte *values, byte rxAlign = 0);
void setBitMask(unsigned char reg, unsigned char mask);
void PCD_SetRegisterBitMask(byte reg, byte mask);
void PCD_ClearRegisterBitMask(byte reg, byte mask);
byte PCD_CalculateCRC(byte *data, byte length, byte *result);
```

Functions for manipulating the MFRC522

```
void PCD_Init();
void PCD_Reset();
void PCD_AntennaOn();
```

Functions for communicating with PICCs

```
byte PCD_TransceiveData(byte *sendData, byte sendLen, byte *backData, byte *backLen, byte *validBits = NULL, byte rxAlign = 0, bool checkCRC = false);


byte PCD_CommunicateWithPICC(byte command, byte waitIRq, byte *sendData, byte
sendLen, byte *backData = NULL, byte *backLen = NULL, byte *validBits = NULL, byte
rxAlign = 0, bool checkCRC = false);

byte PICC_RequestA(byte *bufferATQA, byte *bufferSize);
byte PICC_WakeupA(byte *bufferATQA, byte *bufferSize);
byte PICC_REQA_or_WUPA( byte command, byte *bufferATQA, byte *bufferSize);
byte PICC_Select(Uid *uid, byte validBits = 0);
byte PICC_HaltA();
```

Functions for communicating with MIFARE PICCs

```
byte PCD_Authenticate(byte command, byte blockAddr, MIFARE_Key *key, Uid *uid);
void PCD_StopCrypto1();
byte MIFARE_Read(byte blockAddr, byte *buffer, byte *bufferSize);
byte MIFARE_Write(byte blockAddr, byte *buffer, byte bufferSize);
byte MIFARE_Decrement(byte blockAddr, long delta);
byte MIFARE_Increment(byte blockAddr, long delta);
byte MIFARE_Restore(byte blockAddr);
byte MIFARE_Transfer(byte blockAddr);
byte MIFARE_Ultralight_Write(byte page, byte *buffer, byte bufferSize);
```

Support functions

```
byte PCD_MIFARE_Transceive(   byte *sendData, byte sendLen, bool acceptTimeout =
false);
const char *GetStatusCodeName(byte code);
byte PICC_GetType(byte sak);
const char *PICC_GetTypeName(byte type);
void PICC_DumpToSerial(Uid *uid);
void PICC_DumpMifareClassicToSerial(Uid *uid, byte piccType, MIFARE_Key *key);
void PICC_DumpMifareClassicSectorToSerial(Uid *uid, MIFARE_Key *key, byte sector);
void PICC_DumpMifareUltralightToSerial();
void MIFARE_SetAccessBits(byte *accessBitBuffer, byte g0, byte g1, byte g2, byte g3);
```

Convenience functions - does not add extra functionality

```
bool PICC_IsNewCardPresent();
bool PICC_ReadCardSerial();
```

Detailed documentation – enum and structures

PICC types we can detect. Remember to update PICC_GetTypeName() if you add more.

```
enum PICC_Type
```

Return codes from the functions in this class. Remember to update GetStatusCodeName() if you add more.

```
enum StatusCode
```

A struct used for passing the UID of a PICC.

```
typedef struct {
      byte        size;       Number of bytes in the UID. 4, 7 or 10.
      byte        uidByte[10];
      byte        sak;        The SAK (Select acknowledge) byte returned
                              from the PICC after successful selection.
      } Uid
```

A struct used for passing a MIFARE Crypto1 key

```
typedef struct {
      byte        keyByte[MF_KEY_SIZE];
      } MIFARE_Key;
```

Example:

```
// Prepare key - all keys are set to FFFFFFFFFFFFh at chip delivery from the factory.
MFRC522::MIFARE_Key key;
for (byte i = 0; i < 6; i++) key.keyByte[i] = 0xFF;
```

Member variables

Uid uid;	Used by PICC_ReadCardSerial().

Detailed documentation – functions

+Create object instance

```
MFRC522(
byte chipSelectPin,     Arduino pin used for SPI chip select
byte resetPowerDownPin  Arduino pin used for SPI reset
);
```

Example:

```
#include <SPI.h>
#include <MFRC522.h>
#define SS_PIN 10    //Arduino Uno
#define RST_PIN 9
MFRC522 mfrc522(SS_PIN, RST_PIN);        // Create MFRC522 instance.
```

+Initializes the MFRC522 chip.

`void MFRC522::PCD_Init()`

Example:

```
void setup() {
        Serial.begin(9600);        // Init serial communications with PC
        SPI.begin();               // Init SPI bus
        mfrc522.PCD_Init();        // Init MFRC522 card
}
```

+Set SPI bus to work with MFRC522 chip.

Please call this function if you have changed the SPI config since the MFRC522 constructor was run.

`void MFRC522::setSPIConfig()`

+Performs a soft reset on the MFRC522 chip and waits for it to be ready again.

`void MFRC522::PCD_Reset()`

+Turns the antenna on by enabling pins TX1 and TX2.
After a reset these pins are disabled.

`void MFRC522::PCD_AntennaOn()`

+Transmits a REQuest command, Type A. Invites PICCs in state IDLE to go to READY and prepare for anticollision or selection. 7 bit frame.
Beware: When two PICCs are in the field at the same time I often get STATUS_TIMEOUT - probably due do bad antenna design.
return STATUS_OK on success, STATUS_??? otherwise.

```
byte MFRC522::PICC_RequestA(
byte *bufferATQA, The buffer to store the ATQA (Answer to request) in
byte *bufferSize  Buffer size, at least two bytes. Also number of bytes returned if
STATUS_OK.
```

+Transmits a Wake-UP command, Type A. Invites PICCs in state IDLE and HALT to go to READY(*) and prepare for anticollision or selection. 7 bit frame.
Beware: When two PICCs are in the field at the same time I often get STATUS_TIMEOUT - probably due do bad antenna design.
return STATUS_OK on success, STATUS_??? otherwise.

```
byte MFRC522::PICC_WakeupA(
byte *bufferATQA, The buffer to store the ATQA (Answer to request) in
byte *bufferSize  Buffer size, at least two bytes. Also number of bytes returned if
STATUS_OK.
```

+Transmits SELECT/ANTICOLLISION commands to select a single PICC.
Before calling this function the PICCs must be placed in the READY(*) state by calling PICC_RequestA() or PICC_WakeupA().
On success:

-The chosen PICC is in state ACTIVE(*) and all other PICCs have returned to state IDLE/HALT. (Figure 7 of the ISO/IEC 14443-3 draft.)
-The UID size and value of the chosen PICC is returned in *uid along with the SAK.

A PICC UID consists of 4, 7 or 10 bytes.
Only 4 bytes can be specified in a SELECT command, so for the longer UIDs two or three iterations are used:

UID size	Number of UID bytes	Cascade levels	Example of PICC
single	4	1	MIFARE Classic
double	7	2	MIFARE Ultralight
triple	10	3	Not currently in use?

return STATUS_OK on success, STATUS_??? otherwise.

```
byte MFRC522::PICC_Select(
Uid *uid,                Pointer to Uid struct. Normally output, but can also
                         be used to supply a known UID.

byte validBits           The number of known UID bits supplied in *uid.

                         Normally 0. If set you must also supply uid->size.
```

Description of buffer structure:

```
Byte 0: SEL             Indicates the Cascade Level: PICC_CMD_SEL_CL1,
PICC_CMD_SEL_CL2 or PICC_CMD_SEL_CL3
Byte 1: NVB             Number of Valid Bits (in complete command, not just the UID):
High nibble: complete bytes, Low nibble: Extra bits.
Byte 2: UID-data or CT  See explanation below. CT means Cascade Tag.
Byte 3: UID-data
Byte 4: UID-data
Byte 5: UID-data
Byte 6: BCC             Block Check Character - XOR of bytes 2-5
Byte 7: CRC_A
Byte 8: CRC_A
```

The BCC and CRC_A is only transmitted if we know all the UID bits of the current Cascade Level.
Description of bytes 2-5: (Section 6.5.4 of the ISO/IEC 14443-3 draft: UID contents and cascade levels)

UID size	Cascade level	Byte2	Byte3	Byte4	Byte5
4 bytes	1	uid0	uid1	uid2	uid3

```
7 bytes           1           CT    uid0  uid1  uid2
                  2           uid3  uid4  uid5  uid6
10 bytes          1           CT    uid0  uid1  uid2
                  2           CT    uid3  uid4  uid5
```

+Instructs a PICC in state ACTIVE(*) to go to state HALT.
return STATUS_OK on success, STATUS_??? otherwise.

```
byte PICC_HaltA();
Example:
// Halt PICC
mfrc522.PICC_HaltA();
```

+Executes the MFRC522 MFAuthent command.
This command manages MIFARE authentication to enable a secure communication to any MIFARE Mini, MIFARE 1K and MIFARE 4K card.
The authentication is described in the MFRC522 datasheet section 10.3.1.9 and http://www.nxp.com/documents/data_sheet/MF1S503x.pdf section 10.1. for use with MIFARE Classic PICCs.
The PICC must be selected - ie in state ACTIVE(*) - before calling this function.
Remember to call PCD_StopCrypto1() at the end of communication with the authenticated PICC - otherwise no new communications can start.
All keys are set to FFFFFFFFFFFFh at chip delivery.
return STATUS_OK on success, STATUS_??? otherwise. Probably STATUS_TIMEOUT if you supply the wrong key.

```
byte MFRC522::PCD_AuthenticateI(
byte command,           PICC_CMD_MF_AUTH_KEY_A or PICC_CMD_MF_AUTH_KEY_B
byte blockAddr,         The block number. See numbering in the comments in
                        the .h file.
MIFARE_Key *key,        Pointer to the Crypto1 key to use (6 bytes)
Uid *uid                Pointer to Uid struct. The first 4 bytes of the UID
                        is used.
)
Example:
MFRC522::MIFARE_Key key;
for (byte i = 0; i < 6; i++) key.keyByte[i] = 0xFF;
byte trailerBlock   = 7;
byte status;
if ( ! mfrc522.PICC_ReadCardSerial()) return;
status = mfrc522.PCD_Authenticate(MFRC522::PICC_CMD_MF_AUTH_KEY_A, trailerBlock,
&key, &(mfrc522.uid));
if (status != MFRC522::STATUS_OK) {
   Serial.print("PCD_Authenticate() failed: ");
   Serial.println(mfrc522.GetStatusCodeName(status));
   return;
}
```

+Used to exit the PCD from its authenticated state.
Remember to call PCD_StopCrypto1() at the end of communication with the authenticated PICC - otherwise no new communications can start.

```
void MFRC522::PCD_StopCrypto1()
Example:
// Stop encryption on PCD
mfrc522.PCD_StopCrypto1();
```

+Reads 16 bytes (+ 2 bytes CRC_A) from the active PICC.
For MIFARE Classic the sector containing the block must be authenticated before calling this function.
For MIFARE Ultralight only addresses 00h to 0Fh are decoded.
 * The MF0ICU1 returns a NAK for higher addresses.

```
 * The MF0ICU1 responds to the READ command by sending 16 bytes starting from
      the page address defined by the command argument.
 * For example; if blockAddr is 03h then pages 03h, 04h, 05h, 06h are returned.
 * A roll-back is implemented: If blockAddr is 0Eh, then the contents of pages 0Eh,
0Fh, 00h and 01h are returned.
The buffer must be at least 18 bytes because a CRC_A is also returned.
Checks the CRC_A before returning STATUS_OK.
return STATUS_OK on success, STATUS_??? otherwise.
byte MFRC522::MIFARE_Read(
byte blockAddr,   MIFARE Classic: The block (0-0xff) number.
                  MIFARE Ultralight: The first page to return data from.
byte *buffer,     The buffer to store the data in

byte *bufferSize  Buffer size, at least 18 bytes. Also number of bytes returned
                  if STATUS_OK.
Example:
byte valueBlockA    = 4; byte buffer[18]; byte size = sizeof(buffer);
byte status = mfrc522.MIFARE_Read(valueBlockA, buffer, &size);

+Writes 16 bytes to the active PICC.
For MIFARE Classic the sector containing the block must be authenticated before
calling this function.
For MIFARE Ultralight the operation is called "COMPATIBILITY WRITE".
Even though 16 bytes are transferred to the Ultralight PICC, only the least
significant 4 bytes (bytes 0 to 3) are written to the specified address. It is
recommended to set the remaining bytes 04h to 0Fh to all logic 0.
return STATUS_OK on success, STATUS_??? otherwise.
byte MFRC522::MIFARE_Write(
byte blockAddr,
                  MIFARE Classic: The block (0-0xff) number.
                  MIFARE Ultralight: The page (2-15) to write to.
byte *buffer,     The 16 bytes to write to the PICC
byte bufferSize   Buffer size, must be at least 16 bytes. Exactly 16 bytes are
                   written.
Example:
byte valueBlockA    = 4;
byte value1Block[] = { 1,2,3,4,  5,6,7,8, 9,10,255,12,  13,14,15,16};
status = mfrc522.MIFARE_Write(valueBlockA, value1Block, 16);
if (status != MFRC522::STATUS_OK) {
   Serial.print("MIFARE_Write() failed: ");
   Serial.println(mfrc522.GetStatusCodeName(status));
}


+MIFARE Decrement subtracts the delta from the value of the addressed block, and
stores the result in a volatile memory.
For MIFARE Classic only. The sector containing the block must be authenticated before
calling this function.
Only for blocks in "value block" mode, ie with access bits [C1 C2 C3] = [110] or
[001].
Use MIFARE_Transfer() to store the result in a block.
return STATUS_OK on success, STATUS_??? otherwise.
byte MFRC522::MIFARE_Decrement(
byte blockAddr,   The block (0-0xff) number.
long delta        This number is subtracted from the value of block blockAddr.


+MIFARE Increment adds the delta to the value of the addressed block, and stores the
result in a volatile memory.
For MIFARE Classic only. The sector containing the block must be authenticated before
calling this function.
Only for blocks in "value block" mode, ie with access bits [C1 C2 C3] = [110] or
```

```
[001].
Use MIFARE_Transfer() to store the result in a block.
return STATUS_OK on success, STATUS_??? otherwise.
byte MFRC522::MIFARE_Increment(
byte blockAddr,   The block (0-0xff) number.
long delta        This number is added to the value of block blockAddr.
Example:
// Add 1 to the value of valueBlockA and store the result in valueBlockA.
byte valueBlockA  = 5;
Serial.print("Adding 1 to value of block "); Serial.println(valueBlockA);
byte status = mfrc522.MIFARE_Increment(valueBlockA, 1);
if (status != MFRC522::STATUS_OK) {
      Serial.print("MIFARE_Increment() failed: ");
      Serial.println(mfrc522.GetStatusCodeName(status));
      return;
}
status = mfrc522.MIFARE_Transfer(valueBlockA);
if (status != MFRC522::STATUS_OK) {
      Serial.print("MIFARE_Transfer() failed: ");
      Serial.println(mfrc522.GetStatusCodeName(status));
      return;
}


+MIFARE Restore copies the value of the addressed block into a volatile memory.
For MIFARE Classic only. The sector containing the block must be authenticated before
calling this function.
Only for blocks in "value block" mode, ie with access bits [C1 C2 C3] = [110] or
[001].
Use MIFARE_Transfer() to store the result in a block.
return STATUS_OK on success, STATUS_??? otherwise.
byte MFRC522::MIFARE_Restore(
byte blockAddr    The block (0-0xff) number.
The datasheet describes Restore as a two step operation, but does not explain what
data to transfer in step 2.Doing only a single step does not work, so I chose to
transfer 0L in step two.

+Helper function for the two-step MIFARE Classic protocol operations Decrement,
Increment and Restore.
return STATUS_OK on success, STATUS_??? otherwise.
byte MFRC522::MIFARE_TwoStepHelper(
byte command,     The command to use
byte blockAddr,   The block (0-0xff) number.
long data         The data to transfer in step 2
                                                      )

+MIFARE Transfer writes the value stored in the volatile memory into one MIFARE
Classic block.
For MIFARE Classic only. The sector containing the block must be authenticated before
calling this function.
Only for blocks in "value block" mode, ie with access bits [C1 C2 C3] = [110] or
[001].
return STATUS_OK on success, STATUS_??? otherwise.
byte MFRC522::MIFARE_Transfer(
byte blockAddr    The block (0-0xff) number.


+Returns a string pointer to a status code name.
const char *MFRC522::GetStatusCodeName(
byte code   One of the StatusCode enums.

      code values             return values
```

```
        STATUS_OK:              "Success."
        STATUS_ERROR:           "Error in communication."
        STATUS_COLLISION:       "Collision detected."
        STATUS_TIMEOUT:         "Timeout in communication."
        STATUS_NO_ROOM:         "A buffer is not big enough."
        STATUS_INTERNAL_ERROR:  "Internal error in the code. Should not happen."
        STATUS_INVALID:         "Invalid argument."
        STATUS_CRC_WRONG:       "The CRC_A does not match."
        STATUS_MIFARE_NACK:     "A MIFARE PICC responded with NAK."
        default:                "Unknown error"
```

Example:
```
Serial.println(mfrc522.GetStatusCodeName(status));
```

+Translates the SAK (Select Acknowledge) to a PICC type.
return PICC_Type
byte MFRC522::PICC_GetType(
byte sak   The SAK byte returned from PICC_Select().

sak	return value
sak & 0x04	PICC_TYPE_NOT_COMPLETE
0x09	PICC_TYPE_MIFARE_MINI
0x08	PICC_TYPE_MIFARE_1K
0x18	PICC_TYPE_MIFARE_4K
0x00	PICC_TYPE_MIFARE_UL
0x10 or 0x11	PICC_TYPE_MIFARE_PLUS
0x01	PICC_TYPE_TNP3XXX
sak & 0x20	PICC_TYPE_ISO_14443_4
sak & 0x40	PICC_TYPE_ISO_18092
else	PICC_TYPE_UNKNOWN

+Returns a string pointer to the PICC type name.
const char *MFRC522::PICC_GetTypeName(
byte piccType   One of the PICC_Type enums.

piccType	return value
PICC_TYPE_ISO_14443_4	"PICC compliant with ISO/IEC 14443-4"
PICC_TYPE_ISO_18092:	"PICC compliant with ISO/IEC 18092 (NFC)"
PICC_TYPE_MIFARE_MINI	"MIFARE Mini, 320 bytes"
PICC_TYPE_MIFARE_1K	"MIFARE 1KB"
PICC_TYPE_MIFARE_4K	"MIFARE 4KB"
PICC_TYPE_MIFARE_UL	"MIFARE Ultralight or Ultralight C"
PICC_TYPE_MIFARE_PLUS	"MIFARE Plus"
PICC_TYPE_TNP3XXX	"MIFARE TNP3XXX"
PICC_TYPE_NOT_COMPLETE	"SAK indicates UID is not complete."
PICC_TYPE_UNKNOWN	"Unknown type”

Example:
```
Serial.println(mfrc522.PICC_GetTypeName(piccType));
```

+Dumps debug info about the selected PICC to Serial.
On success the PICC is halted after dumping the data.
For MIFARE Classic the factory default key of 0xFFFFFFFFFFFF is tried.
void MFRC522::PICC_DumpToSerial(
Uid *uid   Pointer to Uid struct returned from a successful PICC_Select().
Example:
```
mfrc522.PICC_DumpToSerial(&(mfrc522.uid));
```

+Dumps memory contents of a sector of a MIFARE Classic PICC.
Uses PCD_Authenticate(), MIFARE_Read() and PCD_StopCrypto1.

```
Always uses PICC_CMD_MF_AUTH_KEY_A because only Key A can always read the sector
trailer access bits.
void MFRC522::PICC_DumpMifareClassicSectorToSerial(
Uid *uid,         Pointer to Uid struct returned from a successful PICC_Select()
MIFARE_Key *key,  Key A for the sector.
byte sector       The sector to dump, 0..39.

+Calculates the bit pattern needed for the specified access bits. In the [C1 C2 C3]
tuples C1 is MSB (=4) and C3 is LSB (=1).
void MFRC522::MIFARE_SetAccessBits(
byte *accessBitBuffer,  Pointer to byte 6, 7 and 8 in the sector trailer.
                        Bytes [0..2] will be set.

byte g0,                Access bits [C1 C2 C3] for block 0
                        (for sectors 0-31) or blocks 0-4 (for sectors 32-39)

byte g1,                Access bits C1 C2 C3] for block 1 (for sectors 0-31)
                        or blocks 5-9 (for sectors 32-39)

byte g2,                Access bits C1 C2 C3] for block 2 (for sectors 0-31)
                        or blocks 10-14 (for sectors 32-39)

byte g3                 Access bits C1 C2 C3] for the sector trailer, block 3
                        (for sectors 0-31) or block 15 (for sectors 32-39)

The access bits are stored in a peculiar fashion.
There are four groups:

g[3]  Access bits for the sector trailer, block 3 (for sectors 0-31) or block 15 (for
sectors 32-39)

g[2]  Access bits for block 2 (for sectors 0-31) or blocks 10-14 (for sectors 32-39)
g[1]  Access bits for block 1 (for sectors 0-31) or blocks 5-9 (for sectors 32-39)
g[0]  Access bits for block 0 (for sectors 0-31) or blocks 0-4 (for sectors 32-39)

Each group has access bits [C1 C2 C3]. In this code C1 is MSB and C3 is LSB.
The four CX bits are stored together in a nibble cx and an inverted nibble cx_.
Example:
// Sector trailer that defines blocks 5 and 6 as Value Blocks and enables key B.
byte trailerBuffer[] = {255,255,255,255,255,255,0,0,0,0, 255,255,255,255,255,255};
// Keep default keys.
// g1=6(i.e.110) => block 5 value block. Key B write&increment, A or B decrement.
// g2=6 => Same thing for block 6.
// g3=3 => Key B must be used to modify the Sector Trailer. Key B becomes valid.
mfrc522.MIFARE_SetAccessBits(&trailerBuffer[6], 0, 6, 6, 3);

+Check if a card is present
bool MFRC522::PICC_IsNewCardPresent()
Returns true if a PICC responds to PICC_CMD_REQA.
Only "new" cards in state IDLE are invited. Sleeping cards in state HALT are ignored.
Example:
// Look for new cards
if ( ! mfrc522.PICC_IsNewCardPresent()) return;



+Read Card Serial
bool MFRC522::PICC_ReadCardSerial()
Simple wrapper around PICC_Select.
Returns true if a UID could be read.The read UID is available in the class variable
uid.
Remember to call PICC_IsNewCardPresent(), PICC_RequestA() or PICC_WakeupA() first.
return true if one selected
```

Now a card is selected. The UID and SAK is in mfrc522.uid.
Example:
```
Serial.print("Card UID:");
for (byte i = 0; i < mfrc522.uid.size; i++) {
    Serial.print(mfrc522.uid.uidByte[i] < 0x10 ? " 0" : " ");
    Serial.print(mfrc522.uid.uidByte[i], HEX);
}
// Dump PICC type
byte piccType = mfrc522.PICC_GetType(mfrc522.uid.sak);
Serial.print("PICC type: ");
Serial.println(mfrc522.PICC_GetTypeName(piccType));
if ( piccType != MFRC522::PICC_TYPE_MIFARE_MINI
     &&  piccType != MFRC522::PICC_TYPE_MIFARE_1K
     &&        piccType != MFRC522::PICC_TYPE_MIFARE_4K) {
     Serial.println("This sample only works with MIFARE Classic cards.");
     return;
}
```

How to convert a normal block into a value block
Usually a normal block has 000 access bits, while a value block has 110 access bits
Example:
```
byte valueBlockA  = 5;
byte valueBlockB  = 6;
byte trailerBlock = 7;
// We need a sector trailer with blocks 5 and 6 as Value Blocks and enables key B.
byte trailerBuffer[] = { 255,255,255,255,255,255,
                        0,0,0,0,255,255,255,255,255,255}; // Keep default keys.
// g1=6 => block 5 as value block. Key B to write&increment, A or B for decrement.
// g2=6 => Same thing for block 6.
// g3=3 => Key B must be used to modify the Sector Trailer. Key B becomes valid.
mfrc522.MIFARE_SetAccessBits(&trailerBuffer[6], 0, 6, 6, 3);
byte status = mfrc522.MIFARE_Write(trailerBlock, trailerBuffer, 16);
if (status != MFRC522::STATUS_OK) {
      Serial.print("MIFARE_Write() failed: ");
      Serial.println(mfrc522.GetStatusCodeName(status));
      return;
}
```

How to setup a Value Block with a value set to zero
Example:
```
byte  blockAddr=5;
byte valueBlock[] = {0,0,0,0, 255,255,255,255, 0,0,0,0,
                        blockAddr,~blockAddr,blockAddr,~blockAddr };
byte status = mfrc522.MIFARE_Write(blockAddr, valueBlock, 16);
if (status != MFRC522::STATUS_OK) {
      Serial.print("MIFARE_Write() failed: ");
      Serial.println(mfrc522.GetStatusCodeName(status));
}
```