/\* Typical pin layout used:

\* -----------------------------------------------------------------------------------------

\* MFRC522 Arduino Arduino Arduino Arduino Arduino

\* Reader/PCD Uno/101 Mega Nano v3 Leonardo/Micro Pro Micro

\* Signal Pin Pin Pin Pin Pin Pin

\* -----------------------------------------------------------------------------------------

\* RST/Reset RST 9 5 D9 RESET/ICSP-5 RST

\* SPI SS SDA(SS) 10 53 D10 10 10

\* SPI MOSI MOSI 11 / ICSP-4 51 D11 ICSP-4 16

\* SPI MISO MISO 12 / ICSP-1 50 D12 ICSP-1 14

\* SPI SCK SCK 13 / ICSP-3 52 D13 ICSP-3 15

\*/

#include <EEPROM.h> // We are going to read and write PICC's UIDs from/to EEPROM

#include <SPI.h> // RC522 Module uses SPI protocol

#include <Wire.h>

/\*

Instead of a Relay you may want to use a servo. Servos can lock and unlock door locks too

Relay will be used by default

\*/

#include <Servo.h>

/\*

For visualizing whats going on hardware we need some leds and to control door lock a relay and a wipe button

(or some other hardware) Used common anode led,digitalWriting HIGH turns OFF led Mind that if you are going

to use common cathode led or just seperate leds, simply comment out #define COMMON\_ANODE,

\*/

#ifndef MFRC522\_h

#define MFRC522\_h

#include <Arduino.h>

#include <SPI.h>

// Firmware data for self-test

// Reference values based on firmware version

// Hint: if needed, you can remove unused self-test data to save flash memory

//

// Version 0.0 (0x90)

// Philips Semiconductors; Preliminary Specification Revision 2.0 - 01 August 2005; 16.1 self-test

const byte MFRC522\_firmware\_referenceV0\_0[] PROGMEM = {

0x00, 0x87, 0x98, 0x0f, 0x49, 0xFF, 0x07, 0x19,

0xBF, 0x22, 0x30, 0x49, 0x59, 0x63, 0xAD, 0xCA,

0x7F, 0xE3, 0x4E, 0x03, 0x5C, 0x4E, 0x49, 0x50,

0x47, 0x9A, 0x37, 0x61, 0xE7, 0xE2, 0xC6, 0x2E,

0x75, 0x5A, 0xED, 0x04, 0x3D, 0x02, 0x4B, 0x78,

0x32, 0xFF, 0x58, 0x3B, 0x7C, 0xE9, 0x00, 0x94,

0xB4, 0x4A, 0x59, 0x5B, 0xFD, 0xC9, 0x29, 0xDF,

0x35, 0x96, 0x98, 0x9E, 0x4F, 0x30, 0x32, 0x8D

};

// Version 1.0 (0x91)

// NXP Semiconductors; Rev. 3.8 - 17 September 2014; 16.1.1 self-test

const byte MFRC522\_firmware\_referenceV1\_0[] PROGMEM = {

0x00, 0xC6, 0x37, 0xD5, 0x32, 0xB7, 0x57, 0x5C,

0xC2, 0xD8, 0x7C, 0x4D, 0xD9, 0x70, 0xC7, 0x73,

0x10, 0xE6, 0xD2, 0xAA, 0x5E, 0xA1, 0x3E, 0x5A,

0x14, 0xAF, 0x30, 0x61, 0xC9, 0x70, 0xDB, 0x2E,

0x64, 0x22, 0x72, 0xB5, 0xBD, 0x65, 0xF4, 0xEC,

0x22, 0xBC, 0xD3, 0x72, 0x35, 0xCD, 0xAA, 0x41,

0x1F, 0xA7, 0xF3, 0x53, 0x14, 0xDE, 0x7E, 0x02,

0xD9, 0x0F, 0xB5, 0x5E, 0x25, 0x1D, 0x29, 0x79

};

// Version 2.0 (0x92)

// NXP Semiconductors; Rev. 3.8 - 17 September 2014; 16.1.1 self-test

const byte MFRC522\_firmware\_referenceV2\_0[] PROGMEM = {

0x00, 0xEB, 0x66, 0xBA, 0x57, 0xBF, 0x23, 0x95,

0xD0, 0xE3, 0x0D, 0x3D, 0x27, 0x89, 0x5C, 0xDE,

0x9D, 0x3B, 0xA7, 0x00, 0x21, 0x5B, 0x89, 0x82,

0x51, 0x3A, 0xEB, 0x02, 0x0C, 0xA5, 0x00, 0x49,

0x7C, 0x84, 0x4D, 0xB3, 0xCC, 0xD2, 0x1B, 0x81,

0x5D, 0x48, 0x76, 0xD5, 0x71, 0x61, 0x21, 0xA9,

0x86, 0x96, 0x83, 0x38, 0xCF, 0x9D, 0x5B, 0x6D,

0xDC, 0x15, 0xBA, 0x3E, 0x7D, 0x95, 0x3B, 0x2F

};

// Clone

// Fudan Semiconductor FM17522 (0x88)

const byte FM17522\_firmware\_reference[] PROGMEM = {

0x00, 0xD6, 0x78, 0x8C, 0xE2, 0xAA, 0x0C, 0x18,

0x2A, 0xB8, 0x7A, 0x7F, 0xD3, 0x6A, 0xCF, 0x0B,

0xB1, 0x37, 0x63, 0x4B, 0x69, 0xAE, 0x91, 0xC7,

0xC3, 0x97, 0xAE, 0x77, 0xF4, 0x37, 0xD7, 0x9B,

0x7C, 0xF5, 0x3C, 0x11, 0x8F, 0x15, 0xC3, 0xD7,

0xC1, 0x5B, 0x00, 0x2A, 0xD0, 0x75, 0xDE, 0x9E,

0x51, 0x64, 0xAB, 0x3E, 0xE9, 0x15, 0xB5, 0xAB,

0x56, 0x9A, 0x98, 0x82, 0x26, 0xEA, 0x2A, 0x62

};

class MFRC522 {

public:

// MFRC522 registers. Described in chapter 9 of the datasheet.

// When using SPI all addresses are shifted one bit left in the "SPI address byte" (section 8.1.2.3)

enum PCD\_Register {

// Page 0: Command and status

// 0x00 // reserved for future use

CommandReg = 0x01 << 1, // starts and stops command execution

ComIEnReg = 0x02 << 1, // enable and disable interrupt request control bits

DivIEnReg = 0x03 << 1, // enable and disable interrupt request control bits

ComIrqReg = 0x04 << 1, // interrupt request bits

DivIrqReg = 0x05 << 1, // interrupt request bits

ErrorReg = 0x06 << 1, // error bits showing the error status of the last command executed

Status1Reg = 0x07 << 1, // communication status bits

Status2Reg = 0x08 << 1, // receiver and transmitter status bits

FIFODataReg = 0x09 << 1, // input and output of 64 byte FIFO buffer

FIFOLevelReg = 0x0A << 1, // number of bytes stored in the FIFO buffer

WaterLevelReg = 0x0B << 1, // level for FIFO underflow and overflow warning

ControlReg = 0x0C << 1, // miscellaneous control registers

BitFramingReg = 0x0D << 1, // adjustments for bit-oriented frames

CollReg = 0x0E << 1, // bit position of the first bit-collision detected on the RF interface

// 0x0F // reserved for future use

// Page 1: Command

// 0x10 // reserved for future use

ModeReg = 0x11 << 1, // defines general modes for transmitting and receiving

TxModeReg = 0x12 << 1, // defines transmission data rate and framing

RxModeReg = 0x13 << 1, // defines reception data rate and framing

TxControlReg = 0x14 << 1, // controls the logical behavior of the antenna driver pins TX1 and TX2

TxASKReg = 0x15 << 1, // controls the setting of the transmission modulation

TxSelReg = 0x16 << 1, // selects the internal sources for the antenna driver

RxSelReg = 0x17 << 1, // selects internal receiver settings

RxThresholdReg = 0x18 << 1, // selects thresholds for the bit decoder

DemodReg = 0x19 << 1, // defines demodulator settings

// 0x1A // reserved for future use

// 0x1B // reserved for future use

MfTxReg = 0x1C << 1, // controls some MIFARE communication transmit parameters

MfRxReg = 0x1D << 1, // controls some MIFARE communication receive parameters

// 0x1E // reserved for future use

SerialSpeedReg = 0x1F << 1, // selects the speed of the serial UART interface

// Page 2: Configuration

// 0x20 // reserved for future use

CRCResultRegH = 0x21 << 1, // shows the MSB and LSB values of the CRC calculation

CRCResultRegL = 0x22 << 1,

// 0x23 // reserved for future use

ModWidthReg = 0x24 << 1, // controls the ModWidth setting?

// 0x25 // reserved for future use

RFCfgReg = 0x26 << 1, // configures the receiver gain

GsNReg = 0x27 << 1, // selects the conductance of the antenna driver pins TX1 and TX2 for modulation

CWGsPReg = 0x28 << 1, // defines the conductance of the p-driver output during periods of no modulation

ModGsPReg = 0x29 << 1, // defines the conductance of the p-driver output during periods of modulation

TModeReg = 0x2A << 1, // defines settings for the internal timer

TPrescalerReg = 0x2B << 1, // the lower 8 bits of the TPrescaler value. The 4 high bits are in TModeReg.

TReloadRegH = 0x2C << 1, // defines the 16-bit timer reload value

TReloadRegL = 0x2D << 1,

TCounterValueRegH = 0x2E << 1, // shows the 16-bit timer value

TCounterValueRegL = 0x2F << 1,

// Page 3: Test Registers

// 0x30 // reserved for future use

TestSel1Reg = 0x31 << 1, // general test signal configuration

TestSel2Reg = 0x32 << 1, // general test signal configuration

TestPinEnReg = 0x33 << 1, // enables pin output driver on pins D1 to D7

TestPinValueReg = 0x34 << 1, // defines the values for D1 to D7 when it is used as an I/O bus

TestBusReg = 0x35 << 1, // shows the status of the internal test bus

AutoTestReg = 0x36 << 1, // controls the digital self-test

VersionReg = 0x37 << 1, // shows the software version

AnalogTestReg = 0x38 << 1, // controls the pins AUX1 and AUX2

TestDAC1Reg = 0x39 << 1, // defines the test value for TestDAC1

TestDAC2Reg = 0x3A << 1, // defines the test value for TestDAC2

TestADCReg = 0x3B << 1 // shows the value of ADC I and Q channels

// 0x3C // reserved for production tests

// 0x3D // reserved for production tests

// 0x3E // reserved for production tests

// 0x3F // reserved for production tests

};

// MFRC522 commands. Described in chapter 10 of the datasheet.

enum PCD\_Command {

PCD\_Idle = 0x00, // no action, cancels current command execution

PCD\_Mem = 0x01, // stores 25 bytes into the internal buffer

PCD\_GenerateRandomID = 0x02, // generates a 10-byte random ID number

PCD\_CalcCRC = 0x03, // activates the CRC coprocessor or performs a self-test

PCD\_Transmit = 0x04, // transmits data from the FIFO buffer

PCD\_NoCmdChange = 0x07, // no command change, can be used to modify the CommandReg register bits without affecting the command, for example, the PowerDown bit

PCD\_Receive = 0x08, // activates the receiver circuits

PCD\_Transceive = 0x0C, // transmits data from FIFO buffer to antenna and automatically activates the receiver after transmission

PCD\_MFAuthent = 0x0E, // performs the MIFARE standard authentication as a reader

PCD\_SoftReset = 0x0F // resets the MFRC522

};

// MFRC522 RxGain[2:0] masks, defines the receiver's signal voltage gain factor (on the PCD).

// Described in 9.3.3.6 / table 98 of the datasheet at http://www.nxp.com/documents/data\_sheet/MFRC522.pdf

enum PCD\_RxGain {

RxGain\_18dB = 0x00 << 4, // 000b - 18 dB, minimum

RxGain\_23dB = 0x01 << 4, // 001b - 23 dB

RxGain\_18dB\_2 = 0x02 << 4, // 010b - 18 dB, it seems 010b is a duplicate for 000b

RxGain\_23dB\_2 = 0x03 << 4, // 011b - 23 dB, it seems 011b is a duplicate for 001b

RxGain\_33dB = 0x04 << 4, // 100b - 33 dB, average, and typical default

RxGain\_38dB = 0x05 << 4, // 101b - 38 dB

RxGain\_43dB = 0x06 << 4, // 110b - 43 dB

RxGain\_48dB = 0x07 << 4, // 111b - 48 dB, maximum

RxGain\_min = 0x00 << 4, // 000b - 18 dB, minimum, convenience for RxGain\_18dB

RxGain\_avg = 0x04 << 4, // 100b - 33 dB, average, convenience for RxGain\_33dB

RxGain\_max = 0x07 << 4 // 111b - 48 dB, maximum, convenience for RxGain\_48dB

};

// Commands sent to the PICC.

enum PICC\_Command {

// 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 (from http://www.mouser.com/ds/2/302/MF1S503x-89574.pdf, Section 9)

// 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.

// The commands used for MIFARE Ultralight (from http://www.nxp.com/documents/data\_sheet/MF0ICU1.pdf, Section 8.6)

// The PICC\_CMD\_MF\_READ and PICC\_CMD\_MF\_WRITE can also be used for MIFARE Ultralight.

PICC\_CMD\_UL\_WRITE = 0xA2 // Writes one 4 byte page to the PICC.

};

// MIFARE constants that does not fit anywhere else

enum MIFARE\_Misc {

MF\_ACK = 0xA, // The MIFARE Classic uses a 4 bit ACK/NAK. Any other value than 0xA is NAK.

MF\_KEY\_SIZE = 6 // A Mifare Crypto1 key is 6 bytes.

};

// PICC types we can detect. Remember to update PICC\_GetTypeName() if you add more.

// last value set to 0xff, then compiler uses less ram, it seems some optimisations are triggered

enum PICC\_Type : byte {

PICC\_TYPE\_UNKNOWN ,

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 Classic protocol, 320 bytes

PICC\_TYPE\_MIFARE\_1K , // MIFARE Classic protocol, 1KB

PICC\_TYPE\_MIFARE\_4K , // MIFARE Classic protocol, 4KB

PICC\_TYPE\_MIFARE\_UL , // MIFARE Ultralight or Ultralight C

PICC\_TYPE\_MIFARE\_PLUS , // MIFARE Plus

PICC\_TYPE\_TNP3XXX , // Only mentioned in NXP AN 10833 MIFARE Type Identification Procedure

PICC\_TYPE\_NOT\_COMPLETE = 0xff // SAK indicates UID is not complete.

};

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

// last value set to 0xff, then compiler uses less ram, it seems some optimisations are triggered

enum StatusCode : byte {

STATUS\_OK , // Success

STATUS\_ERROR , // Error in communication

STATUS\_COLLISION , // Collission 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 = 0xff // A MIFARE PICC responded with NAK.

};

// 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;

// Member variables

Uid uid; // Used by PICC\_ReadCardSerial().

// Size of the MFRC522 FIFO

static const byte FIFO\_SIZE = 64; // The FIFO is 64 bytes.

/////////////////////////////////////////////////////////////////////////////////////

// Functions for setting up the Arduino

/////////////////////////////////////////////////////////////////////////////////////

MFRC522();

MFRC522(byte resetPowerDownPin);

MFRC522(byte chipSelectPin, byte resetPowerDownPin);

/////////////////////////////////////////////////////////////////////////////////////

// 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);

StatusCode PCD\_CalculateCRC(byte \*data, byte length, byte \*result);

/////////////////////////////////////////////////////////////////////////////////////

// Functions for manipulating the MFRC522

/////////////////////////////////////////////////////////////////////////////////////

void PCD\_Init();

void PCD\_Init(byte resetPowerDownPin);

void PCD\_Init(byte chipSelectPin, byte resetPowerDownPin);

void PCD\_Reset();

void PCD\_AntennaOn();

void PCD\_AntennaOff();

byte PCD\_GetAntennaGain();

void PCD\_SetAntennaGain(byte mask);

bool PCD\_PerformSelfTest();

/////////////////////////////////////////////////////////////////////////////////////

// Functions for communicating with PICCs

/////////////////////////////////////////////////////////////////////////////////////

StatusCode PCD\_TransceiveData(byte \*sendData, byte sendLen, byte \*backData, byte \*backLen, byte \*validBits = NULL, byte rxAlign = 0, bool checkCRC = false);

StatusCode 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);

StatusCode PICC\_RequestA(byte \*bufferATQA, byte \*bufferSize);

StatusCode PICC\_WakeupA(byte \*bufferATQA, byte \*bufferSize);

StatusCode PICC\_REQA\_or\_WUPA(byte command, byte \*bufferATQA, byte \*bufferSize);

StatusCode PICC\_Select(Uid \*uid, byte validBits = 0);

StatusCode PICC\_HaltA();

/////////////////////////////////////////////////////////////////////////////////////

// Functions for communicating with MIFARE PICCs

/////////////////////////////////////////////////////////////////////////////////////

StatusCode PCD\_Authenticate(byte command, byte blockAddr, MIFARE\_Key \*key, Uid \*uid);

void PCD\_StopCrypto1();

StatusCode MIFARE\_Read(byte blockAddr, byte \*buffer, byte \*bufferSize);

StatusCode MIFARE\_Write(byte blockAddr, byte \*buffer, byte bufferSize);

StatusCode MIFARE\_Ultralight\_Write(byte page, byte \*buffer, byte bufferSize);

StatusCode MIFARE\_Decrement(byte blockAddr, int32\_t delta);

StatusCode MIFARE\_Increment(byte blockAddr, int32\_t delta);

StatusCode MIFARE\_Restore(byte blockAddr);

StatusCode MIFARE\_Transfer(byte blockAddr);

StatusCode MIFARE\_GetValue(byte blockAddr, int32\_t \*value);

StatusCode MIFARE\_SetValue(byte blockAddr, int32\_t value);

StatusCode PCD\_NTAG216\_AUTH(byte \*passWord, byte pACK[]);

/////////////////////////////////////////////////////////////////////////////////////

// Support functions

/////////////////////////////////////////////////////////////////////////////////////

StatusCode PCD\_MIFARE\_Transceive(byte \*sendData, byte sendLen, bool acceptTimeout = false);

// old function used too much memory, now name moved to flash; if you need char, copy from flash to memory

//const char \*GetStatusCodeName(byte code);

static const \_\_FlashStringHelper \*GetStatusCodeName(StatusCode code);

static PICC\_Type PICC\_GetType(byte sak);

// old function used too much memory, now name moved to flash; if you need char, copy from flash to memory

//const char \*PICC\_GetTypeName(byte type);

static const \_\_FlashStringHelper \*PICC\_GetTypeName(PICC\_Type type);

// Support functions for debuging

void PCD\_DumpVersionToSerial();

void PICC\_DumpToSerial(Uid \*uid);

void PICC\_DumpDetailsToSerial(Uid \*uid);

void PICC\_DumpMifareClassicToSerial(Uid \*uid, PICC\_Type piccType, MIFARE\_Key \*key);

void PICC\_DumpMifareClassicSectorToSerial(Uid \*uid, MIFARE\_Key \*key, byte sector);

void PICC\_DumpMifareUltralightToSerial();

// Advanced functions for MIFARE

void MIFARE\_SetAccessBits(byte \*accessBitBuffer, byte g0, byte g1, byte g2, byte g3);

bool MIFARE\_OpenUidBackdoor(bool logErrors);

bool MIFARE\_SetUid(byte \*newUid, byte uidSize, bool logErrors);

bool MIFARE\_UnbrickUidSector(bool logErrors);

/////////////////////////////////////////////////////////////////////////////////////

// Convenience functions - does not add extra functionality

/////////////////////////////////////////////////////////////////////////////////////

bool PICC\_IsNewCardPresent();

bool PICC\_ReadCardSerial();

private:

byte \_chipSelectPin; // Arduino pin connected to MFRC522's SPI slave select input (Pin 24, NSS, active low)

byte \_resetPowerDownPin; // Arduino pin connected to MFRC522's reset and power down input (Pin 6, NRSTPD, active low)

StatusCode MIFARE\_TwoStepHelper(byte command, byte blockAddr, int32\_t data);

};

#endif

/\*

\* MFRC522.cpp - Library to use ARDUINO RFID MODULE KIT 13.56 MHZ WITH TAGS SPI W AND R BY COOQROBOT.

\* NOTE: Please also check the comments in MFRC522.h - they provide useful hints and background information.

\* Released into the public domain.

\*/

/////////////////////////////////////////////////////////////////////////////////////

// Functions for setting up the Arduino

/////////////////////////////////////////////////////////////////////////////////////

/\*\*

\* Constructor.

\*/

MFRC522::MFRC522() {

} // End constructor

/\*\*

\* Constructor.

\* Prepares the output pins.

\*/

MFRC522::MFRC522( byte resetPowerDownPin ///< Arduino pin connected to MFRC522's reset and power down input (Pin 6, NRSTPD, active low)

): MFRC522(SS, resetPowerDownPin) { // SS is defined in pins\_arduino.h

} // End constructor

/\*\*

\* Constructor.

\* Prepares the output pins.

\*/

MFRC522::MFRC522( byte chipSelectPin, ///< Arduino pin connected to MFRC522's SPI slave select input (Pin 24, NSS, active low)

byte resetPowerDownPin ///< Arduino pin connected to MFRC522's reset and power down input (Pin 6, NRSTPD, active low)

) {

\_chipSelectPin = chipSelectPin;

\_resetPowerDownPin = resetPowerDownPin;

} // End constructor

/////////////////////////////////////////////////////////////////////////////////////

// Basic interface functions for communicating with the MFRC522

/////////////////////////////////////////////////////////////////////////////////////

/\*\*

\* Writes a byte to the specified register in the MFRC522 chip.

\* The interface is described in the datasheet section 8.1.2.

\*/

void MFRC522::PCD\_WriteRegister( byte reg, ///< The register to write to. One of the PCD\_Register enums.

byte value ///< The value to write.

) {

SPI.beginTransaction(SPISettings(SPI\_CLOCK\_DIV4, MSBFIRST, SPI\_MODE0)); // Set the settings to work with SPI bus

digitalWrite(\_chipSelectPin, LOW); // Select slave

SPI.transfer(reg & 0x7E); // MSB == 0 is for writing. LSB is not used in address. Datasheet section 8.1.2.3.

SPI.transfer(value);

digitalWrite(\_chipSelectPin, HIGH); // Release slave again

SPI.endTransaction(); // Stop using the SPI bus

} // End PCD\_WriteRegister()

/\*\*

\* Writes a number of bytes to the specified register in the MFRC522 chip.

\* The interface is described in the datasheet section 8.1.2.

\*/

void MFRC522::PCD\_WriteRegister( byte reg, ///< The register to write to. One of the PCD\_Register enums.

byte count, ///< The number of bytes to write to the register

byte \*values ///< The values to write. Byte array.

) {

SPI.beginTransaction(SPISettings(SPI\_CLOCK\_DIV4, MSBFIRST, SPI\_MODE0)); // Set the settings to work with SPI bus

digitalWrite(\_chipSelectPin, LOW); // Select slave

SPI.transfer(reg & 0x7E); // MSB == 0 is for writing. LSB is not used in address. Datasheet section 8.1.2.3.

for (byte index = 0; index < count; index++) {

SPI.transfer(values[index]);

}

digitalWrite(\_chipSelectPin, HIGH); // Release slave again

SPI.endTransaction(); // Stop using the SPI bus

} // End PCD\_WriteRegister()

/\*\*

\* Reads a byte from the specified register in the MFRC522 chip.

\* The interface is described in the datasheet section 8.1.2.

\*/

byte MFRC522::PCD\_ReadRegister( byte reg ///< The register to read from. One of the PCD\_Register enums.

) {

byte value;

SPI.beginTransaction(SPISettings(SPI\_CLOCK\_DIV4, MSBFIRST, SPI\_MODE0)); // Set the settings to work with SPI bus

digitalWrite(\_chipSelectPin, LOW); // Select slave

SPI.transfer(0x80 | (reg & 0x7E)); // MSB == 1 is for reading. LSB is not used in address. Datasheet section 8.1.2.3.

value = SPI.transfer(0); // Read the value back. Send 0 to stop reading.

digitalWrite(\_chipSelectPin, HIGH); // Release slave again

SPI.endTransaction(); // Stop using the SPI bus

return value;

} // End PCD\_ReadRegister()

/\*\*

\* Reads a number of bytes from the specified register in the MFRC522 chip.

\* The interface is described in the datasheet section 8.1.2.

\*/

void MFRC522::PCD\_ReadRegister( byte reg, ///< The register to read from. One of the PCD\_Register enums.

byte count, ///< The number of bytes to read

byte \*values, ///< Byte array to store the values in.

byte rxAlign ///< Only bit positions rxAlign..7 in values[0] are updated.

) {

if (count == 0) {

return;

}

//Serial.print(F("Reading ")); Serial.print(count); Serial.println(F(" bytes from register."));

byte address = 0x80 | (reg & 0x7E); // MSB == 1 is for reading. LSB is not used in address. Datasheet section 8.1.2.3.

byte index = 0; // Index in values array.

SPI.beginTransaction(SPISettings(SPI\_CLOCK\_DIV4, MSBFIRST, SPI\_MODE0)); // Set the settings to work with SPI bus

digitalWrite(\_chipSelectPin, LOW); // Select slave

count--; // One read is performed outside of the loop

SPI.transfer(address); // Tell MFRC522 which address we want to read

while (index < count) {

if (index == 0 && rxAlign) { // Only update bit positions rxAlign..7 in values[0]

// Create bit mask for bit positions rxAlign..7

byte mask = 0;

for (byte i = rxAlign; i <= 7; i++) {

mask |= (1 << i);

}

// Read value and tell that we want to read the same address again.

byte value = SPI.transfer(address);

// Apply mask to both current value of values[0] and the new data in value.

values[0] = (values[index] & ~mask) | (value & mask);

}

else { // Normal case

values[index] = SPI.transfer(address); // Read value and tell that we want to read the same address again.

}

index++;

}

values[index] = SPI.transfer(0); // Read the final byte. Send 0 to stop reading.

digitalWrite(\_chipSelectPin, HIGH); // Release slave again

SPI.endTransaction(); // Stop using the SPI bus

} // End PCD\_ReadRegister()

/\*\*

\* Sets the bits given in mask in register reg.

\*/

void MFRC522::PCD\_SetRegisterBitMask( byte reg, ///< The register to update. One of the PCD\_Register enums.

byte mask ///< The bits to set.

) {

byte tmp;

tmp = PCD\_ReadRegister(reg);

PCD\_WriteRegister(reg, tmp | mask); // set bit mask

} // End PCD\_SetRegisterBitMask()

/\*\*

\* Clears the bits given in mask from register reg.

\*/

void MFRC522::PCD\_ClearRegisterBitMask( byte reg, ///< The register to update. One of the PCD\_Register enums.

byte mask ///< The bits to clear.

) {

byte tmp;

tmp = PCD\_ReadRegister(reg);

PCD\_WriteRegister(reg, tmp & (~mask)); // clear bit mask

} // End PCD\_ClearRegisterBitMask()

/\*\*

\* Use the CRC coprocessor in the MFRC522 to calculate a CRC\_A.

\*

\* @return STATUS\_OK on success, STATUS\_??? otherwise.

\*/

MFRC522::StatusCode MFRC522::PCD\_CalculateCRC( byte \*data, ///< In: Pointer to the data to transfer to the FIFO for CRC calculation.

byte length, ///< In: The number of bytes to transfer.

byte \*result ///< Out: Pointer to result buffer. Result is written to result[0..1], low byte first.

) {

PCD\_WriteRegister(CommandReg, PCD\_Idle); // Stop any active command.

PCD\_WriteRegister(DivIrqReg, 0x04); // Clear the CRCIRq interrupt request bit

PCD\_SetRegisterBitMask(FIFOLevelReg, 0x80); // FlushBuffer = 1, FIFO initialization

PCD\_WriteRegister(FIFODataReg, length, data); // Write data to the FIFO

PCD\_WriteRegister(CommandReg, PCD\_CalcCRC); // Start the calculation

// Wait for the CRC calculation to complete. Each iteration of the while-loop takes 17.73μs.

// TODO check/modify for other architectures than Arduino Uno 16bit

uint16\_t i = 5000;

byte n;

while (1) {

n = PCD\_ReadRegister(DivIrqReg); // DivIrqReg[7..0] bits are: Set2 reserved reserved MfinActIRq reserved CRCIRq reserved reserved

if (n & 0x04) { // CRCIRq bit set - calculation done

break;

}

if (--i == 0) { // The emergency break. We will eventually terminate on this one after 89ms. Communication with the MFRC522 might be down.

return STATUS\_TIMEOUT;

}

}

PCD\_WriteRegister(CommandReg, PCD\_Idle); // Stop calculating CRC for new content in the FIFO.

// Transfer the result from the registers to the result buffer

result[0] = PCD\_ReadRegister(CRCResultRegL);

result[1] = PCD\_ReadRegister(CRCResultRegH);

return STATUS\_OK;

} // End PCD\_CalculateCRC()

/////////////////////////////////////////////////////////////////////////////////////

// Functions for manipulating the MFRC522

/////////////////////////////////////////////////////////////////////////////////////

/\*\*

\* Initializes the MFRC522 chip.

\*/

void MFRC522::PCD\_Init() {

// Set the chipSelectPin as digital output, do not select the slave yet

pinMode(\_chipSelectPin, OUTPUT);

digitalWrite(\_chipSelectPin, HIGH);

// Set the resetPowerDownPin as digital output, do not reset or power down.

pinMode(\_resetPowerDownPin, OUTPUT);

if (digitalRead(\_resetPowerDownPin) == LOW) { //The MFRC522 chip is in power down mode.

digitalWrite(\_resetPowerDownPin, HIGH); // Exit power down mode. This triggers a hard reset.

// Section 8.8.2 in the datasheet says the oscillator start-up time is the start up time of the crystal + 37,74μs. Let us be generous: 50ms.

delay(50);

}

else { // Perform a soft reset

PCD\_Reset();

}

// When communicating with a PICC we need a timeout if something goes wrong.

// f\_timer = 13.56 MHz / (2\*TPreScaler+1) where TPreScaler = [TPrescaler\_Hi:TPrescaler\_Lo].

// TPrescaler\_Hi are the four low bits in TModeReg. TPrescaler\_Lo is TPrescalerReg.

PCD\_WriteRegister(TModeReg, 0x80); // TAuto=1; timer starts automatically at the end of the transmission in all communication modes at all speeds

PCD\_WriteRegister(TPrescalerReg, 0xA9); // TPreScaler = TModeReg[3..0]:TPrescalerReg, ie 0x0A9 = 169 => f\_timer=40kHz, ie a timer period of 25μs.

PCD\_WriteRegister(TReloadRegH, 0x03); // Reload timer with 0x3E8 = 1000, ie 25ms before timeout.

PCD\_WriteRegister(TReloadRegL, 0xE8);

PCD\_WriteRegister(TxASKReg, 0x40); // Default 0x00. Force a 100 % ASK modulation independent of the ModGsPReg register setting

PCD\_WriteRegister(ModeReg, 0x3D); // Default 0x3F. Set the preset value for the CRC coprocessor for the CalcCRC command to 0x6363 (ISO 14443-3 part 6.2.4)

PCD\_AntennaOn(); // Enable the antenna driver pins TX1 and TX2 (they were disabled by the reset)

} // End PCD\_Init()

/\*\*

\* Initializes the MFRC522 chip.

\*/

void MFRC522::PCD\_Init( byte resetPowerDownPin ///< Arduino pin connected to MFRC522's reset and power down input (Pin 6, NRSTPD, active low)

) {

PCD\_Init(SS, resetPowerDownPin); // SS is defined in pins\_arduino.h

} // End PCD\_Init()

/\*\*

\* Initializes the MFRC522 chip.

\*/

void MFRC522::PCD\_Init( byte chipSelectPin, ///< Arduino pin connected to MFRC522's SPI slave select input (Pin 24, NSS, active low)

byte resetPowerDownPin ///< Arduino pin connected to MFRC522's reset and power down input (Pin 6, NRSTPD, active low)

) {

\_chipSelectPin = chipSelectPin;

\_resetPowerDownPin = resetPowerDownPin;

// Set the chipSelectPin as digital output, do not select the slave yet

PCD\_Init();

} // End PCD\_Init()

/\*\*

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

\*/

void MFRC522::PCD\_Reset() {

PCD\_WriteRegister(CommandReg, PCD\_SoftReset); // Issue the SoftReset command.

// The datasheet does not mention how long the SoftRest command takes to complete.

// But the MFRC522 might have been in soft power-down mode (triggered by bit 4 of CommandReg)

// Section 8.8.2 in the datasheet says the oscillator start-up time is the start up time of the crystal + 37,74μs. Let us be generous: 50ms.

delay(50);

// Wait for the PowerDown bit in CommandReg to be cleared

while (PCD\_ReadRegister(CommandReg) & (1<<4)) {

// PCD still restarting - unlikely after waiting 50ms, but better safe than sorry.

}

} // End PCD\_Reset()

/\*\*

\* Turns the antenna on by enabling pins TX1 and TX2.

\* After a reset these pins are disabled.

\*/

void MFRC522::PCD\_AntennaOn() {

byte value = PCD\_ReadRegister(TxControlReg);

if ((value & 0x03) != 0x03) {

PCD\_WriteRegister(TxControlReg, value | 0x03);

}

} // End PCD\_AntennaOn()

/\*\*

\* Turns the antenna off by disabling pins TX1 and TX2.

\*/

void MFRC522::PCD\_AntennaOff() {

PCD\_ClearRegisterBitMask(TxControlReg, 0x03);

} // End PCD\_AntennaOff()

/\*\*

\* Get the current MFRC522 Receiver Gain (RxGain[2:0]) value.

\* See 9.3.3.6 / table 98 in http://www.nxp.com/documents/data\_sheet/MFRC522.pdf

\* NOTE: Return value scrubbed with (0x07<<4)=01110000b as RCFfgReg may use reserved bits.

\*

\* @return Value of the RxGain, scrubbed to the 3 bits used.

\*/

byte MFRC522::PCD\_GetAntennaGain() {

return PCD\_ReadRegister(RFCfgReg) & (0x07<<4);

} // End PCD\_GetAntennaGain()

/\*\*

\* Set the MFRC522 Receiver Gain (RxGain) to value specified by given mask.

\* See 9.3.3.6 / table 98 in http://www.nxp.com/documents/data\_sheet/MFRC522.pdf

\* NOTE: Given mask is scrubbed with (0x07<<4)=01110000b as RCFfgReg may use reserved bits.

\*/

void MFRC522::PCD\_SetAntennaGain(byte mask) {

if (PCD\_GetAntennaGain() != mask) { // only bother if there is a change

PCD\_ClearRegisterBitMask(RFCfgReg, (0x07<<4)); // clear needed to allow 000 pattern

PCD\_SetRegisterBitMask(RFCfgReg, mask & (0x07<<4)); // only set RxGain[2:0] bits

}

} // End PCD\_SetAntennaGain()

/\*\*

\* Performs a self-test of the MFRC522

\* See 16.1.1 in http://www.nxp.com/documents/data\_sheet/MFRC522.pdf

\*

\* @return Whether or not the test passed. Or false if no firmware reference is available.

\*/

bool MFRC522::PCD\_PerformSelfTest() {

// This follows directly the steps outlined in 16.1.1

// 1. Perform a soft reset.

PCD\_Reset();

// 2. Clear the internal buffer by writing 25 bytes of 00h

byte ZEROES[25] = {0x00};

PCD\_SetRegisterBitMask(FIFOLevelReg, 0x80); // flush the FIFO buffer

PCD\_WriteRegister(FIFODataReg, 25, ZEROES); // write 25 bytes of 00h to FIFO

PCD\_WriteRegister(CommandReg, PCD\_Mem); // transfer to internal buffer

// 3. Enable self-test

PCD\_WriteRegister(AutoTestReg, 0x09);

// 4. Write 00h to FIFO buffer

PCD\_WriteRegister(FIFODataReg, 0x00);

// 5. Start self-test by issuing the CalcCRC command

PCD\_WriteRegister(CommandReg, PCD\_CalcCRC);

// 6. Wait for self-test to complete

byte n;

for (uint8\_t i = 0; i < 0xFF; i++) {

// The datasheet does not specify exact completion condition except

// that FIFO buffer should contain 64 bytes.

// While selftest is initiated by CalcCRC command

// it behaves differently from normal CRC computation,

// so one can't reliably use DivIrqReg to check for completion.

// It is reported that some devices does not trigger CRCIRq flag

// during selftest.

n = PCD\_ReadRegister(FIFOLevelReg);

if (n >= 64) {

break;

}

}

PCD\_WriteRegister(CommandReg, PCD\_Idle); // Stop calculating CRC for new content in the FIFO.

// 7. Read out resulting 64 bytes from the FIFO buffer.

byte result[64];

PCD\_ReadRegister(FIFODataReg, 64, result, 0);

// Auto self-test done

// Reset AutoTestReg register to be 0 again. Required for normal operation.

PCD\_WriteRegister(AutoTestReg, 0x00);

// Determine firmware version (see section 9.3.4.8 in spec)

byte version = PCD\_ReadRegister(VersionReg);

// Pick the appropriate reference values

const byte \*reference;

switch (version) {

case 0x88: // Fudan Semiconductor FM17522 clone

reference = FM17522\_firmware\_reference;

break;

case 0x90: // Version 0.0

reference = MFRC522\_firmware\_referenceV0\_0;

break;

case 0x91: // Version 1.0

reference = MFRC522\_firmware\_referenceV1\_0;

break;

case 0x92: // Version 2.0

reference = MFRC522\_firmware\_referenceV2\_0;

break;

default: // Unknown version

return false; // abort test

}

// Verify that the results match up to our expectations

for (uint8\_t i = 0; i < 64; i++) {

if (result[i] != pgm\_read\_byte(&(reference[i]))) {

return false;

}

}

// Test passed; all is good.

return true;

} // End PCD\_PerformSelfTest()

/////////////////////////////////////////////////////////////////////////////////////

// Functions for communicating with PICCs

/////////////////////////////////////////////////////////////////////////////////////

/\*\*

\* Executes the Transceive command.

\* CRC validation can only be done if backData and backLen are specified.

\*

\* @return STATUS\_OK on success, STATUS\_??? otherwise.

\*/

MFRC522::StatusCode MFRC522::PCD\_TransceiveData( byte \*sendData, ///< Pointer to the data to transfer to the FIFO.

byte sendLen, ///< Number of bytes to transfer to the FIFO.

byte \*backData, ///< NULL or pointer to buffer if data should be read back after executing the command.

byte \*backLen, ///< In: Max number of bytes to write to \*backData. Out: The number of bytes returned.

byte \*validBits, ///< In/Out: The number of valid bits in the last byte. 0 for 8 valid bits. Default NULL.

byte rxAlign, ///< In: Defines the bit position in backData[0] for the first bit received. Default 0.

bool checkCRC ///< In: True => The last two bytes of the response is assumed to be a CRC\_A that must be validated.

) {

byte waitIRq = 0x30; // RxIRq and IdleIRq

return PCD\_CommunicateWithPICC(PCD\_Transceive, waitIRq, sendData, sendLen, backData, backLen, validBits, rxAlign, checkCRC);

} // End PCD\_TransceiveData()

/\*\*

\* Transfers data to the MFRC522 FIFO, executes a command, waits for completion and transfers data back from the FIFO.

\* CRC validation can only be done if backData and backLen are specified.

\*

\* @return STATUS\_OK on success, STATUS\_??? otherwise.

\*/

MFRC522::StatusCode MFRC522::PCD\_CommunicateWithPICC( byte command, ///< The command to execute. One of the PCD\_Command enums.

byte waitIRq, ///< The bits in the ComIrqReg register that signals successful completion of the command.

byte \*sendData, ///< Pointer to the data to transfer to the FIFO.

byte sendLen, ///< Number of bytes to transfer to the FIFO.

byte \*backData, ///< NULL or pointer to buffer if data should be read back after executing the command.

byte \*backLen, ///< In: Max number of bytes to write to \*backData. Out: The number of bytes returned.

byte \*validBits, ///< In/Out: The number of valid bits in the last byte. 0 for 8 valid bits.

byte rxAlign, ///< In: Defines the bit position in backData[0] for the first bit received. Default 0.

bool checkCRC ///< In: True => The last two bytes of the response is assumed to be a CRC\_A that must be validated.

) {

byte n, \_validBits;

uint16\_t i;

// Prepare values for BitFramingReg

byte txLastBits = validBits ? \*validBits : 0;

byte bitFraming = (rxAlign << 4) + txLastBits; // RxAlign = BitFramingReg[6..4]. TxLastBits = BitFramingReg[2..0]

PCD\_WriteRegister(CommandReg, PCD\_Idle); // Stop any active command.

PCD\_WriteRegister(ComIrqReg, 0x7F); // Clear all seven interrupt request bits

PCD\_SetRegisterBitMask(FIFOLevelReg, 0x80); // FlushBuffer = 1, FIFO initialization

PCD\_WriteRegister(FIFODataReg, sendLen, sendData); // Write sendData to the FIFO

PCD\_WriteRegister(BitFramingReg, bitFraming); // Bit adjustments

PCD\_WriteRegister(CommandReg, command); // Execute the command

if (command == PCD\_Transceive) {

PCD\_SetRegisterBitMask(BitFramingReg, 0x80); // StartSend=1, transmission of data starts

}

// Wait for the command to complete.

// In PCD\_Init() we set the TAuto flag in TModeReg. This means the timer automatically starts when the PCD stops transmitting.

// Each iteration of the do-while-loop takes 17.86μs.

// TODO check/modify for other architectures than Arduino Uno 16bit

i = 2000;

while (1) {

n = PCD\_ReadRegister(ComIrqReg); // ComIrqReg[7..0] bits are: Set1 TxIRq RxIRq IdleIRq HiAlertIRq LoAlertIRq ErrIRq TimerIRq

if (n & waitIRq) { // One of the interrupts that signal success has been set.

break;

}

if (n & 0x01) { // Timer interrupt - nothing received in 25ms

return STATUS\_TIMEOUT;

}

if (--i == 0) { // The emergency break. If all other conditions fail we will eventually terminate on this one after 35.7ms. Communication with the MFRC522 might be down.

return STATUS\_TIMEOUT;

}

}

// Stop now if any errors except collisions were detected.

byte errorRegValue = PCD\_ReadRegister(ErrorReg); // ErrorReg[7..0] bits are: WrErr TempErr reserved BufferOvfl CollErr CRCErr ParityErr ProtocolErr

if (errorRegValue & 0x13) { // BufferOvfl ParityErr ProtocolErr

return STATUS\_ERROR;

}

// If the caller wants data back, get it from the MFRC522.

if (backData && backLen) {

n = PCD\_ReadRegister(FIFOLevelReg); // Number of bytes in the FIFO

if (n > \*backLen) {

return STATUS\_NO\_ROOM;

}

\*backLen = n; // Number of bytes returned

PCD\_ReadRegister(FIFODataReg, n, backData, rxAlign); // Get received data from FIFO

\_validBits = PCD\_ReadRegister(ControlReg) & 0x07; // RxLastBits[2:0] indicates the number of valid bits in the last received byte. If this value is 000b, the whole byte is valid.

if (validBits) {

\*validBits = \_validBits;

}

}

// Tell about collisions

if (errorRegValue & 0x08) { // CollErr

return STATUS\_COLLISION;

}

// Perform CRC\_A validation if requested.

if (backData && backLen && checkCRC) {

// In this case a MIFARE Classic NAK is not OK.

if (\*backLen == 1 && \_validBits == 4) {

return STATUS\_MIFARE\_NACK;

}

// We need at least the CRC\_A value and all 8 bits of the last byte must be received.

if (\*backLen < 2 || \_validBits != 0) {

return STATUS\_CRC\_WRONG;

}

// Verify CRC\_A - do our own calculation and store the control in controlBuffer.

byte controlBuffer[2];

MFRC522::StatusCode status = PCD\_CalculateCRC(&backData[0], \*backLen - 2, &controlBuffer[0]);

if (status != STATUS\_OK) {

return status;

}

if ((backData[\*backLen - 2] != controlBuffer[0]) || (backData[\*backLen - 1] != controlBuffer[1])) {

return STATUS\_CRC\_WRONG;

}

}

return STATUS\_OK;

} // End PCD\_CommunicateWithPICC()

/\*\*

\* 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.

\*/

MFRC522::StatusCode 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.

) {

return PICC\_REQA\_or\_WUPA(PICC\_CMD\_REQA, bufferATQA, bufferSize);

} // End PICC\_RequestA()

/\*\*

\* 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.

\*/

MFRC522::StatusCode 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.

) {

return PICC\_REQA\_or\_WUPA(PICC\_CMD\_WUPA, bufferATQA, bufferSize);

} // End PICC\_WakeupA()

/\*\*

\* Transmits REQA or WUPA commands.

\* 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.

\*/

MFRC522::StatusCode MFRC522::PICC\_REQA\_or\_WUPA( byte command, ///< The command to send - PICC\_CMD\_REQA or PICC\_CMD\_WUPA

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.

) {

byte validBits;

MFRC522::StatusCode status;

if (bufferATQA == NULL || \*bufferSize < 2) { // The ATQA response is 2 bytes long.

return STATUS\_NO\_ROOM;

}

PCD\_ClearRegisterBitMask(CollReg, 0x80); // ValuesAfterColl=1 => Bits received after collision are cleared.

validBits = 7; // For REQA and WUPA we need the short frame format - transmit only 7 bits of the last (and only) byte. TxLastBits = BitFramingReg[2..0]

status = PCD\_TransceiveData(&command, 1, bufferATQA, bufferSize, &validBits);

if (status != STATUS\_OK) {

return status;

}

if (\*bufferSize != 2 || validBits != 0) { // ATQA must be exactly 16 bits.

return STATUS\_ERROR;

}

return STATUS\_OK;

} // End PICC\_REQA\_or\_WUPA()

/\*\*

\* 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.

\*/

MFRC522::StatusCode 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.

) {

bool uidComplete;

bool selectDone;

bool useCascadeTag;

byte cascadeLevel = 1;

MFRC522::StatusCode result;

byte count;

byte index;

byte uidIndex; // The first index in uid->uidByte[] that is used in the current Cascade Level.

int8\_t currentLevelKnownBits; // The number of known UID bits in the current Cascade Level.

byte buffer[9]; // The SELECT/ANTICOLLISION commands uses a 7 byte standard frame + 2 bytes CRC\_A

byte bufferUsed; // The number of bytes used in the buffer, ie the number of bytes to transfer to the FIFO.

byte rxAlign; // Used in BitFramingReg. Defines the bit position for the first bit received.

byte txLastBits; // Used in BitFramingReg. The number of valid bits in the last transmitted byte.

byte \*responseBuffer;

byte responseLength;

// 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 are 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

// 3 uid6 uid7 uid8 uid9

// Sanity checks

if (validBits > 80) {

return STATUS\_INVALID;

}

// Prepare MFRC522

PCD\_ClearRegisterBitMask(CollReg, 0x80); // ValuesAfterColl=1 => Bits received after collision are cleared.

// Repeat Cascade Level loop until we have a complete UID.

uidComplete = false;

while (!uidComplete) {

// Set the Cascade Level in the SEL byte, find out if we need to use the Cascade Tag in byte 2.

switch (cascadeLevel) {

case 1:

buffer[0] = PICC\_CMD\_SEL\_CL1;

uidIndex = 0;

useCascadeTag = validBits && uid->size > 4; // When we know that the UID has more than 4 bytes

break;

case 2:

buffer[0] = PICC\_CMD\_SEL\_CL2;

uidIndex = 3;

useCascadeTag = validBits && uid->size > 7; // When we know that the UID has more than 7 bytes

break;

case 3:

buffer[0] = PICC\_CMD\_SEL\_CL3;

uidIndex = 6;

useCascadeTag = false; // Never used in CL3.

break;

default:

return STATUS\_INTERNAL\_ERROR;

break;

}

// How many UID bits are known in this Cascade Level?

currentLevelKnownBits = validBits - (8 \* uidIndex);

if (currentLevelKnownBits < 0) {

currentLevelKnownBits = 0;

}

// Copy the known bits from uid->uidByte[] to buffer[]

index = 2; // destination index in buffer[]

if (useCascadeTag) {

buffer[index++] = PICC\_CMD\_CT;

}

byte bytesToCopy = currentLevelKnownBits / 8 + (currentLevelKnownBits % 8 ? 1 : 0); // The number of bytes needed to represent the known bits for this level.

if (bytesToCopy) {

byte maxBytes = useCascadeTag ? 3 : 4; // Max 4 bytes in each Cascade Level. Only 3 left if we use the Cascade Tag

if (bytesToCopy > maxBytes) {

bytesToCopy = maxBytes;

}

for (count = 0; count < bytesToCopy; count++) {

buffer[index++] = uid->uidByte[uidIndex + count];

}

}

// Now that the data has been copied we need to include the 8 bits in CT in currentLevelKnownBits

if (useCascadeTag) {

currentLevelKnownBits += 8;

}

// Repeat anti collision loop until we can transmit all UID bits + BCC and receive a SAK - max 32 iterations.

selectDone = false;

while (!selectDone) {

// Find out how many bits and bytes to send and receive.

if (currentLevelKnownBits >= 32) { // All UID bits in this Cascade Level are known. This is a SELECT.

//Serial.print(F("SELECT: currentLevelKnownBits=")); Serial.println(currentLevelKnownBits, DEC);

buffer[1] = 0x70; // NVB - Number of Valid Bits: Seven whole bytes

// Calculate BCC - Block Check Character

buffer[6] = buffer[2] ^ buffer[3] ^ buffer[4] ^ buffer[5];

// Calculate CRC\_A

result = PCD\_CalculateCRC(buffer, 7, &buffer[7]);

if (result != STATUS\_OK) {

return result;

}

txLastBits = 0; // 0 => All 8 bits are valid.

bufferUsed = 9;

// Store response in the last 3 bytes of buffer (BCC and CRC\_A - not needed after tx)

responseBuffer = &buffer[6];

responseLength = 3;

}

else { // This is an ANTICOLLISION.

//Serial.print(F("ANTICOLLISION: currentLevelKnownBits=")); Serial.println(currentLevelKnownBits, DEC);

txLastBits = currentLevelKnownBits % 8;

count = currentLevelKnownBits / 8; // Number of whole bytes in the UID part.

index = 2 + count; // Number of whole bytes: SEL + NVB + UIDs

buffer[1] = (index << 4) + txLastBits; // NVB - Number of Valid Bits

bufferUsed = index + (txLastBits ? 1 : 0);

// Store response in the unused part of buffer

responseBuffer = &buffer[index];

responseLength = sizeof(buffer) - index;

}

// Set bit adjustments

rxAlign = txLastBits; // Having a separate variable is overkill. But it makes the next line easier to read.

PCD\_WriteRegister(BitFramingReg, (rxAlign << 4) + txLastBits); // RxAlign = BitFramingReg[6..4]. TxLastBits = BitFramingReg[2..0]

// Transmit the buffer and receive the response.

result = PCD\_TransceiveData(buffer, bufferUsed, responseBuffer, &responseLength, &txLastBits, rxAlign);

if (result == STATUS\_COLLISION) { // More than one PICC in the field => collision.

byte valueOfCollReg = PCD\_ReadRegister(CollReg); // CollReg[7..0] bits are: ValuesAfterColl reserved CollPosNotValid CollPos[4:0]

if (valueOfCollReg & 0x20) { // CollPosNotValid

return STATUS\_COLLISION; // Without a valid collision position we cannot continue

}

byte collisionPos = valueOfCollReg & 0x1F; // Values 0-31, 0 means bit 32.

if (collisionPos == 0) {

collisionPos = 32;

}

if (collisionPos <= currentLevelKnownBits) { // No progress - should not happen

return STATUS\_INTERNAL\_ERROR;

}

// Choose the PICC with the bit set.

currentLevelKnownBits = collisionPos;

count = (currentLevelKnownBits - 1) % 8; // The bit to modify

index = 1 + (currentLevelKnownBits / 8) + (count ? 1 : 0); // First byte is index 0.

buffer[index] |= (1 << count);

}

else if (result != STATUS\_OK) {

return result;

}

else { // STATUS\_OK

if (currentLevelKnownBits >= 32) { // This was a SELECT.

selectDone = true; // No more anticollision

// We continue below outside the while.

}

else { // This was an ANTICOLLISION.

// We now have all 32 bits of the UID in this Cascade Level

currentLevelKnownBits = 32;

// Run loop again to do the SELECT.

}

}

} // End of while (!selectDone)

// We do not check the CBB - it was constructed by us above.

// Copy the found UID bytes from buffer[] to uid->uidByte[]

index = (buffer[2] == PICC\_CMD\_CT) ? 3 : 2; // source index in buffer[]

bytesToCopy = (buffer[2] == PICC\_CMD\_CT) ? 3 : 4;

for (count = 0; count < bytesToCopy; count++) {

uid->uidByte[uidIndex + count] = buffer[index++];

}

// Check response SAK (Select Acknowledge)

if (responseLength != 3 || txLastBits != 0) { // SAK must be exactly 24 bits (1 byte + CRC\_A).

return STATUS\_ERROR;

}

// Verify CRC\_A - do our own calculation and store the control in buffer[2..3] - those bytes are not needed anymore.

result = PCD\_CalculateCRC(responseBuffer, 1, &buffer[2]);

if (result != STATUS\_OK) {

return result;

}

if ((buffer[2] != responseBuffer[1]) || (buffer[3] != responseBuffer[2])) {

return STATUS\_CRC\_WRONG;

}

if (responseBuffer[0] & 0x04) { // Cascade bit set - UID not complete yes

cascadeLevel++;

}

else {

uidComplete = true;

uid->sak = responseBuffer[0];

}

} // End of while (!uidComplete)

// Set correct uid->size

uid->size = 3 \* cascadeLevel + 1;

return STATUS\_OK;

} // End PICC\_Select()

/\*\*

\* Instructs a PICC in state ACTIVE(\*) to go to state HALT.

\*

\* @return STATUS\_OK on success, STATUS\_??? otherwise.

\*/

MFRC522::StatusCode MFRC522::PICC\_HaltA() {

MFRC522::StatusCode result;

byte buffer[4];

// Build command buffer

buffer[0] = PICC\_CMD\_HLTA;

buffer[1] = 0;

// Calculate CRC\_A

result = PCD\_CalculateCRC(buffer, 2, &buffer[2]);

if (result != STATUS\_OK) {

return result;

}

// Send the command.

// The standard says:

// If the PICC responds with any modulation during a period of 1 ms after the end of the frame containing the

// HLTA command, this response shall be interpreted as 'not acknowledge'.

// We interpret that this way: Only STATUS\_TIMEOUT is a success.

result = PCD\_TransceiveData(buffer, sizeof(buffer), NULL, 0);

if (result == STATUS\_TIMEOUT) {

return STATUS\_OK;

}

if (result == STATUS\_OK) { // That is ironically NOT ok in this case ;-)

return STATUS\_ERROR;

}

return result;

} // End PICC\_HaltA()

/////////////////////////////////////////////////////////////////////////////////////

// Functions for communicating with MIFARE PICCs

/////////////////////////////////////////////////////////////////////////////////////

/\*\*

\* 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() after communicating 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.

\*/

MFRC522::StatusCode MFRC522::PCD\_Authenticate(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 Crypteo1 key to use (6 bytes)

Uid \*uid ///< Pointer to Uid struct. The first 4 bytes of the UID is used.

) {

byte waitIRq = 0x10; // IdleIRq

// Build command buffer

byte sendData[12];

sendData[0] = command;

sendData[1] = blockAddr;

for (byte i = 0; i < MF\_KEY\_SIZE; i++) { // 6 key bytes

sendData[2+i] = key->keyByte[i];

}

// Use the last uid bytes as specified in http://cache.nxp.com/documents/application\_note/AN10927.pdf

// section 3.2.5 "MIFARE Classic Authentication".

// The only missed case is the MF1Sxxxx shortcut activation,

// but it requires cascade tag (CT) byte, that is not part of uid.

for (byte i = 0; i < 4; i++) { // The last 4 bytes of the UID

sendData[8+i] = uid->uidByte[i+uid->size-4];

}

// Start the authentication.

return PCD\_CommunicateWithPICC(PCD\_MFAuthent, waitIRq, &sendData[0], sizeof(sendData));

} // End PCD\_Authenticate()

/\*\*

\* Used to exit the PCD from its authenticated state.

\* Remember to call this function after communicating with an authenticated PICC - otherwise no new communications can start.

\*/

void MFRC522::PCD\_StopCrypto1() {

// Clear MFCrypto1On bit

PCD\_ClearRegisterBitMask(Status2Reg, 0x08); // Status2Reg[7..0] bits are: TempSensClear I2CForceHS reserved reserved MFCrypto1On ModemState[2:0]

} // End 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.

\*/

MFRC522::StatusCode 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.

) {

MFRC522::StatusCode result;

// Sanity check

if (buffer == NULL || \*bufferSize < 18) {

return STATUS\_NO\_ROOM;

}

// Build command buffer

buffer[0] = PICC\_CMD\_MF\_READ;

buffer[1] = blockAddr;

// Calculate CRC\_A

result = PCD\_CalculateCRC(buffer, 2, &buffer[2]);

if (result != STATUS\_OK) {

return result;

}

// Transmit the buffer and receive the response, validate CRC\_A.

return PCD\_TransceiveData(buffer, 4, buffer, bufferSize, NULL, 0, true);

} // End MIFARE\_Read()

/\*\*

\* 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.

\*/

MFRC522::StatusCode 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.

) {

MFRC522::StatusCode result;

// Sanity check

if (buffer == NULL || bufferSize < 16) {

return STATUS\_INVALID;

}

// Mifare Classic protocol requires two communications to perform a write.

// Step 1: Tell the PICC we want to write to block blockAddr.

byte cmdBuffer[2];

cmdBuffer[0] = PICC\_CMD\_MF\_WRITE;

cmdBuffer[1] = blockAddr;

result = PCD\_MIFARE\_Transceive(cmdBuffer, 2); // Adds CRC\_A and checks that the response is MF\_ACK.

if (result != STATUS\_OK) {

return result;

}

// Step 2: Transfer the data

result = PCD\_MIFARE\_Transceive(buffer, bufferSize); // Adds CRC\_A and checks that the response is MF\_ACK.

if (result != STATUS\_OK) {

return result;

}

return STATUS\_OK;

} // End MIFARE\_Write()

/\*\*

\* Writes a 4 byte page to the active MIFARE Ultralight PICC.

\*

\* @return STATUS\_OK on success, STATUS\_??? otherwise.

\*/

MFRC522::StatusCode MFRC522::MIFARE\_Ultralight\_Write( byte page, ///< The page (2-15) to write to.

byte \*buffer, ///< The 4 bytes to write to the PICC

byte bufferSize ///< Buffer size, must be at least 4 bytes. Exactly 4 bytes are written.

) {

MFRC522::StatusCode result;

// Sanity check

if (buffer == NULL || bufferSize < 4) {

return STATUS\_INVALID;

}

// Build commmand buffer

byte cmdBuffer[6];

cmdBuffer[0] = PICC\_CMD\_UL\_WRITE;

cmdBuffer[1] = page;

memcpy(&cmdBuffer[2], buffer, 4);

// Perform the write

result = PCD\_MIFARE\_Transceive(cmdBuffer, 6); // Adds CRC\_A and checks that the response is MF\_ACK.

if (result != STATUS\_OK) {

return result;

}

return STATUS\_OK;

} // End MIFARE\_Ultralight\_Write()

/\*\*

\* 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.

\*/

MFRC522::StatusCode MFRC522::MIFARE\_Decrement( byte blockAddr, ///< The block (0-0xff) number.

int32\_t delta ///< This number is subtracted from the value of block blockAddr.

) {

return MIFARE\_TwoStepHelper(PICC\_CMD\_MF\_DECREMENT, blockAddr, delta);

} // End MIFARE\_Decrement()

/\*\*

\* 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.

\*/

MFRC522::StatusCode MFRC522::MIFARE\_Increment( byte blockAddr, ///< The block (0-0xff) number.

int32\_t delta ///< This number is added to the value of block blockAddr.

) {

return MIFARE\_TwoStepHelper(PICC\_CMD\_MF\_INCREMENT, blockAddr, delta);

} // End MIFARE\_Increment()

/\*\*

\* 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.

\*/

MFRC522::StatusCode 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.

return MIFARE\_TwoStepHelper(PICC\_CMD\_MF\_RESTORE, blockAddr, 0L);

} // End MIFARE\_Restore()

/\*\*

\* Helper function for the two-step MIFARE Classic protocol operations Decrement, Increment and Restore.

\*

\* @return STATUS\_OK on success, STATUS\_??? otherwise.

\*/

MFRC522::StatusCode MFRC522::MIFARE\_TwoStepHelper( byte command, ///< The command to use

byte blockAddr, ///< The block (0-0xff) number.

int32\_t data ///< The data to transfer in step 2

) {

MFRC522::StatusCode result;

byte cmdBuffer[2]; // We only need room for 2 bytes.

// Step 1: Tell the PICC the command and block address

cmdBuffer[0] = command;

cmdBuffer[1] = blockAddr;

result = PCD\_MIFARE\_Transceive( cmdBuffer, 2); // Adds CRC\_A and checks that the response is MF\_ACK.

if (result != STATUS\_OK) {

return result;

}

// Step 2: Transfer the data

result = PCD\_MIFARE\_Transceive( (byte \*)&data, 4, true); // Adds CRC\_A and accept timeout as success.

if (result != STATUS\_OK) {

return result;

}

return STATUS\_OK;

} // End MIFARE\_TwoStepHelper()

/\*\*

\* 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.

\*/

MFRC522::StatusCode MFRC522::MIFARE\_Transfer( byte blockAddr ///< The block (0-0xff) number.

) {

MFRC522::StatusCode result;

byte cmdBuffer[2]; // We only need room for 2 bytes.

// Tell the PICC we want to transfer the result into block blockAddr.

cmdBuffer[0] = PICC\_CMD\_MF\_TRANSFER;

cmdBuffer[1] = blockAddr;

result = PCD\_MIFARE\_Transceive( cmdBuffer, 2); // Adds CRC\_A and checks that the response is MF\_ACK.

if (result != STATUS\_OK) {

return result;

}

return STATUS\_OK;

} // End MIFARE\_Transfer()

/\*\*

\* Helper routine to read the current value from a Value Block.

\*

\* Only for MIFARE Classic and only for blocks in "value block" mode, that

\* is: with access bits [C1 C2 C3] = [110] or [001]. The sector containing

\* the block must be authenticated before calling this function.

\*

\* @param[in] blockAddr The block (0x00-0xff) number.

\* @param[out] value Current value of the Value Block.

\* @return STATUS\_OK on success, STATUS\_??? otherwise.

\*/

MFRC522::StatusCode MFRC522::MIFARE\_GetValue(byte blockAddr, int32\_t \*value) {

MFRC522::StatusCode status;

byte buffer[18];

byte size = sizeof(buffer);

// Read the block

status = MIFARE\_Read(blockAddr, buffer, &size);

if (status == STATUS\_OK) {

// Extract the value

\*value = (int32\_t(buffer[3])<<24) | (int32\_t(buffer[2])<<16) | (int32\_t(buffer[1])<<8) | int32\_t(buffer[0]);

}

return status;

} // End MIFARE\_GetValue()

/\*\*

\* Helper routine to write a specific value into a Value Block.

\*

\* Only for MIFARE Classic and only for blocks in "value block" mode, that

\* is: with access bits [C1 C2 C3] = [110] or [001]. The sector containing

\* the block must be authenticated before calling this function.

\*

\* @param[in] blockAddr The block (0x00-0xff) number.

\* @param[in] value New value of the Value Block.

\* @return STATUS\_OK on success, STATUS\_??? otherwise.

\*/

MFRC522::StatusCode MFRC522::MIFARE\_SetValue(byte blockAddr, int32\_t value) {

byte buffer[18];

// Translate the int32\_t into 4 bytes; repeated 2x in value block

buffer[0] = buffer[ 8] = (value & 0xFF);

buffer[1] = buffer[ 9] = (value & 0xFF00) >> 8;

buffer[2] = buffer[10] = (value & 0xFF0000) >> 16;

buffer[3] = buffer[11] = (value & 0xFF000000) >> 24;

// Inverse 4 bytes also found in value block

buffer[4] = ~buffer[0];

buffer[5] = ~buffer[1];

buffer[6] = ~buffer[2];

buffer[7] = ~buffer[3];

// Address 2x with inverse address 2x

buffer[12] = buffer[14] = blockAddr;

buffer[13] = buffer[15] = ~blockAddr;

// Write the whole data block

return MIFARE\_Write(blockAddr, buffer, 16);

} // End MIFARE\_SetValue()

/\*\*

\* Authenticate with a NTAG216.

\*

\* Only for NTAG216. First implemented by Gargantuanman.

\*

\* @param[in] passWord password.

\* @param[in] pACK result success???.

\* @return STATUS\_OK on success, STATUS\_??? otherwise.

\*/

MFRC522::StatusCode MFRC522::PCD\_NTAG216\_AUTH(byte\* passWord, byte pACK[]) //Authenticate with 32bit password

{

MFRC522::StatusCode result;

byte cmdBuffer[18]; // We need room for 16 bytes data and 2 bytes CRC\_A.

cmdBuffer[0] = 0x1B; //Comando de autentificacion

for (byte i = 0; i<4; i++)

cmdBuffer[i+1] = passWord[i];

result = PCD\_CalculateCRC(cmdBuffer, 5, &cmdBuffer[5]);

if (result!=STATUS\_OK) {

return result;

}

// Transceive the data, store the reply in cmdBuffer[]

byte waitIRq = 0x30; // RxIRq and IdleIRq

byte cmdBufferSize = sizeof(cmdBuffer);

byte validBits = 0;

byte rxlength = 5;

result = PCD\_CommunicateWithPICC(PCD\_Transceive, waitIRq, cmdBuffer, 7, cmdBuffer, &rxlength, &validBits);

pACK[0] = cmdBuffer[0];

pACK[1] = cmdBuffer[1];

if (result!=STATUS\_OK) {

return result;

}

return STATUS\_OK;

} // End PCD\_NTAG216\_AUTH()

/////////////////////////////////////////////////////////////////////////////////////

// Support functions

/////////////////////////////////////////////////////////////////////////////////////

/\*\*

\* Wrapper for MIFARE protocol communication.

\* Adds CRC\_A, executes the Transceive command and checks that the response is MF\_ACK or a timeout.

\*

\* @return STATUS\_OK on success, STATUS\_??? otherwise.

\*/

MFRC522::StatusCode MFRC522::PCD\_MIFARE\_Transceive( byte \*sendData, ///< Pointer to the data to transfer to the FIFO. Do NOT include the CRC\_A.

byte sendLen, ///< Number of bytes in sendData.

bool acceptTimeout ///< True => A timeout is also success

) {

MFRC522::StatusCode result;

byte cmdBuffer[18]; // We need room for 16 bytes data and 2 bytes CRC\_A.

// Sanity check

if (sendData == NULL || sendLen > 16) {

return STATUS\_INVALID;

}

// Copy sendData[] to cmdBuffer[] and add CRC\_A

memcpy(cmdBuffer, sendData, sendLen);

result = PCD\_CalculateCRC(cmdBuffer, sendLen, &cmdBuffer[sendLen]);

if (result != STATUS\_OK) {

return result;

}

sendLen += 2;

// Transceive the data, store the reply in cmdBuffer[]

byte waitIRq = 0x30; // RxIRq and IdleIRq

byte cmdBufferSize = sizeof(cmdBuffer);

byte validBits = 0;

result = PCD\_CommunicateWithPICC(PCD\_Transceive, waitIRq, cmdBuffer, sendLen, cmdBuffer, &cmdBufferSize, &validBits);

if (acceptTimeout && result == STATUS\_TIMEOUT) {

return STATUS\_OK;

}

if (result != STATUS\_OK) {

return result;

}

// The PICC must reply with a 4 bit ACK

if (cmdBufferSize != 1 || validBits != 4) {

return STATUS\_ERROR;

}

if (cmdBuffer[0] != MF\_ACK) {

return STATUS\_MIFARE\_NACK;

}

return STATUS\_OK;

} // End PCD\_MIFARE\_Transceive()

/\*\*

\* Returns a \_\_FlashStringHelper pointer to a status code name.

\*

\* @return const \_\_FlashStringHelper \*

\*/

const \_\_FlashStringHelper \*MFRC522::GetStatusCodeName(MFRC522::StatusCode code ///< One of the StatusCode enums.

) {

switch (code) {

case STATUS\_OK: return F("Success.");

case STATUS\_ERROR: return F("Error in communication.");

case STATUS\_COLLISION: return F("Collission detected.");

case STATUS\_TIMEOUT: return F("Timeout in communication.");

case STATUS\_NO\_ROOM: return F("A buffer is not big enough.");

case STATUS\_INTERNAL\_ERROR: return F("Internal error in the code. Should not happen.");

case STATUS\_INVALID: return F("Invalid argument.");

case STATUS\_CRC\_WRONG: return F("The CRC\_A does not match.");

case STATUS\_MIFARE\_NACK: return F("A MIFARE PICC responded with NAK.");

default: return F("Unknown error");

}

} // End GetStatusCodeName()

/\*\*

\* Translates the SAK (Select Acknowledge) to a PICC type.

\*

\* @return PICC\_Type

\*/

MFRC522::PICC\_Type MFRC522::PICC\_GetType(byte sak ///< The SAK byte returned from PICC\_Select().

) {

// http://www.nxp.com/documents/application\_note/AN10833.pdf

// 3.2 Coding of Select Acknowledge (SAK)

// ignore 8-bit (iso14443 starts with LSBit = bit 1)

// fixes wrong type for manufacturer Infineon (http://nfc-tools.org/index.php?title=ISO14443A)

sak &= 0x7F;

switch (sak) {

case 0x04: return PICC\_TYPE\_NOT\_COMPLETE; // UID not complete

case 0x09: return PICC\_TYPE\_MIFARE\_MINI;

case 0x08: return PICC\_TYPE\_MIFARE\_1K;

case 0x18: return PICC\_TYPE\_MIFARE\_4K;

case 0x00: return PICC\_TYPE\_MIFARE\_UL;

case 0x10:

case 0x11: return PICC\_TYPE\_MIFARE\_PLUS;

case 0x01: return PICC\_TYPE\_TNP3XXX;

case 0x20: return PICC\_TYPE\_ISO\_14443\_4;

case 0x40: return PICC\_TYPE\_ISO\_18092;

default: return PICC\_TYPE\_UNKNOWN;

}

} // End PICC\_GetType()

/\*\*

\* Returns a \_\_FlashStringHelper pointer to the PICC type name.

\*

\* @return const \_\_FlashStringHelper \*

\*/

const \_\_FlashStringHelper \*MFRC522::PICC\_GetTypeName(PICC\_Type piccType ///< One of the PICC\_Type enums.

) {

switch (piccType) {

case PICC\_TYPE\_ISO\_14443\_4: return F("PICC compliant with ISO/IEC 14443-4");

case PICC\_TYPE\_ISO\_18092: return F("PICC compliant with ISO/IEC 18092 (NFC)");

case PICC\_TYPE\_MIFARE\_MINI: return F("MIFARE Mini, 320 bytes");

case PICC\_TYPE\_MIFARE\_1K: return F("MIFARE 1KB");

case PICC\_TYPE\_MIFARE\_4K: return F("MIFARE 4KB");

case PICC\_TYPE\_MIFARE\_UL: return F("MIFARE Ultralight or Ultralight C");

case PICC\_TYPE\_MIFARE\_PLUS: return F("MIFARE Plus");

case PICC\_TYPE\_TNP3XXX: return F("MIFARE TNP3XXX");

case PICC\_TYPE\_NOT\_COMPLETE: return F("SAK indicates UID is not complete.");

case PICC\_TYPE\_UNKNOWN:

default: return F("Unknown type");

}

} // End PICC\_GetTypeName()

/\*\*

\* Dumps debug info about the connected PCD to Serial.

\* Shows all known firmware versions

\*/

void MFRC522::PCD\_DumpVersionToSerial() {

// Get the MFRC522 firmware version

byte v = PCD\_ReadRegister(VersionReg);

Serial.print(F("Firmware Version: 0x"));

Serial.print(v, HEX);

// Lookup which version

switch(v) {

case 0x88: Serial.println(F(" = (clone)")); break;

case 0x90: Serial.println(F(" = v0.0")); break;

case 0x91: Serial.println(F(" = v1.0")); break;

case 0x92: Serial.println(F(" = v2.0")); break;

default: Serial.println(F(" = (unknown)"));

}

// When 0x00 or 0xFF is returned, communication probably failed

if ((v == 0x00) || (v == 0xFF))

Serial.println(F("WARNING: Communication failure, is the MFRC522 properly connected?"));

} // End PCD\_DumpVersionToSerial()

/\*\*

\* 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().

) {

MIFARE\_Key key;

// Dump UID, SAK and Type

PICC\_DumpDetailsToSerial(uid);

// Dump contents

PICC\_Type piccType = PICC\_GetType(uid->sak);

switch (piccType) {

case PICC\_TYPE\_MIFARE\_MINI:

case PICC\_TYPE\_MIFARE\_1K:

case PICC\_TYPE\_MIFARE\_4K:

// All keys are set to FFFFFFFFFFFFh at chip delivery from the factory.

for (byte i = 0; i < 6; i++) {

key.keyByte[i] = 0xFF;

}

PICC\_DumpMifareClassicToSerial(uid, piccType, &key);

break;

case PICC\_TYPE\_MIFARE\_UL:

PICC\_DumpMifareUltralightToSerial();

break;

case PICC\_TYPE\_ISO\_14443\_4:

case PICC\_TYPE\_ISO\_18092:

case PICC\_TYPE\_MIFARE\_PLUS:

case PICC\_TYPE\_TNP3XXX:

Serial.println(F("Dumping memory contents not implemented for that PICC type."));

break;

case PICC\_TYPE\_UNKNOWN:

case PICC\_TYPE\_NOT\_COMPLETE:

default:

break; // No memory dump here

}

Serial.println();

PICC\_HaltA(); // Already done if it was a MIFARE Classic PICC.

} // End PICC\_DumpToSerial()

/\*\*

\* Dumps card info (UID,SAK,Type) about the selected PICC to Serial.

\*/

void MFRC522::PICC\_DumpDetailsToSerial(Uid \*uid ///< Pointer to Uid struct returned from a successful PICC\_Select().

) {

// UID

Serial.print(F("Card UID:"));

for (byte i = 0; i < uid->size; i++) {

if(uid->uidByte[i] < 0x10)

Serial.print(F(" 0"));

else

Serial.print(F(" "));

Serial.print(uid->uidByte[i], HEX);

}

Serial.println();

// SAK

Serial.print(F("Card SAK: "));

if(uid->sak < 0x10)

Serial.print(F("0"));

Serial.println(uid->sak, HEX);

// (suggested) PICC type

PICC\_Type piccType = PICC\_GetType(uid->sak);

Serial.print(F("PICC type: "));

Serial.println(PICC\_GetTypeName(piccType));

} // End PICC\_DumpDetailsToSerial()

/\*\*

\* Dumps memory contents of a MIFARE Classic PICC.

\* On success the PICC is halted after dumping the data.

\*/

void MFRC522::PICC\_DumpMifareClassicToSerial( Uid \*uid, ///< Pointer to Uid struct returned from a successful PICC\_Select().

PICC\_Type piccType, ///< One of the PICC\_Type enums.

MIFARE\_Key \*key ///< Key A used for all sectors.

) {

byte no\_of\_sectors = 0;

switch (piccType) {

case PICC\_TYPE\_MIFARE\_MINI:

// Has 5 sectors \* 4 blocks/sector \* 16 bytes/block = 320 bytes.

no\_of\_sectors = 5;

break;

case PICC\_TYPE\_MIFARE\_1K:

// Has 16 sectors \* 4 blocks/sector \* 16 bytes/block = 1024 bytes.

no\_of\_sectors = 16;

break;

case PICC\_TYPE\_MIFARE\_4K:

// Has (32 sectors \* 4 blocks/sector + 8 sectors \* 16 blocks/sector) \* 16 bytes/block = 4096 bytes.

no\_of\_sectors = 40;

break;

default: // Should not happen. Ignore.

break;

}

// Dump sectors, highest address first.

if (no\_of\_sectors) {

Serial.println(F("Sector Block 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 AccessBits"));

for (int8\_t i = no\_of\_sectors - 1; i >= 0; i--) {

PICC\_DumpMifareClassicSectorToSerial(uid, key, i);

}

}

PICC\_HaltA(); // Halt the PICC before stopping the encrypted session.

PCD\_StopCrypto1();

} // End PICC\_DumpMifareClassicToSerial()

/\*\*

\* 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.

) {

MFRC522::StatusCode status;

byte firstBlock; // Address of lowest address to dump actually last block dumped)

byte no\_of\_blocks; // Number of blocks in sector

bool isSectorTrailer; // Set to true while handling the "last" (ie highest address) in the sector.

// 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 nible cx and an inverted nible cx\_.

byte c1, c2, c3; // Nibbles

byte c1\_, c2\_, c3\_; // Inverted nibbles

bool invertedError; // True if one of the inverted nibbles did not match

byte g[4]; // Access bits for each of the four groups.

byte group; // 0-3 - active group for access bits

bool firstInGroup; // True for the first block dumped in the group

// Determine position and size of sector.

if (sector < 32) { // Sectors 0..31 has 4 blocks each

no\_of\_blocks = 4;

firstBlock = sector \* no\_of\_blocks;

}

else if (sector < 40) { // Sectors 32-39 has 16 blocks each

no\_of\_blocks = 16;

firstBlock = 128 + (sector - 32) \* no\_of\_blocks;

}

else { // Illegal input, no MIFARE Classic PICC has more than 40 sectors.

return;

}

// Dump blocks, highest address first.

byte byteCount;

byte buffer[18];

byte blockAddr;

isSectorTrailer = true;

for (int8\_t blockOffset = no\_of\_blocks - 1; blockOffset >= 0; blockOffset--) {

blockAddr = firstBlock + blockOffset;

// Sector number - only on first line

if (isSectorTrailer) {

if(sector < 10)

Serial.print(F(" ")); // Pad with spaces

else

Serial.print(F(" ")); // Pad with spaces

Serial.print(sector);

Serial.print(F(" "));

}

else {

Serial.print(F(" "));

}

// Block number

if(blockAddr < 10)

Serial.print(F(" ")); // Pad with spaces

else {

if(blockAddr < 100)

Serial.print(F(" ")); // Pad with spaces

else

Serial.print(F(" ")); // Pad with spaces

}

Serial.print(blockAddr);

Serial.print(F(" "));

// Establish encrypted communications before reading the first block

if (isSectorTrailer) {

status = PCD\_Authenticate(PICC\_CMD\_MF\_AUTH\_KEY\_A, firstBlock, key, uid);

if (status != STATUS\_OK) {

Serial.print(F("PCD\_Authenticate() failed: "));

Serial.println(GetStatusCodeName(status));

return;

}

}

// Read block

byteCount = sizeof(buffer);

status = MIFARE\_Read(blockAddr, buffer, &byteCount);

if (status != STATUS\_OK) {

Serial.print(F("MIFARE\_Read() failed: "));

Serial.println(GetStatusCodeName(status));

continue;

}

// Dump data

for (byte index = 0; index < 16; index++) {

if(buffer[index] < 0x10)

Serial.print(F(" 0"));

else

Serial.print(F(" "));

Serial.print(buffer[index], HEX);

if ((index % 4) == 3) {

Serial.print(F(" "));

}

}

// Parse sector trailer data

if (isSectorTrailer) {

c1 = buffer[7] >> 4;

c2 = buffer[8] & 0xF;

c3 = buffer[8] >> 4;

c1\_ = buffer[6] & 0xF;

c2\_ = buffer[6] >> 4;

c3\_ = buffer[7] & 0xF;

invertedError = (c1 != (~c1\_ & 0xF)) || (c2 != (~c2\_ & 0xF)) || (c3 != (~c3\_ & 0xF));

g[0] = ((c1 & 1) << 2) | ((c2 & 1) << 1) | ((c3 & 1) << 0);

g[1] = ((c1 & 2) << 1) | ((c2 & 2) << 0) | ((c3 & 2) >> 1);

g[2] = ((c1 & 4) << 0) | ((c2 & 4) >> 1) | ((c3 & 4) >> 2);

g[3] = ((c1 & 8) >> 1) | ((c2 & 8) >> 2) | ((c3 & 8) >> 3);

isSectorTrailer = false;

}

// Which access group is this block in?

if (no\_of\_blocks == 4) {

group = blockOffset;

firstInGroup = true;

}

else {

group = blockOffset / 5;

firstInGroup = (group == 3) || (group != (blockOffset + 1) / 5);

}

if (firstInGroup) {

// Print access bits

Serial.print(F(" [ "));

Serial.print((g[group] >> 2) & 1, DEC); Serial.print(F(" "));

Serial.print((g[group] >> 1) & 1, DEC); Serial.print(F(" "));

Serial.print((g[group] >> 0) & 1, DEC);

Serial.print(F(" ] "));

if (invertedError) {

Serial.print(F(" Inverted access bits did not match! "));

}

}

if (group != 3 && (g[group] == 1 || g[group] == 6)) { // Not a sector trailer, a value block

int32\_t value = (int32\_t(buffer[3])<<24) | (int32\_t(buffer[2])<<16) | (int32\_t(buffer[1])<<8) | int32\_t(buffer[0]);

Serial.print(F(" Value=0x")); Serial.print(value, HEX);

Serial.print(F(" Adr=0x")); Serial.print(buffer[12], HEX);

}

Serial.println();

}

return;

} // End PICC\_DumpMifareClassicSectorToSerial()

/\*\*

\* Dumps memory contents of a MIFARE Ultralight PICC.

\*/

void MFRC522::PICC\_DumpMifareUltralightToSerial() {

MFRC522::StatusCode status;

byte byteCount;

byte buffer[18];

byte i;

Serial.println(F("Page 0 1 2 3"));

// Try the mpages of the original Ultralight. Ultralight C has more pages.

for (byte page = 0; page < 16; page +=4) { // Read returns data for 4 pages at a time.

// Read pages

byteCount = sizeof(buffer);

status = MIFARE\_Read(page, buffer, &byteCount);

if (status != STATUS\_OK) {

Serial.print(F("MIFARE\_Read() failed: "));

Serial.println(GetStatusCodeName(status));

break;

}

// Dump data

for (byte offset = 0; offset < 4; offset++) {

i = page + offset;

if(i < 10)

Serial.print(F(" ")); // Pad with spaces

else

Serial.print(F(" ")); // Pad with spaces

Serial.print(i);

Serial.print(F(" "));

for (byte index = 0; index < 4; index++) {

i = 4 \* offset + index;

if(buffer[i] < 0x10)

Serial.print(F(" 0"));

else

Serial.print(F(" "));

Serial.print(buffer[i], HEX);

}

Serial.println();

}

}

} // End PICC\_DumpMifareUltralightToSerial()

/\*\*

\* 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)

) {

byte c1 = ((g3 & 4) << 1) | ((g2 & 4) << 0) | ((g1 & 4) >> 1) | ((g0 & 4) >> 2);

byte c2 = ((g3 & 2) << 2) | ((g2 & 2) << 1) | ((g1 & 2) << 0) | ((g0 & 2) >> 1);

byte c3 = ((g3 & 1) << 3) | ((g2 & 1) << 2) | ((g1 & 1) << 1) | ((g0 & 1) << 0);

accessBitBuffer[0] = (~c2 & 0xF) << 4 | (~c1 & 0xF);

accessBitBuffer[1] = c1 << 4 | (~c3 & 0xF);

accessBitBuffer[2] = c3 << 4 | c2;

} // End MIFARE\_SetAccessBits()

/\*\*

\* Performs the "magic sequence" needed to get Chinese UID changeable

\* Mifare cards to allow writing to sector 0, where the card UID is stored.

\*

\* Note that you do not need to have selected the card through REQA or WUPA,

\* this sequence works immediately when the card is in the reader vicinity.

\* This means you can use this method even on "bricked" cards that your reader does

\* not recognise anymore (see MFRC522::MIFARE\_UnbrickUidSector).

\*

\* Of course with non-bricked devices, you're free to select them before calling this function.

\*/

bool MFRC522::MIFARE\_OpenUidBackdoor(bool logErrors) {

// Magic sequence:

// > 50 00 57 CD (HALT + CRC)

// > 40 (7 bits only)

// < A (4 bits only)

// > 43

// < A (4 bits only)

// Then you can write to sector 0 without authenticating

PICC\_HaltA(); // 50 00 57 CD

byte cmd = 0x40;

byte validBits = 7; /\* Our command is only 7 bits. After receiving card response,

this will contain amount of valid response bits. \*/

byte response[32]; // Card's response is written here

byte received;

MFRC522::StatusCode status = PCD\_TransceiveData(&cmd, (byte)1, response, &received, &validBits, (byte)0, false); // 40

if(status != STATUS\_OK) {

if(logErrors) {

Serial.println(F("Card did not respond to 0x40 after HALT command. Are you sure it is a UID changeable one?"));

Serial.print(F("Error name: "));

Serial.println(GetStatusCodeName(status));

}

return false;

}

if (received != 1 || response[0] != 0x0A) {

if (logErrors) {

Serial.print(F("Got bad response on backdoor 0x40 command: "));

Serial.print(response[0], HEX);

Serial.print(F(" ("));

Serial.print(validBits);

Serial.print(F(" valid bits)\r\n"));

}

return false;

}

cmd = 0x43;

validBits = 8;

status = PCD\_TransceiveData(&cmd, (byte)1, response, &received, &validBits, (byte)0, false); // 43

if(status != STATUS\_OK) {

if(logErrors) {

Serial.println(F("Error in communication at command 0x43, after successfully executing 0x40"));

Serial.print(F("Error name: "));

Serial.println(GetStatusCodeName(status));

}

return false;

}

if (received != 1 || response[0] != 0x0A) {

if (logErrors) {

Serial.print(F("Got bad response on backdoor 0x43 command: "));

Serial.print(response[0], HEX);

Serial.print(F(" ("));

Serial.print(validBits);

Serial.print(F(" valid bits)\r\n"));

}

return false;

}

// You can now write to sector 0 without authenticating!

return true;

} // End MIFARE\_OpenUidBackdoor()

/\*\*

\* Reads entire block 0, including all manufacturer data, and overwrites

\* that block with the new UID, a freshly calculated BCC, and the original

\* manufacturer data.

\*

\* It assumes a default KEY A of 0xFFFFFFFFFFFF.

\* Make sure to have selected the card before this function is called.

\*/

bool MFRC522::MIFARE\_SetUid(byte \*newUid, byte uidSize, bool logErrors) {

// UID + BCC byte can not be larger than 16 together

if (!newUid || !uidSize || uidSize > 15) {

if (logErrors) {

Serial.println(F("New UID buffer empty, size 0, or size > 15 given"));

}

return false;

}

// Authenticate for reading

MIFARE\_Key key = {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF};

MFRC522::StatusCode status = PCD\_Authenticate(MFRC522::PICC\_CMD\_MF\_AUTH\_KEY\_A, (byte)1, &key, &uid);

if (status != STATUS\_OK) {

if (status == STATUS\_TIMEOUT) {

// We get a read timeout if no card is selected yet, so let's select one

// Wake the card up again if sleeping

// byte atqa\_answer[2];

// byte atqa\_size = 2;

// PICC\_WakeupA(atqa\_answer, &atqa\_size);

if (!PICC\_IsNewCardPresent() || !PICC\_ReadCardSerial()) {

Serial.println(F("No card was previously selected, and none are available. Failed to set UID."));

return false;

}

status = PCD\_Authenticate(MFRC522::PICC\_CMD\_MF\_AUTH\_KEY\_A, (byte)1, &key, &uid);

if (status != STATUS\_OK) {

// We tried, time to give up

if (logErrors) {

Serial.println(F("Failed to authenticate to card for reading, could not set UID: "));

Serial.println(GetStatusCodeName(status));

}

return false;

}

}

else {

if (logErrors) {

Serial.print(F("PCD\_Authenticate() failed: "));

Serial.println(GetStatusCodeName(status));

}

return false;

}

}

// Read block 0

byte block0\_buffer[18];

byte byteCount = sizeof(block0\_buffer);

status = MIFARE\_Read((byte)0, block0\_buffer, &byteCount);

if (status != STATUS\_OK) {

if (logErrors) {

Serial.print(F("MIFARE\_Read() failed: "));

Serial.println(GetStatusCodeName(status));

Serial.println(F("Are you sure your KEY A for sector 0 is 0xFFFFFFFFFFFF?"));

}

return false;

}

// Write new UID to the data we just read, and calculate BCC byte

byte bcc = 0;

for (uint8\_t i = 0; i < uidSize; i++) {

block0\_buffer[i] = newUid[i];

bcc ^= newUid[i];

}

// Write BCC byte to buffer

block0\_buffer[uidSize] = bcc;

// Stop encrypted traffic so we can send raw bytes

PCD\_StopCrypto1();

// Activate UID backdoor

if (!MIFARE\_OpenUidBackdoor(logErrors)) {

if (logErrors) {

Serial.println(F("Activating the UID backdoor failed."));

}

return false;

}

// Write modified block 0 back to card

status = MIFARE\_Write((byte)0, block0\_buffer, (byte)16);

if (status != STATUS\_OK) {

if (logErrors) {

Serial.print(F("MIFARE\_Write() failed: "));

Serial.println(GetStatusCodeName(status));

}

return false;

}

// Wake the card up again

byte atqa\_answer[2];

byte atqa\_size = 2;

PICC\_WakeupA(atqa\_answer, &atqa\_size);

return true;

}

/\*\*

\* Resets entire sector 0 to zeroes, so the card can be read again by readers.

\*/

bool MFRC522::MIFARE\_UnbrickUidSector(bool logErrors) {

MIFARE\_OpenUidBackdoor(logErrors);

byte block0\_buffer[] = {0x01, 0x02, 0x03, 0x04, 0x04, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};

// Write modified block 0 back to card

MFRC522::StatusCode status = MIFARE\_Write((byte)0, block0\_buffer, (byte)16);

if (status != STATUS\_OK) {

if (logErrors) {

Serial.print(F("MIFARE\_Write() failed: "));

Serial.println(GetStatusCodeName(status));

}

return false;

}

return true;

}

/////////////////////////////////////////////////////////////////////////////////////

// Convenience functions - does not add extra functionality

/////////////////////////////////////////////////////////////////////////////////////

/\*\*

\* 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.

\*

\* @return bool

\*/

bool MFRC522::PICC\_IsNewCardPresent() {

byte bufferATQA[2];

byte bufferSize = sizeof(bufferATQA);

MFRC522::StatusCode result = PICC\_RequestA(bufferATQA, &bufferSize);

return (result == STATUS\_OK || result == STATUS\_COLLISION);

} // End PICC\_IsNewCardPresent()

/\*\*

\* Simple wrapper around PICC\_Select.

\* Returns true if a UID could be read.

\* Remember to call PICC\_IsNewCardPresent(), PICC\_RequestA() or PICC\_WakeupA() first.

\* The read UID is available in the class variable uid.

\*

\* @return bool

\*/

bool MFRC522::PICC\_ReadCardSerial() {

MFRC522::StatusCode result = PICC\_Select(&uid);

return (result == STATUS\_OK);

} // End

#define COMMON\_ANODE

#ifdef COMMON\_ANODE

#define LED\_ON LOW

#define LED\_OFF HIGH

#else

#define LED\_ON HIGH

#define LED\_OFF LOW

#endif

#define whiteLed 3

#define yellowLed 4

#define blueLed 5

#define greenLed 6

#define redLed 7

//#define wipeB 8 // Button pin for WipeMode

boolean match = false; // initialize card match to false

boolean programMode = false; // initialize programming mode to false

boolean replaceMaster = false;

uint8\_t successRead; // Variable integer to keep if we have Successful Read from Reader

byte storedCard[4]; // Stores an ID read from EEPROM

byte readCard[4]; // Stores scanned ID read from RFID Module

byte masterCard[4]; // Stores master card's ID read from EEPROM

int cardCount=0;

int cardTrack=0;

int secondFactor[6]={0,0,0,0,0,0};

// Create MFRC522 instance.

#define SS\_PIN 10

#define RST\_PIN 9

MFRC522 mfrc522(SS\_PIN, RST\_PIN);

void ShowReaderDetails();

uint8\_t getID();

boolean findID( byte find[] );

void writeID( byte a[] );

///////////////////////////////////////// Setup ///////////////////////////////////

void setup() {

//Arduino Pin Configuration

pinMode(redLed, OUTPUT);

pinMode(yellowLed,OUTPUT);

pinMode(blueLed, OUTPUT);

pinMode(greenLed, OUTPUT);

pinMode(whiteLed, OUTPUT);

//Be careful how relay circuit behave on while resetting or power-cycling your Arduino

digitalWrite(redLed, LED\_OFF); // Make sure led is off

digitalWrite(greenLed, LED\_OFF); // Make sure led is off

digitalWrite(blueLed, LED\_OFF); // Make sure led is off

digitalWrite(yellowLed, LED\_OFF); // Make sure led is off

digitalWrite(whiteLed, LED\_OFF); // Make sure led is off

//Protocol Configuration

Serial.begin(9600); // Initialize serial communications with PC

SPI.begin(); // MFRC522 Hardware uses SPI protocol

mfrc522.PCD\_Init(); // Initialize MFRC522 Hardware

//If you set Antenna Gain to Max it will increase reading distance

//mfrc522.PCD\_SetAntennaGain(mfrc522.RxGain\_max);

Serial.println(F("Ajiozi Access Control")); // For debugging purposes

ShowReaderDetails(); // Show details of PCD - MFRC522 Card Reader details

}

///////////////////////////////////////// Main Loop ///////////////////////////////////

void loop () {

do {

successRead = getID(); // sets successRead to 1 when we get read from reader otherwise

if(secondFactor[5]<1){

digitalWrite(redLed, LOW); // Make sure led is off

digitalWrite(greenLed, LOW); // Make sure led is off

digitalWrite(blueLed, LOW); // Make sure led is off

digitalWrite(yellowLed, LOW); // Make sure led is off

digitalWrite(whiteLed, LOW); // Make sure led is off

secondFactor[5]=1;

}

}

while (!successRead); //the program will not go further while you are not getting a successful read

if ( findID(readCard)) { // If scanned card is known delete it

Serial.println(F("I know this PICC, Ignoring..."));

Serial.println(cardTrack);

if(cardTrack==6){

Serial.println("Red Light");

if(secondFactor[0]<1){

secondFactor[0]++;digitalWrite(redLed,HIGH);

}else if(secondFactor[0]>0){

digitalWrite(redLed,LOW);secondFactor[0]=0;

}

}

if(cardTrack==10){

Serial.println("Yellow Light");

if(secondFactor[1]<1){

secondFactor[1]++;digitalWrite(yellowLed,HIGH);

}else if(secondFactor[1]>0){

digitalWrite(yellowLed,LOW);secondFactor[1]=0;

}

}

if(cardTrack==14){

Serial.println("Blue Light");

if(secondFactor[2]<1){

secondFactor[2]++;digitalWrite(blueLed,HIGH);

}else if(secondFactor[2]>0){

digitalWrite(blueLed,LOW);secondFactor[2]=0;

}

}

if(cardTrack==18){

Serial.println("Green Light");

if(secondFactor[3]<1){

secondFactor[3]++;digitalWrite(greenLed,HIGH);

}else if(secondFactor[3]>0){

digitalWrite(greenLed,LOW);secondFactor[3]=0;

}

}

if(cardTrack==22){

Serial.println("White Light");

if(secondFactor[4]<1){

secondFactor[4]++;digitalWrite(whiteLed,HIGH);

}else if(secondFactor[4]>0){

digitalWrite(whiteLed,LOW);secondFactor[4]=0;

}

}

else { // If scanned card is not known add it

if(cardCount<5){

Serial.println(F("I do not know this PICC, adding..."));

writeID(readCard);

}else{

Serial.println(F("Only Five Machine is allowed for control"));

//denied();

}

}

}

}

///////////////////////////////////////// Access Granted ///////////////////////////////////

void granted ( uint16\_t setDelay) {

digitalWrite(blueLed, LED\_OFF); // Turn off blue LED

digitalWrite(redLed, LED\_OFF); // Turn off red LED

digitalWrite(greenLed, LED\_OFF); // Turn on green LED

delay(setDelay); // Hold door lock open for given seconds

delay(1000); // Hold green LED on for a second

}

///////////////////////////////////////// Access Denied ///////////////////////////////////

void denied() {

digitalWrite(greenLed, LED\_OFF); // Make sure green LED is off

digitalWrite(blueLed, LED\_OFF); // Make sure blue LED is off

digitalWrite(redLed, LED\_OFF); // Turn on red LED

delay(1000);

}

///////////////////////////////////////// Get PICC's UID ///////////////////////////////////

uint8\_t getID() {

// Getting ready for Reading PICCs

if ( ! mfrc522.PICC\_IsNewCardPresent()) { //If a new PICC placed to RFID reader continue

return 0;

}

if ( ! mfrc522.PICC\_ReadCardSerial()) { //Since a PICC placed get Serial and continue

return 0;

}

// There are Mifare PICCs which have 4 byte or 7 byte UID care if you use 7 byte PICC

// I think we should assume every PICC as they have 4 byte UID

// Until we support 7 byte PICCs

Serial.println(F("Scanned PICC's UID:"));

for ( uint8\_t i = 0; i < 4; i++) { //

readCard[i] = mfrc522.uid.uidByte[i];

Serial.print(readCard[i], HEX);

}

Serial.println("");

mfrc522.PICC\_HaltA(); // Stop reading

return 1;

}

void ShowReaderDetails() {

// Get the MFRC522 software version

byte v = mfrc522.PCD\_ReadRegister(mfrc522.VersionReg);

Serial.println(F("Ajiozi Technology"));

//Serial.print(v, HEX);

if (v == 0x91)

Serial.print(F("https://www.ajiozi.com"));

else if (v == 0x92)

Serial.print(F("https://www.ajiozi.com"));

else

Serial.print(F(" (unknown),probably a chinese clone?"));

Serial.println("");

// When 0x00 or 0xFF is returned, communication probably failed

if ((v == 0x00) || (v == 0xFF)) {

Serial.println(F("WARNING: Communication failure, is the MFRC522 properly connected?"));

Serial.println(F("SYSTEM HALTED: Check connections."));

// Visualize system is halted

digitalWrite(greenLed, LED\_OFF); // Make sure green LED is off

digitalWrite(blueLed, LED\_OFF); // Make sure blue LED is off

digitalWrite(redLed, LED\_OFF); // Turn on red LED

while (true); // do not go further

}

}

///////////////////////////////////////// Cycle Leds (Program Mode) ///////////////////////////////////

void cycleLeds() {

digitalWrite(redLed, LED\_OFF); // Make sure red LED is off

digitalWrite(greenLed, LED\_ON); // Make sure green LED is on

digitalWrite(blueLed, LED\_OFF); // Make sure blue LED is off

delay(200);

digitalWrite(redLed, LED\_OFF); // Make sure red LED is off

digitalWrite(greenLed, LED\_OFF); // Make sure green LED is off

digitalWrite(blueLed, LED\_ON); // Make sure blue LED is on

delay(200);

digitalWrite(redLed, LED\_ON); // Make sure red LED is on

digitalWrite(greenLed, LED\_OFF); // Make sure green LED is off

digitalWrite(blueLed, LED\_OFF); // Make sure blue LED is off

delay(200);

}

//////////////////////////////////////// Normal Mode Led ///////////////////////////////////

void normalModeOn () {

digitalWrite(blueLed, LED\_ON); // Blue LED ON and ready to read card

digitalWrite(redLed, LED\_OFF); // Make sure Red LED is off

digitalWrite(greenLed, LED\_OFF); // Make sure Green LED is off

}

//////////////////////////////////////// Read an ID from EEPROM //////////////////////////////

void readID( uint8\_t number ) {

uint8\_t start = (number \* 4 ) + 2; // Figure out starting position

for ( uint8\_t i = 0; i < 4; i++ ) { // Loop 4 times to get the 4 Bytes

storedCard[i] = EEPROM.read(start + i); // Assign values read from EEPROM to array

cardTrack = start;

}

}

///////////////////////////////////////// Add ID to EEPROM ///////////////////////////////////

void writeID( byte a[] ) {

if ( !findID( a ) ) { // Before we write to the EEPROM, check to see if we have seen this card before!

uint8\_t num = EEPROM.read(0); // Get the number of used spaces, position 0 stores the number of ID cards

uint8\_t start = ( num \* 4 ) + 6; // Figure out where the next slot starts

num++; // Increment the counter by one

EEPROM.write( 0, num ); // Write the new count to the counter

for ( uint8\_t j = 0; j < 4; j++ ) { // Loop 4 times

EEPROM.write( start + j, a[j] ); // Write the array values to EEPROM in the right position

}

successWrite();

Serial.println(F("Succesfully added ID record to EEPROM"));

}

else {

failedWrite();

Serial.println(F("Failed! There is something wrong with ID or bad EEPROM"));

}

}

///////////////////////////////////////// Remove ID from EEPROM ///////////////////////////////////

void deleteID( byte a[] ) {

if ( !findID( a ) ) { // Before we delete from the EEPROM, check to see if we have this card!

failedWrite(); // If not

Serial.println(F("Failed! There is something wrong with ID or bad EEPROM"));

}

else {

uint8\_t num = EEPROM.read(0); // Get the numer of used spaces, position 0 stores the number of ID cards

uint8\_t slot; // Figure out the slot number of the card

uint8\_t start; // = ( num \* 4 ) + 6; // Figure out where the next slot starts

uint8\_t looping; // The number of times the loop repeats

uint8\_t j;

uint8\_t count = EEPROM.read(0); // Read the first Byte of EEPROM that stores number of cards

slot = findIDSLOT( a ); // Figure out the slot number of the card to delete

start = (slot \* 4) + 2;

looping = ((num - slot) \* 4);

num--; // Decrement the counter by one

EEPROM.write( 0, num ); // Write the new count to the counter

for ( j = 0; j < looping; j++ ) { // Loop the card shift times

EEPROM.write( start + j, EEPROM.read(start + 4 + j)); // Shift the array values to 4 places earlier in the EEPROM

}

for ( uint8\_t k = 0; k < 4; k++ ) { // Shifting loop

EEPROM.write( start + j + k, 0);

}

successDelete();

Serial.println(F("Succesfully removed ID record from EEPROM"));

}

}

///////////////////////////////////////// Check Bytes ///////////////////////////////////

boolean checkTwo ( byte a[], byte b[] ) {

if ( a[0] != 0 ) // Make sure there is something in the array first

match = true; // Assume they match at first

for ( uint8\_t k = 0; k < 4; k++ ) { // Loop 4 times

if ( a[k] != b[k] ) // IF a != b then set match = false, one fails, all fail

match = false;

}

if ( match ) { // Check to see if if match is still true

return true; // Return true

}

else {

return false; // Return false

}

}

///////////////////////////////////////// Find Slot ///////////////////////////////////

uint8\_t findIDSLOT( byte find[] ) {

uint8\_t count = EEPROM.read(0); // Read the first Byte of EEPROM to determine how many cards are already in there

for ( uint8\_t i = 1; i <= count; i++ ) { // Loop once for each EEPROM entry

readID(i); // Read an ID from EEPROM, it is stored in storedCard[4]

if ( checkTwo( find, storedCard ) ) { // Check to see if the storedCard read from EEPROM

// is the same as the find[] ID card passed

return i; // The slot number of the card

break; // Stop looking we found it

}

}

}

///////////////////////////////////////// Find ID From EEPROM ///////////////////////////////////

boolean findID( byte find[] ) {

uint8\_t count = EEPROM.read(0); // Read the first Byte of EEPROM to determine how many cards are already in there

cardCount = count;

for ( uint8\_t i = 1; i <= count; i++ ) { // Loop once for each EEPROM entry

readID(i); // Read an ID from EEPROM, it is stored in storedCard[4]

if ( checkTwo( find, storedCard ) ) { // Check to see if the storedCard read from EEPROM

return true;

break; // Stop looking we found it

}

else { // If not, return false

}

}

return false;

}

///////////////////////////////////////// Write Success to EEPROM ///////////////////////////////////

// Flashes the green LED 3 times to indicate a successful write to EEPROM

void successWrite() {

digitalWrite(blueLed, LED\_OFF); // Make sure blue LED is off

digitalWrite(redLed, LED\_OFF); // Make sure red LED is off

digitalWrite(greenLed, LED\_OFF); // Make sure green LED is on

delay(200);

digitalWrite(greenLed, LED\_ON); // Make sure green LED is on

delay(200);

digitalWrite(greenLed, LED\_OFF); // Make sure green LED is off

delay(200);

digitalWrite(greenLed, LED\_ON); // Make sure green LED is on

delay(200);

digitalWrite(greenLed, LED\_OFF); // Make sure green LED is off

delay(200);

digitalWrite(greenLed, LED\_ON); // Make sure green LED is on

delay(200);

}

///////////////////////////////////////// Write Failed to EEPROM ///////////////////////////////////

// Flashes the red LED 3 times to indicate a failed write to EEPROM

void failedWrite() {

digitalWrite(blueLed, LED\_OFF); // Make sure blue LED is off

digitalWrite(redLed, LED\_OFF); // Make sure red LED is off

digitalWrite(greenLed, LED\_OFF); // Make sure green LED is off

delay(200);

digitalWrite(redLed, LED\_ON); // Make sure red LED is on

delay(200);

digitalWrite(redLed, LED\_OFF); // Make sure red LED is off

delay(200);

digitalWrite(redLed, LED\_ON); // Make sure red LED is on

delay(200);

digitalWrite(redLed, LED\_OFF); // Make sure red LED is off

delay(200);

digitalWrite(redLed, LED\_ON); // Make sure red LED is on

delay(200);

}

///////////////////////////////////////// Success Remove UID From EEPROM ///////////////////////////////////

// Flashes the blue LED 3 times to indicate a success delete to EEPROM

void successDelete() {

digitalWrite(blueLed, LED\_OFF); // Make sure blue LED is off

digitalWrite(redLed, LED\_OFF); // Make sure red LED is off

digitalWrite(greenLed, LED\_OFF); // Make sure green LED is off

delay(200);

digitalWrite(blueLed, LED\_ON); // Make sure blue LED is on

delay(200);

digitalWrite(blueLed, LED\_OFF); // Make sure blue LED is off

delay(200);

digitalWrite(blueLed, LED\_ON); // Make sure blue LED is on

delay(200);

digitalWrite(blueLed, LED\_OFF); // Make sure blue LED is off

delay(200);

digitalWrite(blueLed, LED\_ON); // Make sure blue LED is on

delay(200);

}

////////////////////// Check readCard IF is masterCard ///////////////////////////////////

// Check to see if the ID passed is the master programing card

boolean isMaster( byte test[] ) {

if ( checkTwo( test, masterCard ) )

return true;

else

return false;

}