/\*

\* 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.

\*/

#include <Arduino.h>

#include "MFRC522.h"

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

// Functions for setting up the Arduino

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

/\*\*

\* Constructor.

\*/

MFRC522::MFRC522(): MFRC522(SS, UINT8\_MAX) { // SS is defined in pins\_arduino.h, UINT8\_MAX means there is no connection from Arduino to MFRC522's reset and power down input

} // 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). If there is no connection from the CPU to NRSTPD, set this to UINT8\_MAX. In this case, only soft reset will be used in PCD\_Init().

): 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). If there is no connection from the CPU to NRSTPD, set this to UINT8\_MAX. In this case, only soft reset will be used in PCD\_Init().

) {

\_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( PCD\_Register reg, ///< The register to write to. One of the PCD\_Register enums.

byte value ///< The value to write.

) {

SPI.beginTransaction(SPISettings(MFRC522\_SPICLOCK, MSBFIRST, SPI\_MODE0)); // Set the settings to work with SPI bus

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

SPI.transfer(reg); // 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( PCD\_Register 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(MFRC522\_SPICLOCK, MSBFIRST, SPI\_MODE0)); // Set the settings to work with SPI bus

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

SPI.transfer(reg); // 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( PCD\_Register reg ///< The register to read from. One of the PCD\_Register enums.

) {

byte value;

SPI.beginTransaction(SPISettings(MFRC522\_SPICLOCK, MSBFIRST, SPI\_MODE0)); // Set the settings to work with SPI bus

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

SPI.transfer(0x80 | reg); // 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( PCD\_Register 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; // 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(MFRC522\_SPICLOCK, 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

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

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

byte mask = (0xFF << rxAlign) & 0xFF;

// 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[0] & ~mask) | (value & mask);

index++;

}

while (index < count) {

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( PCD\_Register 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( PCD\_Register 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\_WriteRegister(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. Check for the register to

// indicate that the CRC calculation is complete in a loop. If the

// calculation is not indicated as complete in ~90ms, then time out

// the operation.

const uint32\_t deadline = millis() + 89;

do {

// DivIrqReg[7..0] bits are: Set2 reserved reserved MfinActIRq reserved CRCIRq reserved reserved

byte n = PCD\_ReadRegister(DivIrqReg);

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

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;

}

yield();

}

while (static\_cast<uint32\_t> (millis()) < deadline);

// 89ms passed and nothing happened. Communication with the MFRC522 might be down.

return STATUS\_TIMEOUT;

} // End PCD\_CalculateCRC()

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

// Functions for manipulating the MFRC522

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

/\*\*

\* Initializes the MFRC522 chip.

\*/

void MFRC522::PCD\_Init() {

bool hardReset = false;

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

pinMode(\_chipSelectPin, OUTPUT);

digitalWrite(\_chipSelectPin, HIGH);

// If a valid pin number has been set, pull device out of power down / reset state.

if (\_resetPowerDownPin != UNUSED\_PIN) {

// First set the resetPowerDownPin as digital input, to check the MFRC522 power down mode.

pinMode(\_resetPowerDownPin, INPUT);

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

pinMode(\_resetPowerDownPin, OUTPUT); // Now set the resetPowerDownPin as digital output.

digitalWrite(\_resetPowerDownPin, LOW); // Make sure we have a clean LOW state.

delayMicroseconds(2); // 8.8.1 Reset timing requirements says about 100ns. Let us be generous: 2μsl

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);

hardReset = true;

}

}

if (!hardReset) { // Perform a soft reset if we haven't triggered a hard reset above.

PCD\_Reset();

}

// Reset baud rates

PCD\_WriteRegister(TxModeReg, 0x00);

PCD\_WriteRegister(RxModeReg, 0x00);

// Reset ModWidthReg

PCD\_WriteRegister(ModWidthReg, 0x26);

// 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.

uint8\_t count = 0;

do {

// Wait for the PowerDown bit in CommandReg to be cleared (max 3x50ms)

delay(50);

} while ((PCD\_ReadRegister(CommandReg) & (1 << 4)) && (++count) < 3);

} // 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\_WriteRegister(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;

}

}

// 8. Perform a re-init, because PCD does not work after test.

// Reset does not work as expected.

// "Auto self-test done" does not work as expected.

PCD\_Init();

// Test passed; all is good.

return true;

} // End PCD\_PerformSelfTest()

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

// Power control

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

//IMPORTANT NOTE!!!!

//Calling any other function that uses CommandReg will disable soft power down mode !!!

//For more details about power control, refer to the datasheet - page 33 (8.6)

void MFRC522::PCD\_SoftPowerDown(){//Note : Only soft power down mode is available throught software

byte val = PCD\_ReadRegister(CommandReg); // Read state of the command register

val |= (1<<4);// set PowerDown bit ( bit 4 ) to 1

PCD\_WriteRegister(CommandReg, val);//write new value to the command register

}

void MFRC522::PCD\_SoftPowerUp(){

byte val = PCD\_ReadRegister(CommandReg); // Read state of the command register

val &= ~(1<<4);// set PowerDown bit ( bit 4 ) to 0

PCD\_WriteRegister(CommandReg, val);//write new value to the command register

// wait until PowerDown bit is cleared (this indicates end of wake up procedure)

const uint32\_t timeout = (uint32\_t)millis() + 500;// create timer for timeout (just in case)

while(millis()<=timeout){ // set timeout to 500 ms

val = PCD\_ReadRegister(CommandReg);// Read state of the command register

if(!(val & (1<<4))){ // if powerdown bit is 0

break;// wake up procedure is finished

}

yield();

}

}

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

// 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, ///< nullptr 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 nullptr.

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, ///< nullptr 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.

) {

// 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\_WriteRegister(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

}

// In PCD\_Init() we set the TAuto flag in TModeReg. This means the timer

// automatically starts when the PCD stops transmitting.

//

// Wait here for the command to complete. The bits specified in the

// `waitIRq` parameter define what bits constitute a completed command.

// When they are set in the ComIrqReg register, then the command is

// considered complete. If the command is not indicated as complete in

// ~36ms, then consider the command as timed out.

const uint32\_t deadline = millis() + 36;

bool completed = false;

do {

byte 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.

completed = true;

break;

}

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

return STATUS\_TIMEOUT;

}

yield();

}

while (static\_cast<uint32\_t> (millis()) < deadline);

// 36ms and nothing happened. Communication with the MFRC522 might be down.

if (!completed) {

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;

}

byte \_validBits = 0;

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

if (backData && backLen) {

byte 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 == nullptr || \*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 checkBit;

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 % 8; // The bit to modify

checkBit = (currentLevelKnownBits - 1) % 8;

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

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

}

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), nullptr, 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 == nullptr || \*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, nullptr, 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 == nullptr || 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 == nullptr || 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

{

// TODO: Fix cmdBuffer length and rxlength. They really should match.

// (Better still, rxlength should not even be necessary.)

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 == nullptr || 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("Collision 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\_MIFARE\_DESFIRE: return F("MIFARE DESFire");

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;

case 0x12: Serial.println(F(" = counterfeit chip")); 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\_MIFARE\_DESFIRE:

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;

invertedError = false; // Avoid "unused variable" warning.

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 = sizeof(response);

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, 0x08, 0x04, 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);

// Reset baud rates

PCD\_WriteRegister(TxModeReg, 0x00);

PCD\_WriteRegister(RxModeReg, 0x00);

// Reset ModWidthReg

PCD\_WriteRegister(ModWidthReg, 0x26);

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