/\*

\* Library extends MFRC522.h to support RATS for ISO-14443-4 PICC.

\* RATS - Request for Answer To Select.

\* NOTE: Please also check the comments in MFRC522Extended.h

\* @author JPG-Consulting

\*/

#include "MFRC522Extended.h"

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

// Functions for communicating with PICCs

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

/\*\*

\* 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 MFRC522Extended::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;

// IF SAK bit 6 = 1 then it is ISO/IEC 14443-4 (T=CL)

// A Request ATS command should be sent

// We also check SAK bit 3 is cero, as it stands for UID complete (1 would tell us it is incomplete)

if ((uid->sak & 0x24) == 0x20) {

Ats ats;

result = PICC\_RequestATS(&ats);

if (result == STATUS\_OK) {

// Check the ATS

if (ats.size > 0)

{

// TA1 has been transmitted?

// PPS must be supported...

if (ats.ta1.transmitted)

{

// TA1

// 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | Description

// ---+---+---+---+---+---+---+---+------------------------------------------

// 0 | - | - | - | 0 | - | - | - | Different D for each direction supported

// 1 | - | - | - | 0 | - | - | - | Only same D for both direction supported

// - | x | x | x | 0 | - | - | - | DS (Send D)

// - | - | - | - | 0 | x | x | x | DR (Receive D)

//

// D to bitrate table

// 3 | 2 | 1 | Value

// ---+---+---+-----------------------------

// 1 | - | - | 848 kBaud is supported

// - | 1 | - | 424 kBaud is supported

// - | - | 1 | 212 kBaud is supported

// 0 | 0 | 0 | Only 106 kBaud is supported

//

// Note: 106 kBaud is always supported

//

// I have almost constant timeouts when changing speeds :(

// default never used, so only delarate

//TagBitRates ds = BITRATE\_106KBITS;

//TagBitRates dr = BITRATE\_106KBITS;

TagBitRates ds;

TagBitRates dr;

//// TODO Not working at 848 or 424

//if (ats.ta1.ds & 0x04)

//{

// ds = BITRATE\_848KBITS;

//}

//else if (ats.ta1.ds & 0x02)

//{

// ds = BITRATE\_424KBITS;

//}

//else if (ats.ta1.ds & 0x01)

//{

// ds = BITRATE\_212KBITS;

//}

//else

//{

// ds = BITRATE\_106KBITS;

//}

if (ats.ta1.ds & 0x01)

{

ds = BITRATE\_212KBITS;

}

else

{

ds = BITRATE\_106KBITS;

}

//// Not working at 848 or 424

//if (ats.ta1.dr & 0x04)

//{

// dr = BITRATE\_848KBITS;

//}

//else if (ats.ta1.dr & 0x02)

//{

// dr = BITRATE\_424KBITS;

//}

//else if (ats.ta1.dr & 0x01)

//{

// dr = BITRATE\_212KBITS;

//}

//else

//{

// dr = BITRATE\_106KBITS;

//}

if (ats.ta1.dr & 0x01)

{

dr = BITRATE\_212KBITS;

}

else

{

dr = BITRATE\_106KBITS;

}

PICC\_PPS(ds, dr);

}

}

}

}

return STATUS\_OK;

} // End PICC\_Select()

/\*\*

\* Transmits a Request command for Answer To Select (ATS).

\*

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

\*/

MFRC522::StatusCode MFRC522Extended::PICC\_RequestATS(Ats \*ats)

{

// TODO unused variable

//byte count;

MFRC522::StatusCode result;

byte bufferATS[FIFO\_SIZE];

byte bufferSize = FIFO\_SIZE;

memset(bufferATS, 0, FIFO\_SIZE);

// Build command buffer

bufferATS[0] = PICC\_CMD\_RATS;

// The CID defines the logical number of the addressed card and has a range of 0

// through 14; 15 is reserved for future use (RFU).

//

// FSDI codes the maximum frame size (FSD) that the terminal can receive.

//

// FSDI | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9-F

// ------------+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----------

// FSD (bytes) | 16 | 24 | 32 | 40 | 48 | 64 | 96 | 128 | 256 | RFU > 256

//

bufferATS[1] = 0x50; // FSD=64, CID=0

// Calculate CRC\_A

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

if (result != STATUS\_OK) {

return result;

}

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

result = PCD\_TransceiveData(bufferATS, 4, bufferATS, &bufferSize, NULL, 0, true);

if (result != STATUS\_OK) {

PICC\_HaltA();

}

// Set the ats structure data

ats->size = bufferATS[0];

// T0 byte:

//

// b8 | b7 | b6 | b5 | b4 | b3 | b2 | b1 | Meaning

//----+----+----+----+----+----+----+----+---------------------------

// 0 | ...| ...| ...| ...|... | ...| ...| Set to 0 (RFU)

// 0 | 1 | x | x | ...|... | ...| ...| TC1 transmitted

// 0 | x | 1 | x | ...|... | ...| ...| TB1 transmitted

// 0 | x | x | 1 | ...|... | ...| ...| TA1 transmitted

// 0 | ...| ...| ...| x | x | x | x | Maximum frame size (FSCI)

//

// FSCI | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9-F

// ------------+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----------

// FSC (bytes) | 16 | 24 | 32 | 40 | 48 | 64 | 96 | 128 | 256 | RFU > 256

//

// Default FSCI is 2 (32 bytes)

if (ats->size > 0x01)

{

// TC1, TB1 and TA1 where NOT transmitted

ats->ta1.transmitted = (bool)(bufferATS[1] & 0x40);

ats->tb1.transmitted = (bool)(bufferATS[1] & 0x20);

ats->tc1.transmitted = (bool)(bufferATS[1] & 0x10);

// Decode FSCI

switch (bufferATS[1] & 0x0F)

{

case 0x00:

ats->fsc = 16;

break;

case 0x01:

ats->fsc = 24;

break;

case 0x02:

ats->fsc = 32;

break;

case 0x03:

ats->fsc = 40;

break;

case 0x04:

ats->fsc = 48;

break;

case 0x05:

ats->fsc = 64;

break;

case 0x06:

ats->fsc = 96;

break;

case 0x07:

ats->fsc = 128;

break;

case 0x08:

// This value cannot be hold by a byte

// The reason I ignore it is that MFRC255 FIFO is 64 bytes so this is not a possible value (or atleast it shouldn't)

//ats->fsc = 256;

break;

// TODO: What to do with RFU (Reserved for future use)?

default:

break;

}

// TA1

if (ats->ta1.transmitted)

{

ats->ta1.sameD = (bool)(bufferATS[2] & 0x80);

ats->ta1.ds = (TagBitRates)((bufferATS[2] & 0x70) >> 4);

ats->ta1.dr = (TagBitRates)(bufferATS[2] & 0x07);

}

else

{

// Default TA1

ats->ta1.ds = BITRATE\_106KBITS;

ats->ta1.dr = BITRATE\_106KBITS;

}

// TB1

if (ats->tb1.transmitted)

{

uint8\_t tb1Index = 2;

if (ats->ta1.transmitted)

tb1Index++;

ats->tb1.fwi = (bufferATS[tb1Index] & 0xF0) >> 4;

ats->tb1.sfgi = bufferATS[tb1Index] & 0x0F;

}

else

{

// Defaults for TB1

ats->tb1.fwi = 0; // TODO: Don't know the default for this!

ats->tb1.sfgi = 0; // The default value of SFGI is 0 (meaning that the card does not need any particular SFGT)

}

// TC1

if (ats->tc1.transmitted)

{

uint8\_t tc1Index = 2;

if (ats->ta1.transmitted)

tc1Index++;

if (ats->tb1.transmitted)

tc1Index++;

ats->tc1.supportsCID = (bool)(bufferATS[tc1Index] & 0x02);

ats->tc1.supportsNAD = (bool)(bufferATS[tc1Index] & 0x01);

}

else

{

// Defaults for TC1

ats->tc1.supportsCID = true;

ats->tc1.supportsNAD = false;

}

}

else

{

// TC1, TB1 and TA1 where NOT transmitted

ats->ta1.transmitted = false;

ats->tb1.transmitted = false;

ats->tc1.transmitted = false;

// Default FSCI

ats->fsc = 32; // Defaults to FSCI 2 (32 bytes)

// Default TA1

ats->ta1.sameD = false;

ats->ta1.ds = BITRATE\_106KBITS;

ats->ta1.dr = BITRATE\_106KBITS;

// Defaults for TB1

ats->tb1.transmitted = false;

ats->tb1.fwi = 0; // TODO: Don't know the default for this!

ats->tb1.sfgi = 0; // The default value of SFGI is 0 (meaning that the card does not need any particular SFGT)

// Defaults for TC1

ats->tc1.transmitted = false;

ats->tc1.supportsCID = true;

ats->tc1.supportsNAD = false;

}

memcpy(ats->data, bufferATS, bufferSize - 2);

return result;

} // End PICC\_RequestATS()

/\*\*

\* Transmits Protocol and Parameter Selection Request (PPS) without parameter 1

\*

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

\*/

MFRC522::StatusCode MFRC522Extended::PICC\_PPS()

{

StatusCode result;

byte ppsBuffer[4];

byte ppsBufferSize = 4;

// Start byte: The start byte (PPS) consists of two parts:

// –The upper nibble(b8–b5) is set to’D'to identify the PPS. All other values are RFU.

// -The lower nibble(b4–b1), which is called the ‘card identifier’ (CID), defines the logical number of the addressed card.

ppsBuffer[0] = 0xD0; // CID is hardcoded as 0 in RATS

ppsBuffer[1] = 0x00; // PPS0 indicates whether PPS1 is present

// Calculate CRC\_A

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

if (result != STATUS\_OK) {

return result;

}

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

result = PCD\_TransceiveData(ppsBuffer, 4, ppsBuffer, &ppsBufferSize, NULL, 0, true);

if (result == STATUS\_OK)

{

// Enable CRC for T=CL

byte txReg = PCD\_ReadRegister(TxModeReg) | 0x80;

byte rxReg = PCD\_ReadRegister(RxModeReg) | 0x80;

PCD\_WriteRegister(TxModeReg, txReg);

PCD\_WriteRegister(RxModeReg, rxReg);

}

return result;

} // End PICC\_PPS()

/\*\*

\* Transmits Protocol and Parameter Selection Request (PPS)

\*

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

\*/

MFRC522::StatusCode MFRC522Extended::PICC\_PPS(TagBitRates sendBitRate, ///< DS

TagBitRates receiveBitRate ///< DR

) {

StatusCode result;

// TODO not used

//byte txReg = PCD\_ReadRegister(TxModeReg) & 0x8F;

//byte rxReg = PCD\_ReadRegister(RxModeReg) & 0x8F;

byte ppsBuffer[5];

byte ppsBufferSize = 5;

// Start byte: The start byte (PPS) consists of two parts:

// –The upper nibble(b8–b5) is set to’D'to identify the PPS. All other values are RFU.

// -The lower nibble(b4–b1), which is called the ‘card identifier’ (CID), defines the logical number of the addressed card.

ppsBuffer[0] = 0xD0; // CID is hardcoded as 0 in RATS

ppsBuffer[1] = 0x11; // PPS0 indicates whether PPS1 is present

// Bit 8 - Set to '0' as MFRC522 allows different bit rates for send and receive

// Bit 4 - Set to '0' as it is Reserved for future use.

//ppsBuffer[2] = (((sendBitRate & 0x03) << 4) | (receiveBitRate & 0x03)) & 0xE7;

ppsBuffer[2] = (((sendBitRate & 0x03) << 2) | (receiveBitRate & 0x03)) & 0xE7;

// Calculate CRC\_A

result = PCD\_CalculateCRC(ppsBuffer, 3, &ppsBuffer[3]);

if (result != STATUS\_OK) {

return result;

}

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

result = PCD\_TransceiveData(ppsBuffer, 5, ppsBuffer, &ppsBufferSize, NULL, 0, true);

if (result == STATUS\_OK)

{

// Make sure it is an answer to our PPS

// We should receive our PPS byte and 2 CRC bytes

if ((ppsBufferSize == 3) && (ppsBuffer[0] == 0xD0)) {

byte txReg = PCD\_ReadRegister(TxModeReg) & 0x8F;

byte rxReg = PCD\_ReadRegister(RxModeReg) & 0x8F;

// Set bit rate and enable CRC for T=CL

txReg = (txReg & 0x8F) | ((receiveBitRate & 0x03) << 4) | 0x80;

rxReg = (rxReg & 0x8F) | ((sendBitRate & 0x03) << 4) | 0x80;

rxReg &= 0xF0; //Enforce although this should be set already

// From ConfigIsoType

//rxReg |= 0x06;

PCD\_WriteRegister(TxModeReg, txReg);

PCD\_WriteRegister(RxModeReg, rxReg);

// At 212kBps

switch (sendBitRate) {

case BITRATE\_212KBITS:

{

//PCD\_WriteRegister(ModWidthReg, 0x13);

PCD\_WriteRegister(ModWidthReg, 0x15);

}

break;

case BITRATE\_424KBITS:

{

PCD\_WriteRegister(ModWidthReg, 0x0A);

}

break;

case BITRATE\_848KBITS:

{

PCD\_WriteRegister(ModWidthReg, 0x05);

}

break;

default:

{

PCD\_WriteRegister(ModWidthReg, 0x26); // Default value

}

break;

}

//PCD\_WriteRegister(RxThresholdReg, 0x84); // ISO-14443.4 Type A (default)

//PCD\_WriteRegister(ControlReg, 0x10);

delayMicroseconds(10);

}

else

{

return STATUS\_ERROR;

}

}

return result;

} // End PICC\_PPS()

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

// Functions for communicating with ISO/IEC 14433-4 cards

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

MFRC522::StatusCode MFRC522Extended::TCL\_Transceive(PcbBlock \*send, PcbBlock \*back)

{

MFRC522::StatusCode result;

byte inBuffer[FIFO\_SIZE];

byte inBufferSize = FIFO\_SIZE;

byte outBuffer[send->inf.size + 5]; // PCB + CID + NAD + INF + EPILOGUE (CRC)

byte outBufferOffset = 1;

byte inBufferOffset = 1;

// Set the PCB byte

outBuffer[0] = send->prologue.pcb;

// Set the CID byte if available

if (send->prologue.pcb & 0x08) {

outBuffer[outBufferOffset] = send->prologue.cid;

outBufferOffset++;

}

// Set the NAD byte if available

if (send->prologue.pcb & 0x04) {

outBuffer[outBufferOffset] = send->prologue.nad;

outBufferOffset++;

}

// Copy the INF field if available

if (send->inf.size > 0) {

memcpy(&outBuffer[outBufferOffset], send->inf.data, send->inf.size);

outBufferOffset += send->inf.size;

}

// Is the CRC enabled for transmission?

byte txModeReg = PCD\_ReadRegister(TxModeReg);

if ((txModeReg & 0x80) != 0x80) {

// Calculate CRC\_A

result = PCD\_CalculateCRC(outBuffer, outBufferOffset, &outBuffer[outBufferOffset]);

if (result != STATUS\_OK) {

return result;

}

outBufferOffset += 2;

}

// Transceive the block

result = PCD\_TransceiveData(outBuffer, outBufferOffset, inBuffer, &inBufferSize);

if (result != STATUS\_OK) {

return result;

}

// We want to turn the received array back to a PcbBlock

back->prologue.pcb = inBuffer[0];

// CID byte is present?

if (send->prologue.pcb & 0x08) {

back->prologue.cid = inBuffer[inBufferOffset];

inBufferOffset++;

}

// NAD byte is present?

if (send->prologue.pcb & 0x04) {

back->prologue.nad = inBuffer[inBufferOffset];

inBufferOffset++;

}

// Check if CRC is taken care of by MFRC522

byte rxModeReg = PCD\_ReadRegister(TxModeReg);

if ((rxModeReg & 0x80) != 0x80) {

Serial.print("CRC is not taken care of by MFRC522: ");

Serial.println(rxModeReg, HEX);

// Check the CRC

// We need at least the CRC\_A value.

if ((int)(inBufferSize - inBufferOffset) < 2) {

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(inBuffer, inBufferSize - 2, controlBuffer);

if (status != STATUS\_OK) {

return status;

}

if ((inBuffer[inBufferSize - 2] != controlBuffer[0]) || (inBuffer[inBufferSize - 1] != controlBuffer[1])) {

return STATUS\_CRC\_WRONG;

}

// Take away the CRC bytes

inBufferSize -= 2;

}

// Got more data?

if (inBufferSize > inBufferOffset) {

if ((inBufferSize - inBufferOffset) > back->inf.size) {

return STATUS\_NO\_ROOM;

}

memcpy(back->inf.data, &inBuffer[inBufferOffset], inBufferSize - inBufferOffset);

back->inf.size = inBufferSize - inBufferOffset;

} else {

back->inf.size = 0;

}

// If the response is a R-Block check NACK

if (((inBuffer[0] & 0xC0) == 0x80) && (inBuffer[0] & 0x20)) {

return STATUS\_MIFARE\_NACK;

}

return result;

}

/\*\*

\* Send an I-Block (Application)

\*/

MFRC522::StatusCode MFRC522Extended::TCL\_Transceive(TagInfo \*tag, byte \*sendData, byte sendLen, byte \*backData, byte \*backLen)

{

MFRC522::StatusCode result;

PcbBlock out;

PcbBlock in;

byte outBuffer[FIFO\_SIZE];

byte outBufferSize = FIFO\_SIZE;

byte totalBackLen = \*backLen;

// This command sends an I-Block

out.prologue.pcb = 0x02;

if (tag->ats.tc1.supportsCID) {

out.prologue.pcb |= 0x08;

out.prologue.cid = 0x00; // CID is curentlly hardcoded as 0x00

}

// This command doe not support NAD

out.prologue.pcb &= 0xFB;

out.prologue.nad = 0x00;

// Set the block number

if (tag->blockNumber) {

out.prologue.pcb |= 0x01;

}

// Do we have data to send?

if (sendData && (sendLen > 0)) {

out.inf.size = sendLen;

out.inf.data = sendData;

} else {

out.inf.size = 0;

out.inf.data = NULL;

}

// Initialize the receiving data

// TODO Warning: Value escapes the local scope

in.inf.data = outBuffer;

in.inf.size = outBufferSize;

result = TCL\_Transceive(&out, &in);

if (result != STATUS\_OK) {

return result;

}

// Swap block number on success

tag->blockNumber = !tag->blockNumber;

if (backData && (backLen > 0)) {

if (\*backLen < in.inf.size)

return STATUS\_NO\_ROOM;

\*backLen = in.inf.size;

memcpy(backData, in.inf.data, in.inf.size);

}

// Check chaining

if ((in.prologue.pcb & 0x10) == 0x00)

return result;

// Result is chained

// Send an ACK to receive more data

// TODO: Should be checked I've never needed to send an ACK

while (in.prologue.pcb & 0x10) {

byte ackData[FIFO\_SIZE];

byte ackDataSize = FIFO\_SIZE;

result = TCL\_TransceiveRBlock(tag, true, ackData, &ackDataSize);

if (result != STATUS\_OK)

return result;

if (backData && (backLen > 0)) {

if ((\*backLen + ackDataSize) > totalBackLen)

return STATUS\_NO\_ROOM;

memcpy(&(backData[\*backLen]), ackData, ackDataSize);

\*backLen += ackDataSize;

}

}

return result;

} // End TCL\_Transceive()

/\*\*

\* Send R-Block to the PICC.

\*/

MFRC522::StatusCode MFRC522Extended::TCL\_TransceiveRBlock(TagInfo \*tag, bool ack, byte \*backData, byte \*backLen)

{

MFRC522::StatusCode result;

PcbBlock out;

PcbBlock in;

byte outBuffer[FIFO\_SIZE];

byte outBufferSize = FIFO\_SIZE;

// This command sends an R-Block

if (ack)

out.prologue.pcb = 0xA2; // ACK

else

out.prologue.pcb = 0xB2; // NAK

if (tag->ats.tc1.supportsCID) {

out.prologue.pcb |= 0x08;

out.prologue.cid = 0x00; // CID is curentlly hardcoded as 0x00

}

// This command doe not support NAD

out.prologue.pcb &= 0xFB;

out.prologue.nad = 0x00;

// Set the block number

if (tag->blockNumber) {

out.prologue.pcb |= 0x01;

}

// No INF data for R-Block

out.inf.size = 0;

out.inf.data = NULL;

// Initialize the receiving data

// TODO Warning: Value escapes the local scope

in.inf.data = outBuffer;

in.inf.size = outBufferSize;

result = TCL\_Transceive(&out, &in);

if (result != STATUS\_OK) {

return result;

}

// Swap block number on success

tag->blockNumber = !tag->blockNumber;

if (backData && backLen) {

if (\*backLen < in.inf.size)

return STATUS\_NO\_ROOM;

\*backLen = in.inf.size;

memcpy(backData, in.inf.data, in.inf.size);

}

return result;

} // End TCL\_TransceiveRBlock()

/\*\*

\* Send an S-Block to deselect the card.

\*/

MFRC522::StatusCode MFRC522Extended::TCL\_Deselect(TagInfo \*tag)

{

MFRC522::StatusCode result;

byte outBuffer[4];

byte outBufferSize = 1;

byte inBuffer[FIFO\_SIZE];

byte inBufferSize = FIFO\_SIZE;

outBuffer[0] = 0xC2;

if (tag->ats.tc1.supportsCID)

{

outBuffer[0] |= 0x08;

outBuffer[1] = 0x00; // CID is hardcoded

outBufferSize = 2;

}

result = PCD\_TransceiveData(outBuffer, outBufferSize, inBuffer, &inBufferSize);

if (result != STATUS\_OK) {

return result;

}

// TODO:Maybe do some checks? In my test it returns: CA 00 (Same data as I sent to my card)

return result;

} // End TCL\_Deselect()

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

// Support functions

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

/\*\*

\* Get the PICC type.

\*

\* @return PICC\_Type

\*/

MFRC522::PICC\_Type MFRC522Extended::PICC\_GetType(TagInfo \*tag ///< The TagInfo 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)

byte sak = tag->uid.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:

if (tag->atqa == 0x0344)

return PICC\_TYPE\_MIFARE\_DESFIRE;

return PICC\_TYPE\_ISO\_14443\_4;

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

default: return PICC\_TYPE\_UNKNOWN;

}

} // End PICC\_GetType()

/\*\*

\* 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 MFRC522Extended::PICC\_DumpToSerial(TagInfo \*tag)

{

MIFARE\_Key key;

// Dump UID, SAK and Type

PICC\_DumpDetailsToSerial(tag);

// Dump contents

PICC\_Type piccType = MFRC522::PICC\_GetType(tag->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(&tag->uid, piccType, &key);

break;

case PICC\_TYPE\_MIFARE\_UL:

PICC\_DumpMifareUltralightToSerial();

break;

case PICC\_TYPE\_ISO\_14443\_4:

case PICC\_TYPE\_MIFARE\_DESFIRE:

PICC\_DumpISO14443\_4(tag);

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

break;

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.

}

/\*\*

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

\*/

void MFRC522Extended::PICC\_DumpDetailsToSerial(TagInfo \*tag ///< Pointer to TagInfo struct returned from a successful PICC\_Select().

) {

// ATQA

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

if (((tag->atqa & 0xFF00u) >> 8) < 0x10)

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

Serial.print((tag->atqa & 0xFF00u) >> 8, HEX);

if ((tag->atqa & 0x00FFu) < 0x10)

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

else

Serial.print(F(" "));

Serial.println(tag->atqa & 0x00FFu, HEX);

// UID

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

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

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

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

else

Serial.print(F(" "));

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

}

Serial.println();

// SAK

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

if (tag->uid.sak < 0x10)

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

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

// (suggested) PICC type

PICC\_Type piccType = PICC\_GetType(tag);

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

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

} // End PICC\_DumpDetailsToSerial()

/\*\*

\* Dumps memory contents of a ISO-14443-4 PICC.

\*/

void MFRC522Extended::PICC\_DumpISO14443\_4(TagInfo \*tag)

{

// ATS

if (tag->ats.size > 0x00) { // The first byte is the ATS length including the length byte

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

for (byte offset = 0; offset < tag->ats.size; offset++) {

if (tag->ats.data[offset] < 0x10)

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

else

Serial.print(F(" "));

Serial.print(tag->ats.data[offset], HEX);

}

Serial.println();

}

} // End PICC\_DumpISO14443\_4

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

// 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 MFRC522Extended::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);

if (result == STATUS\_OK || result == STATUS\_COLLISION) {

tag.atqa = ((uint16\_t)bufferATQA[1] << 8) | bufferATQA[0];

tag.ats.size = 0;

tag.ats.fsc = 32; // default FSC value

// Defaults for TA1

tag.ats.ta1.transmitted = false;

tag.ats.ta1.sameD = false;

tag.ats.ta1.ds = MFRC522Extended::BITRATE\_106KBITS;

tag.ats.ta1.dr = MFRC522Extended::BITRATE\_106KBITS;

// Defaults for TB1

tag.ats.tb1.transmitted = false;

tag.ats.tb1.fwi = 0; // TODO: Don't know the default for this!

tag.ats.tb1.sfgi = 0; // The default value of SFGI is 0 (meaning that the card does not need any particular SFGT)

// Defaults for TC1

tag.ats.tc1.transmitted = false;

tag.ats.tc1.supportsCID = true;

tag.ats.tc1.supportsNAD = false;

memset(tag.ats.data, 0, FIFO\_SIZE - 2);

tag.blockNumber = false;

return true;

}

return false;

} // 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 MFRC522Extended::PICC\_ReadCardSerial() {

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

// Backward compatibility

uid.size = tag.uid.size;

uid.sak = tag.uid.sak;

memcpy(uid.uidByte, tag.uid.uidByte, sizeof(tag.uid.uidByte));

return (result == STATUS\_OK);

} // End