/\*\*

\* Library to use Arduino MFRC522 module.

\*

\* @authors Dr.Leong, Miguel Balboa, Søren Thing Andersen, Tom Clement, many more! See GitLog.

\*

\* For more information read the README.

\*

\* Please read this file for an overview and then MFRC522.cpp for comments on the specific functions.

\*/

#ifndef MFRC522\_h

#define MFRC522\_h

#include "require\_cpp11.h"

#include "deprecated.h"

// Enable integer limits

#define \_\_STDC\_LIMIT\_MACROS

#include <stdint.h>

#include <Arduino.h>

#include <SPI.h>

#ifndef MFRC522\_SPICLOCK

#define MFRC522\_SPICLOCK (4000000u) // MFRC522 accept upto 10MHz, set to 4MHz.

#endif

// Firmware data for self-test

// Reference values based on firmware version

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

//

// Version 0.0 (0x90)

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

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

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

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

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

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

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

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

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

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

};

// Version 1.0 (0x91)

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

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

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

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

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

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

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

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

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

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

};

// Version 2.0 (0x92)

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

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

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

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

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

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

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

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

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

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

};

// Clone

// Fudan Semiconductor FM17522 (0x88)

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

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

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

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

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

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

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

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

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

};

class MFRC522 {

public:

// Size of the MFRC522 FIFO

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

// Default value for unused pin

static constexpr uint8\_t UNUSED\_PIN = UINT8\_MAX;

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

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

enum PCD\_Register : byte {

// Page 0: Command and status

// 0x00 // reserved for future use

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

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

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

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

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

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

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

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

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

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

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

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

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

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

// 0x0F // reserved for future use

// Page 1: Command

// 0x10 // reserved for future use

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

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

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

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

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

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

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

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

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

// 0x1A // reserved for future use

// 0x1B // reserved for future use

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

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

// 0x1E // reserved for future use

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

// Page 2: Configuration

// 0x20 // reserved for future use

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

CRCResultRegL = 0x22 << 1,

// 0x23 // reserved for future use

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

// 0x25 // reserved for future use

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

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

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

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

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

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

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

TReloadRegL = 0x2D << 1,

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

TCounterValueRegL = 0x2F << 1,

// Page 3: Test Registers

// 0x30 // reserved for future use

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

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

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

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

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

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

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

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

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

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

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

// 0x3C // reserved for production tests

// 0x3D // reserved for production tests

// 0x3E // reserved for production tests

// 0x3F // reserved for production tests

};

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

enum PCD\_Command : byte {

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

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

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

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

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

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

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

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

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

PCD\_SoftReset = 0x0F // resets the MFRC522

};

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

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

enum PCD\_RxGain : byte {

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

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

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

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

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

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

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

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

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

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

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

};

// Commands sent to the PICC.

enum PICC\_Command : byte {

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

PICC\_CMD\_REQA = 0x26, // REQuest command, Type A. Invites PICCs in state IDLE to go to READY and prepare for anticollision or selection. 7 bit frame.

PICC\_CMD\_WUPA = 0x52, // Wake-UP command, Type A. Invites PICCs in state IDLE and HALT to go to READY(\*) and prepare for anticollision or selection. 7 bit frame.

PICC\_CMD\_CT = 0x88, // Cascade Tag. Not really a command, but used during anti collision.

PICC\_CMD\_SEL\_CL1 = 0x93, // Anti collision/Select, Cascade Level 1

PICC\_CMD\_SEL\_CL2 = 0x95, // Anti collision/Select, Cascade Level 2

PICC\_CMD\_SEL\_CL3 = 0x97, // Anti collision/Select, Cascade Level 3

PICC\_CMD\_HLTA = 0x50, // HaLT command, Type A. Instructs an ACTIVE PICC to go to state HALT.

PICC\_CMD\_RATS = 0xE0, // Request command for Answer To Reset.

// The commands used for MIFARE Classic (from http://www.mouser.com/ds/2/302/MF1S503x-89574.pdf, Section 9)

// Use PCD\_MFAuthent to authenticate access to a sector, then use these commands to read/write/modify the blocks on the sector.

// The read/write commands can also be used for MIFARE Ultralight.

PICC\_CMD\_MF\_AUTH\_KEY\_A = 0x60, // Perform authentication with Key A

PICC\_CMD\_MF\_AUTH\_KEY\_B = 0x61, // Perform authentication with Key B

PICC\_CMD\_MF\_READ = 0x30, // Reads one 16 byte block from the authenticated sector of the PICC. Also used for MIFARE Ultralight.

PICC\_CMD\_MF\_WRITE = 0xA0, // Writes one 16 byte block to the authenticated sector of the PICC. Called "COMPATIBILITY WRITE" for MIFARE Ultralight.

PICC\_CMD\_MF\_DECREMENT = 0xC0, // Decrements the contents of a block and stores the result in the internal data register.

PICC\_CMD\_MF\_INCREMENT = 0xC1, // Increments the contents of a block and stores the result in the internal data register.

PICC\_CMD\_MF\_RESTORE = 0xC2, // Reads the contents of a block into the internal data register.

PICC\_CMD\_MF\_TRANSFER = 0xB0, // Writes the contents of the internal data register to a block.

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

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

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

};

// MIFARE constants that does not fit anywhere else

enum MIFARE\_Misc {

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

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

};

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

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

enum PICC\_Type : byte {

PICC\_TYPE\_UNKNOWN ,

PICC\_TYPE\_ISO\_14443\_4 , // PICC compliant with ISO/IEC 14443-4

PICC\_TYPE\_ISO\_18092 , // PICC compliant with ISO/IEC 18092 (NFC)

PICC\_TYPE\_MIFARE\_MINI , // MIFARE Classic protocol, 320 bytes

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

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

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

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

PICC\_TYPE\_MIFARE\_DESFIRE, // MIFARE DESFire

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

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

};

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

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

enum StatusCode : byte {

STATUS\_OK , // Success

STATUS\_ERROR , // Error in communication

STATUS\_COLLISION , // Collission detected

STATUS\_TIMEOUT , // Timeout in communication.

STATUS\_NO\_ROOM , // A buffer is not big enough.

STATUS\_INTERNAL\_ERROR , // Internal error in the code. Should not happen ;-)

STATUS\_INVALID , // Invalid argument.

STATUS\_CRC\_WRONG , // The CRC\_A does not match

STATUS\_MIFARE\_NACK = 0xff // A MIFARE PICC responded with NAK.

};

// A struct used for passing the UID of a PICC.

typedef struct {

byte size; // Number of bytes in the UID. 4, 7 or 10.

byte uidByte[10];

byte sak; // The SAK (Select acknowledge) byte returned from the PICC after successful selection.

} Uid;

// A struct used for passing a MIFARE Crypto1 key

typedef struct {

byte keyByte[MF\_KEY\_SIZE];

} MIFARE\_Key;

// Member variables

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

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

// Functions for setting up the Arduino

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

MFRC522();

MFRC522(byte resetPowerDownPin);

MFRC522(byte chipSelectPin, byte resetPowerDownPin);

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

// Basic interface functions for communicating with the MFRC522

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

void PCD\_WriteRegister(PCD\_Register reg, byte value);

void PCD\_WriteRegister(PCD\_Register reg, byte count, byte \*values);

byte PCD\_ReadRegister(PCD\_Register reg);

void PCD\_ReadRegister(PCD\_Register reg, byte count, byte \*values, byte rxAlign = 0);

void PCD\_SetRegisterBitMask(PCD\_Register reg, byte mask);

void PCD\_ClearRegisterBitMask(PCD\_Register reg, byte mask);

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

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

// Functions for manipulating the MFRC522

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

void PCD\_Init();

void PCD\_Init(byte resetPowerDownPin);

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

void PCD\_Reset();

void PCD\_AntennaOn();

void PCD\_AntennaOff();

byte PCD\_GetAntennaGain();

void PCD\_SetAntennaGain(byte mask);

bool PCD\_PerformSelfTest();

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

// Power control functions

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

void PCD\_SoftPowerDown();

void PCD\_SoftPowerUp();

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

// Functions for communicating with PICCs

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

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

StatusCode PCD\_CommunicateWithPICC(byte command, byte waitIRq, byte \*sendData, byte sendLen, byte \*backData = nullptr, byte \*backLen = nullptr, byte \*validBits = nullptr, byte rxAlign = 0, bool checkCRC = false);

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

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

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

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

StatusCode PICC\_HaltA();

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

// Functions for communicating with MIFARE PICCs

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

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

void PCD\_StopCrypto1();

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

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

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

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

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

StatusCode MIFARE\_Restore(byte blockAddr);

StatusCode MIFARE\_Transfer(byte blockAddr);

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

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

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

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

// Support functions

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

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

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

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

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

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

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

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

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

// Support functions for debuging

void PCD\_DumpVersionToSerial();

void PICC\_DumpToSerial(Uid \*uid);

void PICC\_DumpDetailsToSerial(Uid \*uid);

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

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

void PICC\_DumpMifareUltralightToSerial();

// Advanced functions for MIFARE

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

bool MIFARE\_OpenUidBackdoor(bool logErrors);

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

bool MIFARE\_UnbrickUidSector(bool logErrors);

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

// Convenience functions - does not add extra functionality

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

virtual bool PICC\_IsNewCardPresent();

virtual bool PICC\_ReadCardSerial();

protected:

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

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

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

};

#endif