# MyLD2410.h

#ifndef MY\_LD2410\_H

#define MY\_LD2410\_H

/\*

MyLD2410 library

An Arduino library for the LD2410 presence sensor, including HLK-LD2410B and HLK-LD2410C.

https://github.com/iavorvel/MyLD2410

\*/

/\*\*

\* @file MyLD2410.h

\*/

#include <Arduino.h>

#define LD2410\_BAUD\_RATE 256000

#define LD2410\_BUFFER\_SIZE 0x40

#define LD2410\_LATEST\_FIRMWARE "2.44"

/\*\*

\* @brief The auxiliary light control status

\*/

enum class LightControl

{

NOT\_SET = -1,

NO\_LIGHT\_CONTROL,

LIGHT\_BELOW\_THRESHOLD,

LIGHT\_ABOVE\_THRESHOLD

};

/\*\*

\* @brief The auxiliary output control status

\*/

enum class OutputControl

{

NOT\_SET = -1,

DEFAULT\_LOW,

DEFAULT\_HIGH,

};

/\*\*

\* @brief The status of the auto-thresholds routine

\*/

enum class AutoStatus

{

NOT\_SET = -1,

NOT\_IN\_PROGRESS,

IN\_PROGRESS,

COMPLETED

};

class MyLD2410

{

public:

enum Response

{

FAIL = 0,

ACK,

DATA

};

struct ValuesArray

{

byte values[9];

byte N = 0;

void setN(byte n)

{

N = (n <= 8) ? n : 8;

}

ValuesArray &operator=(const ValuesArray &other)

{

if (this != &other)

{

N = other.N;

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

values[i] = other.values[i];

}

return \*this;

}

template <typename ByteHandler>

void forEach(ByteHandler func) const

{

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

func(values[i]);

}

};

struct SensorData

{

byte status;

unsigned long timestamp;

unsigned long mTargetDistance;

byte mTargetSignal;

unsigned long sTargetDistance;

byte sTargetSignal;

unsigned long distance;

// Enhanced data

ValuesArray mTargetSignals;

ValuesArray sTargetSignals;

};

private:

SensorData sData;

ValuesArray stationaryThresholds;

ValuesArray movingThresholds;

byte maxRange = 0;

byte noOne\_window = 0;

byte lightLevel = 0;

byte outLevel = 0;

byte lightThreshold = 0;

LightControl lightControl = LightControl::NOT\_SET;

OutputControl outputControl = OutputControl::NOT\_SET;

AutoStatus autoStatus = AutoStatus::NOT\_SET;

unsigned long version = 0;

unsigned long bufferSize = 0;

byte MAC[6];

String MACstr = "";

String firmware = "";

byte firmwareMajor = 0;

byte firmwareMinor = 0;

int fineRes = -1;

bool isEnhanced = false;

bool isConfig = false;

unsigned long timeout = 2000;

unsigned long dataLifespan = 500;

byte inBuf[LD2410\_BUFFER\_SIZE];

byte inBufI = 0;

byte headBuf[4];

byte headBufI = 0;

Stream \*sensor;

bool \_debug = false;

bool isDataValid();

bool readFrame();

bool sendCommand(const byte \*command);

bool processAck();

bool processData();

public:

/\*\*

\* @brief Construct a new MyLD2410 object

\*

\* @param serial - a reference to a stream object (sensorSerial)

\* @param debug - a flag that controls whether debug data will be sent to Serial

\*/

MyLD2410(Stream &serial, bool debug = false);

// CONTROLS

/\*\*

\* @brief Call this function in setup() to ascertain whether the device is responding

\*/

bool begin();

/\*\*

\* @brief Call this function to gracefully close the sensor. Useful for entering sleep mode.

\*/

void end();

/\*\*

\* @brief Set the debug flag

\*

\*/

void debugOn();

/\*\*

\* @brief Reset the debug flag

\*

\*/

void debugOff();

/\*\*

@brief Call this function in the main loop

@return MyLD2410::DATA = (evaluates to true) if the latest frame contained data

@return MyLD2410::ACK = (evaluates to true) if the latest frame contained a reply to a command

@return MyLD2410::FAIL = (evaluates to false) if no useful info was processed

\*/

Response check();

// GETTERS

/\*\*

\* @brief Check whether the device is in config mode

\* (accepts commands)

\*/

bool inConfigMode();

/\*\*

\* @brief Check whether the device is in basic mode

\* (continuously sends basic presence data)

\*/

bool inBasicMode();

/\*\*

\* @brief Check whether the device is in enhanced mode

\* (continuously sends enhanced presence data)

\*/

bool inEnhancedMode();

/\*\*

\* @brief Get the status of the sensor:

\* 0 - No presence;

\* 1 - Moving only;

\* 2 - Stationary only;

\* 3 - Both moving and stationary;

\* 4 - Auto thresholds in progress;

\* 5 - Auto thresholds successful;

\* 6 - Auto thresholds failed;

\* 255 - The sensor status is invalid

\*

\* @return byte

\*/

byte getStatus();

/\*\*

\* @brief Get the presence status as a c-string

\*

\* @return const char\* :

\* "No target",

\* "Moving only",

\* "Stationary only",

\* "Both moving and stationary",

\* "Auto thresholds in progress",

\* "Auto thresholds successful",

\* "Auto thresholds failed".

\*/

const char \*statusString();

/\*\*

\* @brief Check whether presence was detected in the latest frame

\*/

bool presenceDetected();

/\*\*

\* @brief Check whether a stationary target was detected in the latest frame

\*/

bool stationaryTargetDetected();

/\*\*

\* @brief Get the distance to the stationary target in [cm]

\*

\* @return unsigned long - distance in [cm]

\*/

unsigned long stationaryTargetDistance();

/\*\*

\* @brief Get the signal from the stationary target

\*

\* @return byte - signal value [0 - 100]

\*/

byte stationaryTargetSignal();

/\*\*

\* @brief Get the Stationary Signals object, if in enhanced mode

\*

\* @return const MyLD2410::ValuesArray& - the signals for each detection gate

\*/

const ValuesArray &getStationarySignals();

/\*\*

\* @brief Check whether a moving target was detected in the latest frame

\*/

bool movingTargetDetected();

/\*\*

\* @brief Get the distance to the moving target in [cm]

\*

\* @return unsigned long - distance in [cm]

\*/

unsigned long movingTargetDistance();

/\*\*

\* @brief Get the signal from the moving target

\*

\* @return byte - signal value [0 - 100]

\*/

byte movingTargetSignal();

/\*\*

\* @brief Get the Moving Signals object, if in enhanced mode

\*

\* @return const MyLD2410::ValuesArray& - the signals for each detection gate

\*/

const ValuesArray &getMovingSignals();

/\*\*

\* @brief Get the detected distance

\*

\* @return unsigned long - distance in [cm]

\*/

unsigned long detectedDistance();

/\*\*

\* @brief Get the Bluetooth MAC address as an array byte[6]

\*

\* @return const byte\*

\*/

const byte \*getMAC();

/\*\*

\* @brief Get the Bluetooth MAC address as a String

\*

\* @return String

\*/

String getMACstr();

/\*\*

\* @brief Get the Firmware as a String

\*

\* @return String

\*/

String getFirmware();

/\*\*

\* @brief Get the Firmware Major

\*

\* @return byte

\*/

byte getFirmwareMajor();

/\*\*

\* @brief Get the Firmware Minor

\*

\* @return byte

\*/

byte getFirmwareMinor();

/\*\*

\* @brief Get the protocol version

\*

\* @return unsigned long

\*/

unsigned long getVersion();

/\*\*

\* @brief Get the SensorData object

\*

\* @return const SensorData&

\*/

const SensorData &getSensorData();

/\*\*

\* @brief Get the sensor resolution (gate-width) in [cm]

\*

\* @return byte either 20 or 75 on success, 0 on failure

\*/

byte getResolution();

// parameters

/\*\*

\* @brief Get the detection thresholds for moving targets

\*

\* @return const ValuesArray&

\*/

const ValuesArray &getMovingThresholds();

/\*\*

\* @brief Get the detection thresholds for stationary targets

\*

\* @return const ValuesArray&

\*/

const ValuesArray &getStationaryThresholds();

/\*\*

\* @brief Get the maximum detection gate

\*

\* @return byte

\*/

byte getRange();

/\*\*

\* @brief Get the maximum detection range in [cm]

\*

\* @return unsigned long

\*/

unsigned long getRange\_cm();

/\*\*

\* @brief Get the time-lag of "no presence" in [s].

\* The sensor begins reporting "no presence"

\* only after no motion has been detected for that many seconds.

\*

\* @return byte

\*/

byte getNoOneWindow();

// end parameters

// REQUESTS

/\*\*

\* @brief Request config mode

\*

\* @param enable [true]/false

\* @return true on success

\*/

bool configMode(bool enable = true);

/\*\*

\* @brief Request enhanced mode

\*

\* @param enable [true]/false

\* @return true on success

\*/

bool enhancedMode(bool enable = true);

/\*\*

\* @brief Request the current auxiliary configuration

\*

\* @return true on success

\*/

bool requestAuxConfig();

/\*\*

\* @brief Begin the automatic threshold detection routine

\* (firmware >= 2.44)

\*

\* @param \_timeout - allow for timeout [s] to leave the room

\* @return true on success

\*/

bool autoThresholds(byte \_timeout = 10);

/\*\*

\* @brief Get the status of the automatic threshold detection routine

\* (firmware >= 2.44)

\*

\* @return AutoStatus

\*/

AutoStatus getAutoStatus();

/\*\*

\* @brief Request the Bluetooth MAC address

\*

\* @return true on success

\*/

bool requestMAC();

/\*\*

\* @brief Request the Firmware

\*

\* @return true on success

\*/

bool requestFirmware();

/\*\*

\* @brief Request the resolution (gate-width)

\*

\* @return true on success

\*/

bool requestResolution();

/\*\*

\* @brief Set the resolution of the sensor

\*

\* @param fine true=20cm; [false]=75cm

\* @return true on success

\*/

bool setResolution(bool fine = false);

/\*\*

\* @brief Request the sensor parameters:

\* range, motion thresholds, stationary thresholds, no-one window

\*

\* @return true on success

\*/

bool requestParameters();

/\*\*

\* @brief Set the gate parameters for a particular gate, or for all gates at once

\*

\* @param gate the gate to configure;

\* pass a value greater than 8 (e.g 0xFF) to apply the same thresholds to all gates

\* @param movingThreshold [0 - 100]

\* @param stationaryThreshold [0 - 100]

\* @return true on success

\*/

bool setGateParameters(byte gate, byte movingThreshold, byte stationaryThreshold);

/\*\*

\* @brief Set the moving target threshold for a particular gate

\*

\* @param gate the gate to configure [0 - 8]

\* @param movingThreshold [0 - 100]

\* @return true on success

\*/

bool setMovingThreshold(byte gate, byte movingThreshold);

/\*\*

\* @brief Set the stationary target threshold for a particular gate

\*

\* @param gate the gate to configure [0 - 8]

\* @param stationaryhreshold [0 - 100]

\* @return true on success

\*/

bool setStationaryThreshold(byte gate, byte stationaryThreshold);

/\*\*

\* @brief Set the parameters for all gates at once, as well as the no-one window

\*

\* @param moving\_thresholds as a ValueArray

\* @param stationary\_thresholds as a ValueArray

\* @param noOneWindow

\* @return true on success

\*/

bool

setGateParameters(const ValuesArray &moving\_thresholds, const ValuesArray &stationary\_thresholds, byte noOneWindow = 5);

/\*\*

\* @brief Set the detection range for moving targets, stationary targets, as well as the no-one window

\*

\* @param movingGate

\* @param stationaryGate

\* @param noOneWindow

\* @return true on success

\*/

bool setMaxGate(byte movingGate, byte stationaryGate, byte noOneWindow = 5);

/\*\*

\* @brief Set the no-one window parameter

\*

\* @param noOneWindow in [s]

\* @return true on success

\*/

bool setNoOneWindow(byte noOneWindow);

/\*\*

\* @brief Set the maximum moving gate

\*

\* @param movingGate

\* @return true on success

\*/

bool setMaxMovingGate(byte movingGate);

/\*\*

\* @brief Set the maximum stationary gate

\*

\* @param stationaryGate

\* @return true on success

\*/

bool setMaxStationaryGate(byte stationaryGate);

/\*\*

\* @brief Get the maximum moving gate

\*

\* @return the maximum moving-target gate

\*/

byte getMaxMovingGate();

/\*\*

\* @brief Get the maximum stationary gate

\*

\* @return the maximum stationary-target gate

\*/

byte getMaxStationaryGate();

/\*\*

\* @brief Request reset to factory default parameters

\*

\* @return true on success

\*/

bool requestReset();

/\*\*

\* @brief Request reboot

\*

\* @return true on success

\*/

bool requestReboot();

/\*\*

\* @brief Turn Bluetooth ON

\*

\* @return true on success

\*/

bool requestBTon();

/\*\*

\* @brief Turn Bluetooth OFF

\*

\* @return true on success

\*/

bool requestBToff();

/\*\*

\* @brief Set a new BT password.

\*

\* The BT password must be 6 characters long. If the string is shorter, it will be padded with spaces '\20'. If it is longer, only the first 6 characters will be used.

\*

\* @param passwd c-string

\* @return true on success

\*/

bool setBTpassword(const char \*passwd);

/\*\*

\* @brief Set a new BT password.

\*

\* The BT password must be 6 characters long. If the string is shorter, it will be padded with spaces '\20'. If it is longer, only the first 6 characters will be used.

\*

\* @param passwd Arduino String

\* @return true on success

\*/

bool setBTpassword(const String &passwd);

/\*\*

\* @brief Reset the BT password

\*

\* @return true on success

\*/

bool resetBTpassword();

/\*\*

\* @brief Reset the serial baud rate. The sensor reboots at the new rate on success

\*

\* @return true on success

\*/

bool setBaud(byte baud);

/\*\*

\* @brief Get the Light Level

\*

\* @return byte

\*/

byte getLightLevel();

/\*\*

\* @brief Get the Light Control parameter

\*

\* @return LightControl enum

\*/

LightControl getLightControl();

/\*\*

\* @brief Set the Auxiliary Control parameters

\*

\* @param light\_control

\* @param light\_threshold

\* @param output\_control

\*

\* @return true on success

\*/

bool setAuxControl(LightControl light\_control, byte light\_threshold, OutputControl output\_control);

/\*\*

\* @brief Reset the Auxiliary Control parameters to their default values

\*

\* @return true on success

\*/

bool resetAuxControl();

/\*\*

\* @brief Get the Light Threshold

\*

\* @return byte

\*/

byte getLightThreshold();

/\*\*

\* @brief Get the Output Control parameter

\*

\* @return OutputControl enum

\*/

OutputControl getOutputControl();

/\*\*

\* @brief Get the Light Level

\*

\* @return byte

\*/

byte getOutLevel();

};

#endif // MY\_LD2410\_H  
  
  
MyLD2410.cpp

#include "MyLD2410.h"

/\*\*\* BEGIN LD2410 namespace \*\*\*/

namespace LD2410

{

const char \*tStatus[7]{

"No target",

"Moving only",

"Stationary only",

"Both moving and stationary",

"Auto thresholds in progress",

"Auto thresholds successful",

"Auto thresholds failed"};

const byte headData[4]{0xF4, 0xF3, 0xF2, 0xF1};

const byte tailData[4]{0xF8, 0xF7, 0xF6, 0xF5};

const byte headConfig[4]{0xFD, 0xFC, 0xFB, 0xFA};

const byte tailConfig[4]{4, 3, 2, 1};

const byte configEnable[6]{4, 0, 0xFF, 0, 1, 0};

const byte configDisable[4]{2, 0, 0xFE, 0};

const byte MAC[6]{4, 0, 0xA5, 0, 1, 0};

const byte firmware[4]{2, 0, 0xA0, 0};

const byte res[4]{2, 0, 0xAB, 0};

const byte resCoarse[6]{4, 0, 0xAA, 0, 0, 0};

const byte resFine[6]{4, 0, 0xAA, 0, 1, 0};

const byte changeBaud[6]{4, 0, 0xA1, 0, 7, 0};

const byte reset[4]{2, 0, 0xA2, 0};

const byte reboot[4]{2, 0, 0xA3, 0};

const byte BTon[6]{4, 0, 0xA4, 0, 1, 0};

const byte BToff[6]{4, 0, 0xA4, 0, 0, 0};

const byte BTpasswd[10]{8, 0, 0xA9, 0, 0x48, 0x69, 0x4C, 0x69, 0x6E, 0x6B};

const byte param[4]{2, 0, 0x61, 0};

const byte engOn[4]{2, 0, 0x62, 0};

const byte engOff[4]{2, 0, 0x63, 0};

const byte auxQuery[4]{2, 0, 0xAE, 0};

const byte auxConfig[8]{6, 0, 0xAD, 0, 0, 0x80, 0, 0};

const byte autoBegin[6]{4, 0, 0x0B, 0, 0x0A, 0};

const byte autoQuery[4]{2, 0, 0x1B, 0};

byte gateParam[0x16]{0x14, 0, 0x64, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0};

byte maxGate[0x16]{0x14, 0, 0x60, 0, 0, 0, 8, 0, 0, 0, 1, 0, 8, 0, 0, 0, 2, 0, 5, 0, 0, 0};

String byte2hex(byte b, bool addZero = true)

{

String bStr(b, HEX);

bStr.toUpperCase();

if (addZero && (bStr.length() == 1))

return "0" + bStr;

return bStr;

}

void printBuf(const byte \*buf, byte size)

{

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

{

Serial.print(byte2hex(buf[i]));

Serial.print(' ');

}

Serial.println();

Serial.flush();

}

bool bufferEndsWith(const byte \*buf, int iMax, const byte \*other)

{

for (int j = 3; j >= 0; j--)

{

if (--iMax < 0)

iMax = 3;

if (buf[iMax] != other[j])

return false;

}

return true;

}

}

/\*\*\* END LD2410 namespace \*\*\*/

MyLD2410::Response MyLD2410::check()

{

while (sensor->available())

{

headBuf[headBufI++] = byte(sensor->read());

headBufI %= 4;

if (LD2410::bufferEndsWith(headBuf, headBufI, LD2410::headConfig) && processAck())

return ACK;

if (LD2410::bufferEndsWith(headBuf, headBufI, LD2410::headData) && processData())

return DATA;

}

return FAIL;

}

bool MyLD2410::sendCommand(const byte \*command)

{

byte size = command[0] + 2;

// LD2410::printBuf(command, size);

sensor->write(LD2410::headConfig, 4);

sensor->write(command, size);

sensor->write(LD2410::tailConfig, 4);

sensor->flush();

unsigned long giveUp = millis() + timeout;

while (millis() < giveUp)

{

while (sensor->available())

{

headBuf[headBufI++] = byte(sensor->read());

headBufI %= 4;

if (LD2410::bufferEndsWith(headBuf, headBufI, LD2410::headConfig))

return processAck();

}

}

return false;

}

bool MyLD2410::readFrame()

{

int frameSize = -1, bytes = 2;

if (bytes > 0)

{

inBufI = 0;

while (bytes)

{

if (sensor->available())

{

inBuf[inBufI++] = byte(sensor->read());

bytes--;

}

}

frameSize = 0;

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

{

frameSize |= inBuf[i] << i \* 8;

}

}

if (frameSize <= 0)

return false;

frameSize += 4;

inBufI = 0;

while (frameSize > 0)

{

if (sensor->available())

{

inBuf[inBufI++] = byte(sensor->read());

frameSize--;

}

}

return true;

}

bool MyLD2410::processAck()

{

if (!readFrame())

return false;

if (\_debug)

LD2410::printBuf(inBuf, inBufI);

if (!LD2410::bufferEndsWith(inBuf, inBufI, LD2410::tailConfig))

return false;

unsigned long command = inBuf[0] | (inBuf[1] << 8);

if (inBuf[2] | (inBuf[3] << 8))

return false;

switch (command)

{

case 0x1FF: // entered config mode

isConfig = true;

version = inBuf[4] | (inBuf[5] << 8);

bufferSize = inBuf[6] | (inBuf[7] << 8);

break;

case 0x1FE: // exited config mode

isConfig = false;

break;

case 0x1A5: // MAC

for (int i = 0; i < 6; i++)

MAC[i] = inBuf[i + 4];

MACstr = LD2410::byte2hex(MAC[0]);

for (int i = 1; i < 6; i++)

MACstr += ":" + LD2410::byte2hex(MAC[i]);

break;

case 0x1A0: // Firmware

firmware = LD2410::byte2hex(inBuf[7], false);

firmware += "." + LD2410::byte2hex(inBuf[6]);

firmware += "." + LD2410::byte2hex(inBuf[11]);

firmware += LD2410::byte2hex(inBuf[10]);

firmware += LD2410::byte2hex(inBuf[9]);

firmware += LD2410::byte2hex(inBuf[8]);

firmwareMajor = inBuf[7];

firmwareMinor = inBuf[6];

break;

case 0x1AB: // Query Resolution

fineRes = (inBuf[4]);

break;

case 0x1AE: // Query auxiliary control parameters

lightControl = LightControl(inBuf[4]);

lightThreshold = inBuf[5];

outputControl = OutputControl(inBuf[6]);

break;

case 0x11B:

autoStatus = AutoStatus(inBuf[4]);

break;

case 0x1A3: // Reboot

isEnhanced = false;

isConfig = false;

break;

case 0x161: // Query parameters

maxRange = inBuf[5];

movingThresholds.setN(inBuf[6]);

stationaryThresholds.setN(inBuf[7]);

for (byte i = 0; i <= movingThresholds.N; i++)

movingThresholds.values[i] = inBuf[8 + i];

for (byte i = 0; i <= stationaryThresholds.N; i++)

stationaryThresholds.values[i] = inBuf[17 + i];

noOne\_window = inBuf[26] | (inBuf[27] << 8);

break;

case 0x162:

isEnhanced = true;

break;

case 0x163:

isEnhanced = false;

break;

case 0x164:

if (LD2410::gateParam[7] == 0xFF)

LD2410::gateParam[7] = 0;

}

return (true);

}

bool MyLD2410::processData()

{

if (!readFrame())

return false;

if (\_debug)

LD2410::printBuf(inBuf, inBufI);

if (!LD2410::bufferEndsWith(inBuf, inBufI, LD2410::tailData))

return false;

if (((inBuf[0] == 1) || (inBuf[0] == 2)) && (inBuf[1] == 0xAA))

{ // Basic mode and Enhanced

sData.timestamp = millis();

sData.status = inBuf[2] & 7;

sData.mTargetDistance = inBuf[3] | (inBuf[4] << 8);

sData.mTargetSignal = inBuf[5];

sData.sTargetDistance = inBuf[6] | (inBuf[7] << 8);

sData.sTargetSignal = inBuf[8];

sData.distance = inBuf[9] | (inBuf[10] << 8);

if (inBuf[0] == 1)

{ // Enhanced mode only

isEnhanced = true;

sData.mTargetSignals.setN(inBuf[11]);

sData.sTargetSignals.setN(inBuf[12]);

byte \*p = inBuf + 13;

for (byte i = 0; i <= sData.mTargetSignals.N; i++)

sData.mTargetSignals.values[i] = \*(p++);

for (byte i = 0; i <= sData.sTargetSignals.N; i++)

sData.sTargetSignals.values[i] = \*(p++);

lightLevel = \*(p++);

outLevel = \*p;

}

else

{ // Basic mode only

isEnhanced = false;

sData.mTargetSignals.setN(0);

sData.sTargetSignals.setN(0);

lightLevel = 0;

outLevel = 0;

}

}

else

return false;

return true;

}

/\*\*

@brief Construct from a serial stream object

\*/

MyLD2410::MyLD2410(Stream &serial, bool debug)

{

sensor = &serial;

\_debug = debug;

}

bool MyLD2410::begin()

{

// Wait for the sensor to come online, or to timeout.

unsigned long giveUp = millis() + timeout;

bool online = false;

isConfig = false;

sendCommand(LD2410::configDisable);

while (millis() < giveUp)

{

if (check())

{

online = true;

break;

}

delay(110);

}

return online;

}

void MyLD2410::end()

{

isConfig = false;

isEnhanced = false;

}

void MyLD2410::debugOn()

{

\_debug = true;

}

void MyLD2410::debugOff()

{

\_debug = false;

}

bool MyLD2410::inConfigMode()

{

return isConfig;

}

bool MyLD2410::inBasicMode()

{

return !isEnhanced;

}

bool MyLD2410::inEnhancedMode()

{

return isEnhanced;

}

byte MyLD2410::getStatus()

{

return (isDataValid()) ? sData.status : 0xFF;

}

const char \*MyLD2410::statusString()

{

return LD2410::tStatus[sData.status];

}

bool MyLD2410::isDataValid()

{

return (millis() - sData.timestamp < dataLifespan);

}

bool MyLD2410::presenceDetected()

{

return isDataValid() && (sData.status) && (sData.status < 4); // 1,2,3

}

bool MyLD2410::stationaryTargetDetected()

{

return isDataValid() && ((sData.status == 2) || (sData.status == 3)); // 2,3

}

unsigned long MyLD2410::stationaryTargetDistance()

{

return sData.sTargetDistance;

}

byte MyLD2410::stationaryTargetSignal()

{

return sData.sTargetSignal;

}

const MyLD2410::ValuesArray &MyLD2410::getStationarySignals()

{

return sData.sTargetSignals;

}

bool MyLD2410::movingTargetDetected()

{

return isDataValid() && ((sData.status == 1) || (sData.status == 3)); // 1,3

}

unsigned long MyLD2410::movingTargetDistance()

{

return sData.mTargetDistance;

}

byte MyLD2410::movingTargetSignal()

{

return sData.mTargetSignal;

}

const MyLD2410::ValuesArray &MyLD2410::getMovingSignals()

{

return sData.mTargetSignals;

}

unsigned long MyLD2410::detectedDistance()

{

return sData.distance;

}

const byte \*MyLD2410::getMAC()

{

if (MACstr.length() == 0)

requestMAC();

return MAC;

}

String MyLD2410::getMACstr()

{

if (MACstr.length() == 0)

requestMAC();

return MACstr;

}

String MyLD2410::getFirmware()

{

if (firmware.length() == 0)

requestFirmware();

return firmware;

}

byte MyLD2410::getFirmwareMajor()

{

if (!firmwareMajor)

requestFirmware();

return firmwareMajor;

}

byte MyLD2410::getFirmwareMinor()

{

if (!firmwareMajor)

requestFirmware();

return firmwareMinor;

}

unsigned long MyLD2410::getVersion()

{

if (version == 0)

{

configMode();

configMode(false);

}

return version;

}

const MyLD2410::SensorData &MyLD2410::getSensorData()

{

return sData;

}

const MyLD2410::ValuesArray &MyLD2410::getMovingThresholds()

{

if (!maxRange)

requestParameters();

return movingThresholds;

}

const MyLD2410::ValuesArray &MyLD2410::getStationaryThresholds()

{

if (!maxRange)

requestParameters();

return stationaryThresholds;

}

byte MyLD2410::getRange()

{

if (!maxRange)

requestParameters();

return maxRange;

}

unsigned long MyLD2410::getRange\_cm()

{

return (getRange() + 1) \* getResolution();

}

byte MyLD2410::getNoOneWindow()

{

if (!maxRange)

requestParameters();

return noOne\_window;

}

bool MyLD2410::configMode(bool enable)

{

if (enable && !isConfig)

return sendCommand(LD2410::configEnable);

if (!enable && isConfig)

return sendCommand(LD2410::configDisable);

return false;

}

bool MyLD2410::enhancedMode(bool enable)

{

if (isConfig)

return sendCommand(((enable) ? LD2410::engOn : LD2410::engOff));

else

return configMode() && sendCommand(((enable) ? LD2410::engOn : LD2410::engOff)) && configMode(false);

}

bool MyLD2410::requestAuxConfig()

{

if (isConfig)

return sendCommand(LD2410::auxQuery);

return configMode() && sendCommand(LD2410::auxQuery) && configMode(false);

}

bool MyLD2410::autoThresholds(byte \_timeout)

{

byte cmd[6];

memcpy(cmd, LD2410::autoBegin, 6);

if (\_timeout)

cmd[4] = \_timeout;

if (isConfig)

return sendCommand(cmd);

return configMode() && sendCommand(cmd) && configMode(false);

}

AutoStatus MyLD2410::getAutoStatus()

{

bool res = false;

if (isConfig)

res = sendCommand(LD2410::autoQuery);

else

res = configMode() && sendCommand(LD2410::autoQuery) && configMode(false);

if (res)

return autoStatus;

return AutoStatus::NOT\_SET;

}

bool MyLD2410::requestMAC()

{

if (isConfig)

return sendCommand(LD2410::MAC);

return configMode() && sendCommand(LD2410::MAC) && configMode(false);

}

bool MyLD2410::requestFirmware()

{

if (isConfig)

return sendCommand(LD2410::firmware);

return configMode() && sendCommand(LD2410::firmware) && configMode(false);

}

bool MyLD2410::requestResolution()

{

if (isConfig)

return sendCommand(LD2410::res);

return configMode() && sendCommand(LD2410::res) && configMode(false);

}

bool MyLD2410::setResolution(bool fine)

{

if (isConfig && sendCommand(((fine) ? LD2410::resFine : LD2410::resCoarse)))

return sendCommand(LD2410::res);

return configMode() && sendCommand(((fine) ? LD2410::resFine : LD2410::resCoarse)) && sendCommand(LD2410::res) && configMode(false);

}

bool MyLD2410::requestParameters()

{

if (isConfig)

return sendCommand(LD2410::param);

return configMode() && sendCommand(LD2410::param) && configMode(false);

}

bool MyLD2410::setGateParameters(byte gate, byte movingThreshold, byte stationaryThreshold)

{

if (movingThreshold > 100)

movingThreshold = 100;

if (stationaryThreshold > 100)

stationaryThreshold = 100;

byte \*cmd = LD2410::gateParam;

if (gate > 8)

{

cmd[6] = 0xFF;

cmd[7] = 0xFF;

}

else

{

cmd[6] = gate;

cmd[7] = 0;

}

cmd[12] = movingThreshold;

cmd[18] = stationaryThreshold;

if (isConfig)

return sendCommand(cmd) && sendCommand(LD2410::param);

return configMode() && sendCommand(cmd) && sendCommand(LD2410::param) && configMode(false);

}

bool MyLD2410::setMovingThreshold(byte gate, byte movingThreshold)

{

if (gate > 8)

return false;

if (!stationaryThresholds.N)

requestParameters();

return setGateParameters(gate, movingThreshold, stationaryThresholds.values[gate]);

}

bool MyLD2410::setStationaryThreshold(byte gate, byte stationaryThreshold)

{

if (gate > 8)

return false;

if (!movingThresholds.N)

requestParameters();

return setGateParameters(gate, movingThresholds.values[gate], stationaryThreshold);

}

bool MyLD2410::setMaxGate(byte movingGate, byte staticGate, byte noOneWindow)

{

if (movingGate > 8)

movingGate = 8;

if (staticGate > 8)

staticGate = 8;

byte \*cmd = LD2410::maxGate;

cmd[6] = movingGate;

cmd[12] = staticGate;

cmd[18] = noOneWindow;

if (isConfig && sendCommand(cmd))

return sendCommand(LD2410::param);

return configMode() && sendCommand(cmd) && sendCommand(LD2410::param) && configMode(false);

}

bool MyLD2410::setGateParameters(

const ValuesArray &moving\_thresholds,

const ValuesArray &stationary\_thresholds,

byte noOneWindow)

{

if (!isConfig)

configMode();

bool success = isConfig;

if (success)

{

for (byte i = 0; i < 9; i++)

{

if (!setGateParameters(i, moving\_thresholds.values[i], stationary\_thresholds.values[i]))

{

success = false;

break;

}

delay(20);

}

}

return success && setMaxGate(moving\_thresholds.N, stationary\_thresholds.N, noOneWindow) && configMode(false);

}

bool MyLD2410::setNoOneWindow(byte noOneWindow)

{

if (!maxRange)

requestParameters();

if (noOne\_window == noOneWindow)

return true;

return setMaxGate(movingThresholds.N, stationaryThresholds.N, noOneWindow);

}

bool MyLD2410::setMaxMovingGate(byte movingGate)

{

if (!maxRange)

requestParameters();

if (movingThresholds.N == movingGate)

return true;

if (!noOne\_window)

noOne\_window = 5;

if (movingGate > 8)

movingGate = 8;

return setMaxGate(movingGate, stationaryThresholds.N, noOne\_window);

}

bool MyLD2410::setMaxStationaryGate(byte stationaryGate)

{

if (!maxRange)

requestParameters();

if (stationaryThresholds.N == stationaryGate)

return true;

if (!noOne\_window)

noOne\_window = 5;

if (stationaryGate > 8)

stationaryGate = 8;

return setMaxGate(movingThresholds.N, stationaryGate, noOne\_window);

}

byte MyLD2410::getMaxMovingGate()

{

if (!movingThresholds.N)

requestParameters();

return movingThresholds.N;

}

byte MyLD2410::getMaxStationaryGate()

{

if (!stationaryThresholds.N)

requestParameters();

return stationaryThresholds.N;

}

bool MyLD2410::requestReset()

{

if (isConfig)

return sendCommand(LD2410::reset) && sendCommand(LD2410::param) && sendCommand(LD2410::res);

return configMode() && sendCommand(LD2410::reset) && sendCommand(LD2410::param) && sendCommand(LD2410::res) && configMode(false);

}

bool MyLD2410::requestReboot()

{

if (isConfig)

return sendCommand(LD2410::reboot);

return configMode() && sendCommand(LD2410::reboot);

}

bool MyLD2410::requestBTon()

{

if (isConfig)

return sendCommand(LD2410::BTon);

return configMode() && sendCommand(LD2410::BTon) && configMode(false);

}

bool MyLD2410::requestBToff()

{

if (isConfig)

return sendCommand(LD2410::BToff);

return configMode() && sendCommand(LD2410::BToff) && configMode(false);

}

bool MyLD2410::setBTpassword(const char \*passwd)

{

byte cmd[10];

for (unsigned int i = 0; i < 4; i++)

cmd[i] = LD2410::BTpasswd[i];

for (unsigned int i = 0; i < 6; i++)

{

if (i < strlen(passwd))

cmd[4 + i] = byte(passwd[i]);

else

cmd[4 + i] = byte(' ');

}

if (isConfig)

return sendCommand(cmd);

return configMode() && sendCommand(cmd) && configMode(false);

}

bool MyLD2410::setBTpassword(const String &passwd)

{

return setBTpassword(passwd.c\_str());

}

bool MyLD2410::resetBTpassword()

{

if (isConfig)

return sendCommand(LD2410::BTpasswd);

return configMode() && sendCommand(LD2410::BTpasswd) && configMode(false);

}

bool MyLD2410::setBaud(byte baud)

{

if ((baud < 1) || (baud > 8))

return false;

byte cmd[6];

memcpy(cmd, LD2410::changeBaud, 6);

cmd[4] = baud;

if (isConfig)

return sendCommand(cmd) && requestReboot();

return configMode() && sendCommand(cmd) && requestReboot();

}

byte MyLD2410::getResolution()

{

if (fineRes >= 0)

return ((fineRes == 1) ? 20 : 75);

if (isConfig)

{

if (sendCommand(LD2410::res))

return getResolution();

}

else

{

if (configMode() && sendCommand(LD2410::res) && configMode(false))

return getResolution();

}

return 0;

}

byte MyLD2410::getLightLevel()

{

return lightLevel;

}

LightControl MyLD2410::getLightControl()

{

if (lightControl == LightControl::NOT\_SET)

requestAuxConfig();

return lightControl;

}

byte MyLD2410::getLightThreshold()

{

if (lightControl == LightControl::NOT\_SET)

requestAuxConfig();

return lightThreshold;

}

OutputControl MyLD2410::getOutputControl()

{

if (outputControl == OutputControl::NOT\_SET)

requestAuxConfig();

return outputControl;

}

bool MyLD2410::setAuxControl(

LightControl light\_control,

byte light\_threshold,

OutputControl output\_control)

{

byte cmd[8];

memcpy(cmd, LD2410::auxConfig, 8);

cmd[4] = byte(light\_control);

cmd[5] = light\_threshold;

cmd[6] = byte(output\_control);

if (isConfig)

return sendCommand(cmd) && requestAuxConfig();

return configMode() && sendCommand(cmd) && requestAuxConfig() && configMode(false);

}

bool MyLD2410::resetAuxControl()

{

if (isConfig)

return sendCommand(LD2410::auxConfig) && requestAuxConfig();

return configMode() && sendCommand(LD2410::auxConfig) && requestAuxConfig() && configMode(false);

}

byte MyLD2410::getOutLevel()

{

return outLevel;

}