#include <SoftwareSerial.h>

#include <EEPROM.h>

#include "Shared.h"

#include "Packet.h"

#include "PacketSenderReceiver.h"

/\*\*\*\*\*\* Uncomment the current slave, comment others \*\*\*\*\*\*/

//#define SLAVE\_NANO4LED\_RELAY

//#define SLAVE\_PROMINIBLUE

//#define SLAVE\_PROMINIBLACK

#define SLAVE\_PROMINIBLACK\_LEDSTRIP

/\*\*\*\*\*\* Unique for each slave \*\*\*\*\*\*/

// UNIQUE\_FACTORY\_ID: An 7-byte integer to identify each slave node on the planet. (ufid)

// DEVICE\_INFO: Non-private information about this slave.

#ifdef SLAVE\_NANO4LED\_RELAY

#define UNIQUE\_FACTORY\_ID {0xFF, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}

#define DEVICE\_TYPE {0x11, 0x0, 0x0, 0x0}

#endif

#ifdef SLAVE\_PROMINIBLUE

#define UNIQUE\_FACTORY\_ID {0xFF, 0x1, 0x0, 0x0, 0x0, 0x0, 0x0}

#define DEVICE\_TYPE {0x12, 0x0, 0x0, 0x0}

#endif

#ifdef SLAVE\_PROMINIBLACK\_LAMPS

#define UNIQUE\_FACTORY\_ID {0xFF, 0xB, 0x0, 0x0, 0x0, 0x0, 0x0}

#define DEVICE\_TYPE {0x12, 0x0, 0x0, 0x0}

#endif

#ifdef SLAVE\_PROMINIBLACK\_LEDSTRIP

#define UNIQUE\_FACTORY\_ID {0xFF, 0x2, 0x2, 0x0, 0x0, 0x0, 0x0}

#define DEVICE\_TYPE {0x12, 0x0, 0x0, 0x0}

#endif

/\*\*\*\*\*\* NOT Unique for each slave \*\*\*\*\*\*/

#define DEBUG\_PIN LED\_BUILTIN

#define STATUS\_LED\_PIN 13

// Note: HC12 TX to RX and RX to TX.

#define TX\_PIN 11

#define RX\_PIN 10

#define PROPERTY\_COUNT 64

#define MAX\_CONCURRENT\_REQUESTS 2

unsigned char getAddress();

SoftwareSerial ss = SoftwareSerial(RX\_PIN, TX\_PIN);

PacketSenderReceiver sr = PacketSenderReceiver(&ss, true, getAddress());

Packet temp;

//\_\_attribute\_\_((section(".noinit")))

unsigned char startupMode = 0;

//unsigned char properties[PROPERTY\_COUNT];

unsigned long lastLedBlink = 0;

unsigned int ledBlinks = 0;

unsigned int ledBlinkInterval = 200;

void led(int blinks, int interval = 200)

{

ledBlinks = blinks \* 2;

ledBlinkInterval = interval;

}

void setupSlave()

{

#ifdef SLAVE\_NANO4LED\_RELAY

pinMode(3, OUTPUT);

pinMode(5, OUTPUT);

pinMode(6, OUTPUT);

pinMode(9, OUTPUT);

pinMode(8, OUTPUT);

#endif

#ifdef SLAVE\_PROMINIBLUE

for (unsigned char i = 2; i <= 9; i++)

{

pinMode(i, OUTPUT);

digitalWrite(i, true);

}

#endif

#ifdef SLAVE\_PROMINIBLACK\_LAMPS

for(int i = 0; i < 8; i++)

{

pinMode(i + 2, OUTPUT);

}

#endif

#ifdef SLAVE\_PROMINIBLACK\_LEDSTRIP

pinMode(2, OUTPUT);

digitalWrite(2, false);

pinMode(3, OUTPUT);

pinMode(5, OUTPUT);

pinMode(6, OUTPUT);

#endif

}

void propertyUpdate()

{

#ifdef SLAVE\_NANO4LED\_RELAY

analogWrite(3, getProperty(0));

analogWrite(5, getProperty(1));

analogWrite(6, getProperty(2));

analogWrite(9, getProperty(3));

analogWrite(8, getProperty(4));

#endif

#ifdef SLAVE\_PROMINIBLUE

for (unsigned char i = 2; i <= 9; i++)

{

//digitalWrite(i, properties[i] > 0);

digitalWrite(i, getProperty(i) > 0);

}

#endif

#ifdef SLAVE\_PROMINIBLACK\_LAMPS

for(int i = 0; i < 8; i++)

{

digitalWrite(i + 2, getProperty(i));

}

#endif

#ifdef SLAVE\_PROMINIBLACK\_LEDSTRIP

analogWrite(3, getProperty(0));

analogWrite(5, getProperty(1));

analogWrite(6, getProperty(2));

#endif

}

unsigned char refreshLiveData(unsigned char liveData[16])

{

for(unsigned char i = 1; i <= 4; i++)

liveData[i - 1] = i;

return 4;

}

void setup()

{

pinMode(DEBUG\_PIN, OUTPUT);

pinMode(STATUS\_LED\_PIN, OUTPUT);

digitalWrite(DEBUG\_PIN, false);

digitalWrite(STATUS\_LED\_PIN, true);

Serial.begin(19200);

veryCoolSplashScreen();

Serial.print("----> My address (slave): ");

Serial.println(getAddress());

Serial.print("----> Registered: ");

Serial.println(getRegistered() ? "yeah" : "nope");

Serial.print("----> My master address: ");

Serial.println(getMaster());

Serial.print("----> Unique factory id (ufid): ");

unsigned char ufid[7] = UNIQUE\_FACTORY\_ID;

for (unsigned char i = 0; i < 7; i++)

{

Serial.print(ufid[i]);

Serial.print(' ');

}

Serial.println();

Serial.print("----> Starting mode: ");

Serial.println(++startupMode);

Serial.println("----> Starting...");

setupSlave();

propertyUpdate();

ss.begin(4800);

//setAddress(0);

if (!getRegistered())

led(5000000);

Serial.println("\t-> OK");

}

void loop()

{

if (ledBlinks > 0 && (millis() - lastLedBlink) > ledBlinkInterval)

{

digitalWrite(DEBUG\_PIN, ledBlinks % 2 == 0);

ledBlinks--;

lastLedBlink = millis();

}

if (sr.receive(&temp))

{

Serial.print("Received packet: ");

temp.printToSerial();

Serial.println();

if (temp.needsResponse())

{

Serial.println("Packet needs response");

processRequest(temp.getMaster(), temp.getData(), temp.getDataLength(), temp.getSlave() == 0);

}

}

#ifdef NANO4LED\_RELAY

if (getProperty(10) != 0)

{

analogWrite(3, random(256));

analogWrite(5, random(256));

analogWrite(6, random(256));

analogWrite(9, random(256));

}

#endif

}

void processRequest(unsigned char fromMaster, unsigned char\* data, unsigned char len, bool isBroadcast)

{

bool reg = getRegistered();

// Bind command when unregistered

if (!reg && len == 9 && data[0] == 0x10)

{

unsigned char ufid[7] = UNIQUE\_FACTORY\_ID;

for (int i = 0; i < 7; i++)

if (data[i + 1] != ufid[i])

return false;

setMaster(fromMaster);

setAddress(data[8]);

sr = PacketSenderReceiver(&ss, true, getAddress());

Serial.print("Registered master: ");

Serial.println(getMaster());

Serial.print("My address: ");

Serial.println(getAddress());

led(10, 50);

unsigned char resp[] = DEVICE\_TYPE;

sr.answer(&temp, resp, sizeof(resp));

return;

}

else if (!reg || fromMaster != getMaster() || isBroadcast)

{

if (isBroadcast)

Serial.println("Broadcast got ignored (WIP?)");

return;

}

// Set, set-range command

if (len >= 3 && data[0] == 0x20)

{

unsigned char startPos = data[1];

unsigned char propDataLen = len - 2;

if (startPos + propDataLen - 1 < PROPERTY\_COUNT)

{

//memcpy(&properties[startPos], &data[2], propDataLen);

setProperties(startPos, &data[2], propDataLen);

Serial.print(propDataLen);

Serial.print(" properties were updated: ");

for (unsigned char i = startPos; i < startPos + propDataLen; i++)

{

Serial.print('[');

Serial.print(i);

Serial.print(" = ");

//Serial.print(properties[i]);

Serial.print(getProperty(i));

Serial.print("] ");

}

Serial.println();

led(2, 25);

propertyUpdate();

unsigned char resp[] = {0xFF};

sr.answer(&temp, resp, sizeof(resp));

return;

}

}

// Ping command

else if (len == 1 && data[0] == 0x1)

{

Serial.println("<-- Me is got being pinged, yay!");

led(25, 50);

unsigned char resp[] = {0xFF};

sr.answer(&temp, resp, sizeof(resp));

return;

}

// Unbind command

else if (len == 1 && data[0] == 0x2)

{

unsigned char resp[] = {0xFF};

sr.answer(&temp, resp, sizeof(resp));

setAddress(0);

sr = PacketSenderReceiver(&ss, true, getAddress());

Serial.println("Device is now unbound.");

led(500000);

return;

}

// Bind command while registered

else if (len == 9 && data[0] == 0x10)

{

Serial.print("Device bind request while bound: ");

unsigned char ufid[7] = UNIQUE\_FACTORY\_ID;

if (memcmp(&data[1], &ufid[0], 7) == 0)

{

Serial.println("for me.");

unsigned char resp[] = DEVICE\_TYPE;

sr.answer(&temp, resp, sizeof(resp));

return;

}

else

{

Serial.println("not for me.");

}

}

// Refresh command.

else if (len == 1 && data[0] == 0x15)

{

static unsigned char resp[16];

memset(resp, 0, sizeof(resp));

unsigned char dataLen = refreshLiveData(resp);

Serial.print("Sending ");

Serial.print(dataLen);

Serial.println(" bytes of live data.");

sr.answer(&temp, resp, dataLen);

return;

}

// Mark request as failed.

unsigned char resp[] = {0x0};

sr.answer(&temp, resp, sizeof(resp));

return;

}

unsigned char getAddress()

{

return 0xFF - EEPROM.read(0);

}

bool getRegistered()

{

unsigned char s = getAddress();

return s > 0 && s < 64;

}

unsigned char getMaster()

{

return 0xFF - EEPROM.read(1);

}

void setAddress(unsigned char addr)

{

EEPROM.write(0, 0xFF - addr);

}

void setMaster(unsigned char masterAddress)

{

EEPROM.write(1, 0xFF - masterAddress);

}

unsigned char getProperty(unsigned char address)

{

return EEPROM.read(address + 100);

}

void getProperties(unsigned char address, unsigned char\* store, unsigned char len)

{

for (unsigned char i = address, j = 0; i < address + len; i++, j++)

{

store[j] = EEPROM.read(i + 100);

}

}

void setProperty(unsigned char address, unsigned char value)

{

EEPROM.update(address + 100, value);

}

void setProperties(unsigned char address, unsigned char\* values, unsigned char len)

{

for (unsigned char i = address, j = 0; i < address + len; i++, j++)

{

EEPROM.update(i + 100, values[j]);

}

}

#include "Logger.h"

#include "Arduino.h"

Logger::Logger()

{

this->enable = true;

}

void Logger::log(char\* s)

{

if (enable)

Serial.print(s);

}

void Logger::log(char s)

{

if (enable)

Serial.print(s);

}

void Logger::log(int i)

{

if (enable)

Serial.print(i, DEC);

}

void Logger::log(unsigned char i)

{

if (enable)

Serial.print(i, DEC);

}

void Logger::log(long i)

{

if (enable)

Serial.print(i, DEC);

}

void Logger::logln(char\* s)

{

if (enable)

Serial.println(s);

}

void Logger::logln(unsigned char i)

{

if (enable)

Serial.println(i, DEC);

}

void Logger::logln(int i)

{

if (enable)

Serial.println(i, DEC);

}

void Logger::logln(long i)

{

if (enable)

Serial.println(i, DEC);

}

void Logger::logln(char s)

{

if (enable)

Serial.println(s, DEC);

}

void Logger::logln()

{

if (enable)

Serial.println();

}

#ifndef Logger\_h

#define Logger\_h

#include "Arduino.h"

class Logger

{

public:

Logger();

void log(char\* s);

void log(char s);

void log(unsigned char i);

void log(int i);

void log(long i);

void logln(char\* s);

void logln(char s);

void logln(unsigned char i);

void logln(int i);

void logln(long i);

void logln();

bool enable;

};

#endif

#include "Packet.h"

#include "SoftwareSerial.h"

#include "Arduino.h"

unsigned char Packet::identifier = 0x69;

Packet::Packet()

{

memset(this->data, 0, 20);

}

Packet::Packet(unsigned char\* data, unsigned char len)

{

memset(this->data, 0, 20);

memcpy(this->data, data, len);

}

Packet::Packet(unsigned char slaveAddress, unsigned char masterAddress, unsigned char\* data,

unsigned char len, PacketType type, unsigned char multiPurposeByte)

{

memset(this->data, 0, 20);

if (len > 0)

memcpy(&this->data[4], data, len);

this->data[0] = Packet::identifier;

this->data[1] = slaveAddress & 0x3F;

this->data[2] = ((masterAddress & 0x3) << 6) | ((type & 0x3) << 4) | (len & 0xF);

this->data[3] = multiPurposeByte;

this->data[1] |= getCurrentCRC() << 6;

}

unsigned char Packet::getCurrentCRC()

{

unsigned char crc = ~Packet::identifier;

for (int i = 2; i < 20; i++)

crc ^= this->data[i];

crc ^= this->data[1] & 0x3F;

return (crc ^ (crc >> 2) ^ (crc >> 4) ^ (crc >> 6)) & 0x3;

}

bool Packet::hasValidIntegrity()

{

return this->getCurrentCRC() == this->getCRC();

}

void Packet::sendViaSoftware(SoftwareSerial\* ss)

{

ss->write(this->data, this->getDataLength() + 4);

}

void Packet::printToSerial()

{

Serial.print('[');

Serial.print(this->getType());

Serial.print(", CRC: ");

Serial.print(this->getCRC());

Serial.print(" =?= ");

Serial.print(this->getCurrentCRC());

Serial.print(", slave: ");

Serial.print(this->getSlave());

Serial.print(", master: ");

Serial.print(this->getMaster());

Serial.print(", data(");

Serial.print(this->getDataLength());

Serial.print("): ");

for (int i = 4, ii = this->getDataLength(); i < 20 && i < (ii + 4); i++)

{

Serial.print(this->data[i], DEC);

Serial.print(' ');

}

Serial.print(']');

}

char Packet::getIdentifier()

{

return this->data[0];

}

unsigned char Packet::getCRC()

{

return this->data[1] >> 6;

}

unsigned char Packet::getSlave()

{

return this->data[1] & 0x3F;

}

unsigned char Packet::getMaster()

{

return this->data[2] >> 6;

}

unsigned char Packet::getRawType()

{

return (this->data[2] >> 4) & 0x3;

}

PacketType Packet::getType()

{

return static\_cast<PacketType>(this->getRawType());

}

unsigned char Packet::getMultiPurposeByte()

{

return this->data[3];

}

unsigned char\* Packet::getData()

{

return &this->data[4];

}

unsigned char Packet::getDataLength()

{

return this->data[2] & 0xF;

}

void Packet::recalculateCRC()

{

this->data[1] |= this->getCurrentCRC() << 6;

}

bool Packet::needsResponse()

{

return this->getRawType() == 0;

}

#ifndef Packet\_h

#define Packet\_h

#include "SoftwareSerial.h"

#include "Arduino.h"

enum PacketType

{

DataRequest,

Push,

Answer,

PleaseResend

};

class Packet

{

public:

Packet();

Packet(unsigned char\* data, unsigned char len = 20);

Packet(unsigned char slaveAddress, unsigned char masterAddress, unsigned char\* data, unsigned char len, PacketType type, unsigned char multiPurposeByte = 0x0);

unsigned char data[20];

static unsigned char identifier;

void sendViaSoftware(SoftwareSerial\* ss);

void printToSerial();

char getIdentifier();

unsigned char getCRC();

unsigned char getSlave();

unsigned char getMaster();

unsigned char getRawType();

PacketType getType();

unsigned char getDataLength();

unsigned char\* getData();

unsigned char getMultiPurposeByte();

unsigned char getCurrentCRC();

bool hasValidIntegrity();

void recalculateCRC();

bool needsResponse();

};

#endif

#include "PacketSenderReceiver.h"

#include "Request.h"

#include "Packet.h"

#include "SoftwareSerial.h"

#include "Logger.h"

#include "Arduino.h"

//Request PacketSenderReceiver::nullRequest;

PacketSenderReceiver::PacketSenderReceiver(SoftwareSerial\* serial, bool isSlave, unsigned char address)

{

this->serial = serial;

this->address = address;

this->isSlave = isSlave;

this->receiving = false;

this->dataPosition = 0;

this->incomingLength = 0;

}

bool PacketSenderReceiver::receiveAny(Packet\* p)

{

if (this->receiving)

{

while (this->serial->available() > 0 && this->dataPosition < this->incomingLength)

p->data[this->dataPosition++] = this->serial->read();

if (this->dataPosition >= this->incomingLength)

{

this->receiving = false;

return true;

}

else

{

return false;

}

}

else

{

if (this->serial->available() < 4)

return false;

while (this->serial->peek() != Packet::identifier)

{

if (this->serial->available() == 0)

return false;

this->log("Out of sync: ");

this->log(this->serial->peek());

this->log(" != ");

this->logln(Packet::identifier);

this->serial->read();

}

if (this->serial->available() < 4)

return false;

this->receiving = true;

memset(p->data, 0, 20);

for (this->dataPosition = 0; this->dataPosition < 4; this->dataPosition++)

p->data[this->dataPosition] = this->serial->read();

this->incomingLength = (p->data[2] & 0xF) + 4;

if (this->incomingLength == 4)

{

this->receiving = false;

return true;

}

return false;//receiveAny(p);

}

}

bool PacketSenderReceiver::receive(Packet\* packet)

{

if (receiveAny(packet))

{

// Check if the received packet is not a broadcast.

if (!((this->isSlave && packet->getSlave() == 0x0) || (!this->isSlave && packet->getMaster() == 0x0)))

{

// Check if this packet is for me or not.

if ((this->isSlave && this->address != packet->getSlave()) || (!this->isSlave && this->address != packet->getMaster()))

{

// This packet is not for me.

this->log("Not for me. ");

this->log(this->address);

this->log(" != (master: ");

this->log(packet->getMaster());

this->log(", slave: ");

this->log(packet->getSlave());

this->logln(")");

return false;

}

}

// Ask for resend if the packet has a false integrity.

if (!packet->hasValidIntegrity() && packet->getType() != PleaseResend)

{

this->send(this->isSlave ? packet->getMaster() : packet->getSlave(), {}, 0, PleaseResend);

this->log("Faulty integrity: ");

packet->printToSerial();

this->logln();

return false;

}

// Resend packet if it is a PleaseResend packet.

if (packet->getType() == PleaseResend)

{

this->resendLastPacket();

this->log("\_");

return false;

}

if (packet->getType() == Answer)

{

/\*this->log("Received request answer: ");

packet->printToSerial();

this->logln();\*/

Request\* r = this->getRequestWithId(packet->getMultiPurposeByte());

if (r)// != &PacketSenderReceiver::nullRequest

{

r->answered(packet->getData(), packet->getDataLength());

return false;

}

}

return true;

}

else

{

return false;

}

}

void PacketSenderReceiver::send(Packet packet)

{

this->lastSentPacket = packet;

this->lastSentMillis = millis();

packet.sendViaSoftware(this->serial);

}

void PacketSenderReceiver::send(unsigned char to, unsigned char\* data, unsigned char len, PacketType type, unsigned char multiPurposeByte)

{

if (this->isSlave)

{

Packet p = Packet(this->address, to, data, len, type, multiPurposeByte);

this->send(p);

}

else

{

Packet p = Packet(to, this->address, data, len, type, multiPurposeByte);

this->send(p);

}

}

void PacketSenderReceiver::broadcast(unsigned char\* data, unsigned char len, PacketType type, unsigned char multiPurposeByte)

{

if (this->isSlave)

{

Packet p = Packet(this->address, 0x0, data, len, type, multiPurposeByte);

this->send(p);

}

else

{

Packet p = Packet(0x0, this->address, data, len, type, multiPurposeByte);

this->send(p);

}

}

void PacketSenderReceiver::resendLastPacket()

{

this->send(this->lastSentPacket);

}

Request\* PacketSenderReceiver::getNewRequest(unsigned char fromAddress, ResponseHandler handler, unsigned char\* data, unsigned char len, void\* state)

{

for (int i = 0; i < MAX\_CONCURRENT\_REQUESTS; i++)

{

if (this->requests[i].mayGetDisposed())

{

this->requests[i].use(fromAddress, handler, data, len, state);

//\*(this->requests[i]) = Request(fromAddress, handler, data, len, state);

return &this->requests[i];

}

}

this->logln("Fatal!! Ran out of requests! Increase MAX\_CONCURRENT\_REQUESTS!");

return nullptr;//&PacketSenderReceiver::nullRequest;

}

void PacketSenderReceiver::resendUnansweredRequests()

{

for (int i = 0; i < MAX\_CONCURRENT\_REQUESTS; i++)

{

if (this->requests[i].shouldGetResend())

{

this->requests[i].resendTries++;

if (this->requests[i].resendTries == REQUEST\_MAX\_RESENDS + 1)

{

this->logln("!");//this->logln("Request disposed.");

this->requests[i].noAnswer();

}

else

{

this->log(".");

this->sendRequest(&this->requests[i]);

}

}

}

}

unsigned char PacketSenderReceiver::sendRequest(Request\* request)

{

if (this->isSlave)

{

Packet p = Packet(this->address, request->fromAddress, request->sentData, request->sentDataLength, DataRequest, request->id);

this->send(p);

}

else

{

Packet p = Packet(request->fromAddress, this->address, request->sentData, request->sentDataLength, DataRequest, request->id);

this->send(p);

}

request->sentMillis = millis();

return request->id;

}

unsigned char PacketSenderReceiver::sendRequest(unsigned char to, ResponseHandler handler, unsigned char\* data, unsigned char len, void\* state)

{

Request\* request = this->getNewRequest(to, handler, data, len, state);

if (!request)//== &PacketSenderReceiver::nullRequest

return 0;

return this->sendRequest(request);

}

Request\* PacketSenderReceiver::getRequestWithId(unsigned char id)

{

for (int i = 0; i < MAX\_CONCURRENT\_REQUESTS; i++)

{

if (this->requests[i].used && this->requests[i].id == id)

{

return &this->requests[i];

}

}

return nullptr;//&PacketSenderReceiver::nullRequest;

}

void PacketSenderReceiver::answer(Packet\* toAnswer, unsigned char\* respData, unsigned char respLen)

{

if (!toAnswer->needsResponse())

return;

this->send(this->isSlave ? toAnswer->getMaster() : toAnswer->getSlave(), respData, respLen, Answer, toAnswer->getMultiPurposeByte());

}

#ifndef PacketSenderReceiver\_h

#define PacketSenderReceiver\_h

#ifndef MAX\_CONCURRENT\_REQUESTS

#define MAX\_CONCURRENT\_REQUESTS 10

#endif

#include "Request.h"

#include "Packet.h"

#include "SoftwareSerial.h"

#include "Logger.h"

#include "Arduino.h"

class PacketSenderReceiver : public Logger

{

public:

PacketSenderReceiver(SoftwareSerial\* serial, bool isSlave, unsigned char address);

bool receiveAny(Packet\* packet);

bool receive(Packet\* packet);

void send(Packet packet);

void send(unsigned char to, unsigned char\* data, unsigned char len, PacketType type = Push, unsigned char multiPurposeByte = 0x0);

void broadcast(unsigned char\* data, unsigned char len, PacketType type = Push, unsigned char multiPurposeByte = 0x0);

void resendLastPacket();

SoftwareSerial\* serial;

unsigned char address;

bool isSlave;

Packet lastSentPacket;

unsigned long lastSentMillis;

static Request nullRequest;

Request requests[MAX\_CONCURRENT\_REQUESTS];

unsigned char sendRequest(Request\* request);

unsigned char sendRequest(unsigned char to, ResponseHandler handler, unsigned char\* data, unsigned char len, void\* state = nullptr);

void resendUnansweredRequests();

Request\* getRequestWithId(unsigned char id);

void answer(Packet\* toAnswer, unsigned char\* respData, unsigned char respLen);

private:

bool receiving;

unsigned char dataPosition;

unsigned char incomingLength;

Request\* getNewRequest(unsigned char fromAddress, ResponseHandler handler, unsigned char\* data, unsigned char len, void\* state = nullptr);

};

#endif

#include "Request.h"

#include "Packet.h"

#include "SoftwareSerial.h"

#include "Logger.h"

unsigned char Request::currentId = 0;

Request::Request()

{

this->used = false;

}

Request::Request(unsigned char fromAddress, ResponseHandler handler, unsigned char\* data, unsigned char len, void\* state)

{

this->use(fromAddress, handler, data, len, state);

}

void Request::use(unsigned char fromAddress, ResponseHandler handler, unsigned char\* data, unsigned char len, void\* state)

{

this->handler = handler;

this->fromAddress = fromAddress;

this->id = Request::currentId++;

if (Request::currentId >= 128)

Request::currentId = 0;

this->gotAnswered = false;

this->sentMillis = millis();

this->createdMillis = this->sentMillis;

this->used = true;

this->resendTries = 0;

this->state = state;

memset(this->sentData, 0, 20);

memcpy(this->sentData, data, len);

this->sentDataLength = len;

}

bool Request::shouldGetResend()

{

if (this->gotAnswered || !this->used || this->resendTries > REQUEST\_MAX\_RESENDS)

return false;

return (millis() - this->sentMillis) > REQUEST\_TRY\_INTERVAL;

}

bool Request::mayGetDisposed()

{

return !used || this->resendTries > REQUEST\_MAX\_RESENDS || (millis() - this->createdMillis) > REQUEST\_MAX\_LIFETIME;

}

void Request::answered(unsigned char\* respData, unsigned char respLen)

{

memset(this->response, 0 , sizeof(this->response));

memcpy(this->response, respData, respLen);

this->responseLength = respLen;

if (respLen == 0)

{

this->handler(Failed, this);

}

else if (respLen == 1)

{

this->handler(respData[0] == 0xff ? Okay : Failed, this);

}

else

{

this->handler(Okay, this);

}

this->used = false;

}

void Request::noAnswer()

{

memset(this->response, 0 , sizeof(this->response));

this->responseLength = 0;

this->handler(NoResponse, this);

this->used = false;

}

#ifndef Request\_h

#define Request\_h

#ifndef REQUEST\_MAX\_RESENDS

#define REQUEST\_MAX\_RESENDS 10

#endif

#ifndef REQUEST\_TRY\_INTERVAL

#define REQUEST\_TRY\_INTERVAL 350

#endif

#ifndef REQUEST\_MAX\_LIFETIME

#define REQUEST\_MAX\_LIFETIME 30000

#endif

#include "Packet.h"

#include "SoftwareSerial.h"

#include "Logger.h"

enum ResponseStatus

{

NoResponse,

Failed,

Okay

};

class Request;

typedef void (\*ResponseHandler)(ResponseStatus status, Request\* requested);

class Request

{

public:

Request();

Request(unsigned char fromAddress, ResponseHandler handler, unsigned char\* data, unsigned char len, void\* state = nullptr);

ResponseHandler handler;

unsigned char fromAddress;

unsigned char id;

unsigned long sentMillis;

unsigned long createdMillis;

bool gotAnswered;

static unsigned char currentId;

bool shouldGetResend();

bool used;

bool mayGetDisposed();

void use(unsigned char fromAddress, ResponseHandler handler, unsigned char\* data, unsigned char len, void\* state = nullptr);

void answered(unsigned char\* respData, unsigned char respLen);

void noAnswer();

unsigned char sentData[20];

unsigned char sentDataLength;

unsigned char response[16];

unsigned char responseLength;

unsigned char resendTries;

void\* state;

};

#endif

#ifndef Shared\_h

#define Shared\_h

void veryCoolSplashScreen()

{

Serial.println();

Serial.println(" \_ \_ \_\_\_\_ \_\_\_\_ ");

Serial.println(" |'| |'| U /\"\_\_\_| U| \_\"\\ u ");

Serial.println(" /| |\_| |\\ \\| | u \\| |\_) |/ ");

Serial.println(" U| \_ |u | |/\_\_ | \_\_/ ");

Serial.println(" |\_| |\_| \\\_\_\_\_| |\_| ");

Serial.println(" // \\\\ \_// \\ \\ ||>>\_ ");

Serial.println(" (\_\") (\"\_) (\_\_)(\_\_) (\_\_)\_\_)");

Serial.println("Home Control Protocol - v0.4.0");

Serial.println("\tby Stijn Rogiest (c) 2019");

Serial.println();

}

#endif

@echo off

copy /Y Packet.cpp .\..\HCP\_CU\_v4\Packet.cpp

copy /Y Packet.h .\..\HCP\_CU\_v4\Packet.h

copy /Y PacketSenderReceiver.cpp .\..\HCP\_CU\_v4\PacketSenderReceiver.cpp

copy /Y PacketSenderReceiver.h .\..\HCP\_CU\_v4\PacketSenderReceiver.h

copy /Y Logger.cpp .\..\HCP\_CU\_v4\Logger.cpp

copy /Y Logger.h .\..\HCP\_CU\_v4\Logger.h

copy /Y Request.cpp .\..\HCP\_CU\_v4\Request.cpp

copy /Y Request.h .\..\HCP\_CU\_v4\Request.h

copy /Y Shared.h .\..\HCP\_CU\_v4\Shared.h

#!/bin/sh

cp Packet.cpp ./../HCP\_CU\_v4/Packet.cpp

cp Packet.h ./../HCP\_CU\_v4/Packet.h

cp PacketSenderReceiver.cpp ./../HCP\_CU\_v4/PacketSenderReceiver.cpp

cp PacketSenderReceiver.h ./../HCP\_CU\_v4/PacketSenderReceiver.h

cp Logger.cpp ./../HCP\_CU\_v4/Logger.cpp

cp Logger.h ./../HCP\_CU\_v4/Logger.h

cp Request.cpp ./../HCP\_CU\_v4/Request.cpp

cp Request.h ./../HCP\_CU\_v4/Request.h

cp Shared.h ./../HCP\_CU\_v4/Shared.h

#include "Device.h"

#include "Arduino.h"

#include "Print.h"

Device::Device(unsigned char fromBytes[118])

{

memcpy(this->name, &fromBytes[0], 25);

memcpy(this->liveDeviceInfo, &fromBytes[25], 16);

memcpy(this->uniqueFactoryId, &fromBytes[41], 7);

memcpy(this->knownProperties, &fromBytes[48], 64);

memcpy(&this->deviceType, &fromBytes[112], 4);

this->address = fromBytes[116];

this->working = (fromBytes[117] & 0x1) == 0x1;

this->online = (fromBytes[117] & 0x2) == 0x2;

}

Device::Device(unsigned char uniqueFactoryId[7], unsigned char address, char name[25])

{

memcpy(this->name, name, sizeof(this->name));

memset(this->liveDeviceInfo, 0, sizeof(this->liveDeviceInfo));

memcpy(this->uniqueFactoryId, uniqueFactoryId, 7);

this->address = address;

this->working = false;

this->online = false;

}

void Device::printTo(Print& dest)

{

dest.print("(address: ");

dest.print(this->address);

dest.print(", name: ");

dest.print(this->name);

dest.print(" (");

dest.print(this->working ? "" : "NOT WORKING, ");

dest.print(this->online ? "ONLINE" : "OFFLINE");

dest.print("), ufid: ");

for(unsigned char i = 0; i < 7; i++)

{

dest.print(this->uniqueFactoryId[i], DEC);

dest.print(' ');

}

dest.print(", type: ");

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

{

dest.print(this->deviceType[i]);

dest.print(' ');

}

dest.print(", live: ");

for(unsigned char i = 0; i < 16; i++)

{

dest.print(this->liveDeviceInfo[i]);

dest.print(' ');

}

dest.print(", prop: ");

for(unsigned char i = 0; i < 64; i++)

{

if (this->knownProperties[i] != 0x0)

{

dest.print(i);

dest.print('=');

dest.print(this->knownProperties[i]);

dest.print(' ');

}

}

dest.print(')');

}

void Device::printJSONTo(Print& dest)

{

// [0]: name

dest.print("{\n\"name\":\"");

dest.print(this->name);

dest.print("\",\n\"address\":");

// [1]: address

dest.print(this->address);

dest.print(",\n\"ufid\": [");

// [2]: uniqueFactoryId

for(unsigned char j = 0; j < 7; j++)

{

if (j != 0)

dest.print(',');

dest.print(this->uniqueFactoryId[j]);

}

dest.print("],\n\"type\": [");

// [3]: deviceType

for(unsigned char j = 0; j < 4; j++)

{

if (j != 0)

dest.print(',');

dest.print(this->deviceType[j]);

}

dest.print("],\n\"knownProperties\": [");

// [4]: knownProperties

for(unsigned char j = 0; j < 64; j++)

{

if (j != 0)

dest.print(',');

dest.print(this->knownProperties[j]);

}

dest.print("],\n\"liveData\": [");

// [5]: liveDeviceInfo

for(unsigned char j = 0; j < 16; j++)

{

if (j != 0)

dest.print(',');

dest.print(this->liveDeviceInfo[j]);

}

dest.print("],\n\"online\": ");

// [6]: online

dest.print(this->online ? "true" : "false");

dest.print(",\n\"working\": ");

// [7]: working

dest.print(this->working ? "true" : "false");

dest.print("\n}");

}

unsigned char\* Device::getBytes()

{

static unsigned char bytes[118];

memset(bytes, 0x0, sizeof(bytes));

memcpy(&bytes[0], this->name, 25);

memcpy(&bytes[25], this->liveDeviceInfo, 16);

memcpy(&bytes[41], this->uniqueFactoryId, 7);

memcpy(&bytes[48], this->knownProperties, 64);

memcpy(&bytes[112], &this->deviceType, 4);

bytes[116] = this->address;

bytes[117] |= this->working ? 0x1 : 0x0;

bytes[117] |= this->online ? 0x2 : 0x0;

return bytes;

}

#ifndef Device\_h

#define Device\_h

#include "Print.h"

#define DEVICE\_BYTE\_SIZE 120

class Device

{

public:

Device(unsigned char fromBytes[118]);

Device(unsigned char uniqueFactoryId[7], unsigned char address, char name[25]);

char name[25];

unsigned char uniqueFactoryId[7];

unsigned char liveDeviceInfo[16];

unsigned char knownProperties[64];

unsigned char deviceType[4];

unsigned char address;

bool working;

bool online;

unsigned char\* getBytes();

void printTo(Print& dest);

void printJSONTo(Print& dest);

};

#endif

/\*

Home Control Protocol v0.4.0

by Stijn Rogiest (copyright 2019)

Random console characters legend:

\_: The last packet was resent, caused by faulty integrity at the receiver.

!: The last request did not get answered and was disposed.

.: The last request was resent.

Sources:

https://tttapa.github.io/ESP8266/Chap10%20-%20Simple%20Web%20Server.html

https://www.arduino.cc/en/Reference/EEPROM

http://www.cplusplus.com/doc/tutorial/pointers/

https://www.arduino.cc/en/Reference/softwareSerial

https://stackoverflow.com/questions/3698043/static-variables-in-c

https://randomnerdtutorials.com/esp8266-web-server/

http://arduino.esp8266.com/stable/package\_esp8266com\_index.json

https://en.wikipedia.org/wiki/Multicast\_DNS

https://en.wikipedia.org/wiki/Cyclic\_redundancy\_check#CRC-32\_algorithm

Packet types/prefixes:

0x20: Set slave properties.

0x1: Ping slave.

0x15: Refresh slave live data.

0x10: Bind slave.

0x2: Unbind slave.

\*/

#include "Shared.h"

#include "PacketSenderReceiver.h"

#include "Device.h"

#include "Packet.h"

#include "WebRequest.h"

#include <SoftwareSerial.h>

#include <ESP8266WiFi.h>

#include <WiFiClient.h>

#include <ESP8266WiFiMulti.h>

#include <ESP8266mDNS.h>

#include <ESP8266WebServer.h>

#include <EEPROM.h>

#define DEBUG\_PIN LED\_BUILTIN

// Note: HC12 TX to RX and RX to TX

#define TX\_PIN 14

#define RX\_PIN 12

// This masters addr, can be 1, 2 or 3.

#define MASTER\_ADDRESS 2

#define MAX\_DEVICES 32

#define MAX\_CONCURRENT\_WEBREQUESTS 8

#define WIFI\_MDNS "homecontrol"

SoftwareSerial ss(RX\_PIN, TX\_PIN);

PacketSenderReceiver sr = PacketSenderReceiver(&ss, false, MASTER\_ADDRESS);

Packet temp;

Device\* devices[MAX\_DEVICES];

ESP8266WiFiMulti wifiMulti;

WiFiServer server(80);

WebRequest\* requesters[8];

const unsigned int retryBindMillisInterval = 400;

unsigned long lastRetryBindMillis = 1;

const unsigned int refreshMillisInterval = 2220;

unsigned long lastRefreshMillis = 1;

unsigned long lastLedBlink = 0;

unsigned int ledBlinks = 0;

unsigned int ledBlinkInterval = 200;

void led(int blinks, int interval = 200)

{

ledBlinks = blinks \* 2;

ledBlinkInterval = interval;

}

// Console

unsigned char currentArg = 0;

String args[16];

// Prototypes

unsigned char refreshSlave(unsigned char addr);

unsigned char pingSlave(unsigned char addr);

unsigned char unbindSlave(unsigned char withAddress);

unsigned char setSlaveProperties(unsigned char addr, unsigned char startPos, unsigned char\* values, unsigned char valueCount);

unsigned char bindSlave(unsigned char ufid[7], unsigned char withAddress);

unsigned char bindSlave(unsigned char ufid[7]);

unsigned char rebindSlave(unsigned char ufid[7], unsigned char withAddress);

void setup()

{

pinMode(DEBUG\_PIN, OUTPUT);

digitalWrite(DEBUG\_PIN, false);

Serial.begin(19200);

delay(5000);

veryCoolSplashScreen();

Serial.print("----> My address (master): ");

Serial.println(MASTER\_ADDRESS);

Serial.println("----> Loading devices...");

EEPROM.begin(4096);

//clearRomDevices();

loadDevicesFromRom();

printDevices();

Serial.print("----> Connecting to WiFi");

wifiMulti.addAP("PollenPatatten", "Ziektes123");

wifiMulti.addAP("RogiestHuis", "Vrijdag1!");

wifiMulti.addAP("pollenpattten", "ziektes123");

wifiMulti.addAP("Stijn Rogiest", "HoiDaag2");

while (wifiMulti.run() != WL\_CONNECTED)

{

delay(250);

Serial.print('.');

}

Serial.println();

Serial.print("----> Connected to ");

Serial.println(WiFi.SSID());

Serial.print("----> IP addr: ");

Serial.println(WiFi.localIP());

if (MDNS.begin(WIFI\_MDNS)) // Start the mDNS responder for esp8266.local

{

Serial.print("\t-> mDNS responder started: ");

Serial.println(WIFI\_MDNS);

}

else

{

Serial.println("\t-> FATAL: Error setting up MDNS responder!");

}

server.begin();

Serial.println("----> Starting...");

delay(500);

ss.begin(4800);

Serial.println("\t-> OK");

}

void loop()

{

if (ledBlinks > 0 && (millis() - lastLedBlink) > ledBlinkInterval)

{

digitalWrite(DEBUG\_PIN, ledBlinks % 2 == 0);

ledBlinks--;

lastLedBlink = millis();

}

if (Serial.available() > 0)

{

char c = Serial.read();

if (c == ' ' || c == ',')

{

if (args[currentArg].length() > 0)

{

currentArg++;

args[currentArg] = "";

}

}

else if (c == ';' || c == '\n')

{

command(args, currentArg + 1);

currentArg = 0;

args[currentArg] = "";

}

else

{

args[currentArg] += c;

}

}

if (sr.receive(&temp))

{

led(1);

// Slave is bound.

if (temp.getMultiPurposeByte() == 130)

{

Serial.print("Received bind response from ");

Serial.println(temp.getSlave());

Device\* bound = getDeviceWithAddress(temp.getSlave());

if (bound)

{

Serial.print("----> Slave is now getting bound (1): ");

bound->printTo(Serial);

Serial.println();

bound->working = true;

bound->online = true;

memcpy(bound->deviceType, temp.getData(), temp.getDataLength());

saveDevicesToRom();

Serial.print("----> Slave is now bound (2): ");

bound->printTo(Serial);

Serial.println();

WebRequest\* request = getWebRequestFor(130);

if (request)

{

request->println("okey");

request->close();

}

}

else

{

Serial.println("----> FATAL: Count not let slave work!");

}

}

}

sr.resendUnansweredRequests();

if ((millis() - lastRetryBindMillis) > retryBindMillisInterval)

{

retryNotWorkingBinds();

lastRetryBindMillis = millis();

}

if ((millis() - lastRefreshMillis) > refreshMillisInterval)

{

refreshSlaves();

lastRefreshMillis = millis();

}

WiFiClient newClient = server.available();

if (newClient)

{

Serial.println("New client?");

bool alreadyRequesting = false;

for(unsigned char i = 0; i < MAX\_CONCURRENT\_WEBREQUESTS; i++)

{

if (requesters[i] && requesters[i]->client == newClient)

{

alreadyRequesting = true;

break;

}

}

if (!alreadyRequesting)

{

for(unsigned char i = 0; i < MAX\_CONCURRENT\_WEBREQUESTS; i++)

{

Serial.print("Requester #");

Serial.print(i);

Serial.println(requesters[i] ? ": active" : ": not active");

if (!requesters[i])

{

requesters[i] = new WebRequest(newClient);

Serial.println("New request");

led(2);

break;

}

}

}

}

for(unsigned char i = 0; i < MAX\_CONCURRENT\_WEBREQUESTS; i++)

{

if (requesters[i])

{

requesters[i]->update(requested);

if (requesters[i]->shouldBeDisposed())

{

Serial.println("WebRequest is kermitting suicide... (3)");

delete requesters[i];

requesters[i] = nullptr;

Serial.println("WebRequest kermitted suicide (4)");

}

}

}

/\*if (newClient && (newClient != client) && (!client || !client.connected()))

{

client = newClient;

clientData = "";

}\*/

/\*while (client && client.available())

{

char c = client.read();

if (c == '\r')

continue;

clientData += c;

if (clientData.length() > 2 && c == '\n' && clientData[clientData.length() - 2] == '\n')

{

int i = clientData.indexOf("GET "), j = clientData.indexOf(" HTTP/");

bool open = false;

if (i >= 0 && j >= 0)

{

String request = clientData.substring(i + 4, j);

request.trim();

open = requested(request);

}

if (!open)

client.stop();

}

}\*/

}

void command(String args[16], unsigned char argsLen)

{

if (argsLen >= 4 && args[0] == "prop")

{

Serial.print("----> Trying to set property ");

unsigned char addr = args[1].toInt();

unsigned char startPos = args[2].toInt();

unsigned char data[16] = {0x20, startPos};

for (unsigned char i = 0; i < argsLen - 3; i++)

{

data[i + 2] = args[i + 3].toInt();

Serial.print('[');

Serial.print(startPos + i);

Serial.print(" = ");

Serial.print(args[i + 3]);

Serial.print("] ");

}

Serial.print("of addr ");

Serial.println(addr);

sr.sendRequest(addr, propertySetAnswer, data, argsLen - 1);

}

else if (args[0] == "wifi")

{

Serial.print("----> Connected to ");

Serial.println(WiFi.SSID());

Serial.print("----> IP addr: ");

Serial.println(WiFi.localIP());

}

else if (argsLen == 2 && args[0] == "ping")

{

pingSlave(args[1].toInt());

}

else if (args[0] == "device")

{

if (argsLen == 1 || args[1] == "list")

{

printDevices();

}

else if (argsLen == 2 && args[1] == "unbindall")

{

Serial.println("----> Unbinding all slaves, please wait...");

for(unsigned char i = 0; i < MAX\_DEVICES; i++)

{

if (devices[i])

{

Serial.print("Unbinding slave ");

Serial.print(devices[i]->address);

Serial.println("...");

unbindSlave(devices[i]->address);

delay(300);

}

}

clearRomDevices();

Serial.println("----> All bound slaves are now not bound anymore.");

}

else if (argsLen >= 3 && argsLen <= 9 && args[1] == "bind")

{

unsigned char ufid[7];

memset(ufid, 0x0, sizeof(ufid));

Serial.print("----> Binding slave with ufid [");

for (unsigned char i = 2; i < argsLen; i++)

{

ufid[i - 2] = args[i].toInt();

Serial.print(ufid[i - 2]);

Serial.print(' ');

}

Serial.println(']');

bindSlave(ufid);

}

else if (argsLen == 3 && args[1] == "unbind")

{

unsigned char addr = args[2].toInt();

Serial.print("----> Unbinding slave ");

Serial.print(addr);

Serial.println("...");

unbindSlave(addr);

}

else

{

Serial.println("Command syntax invalid: device [list|bind <ufid...>|unbind <addr>|unbindall]");

}

}

else

{

Serial.print("Unknown command: ");

Serial.print(args[0]);

Serial.print(" (");

Serial.print(argsLen);

Serial.println(")");

}

}

bool requested(WebRequest\* webRequest, String path)

{

WiFiClient& client = webRequest->client;

Serial.println("Requested path: " + path);

String sub[20];

unsigned char subCount = 0;

for(int i = 1; i < path.length() && subCount < 20; i++)

{

char c = path[i];

if (c == '/')

{

subCount++;

continue;

}

sub[subCount] += c;

}

subCount++;

// HEADER

client.println("HTTP/1.1 200 OK");

client.println("Connection: Keep-Alive");

client.println("Keep-Alive: timeout=15, max=1000");

client.println("Content-type: text/html");

client.println();

if (sub[0] == "interface")

{

// CSS + HTML HEAD

client.println("<!DOCTYPE html><html>");

client.println("<head><meta name=\"viewport\" content=\"width=device-width, initial-scale=1\">");

client.println("<link rel=\"icon\" href=\"data:,\">");

client.println("<style>html { font-family: Helvetica; display: inline-block; margin: 0px auto; text-align: center;}");

client.println(".button { background-color: #195B6A; border: none; color: white; padding: 16px 40px;");

client.println("text-decoration: none; font-size: 30px; margin: 2px; cursor: pointer;}");

client.println(".button2 { background-color: #77878A; }</style></head>");

// HTML

client.println("<body><h1>Home Control</h1>");

client.println("<p>TESTING:</p>");

client.println("<p><a href=\"/nice\"><button class=\"button\">OKE COOL</button></a></p>");

client.println("</body></html>");

client.println();

return false;

}

else if (sub[0] == "test")

{

client.println("okey");

return false;

}

else if (sub[0] == "deviceList")

{

client.print("{\n\"devices\": [");

for(unsigned char i = 0; i < MAX\_DEVICES; i++)

{

if (devices[i])

{

if (i != 0)

client.print(",\n");

devices[i]->printJSONTo(client);

}

}

client.print("],\n\"deviceNames\": [");

for(unsigned char i = 0; i < MAX\_DEVICES; i++)

{

if (devices[i])

{

if (i != 0)

client.print(",");

client.print('\"');

client.print(devices[i]->name);

client.print('\"');

}

}

client.print("]\n}");

return false;

}

else if (sub[0] == "device" && subCount == 2)

{

unsigned char addr = sub[1].toInt();

Device\* d = getDeviceWithAddress(addr);

if (d)

{

d->printJSONTo(client);

}

return false;

}

else if (sub[0] == "setDeviceName" && subCount == 3)

{

unsigned char addr = sub[1].toInt();

Device\* d = getDeviceWithAddress(addr);

if (d && sub[2].length() > 1 && sub[2].length() < 25)

{

sub[2].toCharArray(d->name, sub[2].length() + 1);

saveDevicesToRom();

client.println("okey");

}

else

{

client.println("not okey");

}

return false;

}

else if (sub[0] == "ping" && subCount == 2)

{

unsigned char addr = sub[1].toInt();

webRequest->requestId = pingSlave(addr);

return true;

}

else if (sub[0] == "bind" && subCount > 1 && subCount <= 8)

{

unsigned char ufid[7];

memset(ufid, 0x0, sizeof(ufid));

for (unsigned char i = 1; i < subCount; i++)

ufid[i - 1] = sub[i].toInt();

webRequest->requestId = bindSlave(ufid);

return true;

}

else if (sub[0] == "unbind" && subCount == 2)

{

unsigned char addr = sub[1].toInt();

webRequest->requestId = unbindSlave(addr);

return true;

}

else if (sub[0] == "prop" && subCount > 3 && subCount < 20)

{

unsigned char addr = sub[1].toInt();

unsigned char startPos = sub[2].toInt();

unsigned char data[16] = {0x20, startPos};

for (unsigned char i = 0; i < subCount - 3; i++)

data[i + 2] = sub[i + 3].toInt();

webRequest->requestId = sr.sendRequest(addr, propertySetAnswer, data, subCount - 1);

return true;

}

else

{

client.println("nope");

return false;

}

return false;

}

unsigned char setSlaveProperties(unsigned char addr, unsigned char startPos, unsigned char\* values, unsigned char valueCount)

{

if (valueCount == 0)

return 0xFF;

unsigned char data[16] = {0x20, startPos};

for (unsigned char i = 0; i < valueCount && i < 14; i++)

data[i + 2] = values[i];

return sr.sendRequest(addr, propertySetAnswer, data, valueCount + 2);

}

void propertySetAnswer(ResponseStatus status, Request\* requested)

{

if (status == Okay)

{

Serial.print("\t-> Propery for slave ");

Serial.print(requested->fromAddress);

Serial.println(" was set successfully!");

Device\* setDevice = getDeviceWithAddress(requested->fromAddress);

if (setDevice)

{

unsigned char startPos = requested->sentData[1];

unsigned char valueCount = requested->sentDataLength - 2;

for(unsigned char i = 0; i < valueCount; i++)

setDevice->knownProperties[startPos + i] = requested->sentData[i + 2];

}

}

WebRequest\* request = getWebRequestFor(requested->id);

if (request)

{

request->println(static\_cast<int>(status));

request->close();

}

}

unsigned char pingSlave(unsigned char addr)

{

unsigned char data[1] = {0x1};

return sr.sendRequest(addr, pingAnswer, data, sizeof(data));

}

void pingAnswer(ResponseStatus status, Request\* requested)

{

Serial.print("\t-> Slave ");

Serial.print(requested->fromAddress);

Serial.print(" was pinged: ");

Serial.println(status == Okay ? "Okay" : (status == Failed ? "Failed" : "No response"));

Device\* dev = getDeviceWithAddress(requested->fromAddress);

if (dev)

{

bool online = status == Okay;

if (dev->online != online)

{

dev->online = online;

saveDevicesToRom();

}

}

WebRequest\* request = getWebRequestFor(requested->id);

if (request)

{

request->println(static\_cast<int>(status));

request->close();

}

/\*if (requested->state)

{

WiFiClient\* wc = (WiFiClient\*)requested->state;

wc->println(status);

wc->stop();

}\*/

}

unsigned char refreshSlave(unsigned char addr)

{

unsigned char data[1] = {0x15};

return sr.sendRequest(addr, refreshAnswer, data, sizeof(data));

}

void refreshAnswer(ResponseStatus status, Request\* requested)

{

Device\* dev = getDeviceWithAddress(requested->fromAddress);

if (dev)

{

bool online = status != NoResponse;

if (online)

{

Serial.print("Received ");

Serial.print(requested->responseLength);

Serial.println(" bytes for live data.");

for(unsigned char i = 0; i < requested->responseLength; i++)

dev->liveDeviceInfo[i] = requested->response[i];

}

if (dev->online != online)

{

dev->online = online;

saveDevicesToRom();

}

}

WebRequest\* request = getWebRequestFor(requested->id);

if (request)

{

request->println(static\_cast<int>(status));

request->close();

}

}

unsigned char bindSlave(unsigned char ufid[7])

{

return bindSlave(ufid, getNewAddress());

}

unsigned char bindSlave(unsigned char ufid[7], unsigned char withAddress)

{

for(unsigned char i = 0; i < MAX\_DEVICES; i++)

{

if (devices[i] && (devices[i]->address == withAddress || memcmp(ufid, devices[i]->uniqueFactoryId, 7) == 0))

{

Serial.println("----> Warning: tried to bind 2 slaves with either the same addr or ufid.");

return 0xFF;

}

}

unsigned char id = rebindSlave(ufid, withAddress);

registerNewDevice(ufid, withAddress);

saveDevicesToRom();

return id;

}

unsigned char rebindSlave(unsigned char ufid[7], unsigned char withAddress)

{

unsigned char data[9];

memcpy(&data[1], &ufid[0], 7);

data[0] = 0x10;

data[8] = withAddress;

sr.broadcast(data, sizeof(data), DataRequest, 130);

return 130;

}

unsigned char unbindSlave(unsigned char withAddress)

{

unsigned char data[1] = { 0x2 };

unsigned char id = sr.sendRequest(withAddress, unbindAnswer, data, sizeof(data));

for(unsigned char i = 0; i < MAX\_DEVICES; i++)

{

if (devices[i] && devices[i]->address == withAddress)

{

delete devices[i];

devices[i] = nullptr;

saveDevicesToRom();

Serial.println("\t-> Device is unregistered, waiting for unbind request... (no answer is ok)");

break;

}

}

return id;

}

void unbindAnswer(ResponseStatus status, Request\* requested)

{

if (status == Okay)

{

Serial.print("\t-> Slave ");

Serial.print(requested->fromAddress);

Serial.println(" was successfully unbound from this master.");

}

WebRequest\* request = getWebRequestFor(requested->id);

if (request)

{

request->println(static\_cast<int>(status));

request->close();

}

}

void refreshSlaves()

{

static unsigned char i = 0;

if (i >= MAX\_DEVICES)

i = 0;

for(; i < MAX\_DEVICES; i++)

{

if (devices[i] && devices[i]->working)

{

refreshSlave(devices[i]->address);

i++;

break;

}

}

}

void pingSlaves()

{

static unsigned char i = 0;

if (i >= MAX\_DEVICES)

i = 0;

for(; i < MAX\_DEVICES; i++)

{

if (devices[i] && devices[i]->working)

{

pingSlave(devices[i]->address);

i++;

break;

}

}

}

void retryNotWorkingBinds()

{

static unsigned char i = 0;

if (i >= MAX\_DEVICES)

i = 0;

for(; i < MAX\_DEVICES; i++)

{

if (devices[i] && !(devices[i]->working))

{

Serial.print("----> Trying to let device ");

devices[i]->printTo(Serial);

Serial.println(" work...");

rebindSlave(devices[i]->uniqueFactoryId, devices[i]->address);

/\*unsigned char data[9];

memcpy(&data[1], devices[i]->uniqueFactoryId, 7);

data[0] = 0x10;

data[8] = devices[i]->address;

sr.broadcast(data, sizeof(data), DataRequest, 130);\*/

i++;

break;

}

}

}

unsigned char getNewAddress()

{

unsigned char s = EEPROM.read(0);

if (s == 0xFF)

s = 1;

EEPROM.write(0, ++s);

return s;

}

void printDevices()

{

Serial.println("----> List of devices that are controlled by this master:");

unsigned char deviceCount = 0;

for(unsigned char i = 0; i < MAX\_DEVICES; i++)

{

if (devices[i])

{

Serial.print("\t");

Serial.print(++deviceCount);

Serial.print(": ");

devices[i]->printTo(Serial);

Serial.println();

}

}

}

void loadDevicesFromRom()

{

/\*Serial.print("Size of device: ");

Serial.println(sizeof(Device));\*/

unsigned char deviceCount = 0;

for (int i = 0; i < MAX\_DEVICES; i++)

{

if (EEPROM.read(i \* DEVICE\_BYTE\_SIZE + 100 + DEVICE\_BYTE\_SIZE - 1) == 0xFF)

{

// Device save location is empty

devices[i] = nullptr;

}

else

{

// Device save location is used, read it

unsigned char bytes[DEVICE\_BYTE\_SIZE];

for(int j = 0; j < DEVICE\_BYTE\_SIZE; j++)

bytes[j] = EEPROM.read(i \* DEVICE\_BYTE\_SIZE + 100 + j);

devices[i] = new Device(bytes);

/\*Serial.print("Red device: ");

devices[i]->printToSerial();

Serial.println();\*/

deviceCount++;

}

}

Serial.print("\t-> ");

Serial.print(deviceCount);

Serial.println(" devices were loaded from ROM.");

}

void clearRomDevices()

{

for (int i = 100; i < 100 + MAX\_DEVICES \* DEVICE\_BYTE\_SIZE; i++)

EEPROM.write(i, 0xFF);

for(unsigned char i = 0; i < MAX\_DEVICES; i++)

{

if (devices[i])

{

delete devices[i];

devices[i] = nullptr;

}

}

EEPROM.commit();

Serial.println("\t-> All devices were ereased from ROM.");

}

void saveDevicesToRom()

{

unsigned char deviceCount = 0;

for (int i = 0; i < MAX\_DEVICES; i++)

{

if (devices[i])

{

/\*Serial.print("Saving device ");

Serial.print(i);

Serial.print(": ");

devices[i]->printToSerial();

Serial.println();\*/

unsigned char\* bytes = devices[i]->getBytes();

for(int j = 0; j < DEVICE\_BYTE\_SIZE; j++)

EEPROM.write(i \* DEVICE\_BYTE\_SIZE + 100 + j, bytes[j]);

deviceCount++;

}

else

{

EEPROM.write(i \* DEVICE\_BYTE\_SIZE + 100 + DEVICE\_BYTE\_SIZE - 1, 0xFF);

}

}

EEPROM.commit();

Serial.print("\t-> ");

Serial.print(deviceCount);

Serial.println(" devices were saved to ROM.");

}

Device\* registerNewDevice(unsigned char ufid[7], unsigned char addr)

{

for(unsigned char i = 0; i < MAX\_DEVICES; i++)

{

if (!devices[i])

{

devices[i] = new Device(ufid, addr, "Test");

return devices[i];

}

}

return nullptr;

}

Device\* getDeviceWithAddress(unsigned char addr)

{

for(unsigned char i = 0; i < MAX\_DEVICES; i++)

{

if (devices[i] && devices[i]->address == addr)

return devices[i];

}

return nullptr;

}

WebRequest\* getWebRequestFor(unsigned char requestId)

{

for(int i = 0; i < MAX\_CONCURRENT\_WEBREQUESTS; i++)

{

if (requesters[i] && requesters[i]->requestId == requestId)

return requesters[i];

}

return nullptr;

}

#include "Logger.h"

#include "Arduino.h"

Logger::Logger()

{

this->enable = true;

}

void Logger::log(char\* s)

{

if (enable)

Serial.print(s);

}

void Logger::log(char s)

{

if (enable)

Serial.print(s);

}

void Logger::log(int i)

{

if (enable)

Serial.print(i, DEC);

}

void Logger::log(unsigned char i)

{

if (enable)

Serial.print(i, DEC);

}

void Logger::log(long i)

{

if (enable)

Serial.print(i, DEC);

}

void Logger::logln(char\* s)

{

if (enable)

Serial.println(s);

}

void Logger::logln(unsigned char i)

{

if (enable)

Serial.println(i, DEC);

}

void Logger::logln(int i)

{

if (enable)

Serial.println(i, DEC);

}

void Logger::logln(long i)

{

if (enable)

Serial.println(i, DEC);

}

void Logger::logln(char s)

{

if (enable)

Serial.println(s, DEC);

}

void Logger::logln()

{

if (enable)

Serial.println();

}

#ifndef Logger\_h

#define Logger\_h

#include "Arduino.h"

class Logger

{

public:

Logger();

void log(char\* s);

void log(char s);

void log(unsigned char i);

void log(int i);

void log(long i);

void logln(char\* s);

void logln(char s);

void logln(unsigned char i);

void logln(int i);

void logln(long i);

void logln();

bool enable;

};

#endif

#include "Packet.h"

#include "SoftwareSerial.h"

#include "Arduino.h"

unsigned char Packet::identifier = 0x69;

Packet::Packet()

{

memset(this->data, 0, 20);

}

Packet::Packet(unsigned char\* data, unsigned char len)

{

memset(this->data, 0, 20);

memcpy(this->data, data, len);

}

Packet::Packet(unsigned char slaveAddress, unsigned char masterAddress, unsigned char\* data,

unsigned char len, PacketType type, unsigned char multiPurposeByte)

{

memset(this->data, 0, 20);

if (len > 0)

memcpy(&this->data[4], data, len);

this->data[0] = Packet::identifier;

this->data[1] = slaveAddress & 0x3F;

this->data[2] = ((masterAddress & 0x3) << 6) | ((type & 0x3) << 4) | (len & 0xF);

this->data[3] = multiPurposeByte;

this->data[1] |= getCurrentCRC() << 6;

}

unsigned char Packet::getCurrentCRC()

{

unsigned char crc = ~Packet::identifier;

for (int i = 2; i < 20; i++)

crc ^= this->data[i];

crc ^= this->data[1] & 0x3F;

return (crc ^ (crc >> 2) ^ (crc >> 4) ^ (crc >> 6)) & 0x3;

}

bool Packet::hasValidIntegrity()

{

return this->getCurrentCRC() == this->getCRC();

}

void Packet::sendViaSoftware(SoftwareSerial\* ss)

{

ss->write(this->data, this->getDataLength() + 4);

}

void Packet::printToSerial()

{

Serial.print('[');

Serial.print(this->getType());

Serial.print(", CRC: ");

Serial.print(this->getCRC());

Serial.print(" =?= ");

Serial.print(this->getCurrentCRC());

Serial.print(", slave: ");

Serial.print(this->getSlave());

Serial.print(", master: ");

Serial.print(this->getMaster());

Serial.print(", data(");

Serial.print(this->getDataLength());

Serial.print("): ");

for (int i = 4, ii = this->getDataLength(); i < 20 && i < (ii + 4); i++)

{

Serial.print(this->data[i], DEC);

Serial.print(' ');

}

Serial.print(']');

}

char Packet::getIdentifier()

{

return this->data[0];

}

unsigned char Packet::getCRC()

{

return this->data[1] >> 6;

}

unsigned char Packet::getSlave()

{

return this->data[1] & 0x3F;

}

unsigned char Packet::getMaster()

{

return this->data[2] >> 6;

}

unsigned char Packet::getRawType()

{

return (this->data[2] >> 4) & 0x3;

}

PacketType Packet::getType()

{

return static\_cast<PacketType>(this->getRawType());

}

unsigned char Packet::getMultiPurposeByte()

{

return this->data[3];

}

unsigned char\* Packet::getData()

{

return &this->data[4];

}

unsigned char Packet::getDataLength()

{

return this->data[2] & 0xF;

}

void Packet::recalculateCRC()

{

this->data[1] |= this->getCurrentCRC() << 6;

}

bool Packet::needsResponse()

{

return this->getRawType() == 0;

}

#ifndef Packet\_h

#define Packet\_h

#include "SoftwareSerial.h"

#include "Arduino.h"

enum PacketType

{

DataRequest,

Push,

Answer,

PleaseResend

};

class Packet

{

public:

Packet();

Packet(unsigned char\* data, unsigned char len = 20);

Packet(unsigned char slaveAddress, unsigned char masterAddress, unsigned char\* data, unsigned char len, PacketType type, unsigned char multiPurposeByte = 0x0);

unsigned char data[20];

static unsigned char identifier;

void sendViaSoftware(SoftwareSerial\* ss);

void printToSerial();

char getIdentifier();

unsigned char getCRC();

unsigned char getSlave();

unsigned char getMaster();

unsigned char getRawType();

PacketType getType();

unsigned char getDataLength();

unsigned char\* getData();

unsigned char getMultiPurposeByte();

unsigned char getCurrentCRC();

bool hasValidIntegrity();

void recalculateCRC();

bool needsResponse();

};

#endif

#include "PacketSenderReceiver.h"

#include "Request.h"

#include "Packet.h"

#include "SoftwareSerial.h"

#include "Logger.h"

#include "Arduino.h"

//Request PacketSenderReceiver::nullRequest;

PacketSenderReceiver::PacketSenderReceiver(SoftwareSerial\* serial, bool isSlave, unsigned char address)

{

this->serial = serial;

this->address = address;

this->isSlave = isSlave;

this->receiving = false;

this->dataPosition = 0;

this->incomingLength = 0;

}

bool PacketSenderReceiver::receiveAny(Packet\* p)

{

if (this->receiving)

{

while (this->serial->available() > 0 && this->dataPosition < this->incomingLength)

p->data[this->dataPosition++] = this->serial->read();

if (this->dataPosition >= this->incomingLength)

{

this->receiving = false;

return true;

}

else

{

return false;

}

}

else

{

if (this->serial->available() < 4)

return false;

while (this->serial->peek() != Packet::identifier)

{

if (this->serial->available() == 0)

return false;

this->log("Out of sync: ");

this->log(this->serial->peek());

this->log(" != ");

this->logln(Packet::identifier);

this->serial->read();

}

if (this->serial->available() < 4)

return false;

this->receiving = true;

memset(p->data, 0, 20);

for (this->dataPosition = 0; this->dataPosition < 4; this->dataPosition++)

p->data[this->dataPosition] = this->serial->read();

this->incomingLength = (p->data[2] & 0xF) + 4;

if (this->incomingLength == 4)

{

this->receiving = false;

return true;

}

return false;//receiveAny(p);

}

}

bool PacketSenderReceiver::receive(Packet\* packet)

{

if (receiveAny(packet))

{

// Check if the received packet is not a broadcast.

if (!((this->isSlave && packet->getSlave() == 0x0) || (!this->isSlave && packet->getMaster() == 0x0)))

{

// Check if this packet is for me or not.

if ((this->isSlave && this->address != packet->getSlave()) || (!this->isSlave && this->address != packet->getMaster()))

{

// This packet is not for me.

this->log("Not for me. ");

this->log(this->address);

this->log(" != (master: ");

this->log(packet->getMaster());

this->log(", slave: ");

this->log(packet->getSlave());

this->logln(")");

return false;

}

}

// Ask for resend if the packet has a false integrity.

if (!packet->hasValidIntegrity() && packet->getType() != PleaseResend)

{

this->send(this->isSlave ? packet->getMaster() : packet->getSlave(), {}, 0, PleaseResend);

this->log("Faulty integrity: ");

packet->printToSerial();

this->logln();

return false;

}

// Resend packet if it is a PleaseResend packet.

if (packet->getType() == PleaseResend)

{

this->resendLastPacket();

this->log("\_");

return false;

}

if (packet->getType() == Answer)

{

/\*this->log("Received request answer: ");

packet->printToSerial();

this->logln();\*/

Request\* r = this->getRequestWithId(packet->getMultiPurposeByte());

if (r)// != &PacketSenderReceiver::nullRequest

{

r->answered(packet->getData(), packet->getDataLength());

return false;

}

}

return true;

}

else

{

return false;

}

}

void PacketSenderReceiver::send(Packet packet)

{

this->lastSentPacket = packet;

this->lastSentMillis = millis();

packet.sendViaSoftware(this->serial);

}

void PacketSenderReceiver::send(unsigned char to, unsigned char\* data, unsigned char len, PacketType type, unsigned char multiPurposeByte)

{

if (this->isSlave)

{

Packet p = Packet(this->address, to, data, len, type, multiPurposeByte);

this->send(p);

}

else

{

Packet p = Packet(to, this->address, data, len, type, multiPurposeByte);

this->send(p);

}

}

void PacketSenderReceiver::broadcast(unsigned char\* data, unsigned char len, PacketType type, unsigned char multiPurposeByte)

{

if (this->isSlave)

{

Packet p = Packet(this->address, 0x0, data, len, type, multiPurposeByte);

this->send(p);

}

else

{

Packet p = Packet(0x0, this->address, data, len, type, multiPurposeByte);

this->send(p);

}

}

void PacketSenderReceiver::resendLastPacket()

{

this->send(this->lastSentPacket);

}

Request\* PacketSenderReceiver::getNewRequest(unsigned char fromAddress, ResponseHandler handler, unsigned char\* data, unsigned char len)

{

for (int i = 0; i < MAX\_CONCURRENT\_REQUESTS; i++)

{

if (this->requests[i].mayGetDisposed())

{

this->requests[i].use(fromAddress, handler, data, len);

return &this->requests[i];

}

}

this->logln("Fatal!! Ran out of requests! Increase MAX\_CONCURRENT\_REQUESTS!");

return nullptr;//&PacketSenderReceiver::nullRequest;

}

void PacketSenderReceiver::resendUnansweredRequests()

{

for (int i = 0; i < MAX\_CONCURRENT\_REQUESTS; i++)

{

if (this->requests[i].shouldGetResend())

{

this->requests[i].resendTries++;

if (this->requests[i].resendTries == REQUEST\_MAX\_RESENDS + 1)

{

this->logln("!");//this->logln("Request disposed.");

this->requests[i].noAnswer();

}

else

{

this->log(".");

this->sendRequest(&this->requests[i]);

}

}

}

}

unsigned char PacketSenderReceiver::sendRequest(Request\* request)

{

if (this->isSlave)

{

Packet p = Packet(this->address, request->fromAddress, request->sentData, request->sentDataLength, DataRequest, request->id);

this->send(p);

}

else

{

Packet p = Packet(request->fromAddress, this->address, request->sentData, request->sentDataLength, DataRequest, request->id);

this->send(p);

}

request->sentMillis = millis();

return request->id;

}

unsigned char PacketSenderReceiver::sendRequest(unsigned char to, ResponseHandler handler, unsigned char\* data, unsigned char len)

{

Request\* request = this->getNewRequest(to, handler, data, len);

if (!request)//== &PacketSenderReceiver::nullRequest

return 0;

return this->sendRequest(request);

}

Request\* PacketSenderReceiver::getRequestWithId(unsigned char id)

{

for (int i = 0; i < MAX\_CONCURRENT\_REQUESTS; i++)

{

if (this->requests[i].used && this->requests[i].id == id)

{

return &this->requests[i];

}

}

return nullptr;//&PacketSenderReceiver::nullRequest;

}

void PacketSenderReceiver::answer(Packet\* toAnswer, unsigned char\* respData, unsigned char respLen)

{

if (!toAnswer->needsResponse())

return;

this->send(this->isSlave ? toAnswer->getMaster() : toAnswer->getSlave(), respData, respLen, Answer, toAnswer->getMultiPurposeByte());

}

#ifndef PacketSenderReceiver\_h

#define PacketSenderReceiver\_h

#ifndef MAX\_CONCURRENT\_REQUESTS

#define MAX\_CONCURRENT\_REQUESTS 10

#endif

#include "Request.h"

#include "Packet.h"

#include "SoftwareSerial.h"

#include "Logger.h"

#include "Arduino.h"

class PacketSenderReceiver : public Logger

{

public:

PacketSenderReceiver(SoftwareSerial\* serial, bool isSlave, unsigned char address);

bool receiveAny(Packet\* packet);

bool receive(Packet\* packet);

void send(Packet packet);

void send(unsigned char to, unsigned char\* data, unsigned char len, PacketType type = Push, unsigned char multiPurposeByte = 0x0);

void broadcast(unsigned char\* data, unsigned char len, PacketType type = Push, unsigned char multiPurposeByte = 0x0);

void resendLastPacket();

SoftwareSerial\* serial;

unsigned char address;

bool isSlave;

Packet lastSentPacket;

unsigned long lastSentMillis;

static Request nullRequest;

Request requests[MAX\_CONCURRENT\_REQUESTS];

unsigned char sendRequest(Request\* request);

unsigned char sendRequest(unsigned char to, ResponseHandler handler, unsigned char\* data, unsigned char len);

void resendUnansweredRequests();

Request\* getRequestWithId(unsigned char id);

void answer(Packet\* toAnswer, unsigned char\* respData, unsigned char respLen);

private:

bool receiving;

unsigned char dataPosition;

unsigned char incomingLength;

Request\* getNewRequest(unsigned char fromAddress, ResponseHandler handler, unsigned char\* data, unsigned char len);

};

#endif

#include "Request.h"

#include "Packet.h"

#include "SoftwareSerial.h"

#include "Logger.h"

unsigned char Request::currentId = 0;

Request::Request()

{

this->used = false;

}

Request::Request(unsigned char fromAddress, ResponseHandler handler, unsigned char\* data, unsigned char len)

{

this->use(fromAddress, handler, data, len);

}

void Request::use(unsigned char fromAddress, ResponseHandler handler, unsigned char\* data, unsigned char len)

{

this->handler = handler;

this->fromAddress = fromAddress;

this->id = Request::currentId++;

if (Request::currentId >= 128)

Request::currentId = 0;

this->gotAnswered = false;

this->sentMillis = millis();

this->createdMillis = this->sentMillis;

this->used = true;

this->resendTries = 0;

memset(this->sentData, 0, 20);

memcpy(this->sentData, data, len);

this->sentDataLength = len;

}

bool Request::shouldGetResend()

{

if (this->gotAnswered || !this->used || this->resendTries > REQUEST\_MAX\_RESENDS)

return false;

return (millis() - this->sentMillis) > REQUEST\_TRY\_INTERVAL;

}

bool Request::mayGetDisposed()

{

return !used || this->resendTries > REQUEST\_MAX\_RESENDS || (millis() - this->createdMillis) > REQUEST\_MAX\_LIFETIME;

}

void Request::answered(unsigned char\* respData, unsigned char respLen)

{

memset(this->response, 0 , sizeof(this->response));

memcpy(this->response, respData, respLen);

this->responseLength = respLen;

if (respLen == 0)

{

this->handler(Failed, this);

}

else if (respLen == 1)

{

this->handler(respData[0] == 0xff ? Okay : Failed, this);

}

else

{

this->handler(Okay, this);

}

this->used = false;

}

void Request::noAnswer()

{

memset(this->response, 0 , sizeof(this->response));

this->responseLength = 0;

this->handler(NoResponse, this);

this->used = false;

}

#ifndef Request\_h

#define Request\_h

#ifndef REQUEST\_MAX\_RESENDS

#define REQUEST\_MAX\_RESENDS 10

#endif

#ifndef REQUEST\_TRY\_INTERVAL

#define REQUEST\_TRY\_INTERVAL 350

#endif

#ifndef REQUEST\_MAX\_LIFETIME

#define REQUEST\_MAX\_LIFETIME 30000

#endif

#include "Packet.h"

#include "SoftwareSerial.h"

#include "Logger.h"

enum ResponseStatus

{

NoResponse,

Failed,

Okay

};

class Request;

typedef void (\*ResponseHandler)(ResponseStatus status, Request\* requested);

class Request

{

public:

Request();

Request(unsigned char fromAddress, ResponseHandler handler, unsigned char\* data, unsigned char len);

ResponseHandler handler;

unsigned char fromAddress;

unsigned char id;

unsigned long sentMillis;

unsigned long createdMillis;

bool gotAnswered;

static unsigned char currentId;

bool shouldGetResend();

bool used;

bool mayGetDisposed();

void use(unsigned char fromAddress, ResponseHandler handler, unsigned char\* data, unsigned char len);

void answered(unsigned char\* respData, unsigned char respLen);

void noAnswer();

unsigned char sentData[20];

unsigned char sentDataLength;

unsigned char response[16];

unsigned char responseLength;

unsigned char resendTries;

};

#endif

#ifndef Shared\_h

#define Shared\_h

void veryCoolSplashScreen()

{

Serial.println();

Serial.println(" \_ \_ \_\_\_\_ \_\_\_\_ ");

Serial.println(" |'| |'| U /\"\_\_\_| U| \_\"\\ u ");

Serial.println(" /| |\_| |\\ \\| | u \\| |\_) |/ ");

Serial.println(" U| \_ |u | |/\_\_ | \_\_/ ");

Serial.println(" |\_| |\_| \\\_\_\_\_| |\_| ");

Serial.println(" // \\\\ \_// \\ \\ ||>>\_ ");

Serial.println(" (\_\") (\"\_) (\_\_)(\_\_) (\_\_)\_\_)");

Serial.println("Home Control Protocol - v0.4.0");

Serial.println("\tby Stijn Rogiest (c) 2019");

Serial.println();

}

unsigned char temperatureToByte(float floatTemp)

{

float f = (floatTemp - 10.0f) \* 10.0f;

if (f > 255)

f = 255;

else if (f < 0)

f = 0

return (unsigned char)f;

}

float byteToTemperature(unsigned char byteTemp)

{

return (byteTemp / 10.0f) + 10.0f;

}

unsigned char currentFlowToByte(float floatCurrent)

{

return 0;

}

#endif

#include "WebRequest.h"

#include "WiFiClient.h"

#include "Arduino.h"

WebRequest::WebRequest(WiFiClient client)

{

this->client = client;

this->receivedMillis = millis();

this->clientData = "";

this->requestId = 255;

this->closed = false;

}

void WebRequest::println(String str)

{

this->client.println(str);

}

void WebRequest::print(String str)

{

this->client.print(str);

}

void WebRequest::println(int i)

{

this->client.println(i);

}

void WebRequest::print(int i)

{

this->client.println(i);

}

void WebRequest::close()

{

this->closed = true;

if (this->client)

this->client.stop();

Serial.println("WebRequest kermitted suicide (1)");

}

// funcRequester returns the fact to leave open connection or not.

void WebRequest::update(bool(\*funcRequester)(WebRequest\*, String))

{

if (shouldBeDisposed())

{

Serial.println("WebRequest should kermit suicide (2)");

return;

}

while (client && client.available())

{

char c = client.read();

if (c == '\r')

continue;

clientData += c;

if (clientData.length() > 2 && c == '\n' && clientData[clientData.length() - 2] == '\n')

{

int i = clientData.indexOf("GET "), j = clientData.indexOf(" HTTP/");

bool open = false;

if (i >= 0 && j >= 0)

{

String request = clientData.substring(i + 4, j);

request.trim();

if (request.startsWith("/favicon"))

{

open = false;

}

else

{

open = funcRequester(this, request);

}

}

if (!open)

close();

}

}

}

bool WebRequest::shouldBeDisposed()

{

return closed || !client || !client.connected() || (millis() - receivedMillis) > 20000;

}

#ifndef WebRequest\_h

#define WebRequest\_h

#include "WiFiClient.h"

#include "Arduino.h"

class WebRequest

{

public:

WebRequest(WiFiClient client);

WiFiClient client;

unsigned long receivedMillis;

unsigned char requestId;

String requested;

void println(String str);

void print(String str);

void println(int i);

void print(int i);

void close();

void update(bool(\*funcRequester)(WebRequest\*, String));

bool shouldBeDisposed();

private:

String clientData;

bool closed;

};

#endif