# Appendix 5: MessageHandler.h and MessageHandler.cpp source code

\\psf\Dropbox\Elektrotechniek 4de\Afstuderen ...\Networked LED Driver System\MessageHandler.h 1

```
//
//  MessageHandler.h
//  Networked LED Driver System server
//
//  Created by Winer Bao on 18/02/15.
//  for Tekt Industries Pty. Ltd.
//  Copyright (c) 2015 Winer Bao. All rights reserved.
//

#ifndef __Networked_LED_Driver_System_Server_MessageHandler__
#define __Networked_LED_Driver_System_Server_MessageHandler__

#define _WINSOCK_DEPRECATED_NO_WARNINGS

#include <stdio.h>
#include <sys/types.h>
#include <winsock2.h>
#include <ws2tcpip.h>
#include <queue>
#include <thread>
#include <mutex>
#include <condition_variable>
#include "Definitions.h"
#include "Database.h"

#pragma comment(lib,"ws2_32.lib") //Winsock Library

/* Size of the receive buffer in bytes */
const int BUFSIZE = 65507;

enum Response
{
    WRONGFORMAT,
    ACCESSDENIED,
    NOTREGISTERED,
    NODRIVER,
    REGISTERCONFIFRM,
    ALREADYCONNECTED,
    CHANNELOCCUPIED,
    CHANNELRECONFIGURED,
    DISCONNECTCONFIRM,
    WRONGSETTINGS,
};

class Socket
{
public:
    Socket(Mapping& m);
    ~Socket();
    int setupUDPsocket();
    void checkUDPmsg();
    void processSenderDT(Mapping& map);
    void processSenderCMD(Mapping& map);
    void processDriverDT(Mapping& map);
    void processDriverCMD(Mapping& map);
    void processAdminDT(Mapping& map);
    void processAdminCMD(Mapping& map);
    void sendUDPmsg(Mapping& map);

private:
    struct sockaddr_in myaddr;
    struct sockaddr_in remaddr;
    int addrlen = sizeof(remaddr);
    int recvlen;
    SOCKET s;
    WSADATA wsa;
    char buf[BUFSIZE];
    bool Server_isRunning;
    bool isSenderCMDMessageFormatCorrect(const std::vector<unsigned char>& bufIn) const;
    bool isDriverDTMessageFormatCorrect(const std::vector<unsigned char>& bufIn) const;
    bool isDriverCMDMessageFormatCorrect(const std::vector<unsigned char>& bufIn) const;
    bool isAdminDTMessageFormatCorrect(const std::vector<unsigned char>& bufIn) const;
```

\\psf\Dropbox\Elektrotechniek 4de\Afstuderen ...\Networked LED Driver System\MessageHandler.h 2

```
    bool isAdminCMDMessageFormatCorrect(const std::vector<unsigned char>& bufIn) const;
    void feedbackMessage(const Response& response, const sockaddr_in& msgaddr);

    /* Mutexes */
    std::mutex mtxSenderDT;
    std::mutex mtxSenderCMD;
    std::mutex mtxDriverDT;
    std::mutex mtxDriverCMD;
    std::mutex mtxAdminDT;
    std::mutex mtxAdminCMD;
    std::mutex mtxOut_UDP;
    std::mutex mtxOut_USB;

    /* conditional variables */
    std::condition_variable cond_varsSenderDT;
    std::condition_variable cond_varsSenderCMD;
    std::condition_variable cond_varsDriverDT;
    std::condition_variable cond_varsDriverCMD;
    std::condition_variable cond_varsAdminDT;
    std::condition_variable cond_varsAdminCMD;
    std::condition_variable cond_varsOut_UDP;

    /* message inboxes/outbox */
    std::queue <std::vector<unsigned char>> queueSenderDT;
    std::queue <std::vector<unsigned char>> queueSenderCMD;
    std::queue <std::vector<unsigned char>> queueDriverDT;
    std::queue <std::vector<unsigned char>> queueDriverCMD;
    std::queue <std::vector<unsigned char>> queueAdminDT;
    std::queue <std::vector<unsigned char>> queueAdminCMD;
    std::queue <std::vector<unsigned char>> OutgoingMessageQueue_UDP;

    /* IP and port number storage */
    std::queue <sockaddr_in> sockaddrSenderDT;
    std::queue <sockaddr_in> sockaddrSenderCMD;
    std::queue <sockaddr_in> sockaddrDriverDT;
    std::queue <sockaddr_in> sockaddrDriverCMD;
    std::queue <sockaddr_in> sockaddrAdminDT;
    std::queue <sockaddr_in> sockaddrAdminCMD;
    std::queue <sockaddr_in> sockaddrOut_UDP;
};

#endif /* defined(__Networked_LED_Driver_System_Server_MessageHandler__) */
```

\\psf\Dropbox\Elektrotechniek 4de\Afstuderen ...\Networked LED Driver System\MessageHandler.cpp 1

```
//
//  MessageHandler.cpp
//  Networked LED Driver System server
//
//  Created by Winer Bao on 18/02/15.
//  for Tekt Industries Pty. Ltd.
//  Copyright (c) 2015 Winer Bao. All rights reserved.
//

#include "MessageHandler.h"

// Global variable
// Storage container for all threads
std::vector<std::thread> thread;

// Constructor
// Starts the UDP socket setup and start program threads
Socket::Socket(Mapping& m) : Server_isRunning(true)
{
    /* Initialize UDP socket */
    setupUDPsocket();

    /* Start threads */
    thread.push_back(std::thread(&Socket::checkUDPmsg, this));
    for (int i = 0; i < TOTAL_SENDERDATAPROCESSINGTHREADS; ++i)
    {
        thread.push_back(std::thread(&Socket::processSenderDT, this, std::ref(m)));
    }
    for (int i = 0; i < TOTAL_SENDERCOMMANDPROCESSINGTHREADS; ++i)
    {
        thread.push_back(std::thread(&Socket::processSenderCMD, this, std::ref(m)));
    }
    for (int i = 0; i < TOTAL_DRIVERDATAPROCESSINGTHREADS; ++i)
    {
        thread.push_back(std::thread(&Socket::processDriverDT, this, std::ref(m)));
    }
    for (int i = 0; i < TOTAL_DRIVERCOMMANDPROCESSINGTHREADS; ++i)
    {
        thread.push_back(std::thread(&Socket::processDriverCMD, this, std::ref(m)));
    }
    for (int i = 0; i < TOTAL_ADMINDATAPROCESSINGTHREADS; ++i)
    {
        thread.push_back(std::thread(&Socket::processAdminDT, this, std::ref(m)));
    }
    for (int i = 0; i < TOTAL_ADMINCOMMANDPROCESSINGTHREADS; ++i)
    {
        thread.push_back(std::thread(&Socket::processAdminCMD, this, std::ref(m)));
    }
    thread.push_back(std::thread(&Socket::sendUDPmsg, this, std::ref(m)));
}

// Destructor
// Wakes up all threads and allow them to exit there loop
// All threads are joined before the object is destroyed
Socket::~Socket()
{
    Server_isRunning = false;
    cond_varsSenderDT.notify_all();
    cond_varsSenderCMD.notify_all();
    cond_varsDriverDT.notify_all();
    cond_varsDriverCMD.notify_all();
    cond_varsAdminDT.notify_all();
    cond_varsAdminCMD.notify_all();
    cond_varsOut_UDP.notify_all();
    closesocket(s);

    /* Wait until all threads finishes */
    for (auto& t : thread)
    {
        t.join();
    }
}
```

```
// Creates a socket.
// IP and port are defined in the header file.
// Returns 0 when socket setup is not succesful.
int Socket::setupUDPsocket()
{
    //Initialise winsock
    printf("\nInitialising Winsock...");
    if (WSAStartup(MAKEWORD(2, 2), &wsa) != 0)
    {
        printf("Failed. Error Code : %d", WSAGetLastError());
        exit(EXIT_FAILURE);
    }
    printf("Initialised.\n");

    //Create a socket
    if ((s = socket(AF_INET, SOCK_DGRAM, 0)) == INVALID_SOCKET)
    {
        printf("Could not create socket : %d", WSAGetLastError());
    }
    printf("Socket created.\n");

    memset((char*)&myaddr, 0, sizeof(myaddr));
    myaddr.sin_family = AF_INET;
    myaddr.sin_addr.s_addr = inet_addr(SERVER_ADDR);
    myaddr.sin_port = htons(SERVER_PORT);

    //Bind
    if (bind(s, (struct sockaddr *)&myaddr, sizeof(myaddr)) == SOCKET_ERROR)
    {
        printf("Bind failed with error code : %d", WSAGetLastError());
        exit(EXIT_FAILURE);
    }
    std::cout << "Bind done" << std::endl;
    std::cout << "Server IP: " << SERVER_ADDR << " port: " << SERVER_PORT << std::endl << std::endl;

    return 1;
}

// Waits for incoming UDP message and sorts the message in their appropiate queue
void Socket::checkUDPmsg()
{
    while (Server_isRunning)
    {
        //printf("waiting on port %d\n", PORT);
        recvlen = recvfrom(s, buf, BUFSIZE, 0, (struct sockaddr*)&remaddr, &addrlen);
        //printf("received %d bytes\n", recvlen);

        if (!Server_isRunning)
        {
            return;
        }

        if (recvlen != SOCKET_ERROR)
        {
            std::vector<unsigned char> message;

            for (int i = 0; i < recvlen; i++)
            {
                message.push_back(buf[i]);
            }

            switch (buf[0])
            {
                case SENDERDATA:
                {
                    std::unique_lock<std::mutex> lckIn(mtxSenderDT);
                    queueSenderDT.push(message);
                    sockaddrSenderDT.push(remaddr);
                    cond_varsSenderDT.notify_one();
                    break;
                }
```

```
                case SENDERCMD:
                {
                    std::unique_lock<std::mutex> lckIn(mtxSenderCMD);
                    queueSenderCMD.push(message);
                    sockaddrSenderCMD.push(remaddr);
                    cond_varsSenderCMD.notify_one();
                    break;
                }

                case DRIVERDATA:
                {
                    std::unique_lock<std::mutex> lckIn(mtxDriverDT);
                    queueDriverDT.push(message);
                    sockaddrDriverDT.push(remaddr);
                    cond_varsDriverDT.notify_one();
                    break;
                }

                case DRIVERCMD:
                {
                    std::unique_lock<std::mutex> lckIn(mtxDriverCMD);
                    queueDriverCMD.push(message);
                    sockaddrDriverCMD.push(remaddr);
                    cond_varsDriverCMD.notify_one();
                    break;
                }

                case ADMINDATA:
                {
                    std::unique_lock<std::mutex> lckIn(mtxAdminDT);
                    queueAdminDT.push(message);
                    sockaddrAdminDT.push(remaddr);
                    cond_varsAdminDT.notify_one();
                    break;
                }

                case ADMINCMD:
                {
                    std::unique_lock<std::mutex> lckIn(mtxAdminCMD);
                    queueAdminCMD.push(message);
                    sockaddrAdminCMD.push(remaddr);
                    cond_varsAdminCMD.notify_one();
                    break;
                }

                default:
                {
                    std::cout << "Message type not recognized." << std::endl;
                    break;
                }
            }
        }
        else
        {
            std::cout << "Receive error" << std::endl;
        }
    }
}

// Extract information from the received message.
// The information are saved for logging purposes.
// The message contains color data
void Socket::processSenderDT(Mapping& map)
{
    while (Server_isRunning)
    {
        std::unique_lock<std::mutex> lckIn(mtxSenderDT);
        cond_varsSenderDT.wait(lckIn, [&](){return (queueSenderDT.size() >= 1 || !Server_isRunning); });

        if (!Server_isRunning)
        {
```

\\psf\Dropbox\Elektrotechniek 4de\Afstuderen ...\Networked LED Driver System\MessageHandler.cpp 4

```
            return;
        }

        //std::cout << "SenderDT inbox: " << queueSenderDT.size() << std::endl;
        struct sockaddr_in msgaddr = sockaddrSenderDT.front();
        std::vector<unsigned char>bufIn = queueSenderDT.front();
        sockaddrSenderDT.pop();
        queueSenderDT.pop();
        lckIn.unlock();

        int msgChannel = ByteToInt(bufIn[1], (bufIn[2]));

        if (map.checkSenderDTformat(msgChannel, bufIn))
        {
            if (map.doesChannelExist(msgChannel))
            {
                if (map.isChannelOccupied(msgChannel))
                {
                    if (map.doesChannelBelongToSender(msgChannel, inet_ntoa(msgaddr.sin_addr), ntohs
    (msgaddr.sin_port)))
                    {
                        std::vector<unsigned char> TXbuf;

                        if (map.getConnectionType(msgChannel) == UDP)
                        {
                            map.decodeUDP(msgChannel, TXbuf, bufIn);
                            msgaddr.sin_port = htons(map.getDriverPort(msgChannel));
                            inet_pton(AF_INET, map.getDriverIPaddr(msgChannel).c_str(), &msgaddr.
    sin_addr);

                            std::unique_lock<std::mutex> lckOut(mtxOut_UDP);
                            OutgoingMessageQueue_UDP.push(TXbuf);
                            sockaddrOut_UDP.push(msgaddr);
                            cond_varsOut_UDP.notify_one();
                        }
                        else //if (map.getConnectionType(msgChannel) == USB)
                        {
                            map.decodeUSB(msgChannel, TXbuf, bufIn);
                            std::unique_lock<std::mutex> lckOut(mtxOut_USB);
                            sendUSB_RGBStripData(TXbuf);
                        }
                    }
                    else
                    {
                        printf("Access denied because you're not the owner\n");
                        feedbackMessage(ACCESSDENIED, msgaddr);
                    }
                }
                else
                {
                    printf("You're not registered\n");
                    feedbackMessage(NOTREGISTERED, msgaddr);
                }
            }
            else
            {
                printf("No driver board available for this channel\n");
                feedbackMessage(NODRIVER, msgaddr);
            }
        }
        else
        {
            printf("Sender message wrong format\n");
            feedbackMessage(WRONGFORMAT, msgaddr);
        }
    }
}

// Extract information from the received message.
// The information are saved for logging purposes.
// The message contains a command.
void Socket::processSenderCMD(Mapping& map)
```

\\psf\Dropbox\Elektrotechniek 4de\Afstuderen ...\Networked LED Driver System\MessageHandler.cpp 5

```
{
    while (Server_isRunning)
    {
        std::unique_lock<std::mutex> lckIn(mtxSenderCMD);
        cond_varsSenderCMD.wait(lckIn, [&](){return (queueSenderCMD.size() >= 1 || !Server_isRunning);
     });

        if (!Server_isRunning)
        {
            return;
        }

        //  std::cout << "SenderCMD inbox: " << queueSenderCMD.size() << std::endl;
        struct sockaddr_in msgaddr = sockaddrSenderCMD.front();
        std::vector<unsigned char> bufIn = queueSenderCMD.front();
        sockaddrSenderCMD.pop();
        queueSenderCMD.pop();
        lckIn.unlock();

        if (isSenderCMDMessageFormatCorrect(bufIn))
        {
            switch (bufIn[1])
            {
                case SENDER_CONNECT:
                {
                    int msgChannel = ByteToInt(bufIn[3], bufIn[4]);

                    if (map.doesChannelExist(msgChannel))
                    {
                        if (!map.isChannelOccupied(msgChannel))
                        {
                            class_t class_type;
                            color_mode color;
                            unsigned char msgColor = bufIn[6];
                            unsigned char msgClass = bufIn[8];

                            if (msgClass == CLASS_STRIP)
                            {
                                class_type = STRIP;
                            }
                            else if (msgClass == CLASS_MATRIX)
                            {
                                class_type = MATRIX;
                            }
                            else //if (msgClass == CLASS_RAW)
                            {
                                class_type = RAW;
                            }

                            if (msgColor == COLOR_MONO)
                            {
                                color = MONO;
                            }
                            else //if (msgColor == COLOR_RGB)
                            {
                                color = RGB;
                            }

                            if (msgClass == CLASS_STRIP)
                            {
                                int msgLength = ByteToInt(bufIn[10], bufIn[11]);
                                unsigned char msgType = bufIn[13];
                                unsigned char msgConfig = bufIn[15];
                                led_t type;
                                config_t config;

                                if (msgType == TYPE_ADDRESSABLE)
                                {
                                    type = ADDRESSABLE;
                                }
                                else //if (msgType == TYPE_NONADDRESSABLE)
                                {
```

```
                                type = NONADDRESSABLE;
                            }

                            if (msgConfig == CONFIG_LINE)
                            {
                                config = LINE;
                            }
                            else //if (msgConfig == CONFIG_OTHER)
                            {
                                config = OTHER;
                            }


                            if (map.getClass(msgChannel) == class_type && map.getColorMode
(msgChannel) == color && map.getLength(msgChannel) == msgLength && map.getType(msgChannel) == type &
& map.getConfig(msgChannel) == config)
                            {
                                map.registerSender(msgChannel, inet_ntoa(msgaddr.sin_addr), ntohs
(msgaddr.sin_port));
                                printf("Channel %d just connected\n", msgChannel);
                                feedbackMessage(REGISTERCONFIFRM, msgaddr);
                            }
                            else
                            {
                                printf("Channel %d settings does not match\n", msgChannel);
                                feedbackMessage(WRONGSETTINGS, msgaddr);
                            }
                        }
                        else if (msgClass == CLASS_MATRIX)
                        {
                            int msgWidth = ByteToInt(bufIn[10], bufIn[11]);
                            int msgHeight = ByteToInt(bufIn[13], bufIn[14]);

                            if (map.getClass(msgChannel) == class_type && map.getColorMode
(msgChannel) == color && map.getWidth(msgChannel) == msgWidth && map.getHeight(msgChannel) ==
msgHeight)
                            {
                                map.registerSender(msgChannel, inet_ntoa(msgaddr.sin_addr), ntohs
(msgaddr.sin_port));
                                printf("Channel %d just connected\n", msgChannel);
                                feedbackMessage(REGISTERCONFIFRM, msgaddr);
                            }
                            else
                            {
                                printf("Channel %d settings does not match\n", msgChannel);
                                feedbackMessage(WRONGSETTINGS, msgaddr);
                            }
                        }
                        else if (msgClass == CLASS_RAW)
                        {
                            if (map.getClass(msgChannel) == class_type)
                            {
                                map.registerSender(msgChannel, inet_ntoa(msgaddr.sin_addr), ntohs
(msgaddr.sin_port));
                                printf("Channel %d just connected\n", msgChannel);
                                feedbackMessage(REGISTERCONFIFRM, msgaddr);
                            }
                            else
                            {
                                printf("Channel %d settings does not match\n", msgChannel);
                                feedbackMessage(WRONGSETTINGS, msgaddr);
                            }
                        }
                    }
                    else
                    {
                        if (map.doesChannelBelongToSender(msgChannel, inet_ntoa(msgaddr.sin_addr),
ntohs(msgaddr.sin_port)))
                        {
                            printf("You're already connected to this channel\n");
                            feedbackMessage(ALREADYCONNECTED, msgaddr);
                        }
```

\\psf\Dropbox\Elektrotechniek 4de\Afstuderen ...\Networked LED Driver System\MessageHandler.cpp 7

```
                            else
                            {
                                printf("Channel is already in use\n");
                                feedbackMessage(CHANNELOCCUPIED, msgaddr);
                            }
                        }
                    }
                    else
                    {
                        printf("No driver board available for this channel\n");
                        feedbackMessage(NODRIVER, msgaddr);
                    }
                    break;
                }

                case SENDER_DISCONNECT:
                {
                    int msgChannel = ByteToInt(bufIn[3], bufIn[4]);

                    if (map.doesChannelBelongToSender(msgChannel, inet_ntoa(msgaddr.sin_addr), ntohs ↙
    (msgaddr.sin_port)))
                    {
                        map.deregisterSender(msgChannel);
                        printf("Channel %d just disconnected\n", msgChannel);
                        feedbackMessage(DISCONNECTCONFIRM, msgaddr);
                    }
                    else
                    {
                        printf("Can't disconnect this channel because you're not the owner\n");
                        feedbackMessage(ACCESSDENIED, msgaddr);
                    }
                    break;
                }

                default:
                {
                    break;
                }
            }
        }
        else
        {
            printf("Sender message wrong format\n");
            feedbackMessage(WRONGFORMAT, msgaddr);
        }
    }
}

// Extract information from the received message.
// The information are saved for logging purposes.
// The message contains driver information.
void Socket::processDriverDT(Mapping& map)
{
    while (Server_isRunning)
    {
        std::unique_lock<std::mutex> lckIn(mtxDriverDT);
        cond_varsDriverDT.wait(lckIn, [&](){return (queueDriverDT.size() >= 1 || !Server_isRunning); });

        if (!Server_isRunning)
        {
            return;
        }

        //  std::cout << "DriverDT inbox: " << queueDriverDT.size() << std::endl;
        struct sockaddr_in msgaddr = sockaddrDriverDT.front();
        std::vector<unsigned char> bufIn = queueDriverDT.front();
        sockaddrDriverDT.pop();
        queueDriverDT.pop();
        lckIn.unlock();

        if (isDriverDTMessageFormatCorrect(bufIn))
        {
```

```
            std::vector<unsigned char> TXbuf;

            // Implement in the future

            //std::unique_lock<std::mutex> lckOut(mtxOut_UDP);
            //OutgoingMessageQueue_UDP.push(TXbuf);
            //sockaddrOut_UDP.push(msgaddr);
            //cond_varsOut_UDP.notify_one();
            std::cout << "This is a DRIVERDATA\n" << std::endl;
        }
        else
        {
            printf("Driver message wrong format\n");
            feedbackMessage(WRONGFORMAT, msgaddr);
        }
    }
}

// Extract information from the received message.
// The information are saved for logging purposes.
// The message contains a command.
void Socket::processDriverCMD(Mapping& map)
{
    while (Server_isRunning)
    {
        std::unique_lock<std::mutex> lckIn(mtxDriverCMD);
        cond_varsDriverCMD.wait(lckIn, [&](){return (queueDriverCMD.size() >= 1 || !Server_isRunning);
     });

        if (!Server_isRunning)
        {
            return;
        }

        //  std::cout << "DriverCMD inbox: " << queueDriverCMD.size() << std::endl;
        struct sockaddr_in msgaddr = sockaddrDriverCMD.front();
        std::vector<unsigned char> bufIn = queueDriverCMD.front();
        sockaddrDriverCMD.pop();
        queueDriverCMD.pop();
        lckIn.unlock();

        if (isDriverCMDMessageFormatCorrect(bufIn))
        {
            int msgChannel = ByteToInt(bufIn[3], bufIn[4]);

            switch (bufIn[1])
            {
                case DRIVER_CONNECT:
                {
                    if (!map.isDriverOccupied(msgChannel))
                    {
                        map.registerDriver(msgChannel, msgChannel, inet_ntoa(msgaddr.sin_addr), ntohs
    (msgaddr.sin_port), UDP);
                        printf("Driver %d just connected\n", msgChannel);
                        feedbackMessage(REGISTERCONFIFRM, msgaddr);
                    }
                    else
                    {
                        if (map.doesChannelBelongToDriver(msgChannel, inet_ntoa(msgaddr.sin_addr), ntohs
    (msgaddr.sin_port)))
                        {
                            printf("You're already connected to this channel\n");
                            feedbackMessage(ALREADYCONNECTED, msgaddr);
                        }
                        else
                        {
                            printf("Channel is already in use\n");
                            feedbackMessage(CHANNELOCCUPIED, msgaddr);
                        }
                    }
                    break;
                }
```

```
                case DRIVER_DISCONNECT:
                {
                    if (map.doesChannelBelongToDriver(msgChannel, inet_ntoa(msgaddr.sin_addr), ntohs
    (msgaddr.sin_port)))
                    {
                        map.deregisterDriver(msgChannel);
                        printf("Driver %d just disconnected\n", msgChannel);
                        feedbackMessage(DISCONNECTCONFIRM, msgaddr);
                    }
                    break;
                }

                default:
                {
                    break;
                }
            }
        }
        else
        {
            printf("Driver message wrong format\n");
            feedbackMessage(WRONGFORMAT, msgaddr);
        }
    }
}

// Extract information from the received message.
// The information are saved for logging purposes.
// The message contains a data request.
void Socket::processAdminDT(Mapping& map)
{
    while (Server_isRunning)
    {
        std::unique_lock<std::mutex> lckIn(mtxAdminDT);
        cond_varsAdminDT.wait(lckIn, [&](){return (queueAdminDT.size() >= 1 || !Server_isRunning); });

        if (!Server_isRunning)
        {
            return;
        }

        //  std::cout << "AdminDT inbox: " << queueAdminDT.size() << std::endl;
        struct sockaddr_in msgaddr = sockaddrAdminDT.front();
        std::vector<unsigned char> bufIn = queueAdminDT.front();
        sockaddrAdminDT.pop();
        queueAdminDT.pop();
        lckIn.unlock();

        if (isAdminDTMessageFormatCorrect(bufIn))
        {
            if (map.doesBelongToAdmin(inet_ntoa(msgaddr.sin_addr), ntohs(msgaddr.sin_port)))
            {
                std::vector<unsigned char> TXbuf;
                std::string temp;
                switch (bufIn[1])
                {
                case REQUEST_SENDERDATA:
                {
                    temp = map.getDatabase_SenderInfo();
                    TXbuf.insert(TXbuf.begin(), temp.begin(), temp.end());
                    break;
                }

                case REQUEST_DRIVERDATA:
                {

                    temp = map.getDatabase_DriverInfo();
                    TXbuf.insert(TXbuf.begin(), temp.begin(), temp.end());
                    break;
                }
```

```
                default:
                    break;
                }

                std::unique_lock<std::mutex> lckOut(mtxOut_UDP);
                OutgoingMessageQueue_UDP.push(TXbuf);
                sockaddrOut_UDP.push(msgaddr);
                cond_varsOut_UDP.notify_one();
            }
            else
            {
                printf("Access denied. You're not the admin\n");
                feedbackMessage(ACCESSDENIED, msgaddr);
            }
        }
    }
}

// Extract information from the received message.
// The information are saved for logging purposes.
// The message contains a command.
void Socket::processAdminCMD(Mapping& map)
{
    while (Server_isRunning)
    {
        std::unique_lock<std::mutex> lckIn(mtxAdminCMD);
        cond_varsAdminCMD.wait(lckIn, [&](){return (queueAdminCMD.size() >= 1 || !Server_isRunning); });

        if (!Server_isRunning)
        {
            return;
        }

        //  std::cout << "AdminCMD inbox: " << queueAdminCMD.size() << std::endl;
        struct sockaddr_in msgaddr = sockaddrAdminCMD.front();
        std::vector<unsigned char> bufIn = queueAdminCMD.front();
        sockaddrAdminCMD.pop();
        queueAdminCMD.pop();
        lckIn.unlock();

        if (isAdminCMDMessageFormatCorrect(bufIn))
        {
            switch (bufIn[1])
            {
            case ADMIN_CONNECT:
            {
                if (map.doesAdminExist())
                {
                    if (map.doesBelongToAdmin(inet_ntoa(msgaddr.sin_addr), ntohs(msgaddr.sin_port)))
                    {
                        printf("You're already the admin\n");
                        feedbackMessage(ALREADYCONNECTED, msgaddr);
                    }
                    else
                    {
                        printf("Admin already exists already\n");
                        feedbackMessage(CHANNELOCCUPIED, msgaddr);
                    }
                }
                else
                {
                    map.registerAdmin(inet_ntoa(msgaddr.sin_addr), ntohs(msgaddr.sin_port));
                    printf("Admin just logged in\n");
                    feedbackMessage(REGISTERCONFIFRM, msgaddr);
                }
                break;
            }

            case ADMIN_DISCONNECT:
            {
                if (map.doesBelongToAdmin(inet_ntoa(msgaddr.sin_addr), ntohs(msgaddr.sin_port)))
                {
```

```
                map.deregisterAdmin();
                printf("Admin just logged off\n");
                feedbackMessage(DISCONNECTCONFIRM, msgaddr);
            }
            else
            {
                printf("Can't log off. You're not the admin\n");
                feedbackMessage(ACCESSDENIED, msgaddr);
            }
            break;
        }

        case CHANNELMOD:
        {
            if (map.doesBelongToAdmin(inet_ntoa(msgaddr.sin_addr), ntohs(msgaddr.sin_port)))
            {
                int channel = ByteToInt(bufIn[3], bufIn[4]);
                color_mode color;
                int group = ByteToInt(bufIn[10], bufIn[11]);

                if (bufIn[8] == COLOR_MONO)
                {
                    color = MONO;
                }
                else if (bufIn[8] == COLOR_RGB)
                {
                    color = RGB;
                }
                try
                {
                    switch (bufIn[6])
                    {
                    case CLASS_STRIP:
                    {
                        int length = ByteToInt(bufIn[17], bufIn[18]);
                        led_t type;
                        config_t config;
                        std::vector<unsigned char> DriverNotifyMessage;

                        if (bufIn[13] == TYPE_ADDRESSABLE)
                        {
                            type = ADDRESSABLE;
                        }
                        else //if (bufIn[13] == TYPE_NONADDRESSABLE)
                        {
                            type = NONADDRESSABLE;
                        }

                        if (bufIn[15] == CONFIG_LINE)
                        {
                            config = LINE;
                        }
                        else //if (bufIn[15] == CONFIG_OTHER)
                        {
                            config = OTHER;
                        }

                        map.changeStrip(channel, length, type, config, color, group);
                        printf("Channel %d changed strip\n", channel);
                        feedbackMessage(CHANNELRECONFIGURED, msgaddr);

                        if (map.getConnectionType(channel) == UDP)
                        {
                            DriverNotifyMessage.push_back('#');
                            DriverNotifyMessage.push_back((channel >> 8) & 0xFF);
                            DriverNotifyMessage.push_back(channel & 0xFF);
                            DriverNotifyMessage.push_back('/');
                            DriverNotifyMessage.push_back(SET_LENGTH);
                            DriverNotifyMessage.push_back('/');
                            DriverNotifyMessage.push_back((length >> 8) & 0xFF);
                            DriverNotifyMessage.push_back(length & 0xFF);
```

\\psf\Dropbox\Elektrotechniek 4de\Afstuderen ...\Networked LED Driver System\MessageHandler.cpp 12

```
                                msgaddr.sin_addr.s_addr = inet_addr(map.getDriverIPaddr(channel).c_str
    ());
                                msgaddr.sin_port = htons(map.getDriverPort(channel));

                                std::unique_lock<std::mutex> lckOut(mtxOut_UDP);
                                OutgoingMessageQueue_UDP.push(DriverNotifyMessage);
                                sockaddrOut_UDP.push(msgaddr);
                                cond_varsOut_UDP.notify_one();
                            }
                            break;
                        }

                        case CLASS_MATRIX:
                        {
                            int height = ByteToInt(bufIn[13], bufIn[14]);
                            int width = ByteToInt(bufIn[16], bufIn[17]);
                            map.changeMatrix(channel, width, height, color, group);
                            printf("Channel %d changed matrix\n", channel);
                            feedbackMessage(CHANNELRECONFIGURED, msgaddr);
                            break;
                        }

                        case CLASS_RAW:
                        {
                            map.changeRaw(channel, group);
                            printf("Channel %d changed raw\n", channel);
                            feedbackMessage(CHANNELRECONFIGURED, msgaddr);
                            break;
                        }

                        default:
                            break;
                        }
                    }
                    catch (MyException& e)
                    {
                        std::cout << e.what() << std::endl;
                    }
                }
            default:
                break;
            }
            }
        }
        else
        {
            printf("Admin message wrong format\n");
            feedbackMessage(WRONGFORMAT, msgaddr);
        }
    }
}

// Sends processed UDP message to the driver board.
void Socket::sendUDPmsg(Mapping& map)
{
    while (Server_isRunning)
    {
        std::unique_lock<std::mutex> lckOut(mtxOut_UDP);
        cond_varsOut_UDP.wait(lckOut, [&](){return (OutgoingMessageQueue_UDP.size() >= 1 || !
    Server_isRunning); });

        if (!Server_isRunning)
        {
            return;
        }

        //std::cout << "UDP Outgoing inbox: " << OutgoingMessageQueue_UDP.size() << std::endl;
        std::vector<unsigned char> bufOut = OutgoingMessageQueue_UDP.front();
        struct sockaddr_in receivaddr = sockaddrOut_UDP.front();
        sockaddrOut_UDP.pop();
        OutgoingMessageQueue_UDP.pop();
        lckOut.unlock();
```

```
        if (sendto(s, reinterpret_cast<char*> (&bufOut[0]), bufOut.size(), 0, (struct sockaddr*)&
    receivaddr, sizeof(receivaddr)) < 0)
        {
            perror("sendto failed");
        }
    }
}

// Checks if the message conforms to the communication protocol
bool Socket::isSenderCMDMessageFormatCorrect(const std::vector<unsigned char>& bufIn) const
{
    if (!(bufIn[1] == SENDER_CONNECT || bufIn[1] == SENDER_DISCONNECT))
    {
        return false;
    }

    bool valid = true;

    if (bufIn[1] == SENDER_CONNECT)
    {
        valid = (bufIn[2] == '/');

        if (bufIn[8] == CLASS_STRIP)
        {
            valid &= (bufIn[5] == '/')
                && (bufIn[6] == COLOR_MONO || bufIn[6] == COLOR_RGB)
                && (bufIn[7] == '/')
                && (bufIn[9] == '/')
                && (bufIn[12] == '/')
                && (bufIn[13] == TYPE_ADDRESSABLE || bufIn[13] == TYPE_NONADDRESSABLE)
                && (bufIn[14] == '/')
                && (bufIn[15] == CONFIG_LINE || bufIn[16] == CONFIG_OTHER);
        }
        else if (bufIn[8] == CLASS_MATRIX)
        {
            valid &= (bufIn[5] == '/')
                && (bufIn[6] == COLOR_MONO || bufIn[6] == COLOR_RGB)
                && (bufIn[7] == '/')
                && (bufIn[9] == '/')
                && (bufIn[12] == '/');
        }
        else if (bufIn[8] == CLASS_RAW)
        {
            valid &= (bufIn[5] == '/')
                && (bufIn[6] == COLOR_MONO || bufIn[6] == COLOR_RGB)
                && (bufIn[7] == '/');
        }
        else
        {
            valid = false;
        }
    }
    else if (bufIn[1] == SENDER_DISCONNECT)
    {
        valid = (bufIn[2] == '/');
    }

    return valid;
}

// Checks if the message conforms to the communication protocol
bool Socket::isDriverDTMessageFormatCorrect(const std::vector<unsigned char>& bufIn) const
{
    if (bufIn[2] == '/')
    {
        return true;
    }
    else
    {
        return false;
    }
```

```
}

// Checks if the message conforms to the communication protocol
bool Socket::isDriverCMDMessageFormatCorrect(const std::vector<unsigned char>& bufIn) const
{
    if (!(bufIn[1] == DRIVER_CONNECT || bufIn[1] == DRIVER_DISCONNECT))
    {
        return false;
    }

    if (bufIn[3] == '/')
    {
        return true;
    }
    else
    {
        return false;
    }
}

// Checks if the message conforms to the communication protocol
bool Socket::isAdminDTMessageFormatCorrect(const std::vector<unsigned char>& bufIn) const
{
    if (!(bufIn[1] == REQUEST_SENDERDATA || bufIn[1] == REQUEST_DRIVERDATA))
    {
        return false;
    }

    bool valid = true;

    if (bufIn[1] == REQUEST_SENDERDATA)
    {
        valid = (bufIn[2] == '/');
    }
    else if (bufIn[1] == REQUEST_DRIVERDATA)
    {
        valid = (bufIn[2] == '/');
    }
    else
    {
        valid = false;
    }

    return valid;
}

// Checks if the message conforms to the communication protocol
bool Socket::isAdminCMDMessageFormatCorrect(const std::vector<unsigned char>& bufIn) const
{
    if (!(bufIn[1] == ADMIN_CONNECT || bufIn[1] == ADMIN_DISCONNECT || bufIn[1] == CHANNELMOD))
    {
        return false;
    }

    bool valid = true;

    if (bufIn[1] == ADMIN_CONNECT)
    {
        valid = (bufIn[2] == '/');
    }
    else if (bufIn[1] == ADMIN_DISCONNECT)
    {
        valid = (bufIn[2] == '/');
    }
    else if (bufIn[1] == CHANNELMOD)
    {
        if (bufIn[6] == CLASS_STRIP)
        {
            valid = (bufIn[2] == '/')
                && (bufIn[5] == '/')
                && (bufIn[7] == '/')
                && (bufIn[9] == '/')
```

```
                && (bufIn[12] == '/')
                && (bufIn[14] == '/')
                && (bufIn[16] == '/');
        }
        else if (bufIn[6] == CLASS_MATRIX)
        {
            valid = (bufIn[2] == '/')
                && (bufIn[5] == '/')
                && (bufIn[7] == '/')
                && (bufIn[9] == '/')
                && (bufIn[12] == '/')
                && (bufIn[15] == '/');
        }
        else if (bufIn[6] == CLASS_RAW)
        {
            valid = (bufIn[2] == '/')
                && (bufIn[5] == '/')
                && (bufIn[7] == '/')
                && (bufIn[9] == '/')
                && (bufIn[12] == '/');
        }
        else
        {
            valid = false;
        }
    }

    return valid;
}

// Checks if the message conforms to the communication protocol
void Socket::feedbackMessage(const Response& response, const sockaddr_in& msgaddr)
{
    std::vector<unsigned char> MessageResponse;

    switch (response)
    {
        case WRONGFORMAT:
        {
            std::string str = "Wrong format";
            std::copy(str.begin(), str.end(), back_inserter(MessageResponse));
            break;
        }

        case ACCESSDENIED:
        {
            std::string str = "Access denied";
            std::copy(str.begin(), str.end(), back_inserter(MessageResponse));
            break;
        }

        case NOTREGISTERED:
        {
            std::string str = "Not registered";
            std::copy(str.begin(), str.end(), back_inserter(MessageResponse));
            break;
        }

        case NODRIVER:
        {
            std::string str = "No driver connected to this channel";
            std::copy(str.begin(), str.end(), back_inserter(MessageResponse));
            break;
        }

        case REGISTERCONFIFRM:
        {
            std::string str = "Connect successful";
            std::copy(str.begin(), str.end(), back_inserter(MessageResponse));
            break;
        }
```

\\psf\Dropbox\Elektrotechniek 4de\Afstuderen ...\Networked LED Driver System\MessageHandler.cpp 16

```
        case ALREADYCONNECTED:
        {
            std::string str = "You're already connected to this channel";
            std::copy(str.begin(), str.end(), back_inserter(MessageResponse));
            break;
        }

        case CHANNELOCCUPIED:
        {
            std::string str = "Channel is already taken";
            std::copy(str.begin(), str.end(), back_inserter(MessageResponse));
            break;
        }

        case CHANNELRECONFIGURED:
        {
            std::string str = "Reconfiguration successful";
            std::copy(str.begin(), str.end(), back_inserter(MessageResponse));
            break;
        }

        case DISCONNECTCONFIRM:
        {
            std::string str = "Disconnect successful";
            std::copy(str.begin(), str.end(), back_inserter(MessageResponse));
            break;
        }

        case WRONGSETTINGS:
        {
            std::string str = "Wrong settings";
            std::copy(str.begin(), str.end(), back_inserter(MessageResponse));
            break;
        }
    }

    std::unique_lock<std::mutex> lckOut(mtxOut_UDP);
    OutgoingMessageQueue_UDP.push(MessageResponse);
    sockaddrOut_UDP.push(msgaddr);
    cond_varsOut_UDP.notify_one();
}
```

# Appendix 6: USBdriver.h and USBdriver.cpp source code

\\psf\Dropbox\Elektrotechniek 4de\Afstuderen ... System\Networked LED Driver System\USBdriver.h 1

```
//
//  USBdriver.h
//  Networked LED Driver System server
//
//  Created by Kia Riegel and Winer Bao on 14/02/15.
//  for Tekt Industries Pty. Ltd.
//  Copyright (c) 2015 Winer Bao. All rights reserved.
//

#ifndef USB_driver
#define USB_driver

#include "stdafx.h"

#include <ctime>
#include <chrono>
#include <ftd2xx.h>
#include <stdlib.h>
#include <vector>
#include <string>
#include <iostream>

#define BYTE_WIDTH 8
#define CMD_WIDTH 4             // bits
#define MAX_NUM_CHANNELS 40
#define MAX_NUM_LEDS 300
#define NUM_BYTES_PER_LED 3
#define ONE_SECTOR 1024

// Commands
const unsigned char CMD_LOAD_LINE = 0x0;
const unsigned char CMD_SET_LENGTH = 0x1;

enum exitCodes
{
    BUFFER_PURGE_FAIL,
    CREATE_DEV_INFO_LIST_FAIL,
    INVALID_NUM_CHANNELS,
    NO_DEVICES_FOUND,
    NO_SUPPORTED_DEVICES_FOUND,
    SET_BIT_MODE_FAIL,
    SET_HANDLE_FAIL
};

struct rgbColour {
    unsigned char red;
    unsigned char green;
    unsigned char blue;
};

class Device
{
private:
    FT_DEVICE_LIST_INFO_NODE deviceInfo;
    FT_STATUS ftStatus;
    unsigned int numChannels;

public:
    // constructors
    Device(void);

    // getters
    char* getDeviceDescription(void);
    FT_HANDLE getHandle(void);
    unsigned int getNumChannels(void);

    // setters
    void setDeviceInfo(FT_DEVICE_LIST_INFO_NODE);
    void setFtStatus(FT_STATUS);
    void setHandle(FT_HANDLE);
    void setNumChannels(unsigned int);
};
```

\\psf\Dropbox\Elektrotechniek 4de\Afstuderen ... System\Networked LED Driver System\USBdriver.h 2

```
void scanForUSBdevices();
void setUSBChannelLength(unsigned int chNum, unsigned int len);
unsigned int cmd_set_length(unsigned int ch_num, unsigned int len);
std::vector<unsigned char> cmd_load_data(unsigned int ch_num, unsigned int len, const std::vector
    <unsigned char>& RGBdata);
void sendUSB_RGBStripData(std::vector<unsigned char>& RGBdata);
int openDevices();
void purgeRxTxBuffers(FT_HANDLE ftHandle);
void setBitMode(FT_HANDLE ftHandle, UCHAR ucMask, UCHAR ucMode);
Device* get_deviceList();
DWORD get_numDevs();

#endif /* defined(USB_driver) */
```

\\psf\Dropbox\Elektrotechniek 4de\Afstuderen ... System\Networked LED Driver System\USBdriver.cpp 1

```
//
//  USBdriver.h
//  Networked LED Driver System server
//
//  Created by Kia Riegel and Winer Bao on 14/02/15.
//  for Tekt Industries Pty. Ltd.
//  Copyright (c) 2015 Winer Bao. All rights reserved.
//

#include "USBdriver.h"

FT_HANDLE ftHandle;
FT_STATUS ftStatus;
DWORD EventDWord;
DWORD RxBytes;
DWORD TxBytes;
DWORD BytesWritten;

FT_DEVICE_LIST_INFO_NODE *devInfo;

UCHAR CLKOUT_DIR_MASK = 0xff;
const UCHAR SYNCH_FIFO_245_MODE = 0x40;

Device* deviceList;
DWORD numDevs;

//unsigned char TxBuffer[ONE_SECTOR];
unsigned char TxBuffer[ONE_SECTOR];

Device::Device(void)
{
    numChannels = 0;
}

char* Device::getDeviceDescription(void)
{
    return deviceInfo.Description;
}

FT_HANDLE Device::getHandle(void)
{
    return deviceInfo.ftHandle;
}

unsigned int Device::getNumChannels(void)
{
    return numChannels;
}

void Device::setDeviceInfo(FT_DEVICE_LIST_INFO_NODE devInfo)
{
    deviceInfo = devInfo;
}

void Device::setHandle(FT_HANDLE ftHandleInput)
{
    if (ftHandleInput != NULL)
    {
        deviceInfo.ftHandle = ftHandleInput;
    }
    else
    {
        exit(SET_HANDLE_FAIL);
    }
}

void Device::setNumChannels(unsigned int numChs)
{
    if (numChannels <= MAX_NUM_CHANNELS)
    {
        numChannels = numChs;
    }
```

\\psf\Dropbox\Elektrotechniek 4de\Afstuderen ... System\Networked LED Driver System\USBdriver.cpp 2

```
    else
    {
        exit(INVALID_NUM_CHANNELS);
    }
}

void Device::setFtStatus(FT_STATUS ftStatusInput)
{

    ftStatus = ftStatusInput;
}

void scanForUSBdevices()
{
    using namespace std;

    if (openDevices())
    {
    ftHandle = deviceList[0].getHandle();

    setBitMode(ftHandle, CLKOUT_DIR_MASK, SYNCH_FIFO_245_MODE);
    purgeRxTxBuffers(ftHandle);
    }
}

void setUSBChannelLength(unsigned int chNum, unsigned int len)
{
    using namespace std;
    unsigned int tx_size;

    tx_size = cmd_set_length(chNum, len);
    ftStatus = FT_Write(ftHandle, TxBuffer, tx_size, &BytesWritten);

    if (ftStatus == FT_OK)
    {
        cout << "FT_Write OK. Bytes written: " << BytesWritten << endl;
        // FT_Write OK
    }
    else
    {
        cout << "FT_Write Failed." << endl;
        // FT_Write Failed
    }
}

std::vector<unsigned char> cmd_load_data(unsigned int ch_num, unsigned int len, const std::vector ↙
    <unsigned char>& RGBdata)
{
    using namespace std;
    std::vector<unsigned char> message;

    unsigned int len_temp = len*NUM_BYTES_PER_LED;
    message.push_back(len_temp & 0xFF);
    message.push_back((len_temp >> BYTE_WIDTH) & 0xFF);
    message.push_back((len_temp >> 2 * BYTE_WIDTH) & 0xFF);
    message.push_back((len_temp >> 3 * BYTE_WIDTH) & 0xFF);

    unsigned char ch_num_temp = (ch_num << CMD_WIDTH) & 0xF0;
    message.push_back((ch_num_temp | (CMD_LOAD_LINE & 0x0F)));
    ch_num_temp = (ch_num >> CMD_WIDTH) & 0xFF;
    message.push_back(ch_num_temp);

    unsigned int i;
    int c = 0;
    for (i = 0; i < ((len*NUM_BYTES_PER_LED)); i = i + 3)
    {
        message.push_back(RGBdata[c+1]); //green
        message.push_back(RGBdata[c]); //red
        message.push_back(RGBdata[c + 2]); //blue
        c += 3;
    }
```

```
    return message;
}

unsigned int cmd_set_length(unsigned int ch_num, unsigned int len)
{
    TxBuffer[0] = 0x2;
    TxBuffer[1] = 0x0;
    TxBuffer[2] = 0x0;
    TxBuffer[3] = 0x0;

    unsigned char ch_num_temp = (ch_num << CMD_WIDTH) & 0xF0;
    TxBuffer[4] = (ch_num_temp | (CMD_SET_LENGTH & 0x0F));
    using namespace std;
    ch_num_temp = (ch_num >> CMD_WIDTH) & 0xFF;
    TxBuffer[5] = ch_num_temp;
    unsigned char len_temp = len & 0xFF;
    //  cout << len_temp << endl;
    TxBuffer[6] = len_temp;
    len_temp = (len >> BYTE_WIDTH) & 0xFF;
    //  cout << len_temp << endl;
    TxBuffer[7] = len_temp;

    return 8;
}

void sendUSB_RGBStripData(std::vector<unsigned char>& message)
{
    using namespace std;
    unsigned char* tx_message = &message[0];

    if (message.size() <= ONE_SECTOR)
    {
        ftStatus = FT_Write(ftHandle, tx_message, message.size(), &BytesWritten);

        if (ftStatus == FT_OK)
        {
            //cout << "FT_Write OK. Bytes written: " << BytesWritten << endl;
            // FT_Write OK
        }
        else
        {
            cout << "FT_Write Failed." << endl;
            // FT_Write Failed
        }
    }
    else
    {
        cout << "USB buffer overflow" << endl;
    }
}

int openDevices()
{
    using namespace std;
    // create the device information list
    ftStatus = FT_CreateDeviceInfoList(&numDevs);
    if (ftStatus == FT_OK)
    {
        printf("Number of devices is %d\n\n", numDevs);
    }
    else
    {
        cout << "Unable to create device info list." << endl;
        exit(CREATE_DEV_INFO_LIST_FAIL);
    }

    DWORD numSupportedDevs = 0;
    if (numDevs > 0)
    {
        // allocate storage for list based on numDevs
        devInfo = new FT_DEVICE_LIST_INFO_NODE[(sizeof(FT_DEVICE_LIST_INFO_NODE)*numDevs)];
        // get the device information list
```

\\psf\Dropbox\Elektrotechniek 4de\Afstuderen ... System\Networked LED Driver System\USBdriver.cpp 4

```
        ftStatus = FT_GetDeviceInfoList(devInfo, &numDevs);
        if (ftStatus == FT_OK)
        {
            for (unsigned int i = 0; i < numDevs; i++)
            {
                printf("Dev %d:\n", i);
                printf(" Flags=0x%x\n", devInfo[i].Flags);
                printf(" Type=0x%x\n", devInfo[i].Type);
                printf(" ID=0x%x\n", devInfo[i].ID);
                printf(" LocId=0x%x\n", devInfo[i].LocId);
                printf(" SerialNumber=%s\n", devInfo[i].SerialNumber);
                printf(" Description=%s\n", devInfo[i].Description);
                printf(" ftHandle=0x%x\n\n", devInfo[i].ftHandle);

                if (strcmp(devInfo[i].Description, "UM232H") == 0)
                {
                    // Increment number of compatible devices
                    numSupportedDevs++;
                }
                else if (strcmp(devInfo[i].Description, "Morph-IC-II A") == 0)
                {
                    // Increment number of compatible devices
                    numSupportedDevs++;
                }
                else if (strcmp(devInfo[i].Description, "USB <-> Serial Converter A") == 0)
                {
                    // Increment number of compatible devices
                    numSupportedDevs++;
                }
                else
                {
                    // Unsupported device
                }
            }

            if (numSupportedDevs == 0)
            {
                cout << "No supported devices found." << endl;
                exit(NO_SUPPORTED_DEVICES_FOUND);
            }

            deviceList = new Device[(sizeof(Device)*numSupportedDevs)];

            unsigned int j = 0;
            FT_HANDLE ftHandle;
            for (unsigned int i = 0; i < numDevs; i++)
            {
                if (strcmp(devInfo[i].Description, "UM232H") == 0)
                {
                    ftStatus = FT_OpenEx("UM232H", FT_OPEN_BY_DESCRIPTION, &ftHandle);
                    deviceList[j].setDeviceInfo(devInfo[i]);
                    deviceList[j].setHandle(ftHandle);
                    j++;
                }
                else if (strcmp(devInfo[i].Description, "Morph-IC-II A") == 0)
                {
                    ftStatus = FT_OpenEx("Morph-IC-II A", FT_OPEN_BY_DESCRIPTION, &ftHandle);
                    deviceList[j].setDeviceInfo(devInfo[i]);
                    deviceList[j].setHandle(ftHandle);
                    j++;
                }
                else if (strcmp(devInfo[i].Description, "USB <-> Serial Converter A") == 0)
                {
                    ftStatus = FT_OpenEx("USB <-> Serial Converter A", FT_OPEN_BY_DESCRIPTION, &        ↙
    ftHandle);
                    deviceList[j].setDeviceInfo(devInfo[i]);
                    deviceList[j].setHandle(ftHandle);
                    j++;
                }
            }

            cout << "\nNumber of supported devices opened: " << j << endl;
```

```
            delete[] devInfo;
            return 1;
        }
    }
    else
    {
        cout << "No devices found." << endl;
        return 0;
    }
    return 1;
}


void purgeRxTxBuffers(FT_HANDLE ftHandle)
{
    using namespace std;
    ftStatus = FT_Purge(ftHandle, FT_PURGE_RX | FT_PURGE_TX); // Purge both Rx and Tx buffers
    if (ftStatus == FT_OK)
    {
        // FT_Purge OK
    }
    else
    {
        // FT_Purge failed
        cout << "Purge failed." << endl;
        exit(BUFFER_PURGE_FAIL);
    }
}

void setBitMode(FT_HANDLE ftHandle, UCHAR ucMask, UCHAR ucMode)
{
    using namespace std;
    ftStatus = FT_SetBitMode(ftHandle, ucMask, ucMode);
    if (ftStatus == FT_OK)
    {
        // 0xff written to device
    }
    else
    {
        cout << "FT_SetBitMode failed." << endl;
        exit(SET_BIT_MODE_FAIL);
    }
}

Device* get_deviceList()
{
    return deviceList;
}

DWORD get_numDevs()
{
    return numDevs;
}
```

## Appendix 7: main.cpp source code

\\psf\Dropbox\Elektrotechniek 4de\Afstuderen ...Driver System\Networked LED Driver System\main.cpp 1

```
//
//  main.cpp
//  Networked LED Driver System server
//
//  Created by Winer Bao on 17/02/15.
//  for Tekt Industries Pty. Ltd.
//  Copyright (c) 2015 Winer Bao. All rights reserved.
//

#include <stdio.h>
#include <thread>
#include "Definitions.h"
#include "MessageHandler.h"
#include "Database.h"
#include "ChannelObject.h"
#include "USBdriver.h"

int main(int argc, const char*  argv[])
{
    Mapping m;
    Socket s(m);

    scanForUSBdevices();

    // Add drivers
    if (get_numDevs() > 0)
    {
        char* description = get_deviceList()->getDeviceDescription();
        if (strcmp(description, "USB <-> Serial Converter A") == 0)
        {
            for (int channel = 0; channel<40; channel++)
            {
                m.registerDriver(channel, channel, DRIVER_ADDR, DRIVER_PORT, USB);
                m.changeStrip(channel, STRIP_LENGTH, ADDRESSABLE, LINE, RGB, 1);
                //m.changeMatrix(channel, 32, 8, RGB, 1);
            }
        }
    }
    else
    {
        for (int channel = 0; channel<TOTAL_DRIVERS; channel++)
        {
            m.registerDriver(channel, channel, DRIVER_ADDR, DRIVER_PORT, UDP);
            m.changeStrip(channel, STRIP_LENGTH, ADDRESSABLE, LINE, RGB, 1);
            //m.changeMatrix(channel, 32, 8, RGB, 1);
        }
    }

    std::string input = "";
    while (input != "shutdown")
    {
        getline(std::cin, input);
    }

    std::cout << "server shutdown" << std::endl;
    return 0;
}

// Converts 2 bytes into an integer.
// Useful for recomposition of an integer sent through UDP.
int ByteToInt(unsigned char MSB, unsigned char LSB)
{
    return ((MSB & 0x00FF) << 8) | (LSB & 0x00FF);
}
```

## Appendix 8: Sender_Knob source code

```
/*
previously called UDP_sender_and_knobs_v1_8a

V1.8
Program can connect and disconnect to the server by pressing 's' and 'q' respectively
To reconfigure the channel, press 'r'.
Only sends data when the color is changed

V1.7
Program split into two senders

V1.6
New data format.

V1.5
Class Strip added. Multiple LED strip channels.
Channels are added automatically. User can add differt length of LED strips.

V1.4
Multiple LEDs (strip)

V1.3
Bug "128-159 value = 63" fixed. Send byte[] instead of string

V1.2
RGB data is sent by UDP continuously

V1.1
Added the UPD protocol to send 8 bit data for each Red/Green/Blue channels.
Buttons are made with the ControlP5 library.

V1.0
Created 3 knob to set the value of each Red/Green/Blue channel.
Knobs are custom mdae
*/

import controlP5.*;
import hypermedia.net.*;
import java.util.Map;

ControlP5 cp5;
UDP udp;
HashMap<Integer, Strip> stripChannels;

/* Definitions to increase readability */
final byte ALL           = 0x00;
final byte SINGLE        = 0x01;
final byte MULTIPLE      = 0x02;
final byte SENDER_CONNECT    = 0x00;
final byte SENDER_RECONFIG   = 0x01;
final byte SENDER_DISCONNECT = 0x02;
final byte COLOR_RGB         = 0x07;
final byte CLASS_STRIP       = 0x00;
```

```
final byte TYPE_ADDRESSABLE  = 0x02;
final byte CONFIG_LINE       = 0x04;
final byte CONFIG_OTHER      = 0x05;

final String SERVERADDR  = "127.0.0.1";
final int SERVERPORT     = 21234;

Knob myKnobR;
Knob myKnobG;
Knob myKnobB;

int myColorBackground   = color(225);
int Channels;
int channelSelected;
int connected;

void setup()
{
  frameRate(30);
  udp = new UDP(this, 6500);
  udp.listen(true);

  size(1000,500);
  smooth();
  noStroke();
  stripChannels = new HashMap<Integer, Strip>();
  addStrip(0, 1, 3);
  addStrip(1, 5, 3);
  addStrip(2, 8, 3);
  addStrip(3, 10, 3);

  cp5 = new ControlP5(this);

  myKnobR = cp5.addKnob("Red")
              .setRange(0,255)
              .setValue(0)
              .setPosition(50,70)
              .setRadius(50)
              .setDragDirection(Knob.VERTICAL)
              .setNumberOfTickMarks(10)
              .setTickMarkLength(4)
              .snapToTickMarks(false)
              .setColorForeground(color(155, 0, 0))
              .setColorBackground(color(255, 0, 0))
              .setColorActive(color(255,255,0))
              .setDecimalPrecision(0)
              ;

 myKnobG = cp5.addKnob("Green")
              .setRange(0,255)
              .setValue(0)
              .setPosition(50,200)
              .setRadius(50)
              .setDragDirection(Knob.VERTICAL)
              .setNumberOfTickMarks(10)
              .setTickMarkLength(4)
```

```
            .snapToTickMarks(false)
            .setColorForeground(color(0, 100, 50))
            .setColorBackground(color(0, 200, 50))
            .setColorActive(color(255,255,0))
            .setDecimalPrecision(0)
            ;

  myKnobB = cp5.addKnob("Blue")
              .setRange(0,255)
              .setValue(0)
              .setPosition(50,330)
              .setRadius(50)
              .setDragDirection(Knob.VERTICAL)
              .setNumberOfTickMarks(10)
              .setTickMarkLength(4)
              .snapToTickMarks(false)
              .setColorForeground(color(0, 0, 155))
              .setColorBackground(color(0, 0, 255))
              .setColorActive(color(255,255,0))
              .setDecimalPrecision(0)
              ;
}

void draw()
{
  background(myColorBackground);
  fill(0,100);
  rect(30,40,140,425);

    if(connected == 1)
  {
    fill(0, 235, 0);
    ellipse(970, 30, 50, 50);
  }
  else
  {
    fill(0, 0, 0);
    ellipse(970, 30, 50, 50);
  }

  for (Map.Entry map: stripChannels.entrySet())
  {
    Object temp = map.getValue();
    ((Strip)temp).displayStrip();
  }
}

void controlEvent(ControlEvent theEvent)
{
   if(theEvent.isController())
   {
      if(theEvent.controller().name()  ==  "Red"  ||  theEvent.controller().name()==  "Green"  ||
theEvent.controller().name() == "Blue")
      {
        if(connected == 1)
        {
```

```
          stripChannels.get(channelSelected).DataTX();
        }
      }
    }
}

void mousePressed()
{
  for (Map.Entry map: stripChannels.entrySet())
  {
    Object temp = map.getValue();
    for( int i = 0; i < ((Strip)temp).stripLength; i++)
    {
      if (overCircle(((Strip)temp).virtualLED[i][0], ((Strip)temp).virtualLED[i][1], 50))
      {
        ((Strip)temp).selectedLED = i;
        channelSelected = (Integer)map.getKey();
        myKnobR.setValue(int(((Strip)temp).strip[i][0]));
        myKnobG.setValue(int(((Strip)temp).strip[i][1]));
        myKnobB.setValue(int(((Strip)temp).strip[i][2]));
      }
    }
  }
}

void keyPressed()
{
  if(key == 'q')
  {
   for (Map.Entry map: stripChannels.entrySet())
    {
      Object channel = map.getKey();
      DisconnectServer((Integer)channel);
    }
  }
  else if(key == 's')
  {
    println("key pressed");
    for (Map.Entry map: stripChannels.entrySet())
    {
      Object channel = map.getKey();
      Object strip = map.getValue();
      connectToServer((Integer)channel, ((Strip)strip).stripLength);
    }
  }
  else if(key == 'r')
  {
    for (Map.Entry map: stripChannels.entrySet())
    {
      Object channel = map.getKey();
      Object strip = map.getValue();
      ChannelReconfig((Integer)channel, ((Strip)strip).stripLength);
    }
  }
  else if(key == 't')
  {
```

```
    for (Map.Entry map: stripChannels.entrySet())
    {
      Object strip = map.getValue();
      ((Strip)strip).DataTX();
    }
  }
}

boolean overCircle(int x, int y, int diameter)
{
  float disX = x - mouseX;
  float disY = y - mouseY;
  if(sqrt(sq(disX) + sq(disY)) < diameter/2 )
  {
    return true;
  } else
  {
    return false;
  }
}

/*
  virtualLED[][0] = X coordinates of the LED
  virtualLED[][1] = Y coordinates of the LED
  strip[][0] = Red channel data
  strip[][1] = Green channel data
  strip[][2] = Blue channel data
*/

void receive(byte[] data)
{
  String temp = new String(data);
  println(temp);
  if(match(temp, "Connect successful") != null)
  {
    connected = 1;
  }
  else if(match(temp, "Disconnect successful") != null)
  {
    connected = 0;
  }
   else if(match(temp, "You're already connected to this channel") != null)
  {
    connected = 1;
  }
}

void addStrip(int channel, int Length, int colorChannel)
{
  stripChannels.put(channel, new Strip(Length, colorChannel, channel));
  Channels++;
}

void connectToServer(int channel, int Length)
{
  byte[] msgUDP = new byte[16];
```

```
  msgUDP[0] = '*';
  msgUDP[1] = SENDER_CONNECT;
  msgUDP[2] = '/';
  msgUDP[3] = byte(channel>>8);
  msgUDP[4] = byte(channel);
  msgUDP[5] = '/';
  msgUDP[6] = COLOR_RGB;
  msgUDP[7] = '/';
  msgUDP[8] = CLASS_STRIP;
  msgUDP[9] = '/';
  msgUDP[10] = byte(Length>>8);
  msgUDP[11] = byte(Length);
  msgUDP[12] = '/';
  msgUDP[13] = TYPE_ADDRESSABLE;
  msgUDP[14] = '/';
  msgUDP[15] = CONFIG_LINE;

  String ip       = SERVERADDR;
  int port        = SERVERPORT;

  udp.send(msgUDP, ip, port);
  msgUDP = null;
}

void ChannelReconfig(int channel, int Length)
{
    byte[] msgUDP = new byte[12];
  msgUDP[0] = '*';
  msgUDP[1] = SENDER_RECONFIG;
  msgUDP[2] = '/';
  msgUDP[3] = byte(channel>>8);
  msgUDP[4] = byte(channel);
  msgUDP[5] = '/';
  msgUDP[6] = byte(Length>>8);
  msgUDP[7] = byte(Length);
  msgUDP[8] = '/';
  msgUDP[9] = 0x00;
  msgUDP[10] = '/';
  msgUDP[11] = 0x01;

  String ip       = SERVERADDR;
  int port        = SERVERPORT;

  udp.send(msgUDP, ip, port);
  msgUDP = null;
}

void DisconnectServer(int channel)
{
  byte[] msgUDP = new byte[12];
  msgUDP[0] = '*';
  msgUDP[1] = SENDER_DISCONNECT;
  msgUDP[2] = '/';
  msgUDP[3] = byte(channel>>8);
  msgUDP[4] = byte(channel);
```

```
  String ip       = SERVERADDR;
  int port        = SERVERPORT;

  udp.send(msgUDP, ip, port);
  msgUDP = null;
}

class Strip
{
  int selectedLED;
  int stripLength;
  int colorChannels;
  int channel;
  final int LEDparam = 2;
  byte[][] strip;
  int[][] virtualLED;

  Strip(int Length, int colorChannel, int displayChannel)
  {
    int posX = 200;
    int posY = 75;
    stripLength = Length;
    colorChannels = colorChannel;
    channel      = displayChannel;
    strip        = new byte[stripLength][colorChannels];
    virtualLED    = new int[stripLength][LEDparam];
    for( int i = 0; i < stripLength; i++)
    {
      virtualLED[i][0] = posX;
      posX += 55;
      virtualLED[i][1] = posY + (100*channel);
    }
  }

  void changePixelColor(int pixel, int red, int green, int blue)
  {
    strip[pixel][0] = byte(red);
    strip[pixel][1] = byte(green);
    strip[pixel][2] = byte(blue);
  }

  void displayStrip()
  {
    if(channelSelected == channel)
    {
      fill(255, 0, 0);
      ellipse(virtualLED[selectedLED][0], virtualLED[selectedLED][1], 55, 55);
    }

    for( int i = 0; i < stripLength; i++)
    {
      if(channelSelected == channel)
      {
        strip[selectedLED][0]= byte(myKnobR.getValue());
        strip[selectedLED][1]= byte(myKnobG.getValue());
        strip[selectedLED][2]= byte(myKnobB.getValue());
```

```
    }
    fill(int(strip[i][0]), int(strip[i][1]), int(strip[i][2]));
    ellipse(virtualLED[i][0], virtualLED[i][1], 50, 50);
  }
 }

 void DataTX()
 {
  byte[] msgUDP = new byte[stripLength*colorChannels + 6];
  int temp = 6;
  msgUDP[0] = '>';
  msgUDP[1] = byte(channel>>8);
  msgUDP[2] = byte(channel);
  msgUDP[3] = '/';
  msgUDP[4] = ALL;
  msgUDP[5] = '/';
  for(int i = 0; i < stripLength ; i++)
  {
   for(int j = 0; j < colorChannels; j++)
   {
    msgUDP[temp++] = strip[i][j];
   }
  }

  String ip       = SERVERADDR;
  int port        = SERVERPORT;

  udp.send(msgUDP, ip, port);
  msgUDP = null;
 }
}
```

## Appendix 9: Sender_Webcam source code

```
/*
previously called UDP_sender_video_v1_1

V1.1
send webcam pixels to the server via IP.
Press 's' to connect to the server and press 'q' to disconnect.
The program will only send data when it is connected to the server.

V1.0
Example program that takes the webcam data and pixalates and displays the data on screen.
*/

/* Import libraries */
import processing.video.*;
import hypermedia.net.*;
import java.util.Map;

/* Global Variables */
// UDP object
UDP udp;
// Map to link the channel with a Strip object
HashMap<Integer, Strip> stripChannels;
// Background color of the program
int myColorBackground   = color(225);
// Stores the amount of channels connected
int Channels;
// The current channel being processed
int channelSelected;
// Tells if the program is connected to the server
boolean connected;
//webcam resolution
int camwidth = 1280;
int camheight  = 720;
// Size of each cell in the grid, ratio of window size to video size
int videoScale = 32;
// Number of columns and rows in our system
int cols, rows;
// Variable to hold onto Capture object
Capture video;

/* Definitions to increase readability */
final byte ALL          = 0x00;
final byte SINGLE       = 0x01;
final byte MULTIPLE     = 0x02;
final byte SENDER_CONNECT    = 0x00;
final byte SENDER_RECONFIG   = 0x01;
final byte SENDER_DISCONNECT = 0x02;
final byte COLOR_RGB        = 0x07;
final byte CLASS_STRIP      = 0x00;
final byte TYPE_ADDRESSABLE  = 0x02;
final byte CONFIG_LINE      = 0x04;
final byte CONFIG_OTHER     = 0x05;
```

```
/* Server IP and port number */
final String SERVERADDR  = "192.168.1.27";
final int SERVERPORT     = 21234;


void setup() {
  // Set framerate to 30
  frameRate(30);

  // Open a UDP socket at computer's IP and port number 6500
  // This socket can also receive messages
  udp = new UDP(this, 6500);
  udp.listen(true);

  // Create a map and add 45 Strip objects with a length of 80 each
  stripChannels = new HashMap<Integer, Strip>();
  for (int i=0; i < 23; i++)
  {
    addStrip(i, 40, 3);
  }

  // Define the window size
  size(camwidth, camheight);

  // Initialize columns and rows
  cols = width / videoScale;
  rows = height / videoScale;
  video = new Capture(this, camwidth, camheight);
  video.start();
}

void captureEvent(Capture video) {
  // Read image from the camera
  video.read();
}

void draw() {
  // Load the video data
  video.loadPixels();

  // Begin loop for columns
  for (int i = 0; i < rows; i++) {
    // Begin loop for rows
    for (int j = 0; j < cols; j++) {

      // Where are we, pixel-wise?
      int x = j * videoScale;
      int y = i * videoScale;
      // Looking up the appropriate color in the pixel array
      color c = video.pixels[(j*videoScale) + (i* videoScale * video.width)];
      fill(c);
      stroke(0);
      rect(x, y, videoScale, videoScale);

      // Save the pixel color.
      stripChannels.get(i).changePixelColor(j, c>>16&0xFF, c>>8&0xFF, c&0xFF);
```

```
    }
  }

  // If the program is connected to the server, send pixel data
  if (connected == true)
  {
    for (Map.Entry map: stripChannels.entrySet())
    {
      Object strip = map.getValue();
      ((Strip)strip).DataTX();
    }
  }
}

// The Strip class stores information and has useful functions
// strip[][] stores the color values
class Strip
{
  int selectedLED;
  int stripLength;
  int colorChannels;
  int channel;
  final int LEDparam = 2;
  byte[][] strip;

  Strip(int Length, int colorChannel, int displayChannel)
  {
    int posX = 200;
    int posY = 75;
    stripLength = Length;
    colorChannels = colorChannel;
    channel      = displayChannel;
    strip        = new byte[stripLength][colorChannels];
  }

  void changePixelColor(int pixel, int red, int green, int blue)
  {
    strip[pixel][0] = byte(red);
    strip[pixel][1] = byte(green);
    strip[pixel][2] = byte(blue);
  }

  void DataTX()
  {
    byte[] msgUDP = new byte[stripLength*colorChannels + 6];
    int temp = 6;
    msgUDP[0] = '>';
    msgUDP[1] = byte(channel>>8);
    msgUDP[2] = byte(channel);
    msgUDP[3] = '/';
    msgUDP[4] = ALL;
    msgUDP[5] = '/';
    for(int i = 0; i < stripLength ; i++)
    {
      for(int j = 0; j < colorChannels; j++)
      {
```

```
      msgUDP[temp++] = strip[i][j];
    }
   }

   String ip       = SERVERADDR;
   int port        = SERVERPORT;

   udp.send(msgUDP, ip, port);
   msgUDP = null;
  }
}

void keyPressed()
{
  // Press the q key to disconnect from the server
  if(key == 'q')
  {
   for (Map.Entry map: stripChannels.entrySet())
   {
     Object channel = map.getKey();
     DisconnectServer((Integer)channel);
     connected = false;
     noLoop();
   }
  }
  //Press the s key to connect to the server
  else if(key == 's')
  {
   for (Map.Entry map: stripChannels.entrySet())
   {
     Object channel = map.getKey();
     Object strip = map.getValue();
     connectToServer((Integer)channel, ((Strip)strip).stripLength);
     //((Strip)strip).DataTX();
     loop();
   }
  }
  //Press the r key to reconfigure your strip
  else if(key == 'r')
  {
   for (Map.Entry map: stripChannels.entrySet())
   {
     Object channel = map.getKey();
     Object strip = map.getValue();
     ChannelReconfig((Integer)channel, ((Strip)strip).stripLength);
   }
  }
  // Press the t key to manually transmit the data
  else if(key == 't')
  {
   for (Map.Entry map: stripChannels.entrySet())
   {
     Object strip = map.getValue();
     ((Strip)strip).DataTX();
   }
  }
```

```
}

void receive(byte[] data)
{
  String temp = new String(data);

  if(match(temp, "Connect successful") != null)
  {
    connected = true;
  }
  else if(match(temp, "Disconnect successful") != null)
  {
    connected = false;
  }
   else if(match(temp, "You're already connected to this channel") != null)
  {
    connected = true;
  }
}

void addStrip(int channel, int Length, int colorChannel)
{
  stripChannels.put(channel, new Strip(Length, colorChannel, channel));
  Channels++;
}

void connectToServer(int channel, int Length)
{
  byte[] msgUDP = new byte[16];
  msgUDP[0] = '*';
  msgUDP[1] = SENDER_CONNECT;
  msgUDP[2] = '/';
  msgUDP[3] = byte(channel>>8);
  msgUDP[4] = byte(channel);
  msgUDP[5] = '/';
  msgUDP[6] = COLOR_RGB;
  msgUDP[7] = '/';
  msgUDP[8] = CLASS_STRIP;
  msgUDP[9] = '/';
  msgUDP[10] = byte(Length>>8);
  msgUDP[11] = byte(Length);
  msgUDP[12] = '/';
  msgUDP[13] = TYPE_ADDRESSABLE;
  msgUDP[14] = '/';
  msgUDP[15] = CONFIG_LINE;

  String ip       = SERVERADDR;
  int port        = SERVERPORT;

  udp.send(msgUDP, ip, port);
  msgUDP = null;
}

void ChannelReconfig(int channel, int Length)
{
  byte[] msgUDP = new byte[16];
```

```
  msgUDP[0] = '*';
  msgUDP[1] = SENDER_CONNECT;
  msgUDP[2] = '/';
  msgUDP[3] = byte(channel>>8);
  msgUDP[4] = byte(channel);
  msgUDP[5] = '/';
  msgUDP[6] = COLOR_RGB;
  msgUDP[7] = '/';
  msgUDP[8] = CLASS_STRIP;
  msgUDP[9] = '/';
  msgUDP[10] = byte(Length>>8);
  msgUDP[11] = byte(Length);
  msgUDP[12] = '/';
  msgUDP[13] = TYPE_ADDRESSABLE;
  msgUDP[14] = '/';
  msgUDP[15] = CONFIG_OTHER;

  String ip       = SERVERADDR;
  int port        = SERVERPORT;

  udp.send(msgUDP, ip, port);
  msgUDP = null;
}

void DisconnectServer(int channel)
{
  byte[] msgUDP = new byte[12];
  msgUDP[0] = '*';
  msgUDP[1] = SENDER_DISCONNECT;
  msgUDP[2] = '/';
  msgUDP[3] = byte(channel>>8);
  msgUDP[4] = byte(channel);

  String ip       = SERVERADDR;
  int port        = SERVERPORT;

  udp.send(msgUDP, ip, port);
  msgUDP = null;
}
```

## Appendix 10: Sender_Rainbow source code

```
//previously called UDP_sender_rainbow

/* Import libraries */
import hypermedia.net.*;

/* Global Variables */
// UDP object
UDP udp;
// Map to link the channel with a Strip object
// Background color of the program
int myColorBackground   = color(225);
// Tells if the program is connected to the server
boolean connected;

/* Definitions to increase readability */
final byte ALL              = 0x00;
final byte SINGLE           = 0x01;
final byte MULTIPLE         = 0x02;
final byte SENDER_CONNECT    = 0x00;
final byte SENDER_RECONFIG   = 0x01;
final byte SENDER_DISCONNECT = 0x02;
final byte COLOR_RGB         = 0x07;
final byte CLASS_STRIP       = 0x00;
final byte TYPE_ADDRESSABLE  = 0x02;
final byte CONFIG_LINE       = 0x04;
final byte CONFIG_OTHER      = 0x05;

/* Server IP and port number */
final String SERVERADDR  = "127.0.0.1";
final int SERVERPORT     = 21234;

int shift;
//int ch_number = 7;
int ch_length = 256;

class rgbColour {
  byte red;
  byte green;
  byte blue;
};

void setup() {
  // Set framerate to 60
  frameRate(60);

  // Open a UDP socket at computer's IP and port number 6600
  // This socket can also receive messages
  udp = new UDP(this, 6600);
  udp.listen(true);
}

void draw() {
  // If the program is connected to the server, send pixel data
```

```
  if (connected == true)
  {
    DataTX(23, ch_length);
    DataTX(31, ch_length);
    DataTX(39, ch_length);
  }
}

void DataTX(int channel, int len)
{
  rgbColour pixel = new rgbColour();
  int i;
  byte[] msgUDP = new byte[len*3+6];

  msgUDP[0] = '>';
  msgUDP[1] = byte(channel>>8);
  msgUDP[2] = byte(channel);
  msgUDP[3] = '/';
  msgUDP[4] = ALL;
  msgUDP[5] = '/';

  for( i = 0; i < ch_length; i++)
  {
    pixel = Wheel(byte(i + shift));
    msgUDP[i * 3 + 6] = pixel.green;
    msgUDP[i * 3 + 1 + 6] = pixel.red;
    msgUDP[i * 3 + 2 + 6] = pixel.blue;
  }
  shift++;
  if (shift > 255)
  {
    shift = 0;
  };

  udp.send(msgUDP, SERVERADDR, SERVERPORT);
  msgUDP = null;
}

void keyPressed()
{
  // Press the q key to disconnect from the server
  if(key == 'q')
  {
     DisconnectServer(23);
     DisconnectServer(31);
     DisconnectServer(39);
     connected = false;
     noLoop();
  }
  //Press the s key to connect to the server
  else if(key == 's')
  {
     connectToServer(23, ch_length);
     connectToServer(31, ch_length);
     connectToServer(39, ch_length);
     loop();
```

```
  }
}

void receive(byte[] data)
{
  String temp = new String(data);

  if(match(temp, "Connect successful") != null)
  {
    connected = true;
  }
  else if(match(temp, "Disconnect successful") != null)
  {
    connected = false;
  }
   else if(match(temp, "You're already connected to this channel") != null)
  {
    connected = true;
  }
}

void connectToServer(int channel, int Length)
{
  byte[] msgUDP = new byte[16];
  msgUDP[0] = '*';
  msgUDP[1] = SENDER_CONNECT;
  msgUDP[2] = '/';
  msgUDP[3] = byte(channel>>8);
  msgUDP[4] = byte(channel);
  msgUDP[5] = '/';
  msgUDP[6] = COLOR_RGB;
  msgUDP[7] = '/';
  msgUDP[8] = CLASS_STRIP;
  msgUDP[9] = '/';
  msgUDP[10] = byte(Length>>8);
  msgUDP[11] = byte(Length);
  msgUDP[12] = '/';
  msgUDP[13] = TYPE_ADDRESSABLE;
  msgUDP[14] = '/';
  msgUDP[15] = CONFIG_LINE;

  udp.send(msgUDP, SERVERADDR, SERVERPORT);
  msgUDP = null;
}

void DisconnectServer(int channel)
{
  byte[] msgUDP = new byte[12];
  msgUDP[0] = '*';
  msgUDP[1] = SENDER_DISCONNECT;
  msgUDP[2] = '/';
  msgUDP[3] = byte(channel>>8);
  msgUDP[4] = byte(channel);

  udp.send(msgUDP, SERVERADDR, SERVERPORT);
  msgUDP = null;
```

```
}

// Input a value 0 to 255 to get a color value.
// The colours are a transition r - g - b - back to r.
rgbColour Wheel(byte WheelPos)
{
  rgbColour pixel = new rgbColour();
  WheelPos = byte(255 - WheelPos);
  if (int(WheelPos) < 85)
  {
    pixel.red = byte(255 - WheelPos * 3);
    pixel.green = 0;
    pixel.blue = byte(WheelPos * 3);
    return pixel;
  }
  else if (int(WheelPos) < 170)
  {
    WheelPos -= 85;
    pixel.red = 0;
    pixel.green = byte(WheelPos * 3);
    pixel.blue = byte(255 - WheelPos * 3);
    return pixel;
  }
  else
  {
    WheelPos -= 170;
    pixel.red = byte(WheelPos * 3);
    pixel.green = byte(255 - WheelPos * 3);
    pixel.blue = 0;
    return pixel;
  }
}
```

## Appendix 11: Sender_GIF source code

```
/*
previously called UDP_sender_gif_V1_2
  V1.2 Split image into 3 and send to 3 channels

  V1.1 Loads gifs

  V1.0 Loads pictures instead of gifs
*/

import gifAnimation.*;
import hypermedia.net.*;
import java.util.Map;

final byte ALL              = 0x00;
final byte SINGLE           = 0x01;
final byte MULTIPLE         = 0x02;
final byte SENDER_CONNECT   = 0x00;
final byte SENDER_RECONFIG  = 0x01;
final byte SENDER_DISCONNECT = 0x02;
final byte COLOR_RGB        = 0x07;
final byte CLASS_STRIP      = 0x00;
final byte CLASS_MATRIX     = 0x01;
final byte TYPE_ADDRESSABLE  = 0x02;
final byte CONFIG_LINE      = 0x04;
final byte CONFIG_OTHER     = 0x05;

final String SERVERADDR     = "127.0.0.1";
final int SERVERPORT        = 21234;
final int gif_Xoffset       = -175;
final int gif_Yoffset       = -110;
final int num_panels        = 3;
final int panel_width       = 32;
final int panel_height      = 8;

UDP udp;
HashMap<Integer, Matrix> Channels;
Gif myAnimation;

float xpos;
float ypos;
boolean connected;
int pixel_diameter;

void setup() {
  size(200, 150);
  float aspect_ratio = ((float)panel_height/(float)panel_width);
  float window_height = ((float)width * aspect_ratio)*3;
  size(width, (int)window_height);
  background(255, 204, 0);
  frameRate(30);
  udp = new UDP(this, 6500);
  udp.listen(true);
  myAnimation = new Gif(this, "nyancat.gif");
```

```
  Channels = new HashMap<Integer, Matrix>();
  pixel_diameter = width/panel_width;
  //for (int i=0; i < num_panels; i++)
  //{
  //  Channels.put(i, new Matrix(panel_width, panel_height, i));
  //}
  Channels.put(0, new Matrix(panel_width, panel_height, 23));
  Channels.put(1, new Matrix(panel_width, panel_height, 31));
  Channels.put(2, new Matrix(panel_width, panel_height, 39));
  println("Panels: ", num_panels, " Length: ", panel_width*panel_height, " Pixel diameter: ",
pixel_diameter);
  noLoop();
}

void draw() {
  if(connected)
  {
    image(myAnimation,gif_Xoffset, gif_Yoffset);
    loadPixels();
    int panel_Xoffset = pixel_diameter/2;
    int panel_Yoffset = 0 ;
    for (int i=0; i < num_panels; i++)
    {
      Channels.get(i).update(panel_Xoffset, panel_Yoffset);
      panel_Yoffset += panel_height*pixel_diameter;
    }
    for (int i=0; i < num_panels; i++)
    {
      Channels.get(i).DataTX();
    }
  }
}

class Matrix
{
  int m_width;
  int m_height;
  int channel;
  byte[][] buf;

  Matrix(int panel_width, int panel_height, int displayChannel)
  {
    int posX = 200;
    int posY = 75;
    m_width = panel_width;
    m_height = panel_height;
    channel  = displayChannel;
    buf      = new byte[m_width*m_height][3];
  }

  void changePixelColor(int pixel, float red, float green, float blue)
  {
    buf[pixel][0] = byte(red);
    buf[pixel][1] = byte(green);
    buf[pixel][2] = byte(blue);
  }
```

```
 void update(int start_xpoint, int start_ypoint)
 {
   int pixel_number;
   color c;
   int max_height_loop;
   pixel_number = 0;
   if(((pixel_diameter*m_height) + start_ypoint) < height)
   {
     max_height_loop = ((pixel_diameter*m_height) + start_ypoint);
   }
   else
   {
     max_height_loop = height;
   }

   for( int x = start_xpoint; x < (pixel_diameter*m_width) + start_xpoint; x += (pixel_diameter*2))
   {
     for( int y = start_ypoint; y < max_height_loop; y += pixel_diameter)
     {
       c = pixels[x+(y*width)];
       changePixelColor(pixel_number, red(c), green(c), blue(c));
       pixel_number++;
     }
     pixel_number += m_height;
   }

   pixel_number = (m_height*2)-1;
   for( int x = start_xpoint + pixel_diameter; x < (pixel_diameter*m_width) + start_xpoint +
pixel_diameter; x += (pixel_diameter*2))
   {
     for( int y = start_ypoint; y < max_height_loop; y += pixel_diameter)
     {
       c = pixels[x+(y*width)];
       changePixelColor(pixel_number, red(c), green(c), blue(c));
       pixel_number--;
     }
     pixel_number+= (m_height*3);
   }
 }

 void DataTX()
 {
   byte[] msgUDP = new byte[(m_width*m_height)*3 + 6];
   int temp = 6;
   msgUDP[0] = '>';
   msgUDP[1] = byte(channel>>8);
   msgUDP[2] = byte(channel);
   msgUDP[3] = '/';
   msgUDP[4] = ALL;
   msgUDP[5] = '/';
   for(int i = 0; i < (m_width*m_height) ; i++)
   {
     for(int j = 0; j < 3; j++)
     {
       msgUDP[temp++] = buf[i][j];
```

```
    }
   }

   String ip       = SERVERADDR;
   int port        = SERVERPORT;

   udp.send(msgUDP, ip, port);
   msgUDP = null;
  }
}

void keyPressed()
{
  // Press the q key to disconnect from the server
  if(key == 'q')
  {
   for (Map.Entry map: Channels.entrySet())
    {
     Object panel = map.getValue();
     DisconnectServer(((Matrix)panel).channel);
     connected = false;
     noLoop();
    }
  }
  //Press the s key to connect to the server
  else if(key == 's')
  {
   for (Map.Entry map: Channels.entrySet())
   {
     Object panel = map.getValue();
     connectToServer(((Matrix)panel).channel, ((Matrix)panel).m_width, ((Matrix)panel).m_height);
     loop();
   }
  }
}

void receive(byte[] data)
{
  String temp = new String(data);

  if(match(temp, "Connect successful") != null)
  {
   connected = true;
   myAnimation.play();
  }
  else if(match(temp, "Disconnect successful") != null)
  {
   connected = false;
   myAnimation.stop();
  }
   else if(match(temp, "You're already connected to this channel") != null)
  {
   connected = true;
   myAnimation.play();
  }
}
```

```
void connectToServer(int channel, int m_width, int m_height)
{
  byte[] msgUDP = new byte[15];
  msgUDP[0] = '*';
  msgUDP[1] = SENDER_CONNECT;
  msgUDP[2] = '/';
  msgUDP[3] = byte(channel>>8);
  msgUDP[4] = byte(channel);
  msgUDP[5] = '/';
  msgUDP[6] = COLOR_RGB;
  msgUDP[7] = '/';
  msgUDP[8] = CLASS_MATRIX;
  msgUDP[9] = '/';
  msgUDP[10] = byte(m_width>>8);
  msgUDP[11] = byte(m_width);
  msgUDP[12] = '/';
  msgUDP[13] = byte(m_height>>8);
  msgUDP[14] = byte(m_height);

  String ip       = SERVERADDR;
  int port        = SERVERPORT;

  udp.send(msgUDP, ip, port);
  msgUDP = null;
}

void DisconnectServer(int channel)
{
  byte[] msgUDP = new byte[12];
  msgUDP[0] = '*';
  msgUDP[1] = SENDER_DISCONNECT;
  msgUDP[2] = '/';
  msgUDP[3] = byte(channel>>8);
  msgUDP[4] = byte(channel);

  String ip       = SERVERADDR;
  int port        = SERVERPORT;

  udp.send(msgUDP, ip, port);
  msgUDP = null;
}
```

## Appendix 12: Driver_Receive source code

```
/*
previously called UDP_receiver_v1_5

V1.5
Pre-defined strip lengths. Works with communication protocol.

V1.4
Higher DPI. Send data via IP instead of localhost

V1.3
Receive data from 127.0.0.1:6100

V1.2
Display multiple channel/ LED strips.

V1.1
Receive data and display on (virtual) LED strip

V1.0
Converts bytes for the Red/Green/Blue channels that is received with the UDP protocol
*/

import controlP5.*;
import hypermedia.net.*;
import java.util.Map;
import java.util.HashMap;
import java.util.Iterator;

ControlP5 cp5;
UDP udp;
HashMap<Integer, Strip> stripChannels;

final byte SET_LENGTH       = 0x04;
final String RECEIVER_ADDR  = "192.168.1.27";
final int RECEIVER_PORT     = 6100;

int myColorBackground = color(0);
int messageR = 0;
int messageG = 0;
int messageB = 0;

int Channels;

void setup()
{
  frameRate(30);
  udp = new UDP(this, RECEIVER_PORT, RECEIVER_ADDR);
  udp.listen(true);

  size(600, 400);
  smooth();
  noStroke();
```

```
  stripChannels = new HashMap<Integer, Strip>();
  cp5 = new ControlP5(this);
}

void receive( byte[] data)
{
  synchronized(this)
  {
    int temp = 6;
    int msgChannel = ((data[1]&0x00FF)<<8)|(data[2]&0x00FF);

    if(data[0] == '#' && data[4] == SET_LENGTH)
    {
      int msgStripLength = ((data[6]&0x00FF)<<8) | (data[7]&0x00FF);
      if(!stripChannels.containsKey(msgChannel))
      {
        addStrip(msgChannel, msgStripLength, 3);
      }
      else
      {
        stripChannels.get(msgChannel).changeLength(msgStripLength);
      }
    }
    else if(data[0] == '/')
    {
      for(int i = 0; i < stripChannels.get(msgChannel).stripLength ; i++)
      {
        for(int j = 0; j < stripChannels.get(msgChannel).colorChannels; j++)
        {
          stripChannels.get(msgChannel).strip[i][j] = data[temp++];
        }
      }
    }
  }
}

void draw()
{
  background(myColorBackground);
  synchronized(this)
  {
    for(Map.Entry map : stripChannels.entrySet())
    {
      Object temp = map.getValue();
      ((Strip)temp).displayStrip();
    }
  }
}

void addStrip(int channel, int Length, int colorChannel)
{
  stripChannels.put(channel, new Strip(Length, colorChannel, channel));
  Channels++;
}

class Strip
```

```
{
  int stripLength;
  int colorChannels;
  int channel;
  byte[][] strip;
  int[][] virtualLED;

  Strip(int Length, int colorChannel, int displayChannel)
  {
    int posX = 20;
    int posY = 20;
    stripLength = Length;
    colorChannels = colorChannel;
    channel      = displayChannel;
    strip        = new byte[stripLength][colorChannels];
    virtualLED    = new int[stripLength][2];
    for( int i = 0; i < stripLength; i++)
    {
      virtualLED[i][0] = posX;
      posX += 5;
      virtualLED[i][1] = posY + (5*channel);
    }
  }

  void changeLength(int len)
  {
    stripLength = len;
    strip       = new byte[stripLength][colorChannels];
  }

  void displayStrip()
  {
    for( int i = 0; i < stripLength; i++)
    {
      noStroke();
      fill(int(strip[i][0]), int(strip[i][1]), int(strip[i][2]));
      ellipse(virtualLED[i][0], virtualLED[i][1], 5, 5);
    }
  }
}
```

## Appendix 13: Admin_Control source code

```
/*
previously called UDP_ADMIN_V1_1
V1.1 new class system

V1.0 Admin program
*/

import controlP5.*;
import hypermedia.net.*;

UDP udp;
ControlP5 cp5;

final String ADMIN_ADDR  = "192.168.1.27";
final int ADMIN_PORT     = 7500;
final String SERV_ADDR   = "192.168.1.27";
final int SERV_PORT      = 21234;
int myColorBackground = color(255);

void setup()
{
  size(925, 700);
  frameRate(30);
  noStroke();

  udp = new UDP(this, ADMIN_PORT, ADMIN_ADDR);
  udp.listen(true);
  cp5 = new ControlP5(this);
  setLucidaGrandeFont();
  setupTabs();
  setupDatabases();
  setupLists();
  setupDetailBox();
  setupStatusWindow();
}

void draw()
{
  background(myColorBackground);
  if(CurrentTab == "SENDERS")
  {
    displaySendersTab();
  }
  else if(CurrentTab == "DRIVERS")
  {
    displayDriversTab();
  }
  else if(CurrentTab == "GROUPS")
  {
    displayGroupsTab();
  }
  else if(CurrentTab == "STATUS")
  {
```

```
    displayStatusTab();
  }
}

void controlEvent(ControlEvent theControlEvent)
{
  if (theControlEvent.isTab())
  {
    checkTab(theControlEvent);
  }
  else if(theControlEvent.isController())
  {
    checkdetailButton(theControlEvent);
    checkstatusButton(theControlEvent);
  }
  else if(theControlEvent.isGroup())
  {
    checkDropdownList(theControlEvent);
  }
}

void receive(byte[] data)
{
  String s = new String(data);
  if(s.equals("Connect successful"))
  {
    statuswin.connectionState = 1;
  }
  else if(s.equals("Disconnect successful"))
  {
    statuswin.connectionState = 0;
  }
  else
  {
    String[] type = match(s,"&(.*?)/");
    String[][] m = matchAll(s, "(.*?)/");
    if(type[1].equals("SENDER"))
    {
      addSenderDataToDatabase(m);
    }
    else if(type[1].equals("DRIVER"))
    {
      addDriverDataToDatabase(m);
    }
  }
}

//int selectedField;

sendersList sl1;
driversList dl1;
final int numberSenderItemShown = 17;
final int numberDriverItemShown = 10;

public abstract class List
{
```

```
  public float chWidth, ipWidth, portWidth, otherWidth, totalWidth, rowHeight, totalHeight;
  public float xpos, ypos;
  public VScrollbar scrollbar;
  public ArrayList itemfieldList;
  public int selectedField;
  public List(float xp, float yp, float chw, float ipw, float ptw, float otw, float rh, int nis)
  {
    xpos = xp;
    ypos = yp;
    chWidth = chw;
    ipWidth = ipw;
    portWidth = ptw;
    otherWidth = otw;
    totalWidth = chWidth + ipWidth + portWidth + otherWidth;
    rowHeight = rh;
    totalHeight = nis * rowHeight;
    scrollbar = new VScrollbar(xpos+totalWidth+11, ypos, 16, totalHeight, nis);
    itemfieldList = new ArrayList<ItemField>();
  }
  public abstract void update();
  public abstract void display();
  public void drawListFrame()
  {
    stroke(67,126,247);
    strokeWeight(2);
    fill(132,170,247);
    rect(xpos, ypos, totalWidth, totalHeight);
    noFill();
    rect(xpos, ypos-rowHeight, totalWidth,rowHeight);
    rect(xpos, ypos-rowHeight, chWidth+ipWidth+portWidth,rowHeight);
    rect(xpos, ypos-rowHeight, chWidth+ipWidth,rowHeight);
    rect(xpos, ypos-rowHeight, chWidth,rowHeight);
  }
  public void addFrameText()
  {
    textSize(12);
    fill(0);
    text("field1", xpos+5, ypos+rowHeight/2-rowHeight);
    text("field2", xpos+chWidth+5, ypos+rowHeight/2-rowHeight);
    text("field3", xpos+chWidth+ipWidth+5, ypos+rowHeight/2-rowHeight);
    text("field4", xpos+chWidth+ipWidth+portWidth+5, ypos+rowHeight/2-rowHeight);
  }
}

public abstract class ItemField
{
  public String field1, field2, field3, field4;
  public float chWidth, ipWidth, portWidth, otherWidth, totalWidth, rowHeight;
  public float xpos, ypos;
  public ItemField(float chw, float ipw, float ptw, float otw, float tow, float rh)
  {
    chWidth = chw;
    ipWidth = ipw;
    portWidth = ptw;
    otherWidth = otw;
    totalWidth = tow;
```

```
    rowHeight = rh;
  }
  public abstract void update(int itemnumber);
  public void updateBlank()
  {
    field1 = "";
    field2 = "";
    field3 = "";
    field4 = "";
  }
  public boolean selectItemField(float xp, float yp)
  {
    boolean over;
    if (mouseX > xp && mouseX < xp+totalWidth && mouseY > yp && mouseY < yp+rowHeight)
    {
      over = true;
    }
    else
    {
      over = false;
    }

    if (mousePressed && over)
    {
      return true;
    }
    return false;
  }
  public void display(float xp, float yp, color c)
  {
    xpos = xp;
    ypos = yp;
    strokeWeight(1);
    stroke(67,126,247);
    fill(c);
    //180,214,255
    rect(xpos, ypos, totalWidth, rowHeight);
    rect(xpos, ypos, chWidth+ipWidth+portWidth+otherWidth, rowHeight);
    rect(xpos, ypos, chWidth+ipWidth+portWidth, rowHeight);
    rect(xpos, ypos, chWidth+ipWidth, rowHeight);
    rect(xpos, ypos, chWidth, rowHeight);
    fill(0);
    text(field1, xpos+2, ypos+rowHeight/2);
    text(field2, xpos+chWidth+2, ypos+rowHeight/2);
    text(field3, xpos+chWidth+ipWidth+2, ypos+rowHeight/2);
    text(field4, xpos+chWidth+ipWidth+portWidth+2, ypos+rowHeight/2);
  }
}

class sendersList extends List
{
  public sendersList(float xp, float yp, float chw, float ipw, float ptw, float otw, float rh)
  {
    super(xp, yp, chw, ipw, ptw, otw, rh, numberSenderItemShown);
  }
  public void update()
```

```
 {
   itemfieldList.clear();
   for(int i = 0; i<senderDatabase.size(); i++)
   {
     itemfieldList.add(new SenderItemField(chWidth, ipWidth, portWidth, otherWidth, totalWidth,
rowHeight));
     SenderItemField item = (SenderItemField)itemfieldList.get(i);
     item.update(i);
   }
   if(senderDatabase.size()<numberSenderItemShown)
   {
     for(int j = senderDatabase.size(); j <numberSenderItemShown; j++)
     {
       itemfieldList.add(new SenderItemField(chWidth, ipWidth, portWidth, otherWidth, totalWidth,
rowHeight));
       SenderItemField item = (SenderItemField)itemfieldList.get(j);
       item.updateBlank();
     }
   }
 }
 public void display()
 {
   // Draw the frame, the scrollbar of the list and add column description
   drawListFrame();
   addFrameText();
   scrollbar.update(senderDatabase.size());
   scrollbar.display();

   int itemnumber = 0;
   if(scrollbar.firstItemScroll(senderDatabase.size())<                    (senderDatabase.size()-
numberSenderItemShown))
   {
     itemnumber = scrollbar.firstItemScroll(senderDatabase.size());
   }
   else if(senderDatabase.size() >= numberSenderItemShown)
   {
     itemnumber = (senderDatabase.size() - numberSenderItemShown);
   }

   // Display senders data/info
   for(int i=0; i<numberSenderItemShown; i++)
   {
     SenderItemField item = (SenderItemField)itemfieldList.get(itemnumber);
     if(item.selectItemField(xpos, ypos+i*rowHeight))
     {
       if(itemnumber >= senderDatabase.size())
       {
         if(senderDatabase.size() != 0)
         {
           selectedField = senderDatabase.size()-1;
         }
         else
         {
           selectedField = 0;
         }
       }
```

```
      else
      {
        selectedField = itemnumber;
      }
      item.display(xpos, ypos+i*rowHeight, color(255));

      if(senderDatabase.size() > 0)
      {
        if(senderDatabase.get(selectedField).getClassType() == CLASS_STRIP)
        {
          detailbox_ptr = detail_strip_s;
        }
        else //if(senderDatabase.get(selectedField).getClassType() == CLASS_MATRIX)
        {
          detailbox_ptr = detail_matrix_s;
        }
      }
    }
    else if (selectedField == itemnumber)
    {
      item.display(xpos, ypos+i*rowHeight, color(255));
      if(senderDatabase.size() > 0)
      {
        if(senderDatabase.get(selectedField).getClassType() == CLASS_STRIP)
        {
          detailbox_ptr = detail_strip_s;
        }
        else if(senderDatabase.get(selectedField).getClassType() == CLASS_MATRIX)
        {
          detailbox_ptr = detail_matrix_s;
        }
         else //if(senderDatabase.get(selectedField).getClassType() == CLASS_RAW)
        {
          detailbox_ptr = detail_raw_s;
        }
      }
    }
    else
    {
      item.display(xpos, ypos+i*rowHeight, color(180,214,255));
    }
    itemnumber++;
  }
 }
 public void addFrameText()
 {
  textSize(12);
  fill(0);
  text("Sender", xpos+5, ypos+rowHeight/2-rowHeight);
  text("IP", xpos+chWidth+5, ypos+rowHeight/2-rowHeight);
  text("Port", xpos+chWidth+ipWidth+5, ypos+rowHeight/2-rowHeight);
  text("Logged on since", xpos+chWidth+ipWidth+portWidth+5, ypos+rowHeight/2-rowHeight);
 }
}

class driversList extends List
```

```
{
  public driversList(float xp, float yp, float chw, float ipw, float ptw, float otw, float rh)
  {
    super(xp, yp, chw, ipw, ptw, otw, rh, numberDriverItemShown);
  }
  public void update()
  {
    itemfieldList.clear();
    for(int i = 0; i<driverDatabase.size(); i++)
    {
      itemfieldList.add(new DriverItemField(chWidth, ipWidth, portWidth, otherWidth, totalWidth,
rowHeight));
      DriverItemField item = (DriverItemField)itemfieldList.get(i);
      item.update(i);
    }
    if(driverDatabase.size()<numberDriverItemShown)
    {
      for(int j = driverDatabase.size(); j <numberDriverItemShown; j++)
      {
        itemfieldList.add(new DriverItemField(chWidth, ipWidth, portWidth, otherWidth, totalWidth,
rowHeight));
        DriverItemField item = (DriverItemField)itemfieldList.get(j);
        item.updateBlank();
      }
    }
  }
  public void display()
  {
    // Draw the frame, the scrollbar of the list and add column description
    drawListFrame();
    addFrameText();
    scrollbar.update(driverDatabase.size());
    scrollbar.display();

    int itemnumber = 0;
    if(scrollbar.firstItemScroll(driverDatabase.size())< (driverDatabase.size()-numberDriverItemShown))
    {
      itemnumber = scrollbar.firstItemScroll(driverDatabase.size());
    }
    else if(driverDatabase.size() >= numberDriverItemShown)
    {
      itemnumber = (driverDatabase.size() - numberDriverItemShown);
    }

    // Display senders data/info
    for(int i=0; i<numberDriverItemShown; i++)
    {
      DriverItemField item = (DriverItemField)itemfieldList.get(itemnumber);
      if(item.selectItemField(xpos, ypos+i*rowHeight))
      {
        if(itemnumber >= driverDatabase.size())
        {
          if(driverDatabase.size() > 0)
          {
            selectedField = driverDatabase.size()-1;
          }
```

```
        else
        {
          selectedField = 0;
        }
      }
      else
      {
        selectedField = itemnumber;
      }
      item.display(xpos, ypos+i*rowHeight, color(255));

      if(driverDatabase.size() > 0)
      {
        if(driverDatabase.get(selectedField).getClassType() == CLASS_STRIP)
        {
          detailbox_ptr = detail_strip_d;

cp5.get(Textfield.class,"LENGTH").setText(Integer.toString(((Strip)driverDatabase.get(selectedField)).
getLength()));
        }
        else if(driverDatabase.get(selectedField).getClassType() == CLASS_MATRIX)
        {
          detailbox_ptr = detail_matrix_d;

cp5.get(Textfield.class,"WIDTH").setText(Integer.toString(((Matrix)driverDatabase.get(selectedField)).
getWidth()));

cp5.get(Textfield.class,"HEIGHT").setText(Integer.toString(((Matrix)driverDatabase.get(selectedField)
).getHeight()));
        }
        else //if(driverDatabase.get(selectedField).getClassType() == CLASS_RAW)
        {
          detailbox_ptr = detail_raw_d;
        }
      }
    }
    else if (selectedField == itemnumber)
    {
      item.display(xpos, ypos+i*rowHeight, color(255));

      if(driverDatabase.size() > 0)
      {
        if(driverDatabase.get(selectedField).getClassType() == CLASS_STRIP)
        {
          detailbox_ptr = detail_strip_d;

//cp5.get(Textfield.class,"LENGTH").setText(Integer.toString(((Strip)driverDatabase.get(selectedField)
).getLength()));
        }
        else if(driverDatabase.get(selectedField).getClassType() == CLASS_MATRIX)
        {
          detailbox_ptr = detail_matrix_d;

//cp5.get(Textfield.class,"WIDTH").setText(Integer.toString(((Matrix)driverDatabase.get(selectedField)
).getWidth()));
```

```
//cp5.get(Textfield.class,"HEIGHT").setText(Integer.toString(((Matrix)driverDatabase.get(selectedField
)).getHeight()));
          }
          else //if(driverDatabase.get(selectedField).getClassType() == CLASS_RAW)
          {
            detailbox_ptr = detail_raw_d;
          }
        }
      }
      else
      {
        item.display(xpos, ypos+i*rowHeight, color(180,214,255));
      }
      itemnumber++;
    }
  }
  public void addFrameText()
  {
    textSize(12);
    fill(0);
    text("Driver", xpos+5, ypos+rowHeight/2-rowHeight);
    text("IP", xpos+chWidth+5, ypos+rowHeight/2-rowHeight);
    text("Port", xpos+chWidth+ipWidth+5, ypos+rowHeight/2-rowHeight);
    text("Logged on since", xpos+chWidth+ipWidth+portWidth+5, ypos+rowHeight/2-rowHeight);
  }
}

class SenderItemField extends ItemField
{
  public SenderItemField(float chw, float ipw, float ptw, float otw, float tow, float rh)
  {
    super(chw, ipw, ptw, otw, tow, rh);
  }
  public void update(int itemnumber)
  {
    field1 = (String)(senderDatabase.get(itemnumber)).getledName();
    field2 = (String)(senderDatabase.get(itemnumber)).getSenderIP();
    field3 = Integer.toString((senderDatabase.get(itemnumber)).getSenderPort());
    field4 = (String)(senderDatabase.get(itemnumber)).getOther();
  }
}

class DriverItemField extends ItemField
{
  public DriverItemField(float chw, float ipw, float ptw, float otw, float tow, float rh)
  {
    super(chw, ipw, ptw, otw, tow, rh);
  }
  public void update(int itemnumber)
  {
    field1 = (String)(driverDatabase.get(itemnumber)).getledName();
    field2 = (String)(driverDatabase.get(itemnumber)).getDriverIP();
    field3 = Integer.toString((driverDatabase.get(itemnumber)).getDriverPort());
    field4 = (String)(driverDatabase.get(itemnumber)).getOther();
  }
```

```
}

// Scrollbar Class
class VScrollbar
{
  public float swidth, sheight;    // width and height of bar
  public float xpos, ypos;       // x and y position of bar
  public float spos, newspos;    // y position of slider
  public float sposMin, sposMax; // max and min values of slider
  public int loose;              // how loose/heavy
  public boolean over;           // is the mouse over the slider?
  public boolean locked;
  public float ratio;
  public int numberItemShown;
  public VScrollbar (float xp, float yp, float sw, float sh, int nis)
  {
    swidth = sw;
    sheight = sh;
    xpos = xp-swidth/2;
    ypos = yp;
    spos = ypos;
    newspos = spos;
    sposMin = ypos;
    sposMax = ypos + sheight - 100;
    ratio = (float)sheight/ (sposMax-ypos);
    numberItemShown = nis;
  }
  public void update(int totalItems)
  {
    if(totalItems <numberItemShown)
    {
      totalItems = numberItemShown;
    }
    sposMax = ypos + sheight - (numberItemShown*(sheight/totalItems));
    ratio = (float)sheight/ (sposMax-ypos);

    if (overEvent())
    {
      over = true;
    }
    else
    {
      over = false;
    }

    if (mousePressed && over)
    {
      locked = true;
    }

    if (!mousePressed)
    {
      locked = false;
    }

    if (locked)
```

```
  {
    newspos = constrain(mouseY - (numberItemShown*(sheight/totalItems))/2, sposMin, sposMax);
  }

  if (abs(newspos - spos) > 1)
  {
    spos = newspos;
  }
}
public float constrain(float val, float minv, float maxv)
{
  return min(max(val, minv), maxv);
}
public boolean overEvent()
{
  if (mouseX > xpos && mouseX < xpos+swidth && mouseY > ypos && mouseY < ypos+sheight)
  {
    return true;
  }
  else
  {
    return false;
  }
}
public void display()
{
  noStroke();
  fill(204);
  rect(xpos, ypos, swidth, sheight);

  if (over || locked)
  {
    fill(0, 0, 0);
  }
  else
  {
    fill(102, 102, 102);
  }
  rect(xpos, spos, swidth, ypos+sheight - sposMax);
}
public float getPos()
{
  // Convert spos to be values between
  // 0 and the total height of the scrollbar
  return (spos-ypos)*ratio ;
}
public int firstItemScroll(int totalItems)
{
  if(totalItems <= numberItemShown)
  {
    return 0;
  }
  else
  {
    return (int)(getPos()/(float)(sheight/(totalItems-numberItemShown)));
  }
```

```
  }
}

void setupLists()
{
  sl1 = new sendersList(50, 130, 200, 200, 200, 200, 30);
  dl1 = new driversList(50, 130, 200, 200, 200, 200, 30);
}

String CurrentTab = "SENDERS";

void setupTabs()
{
  // Set beginning position of the tabs
 cp5.window().setPositionOfTabs(new PVector(50,50,0));

 // Add tabs
 cp5.addTab("DRIVERS")
     .setColorBackground(color(0, 160, 100))
     .setColorLabel(color(255))
     .setColorActive(color(255,128,0))
     .setWidth(125)
     .setHeight(30)
     ;

  cp5.addTab("GROUPS")
     .setColorBackground(color(0, 160, 100))
     .setColorLabel(color(255))
     .setColorActive(color(255,128,0))
     .setWidth(125)
     .setHeight(30)
     ;

   cp5.addTab("STATUS")
     .setColorBackground(color(0, 160, 100))
     .setColorLabel(color(255))
     .setColorActive(color(255,128,0))
     .setWidth(125)
     .setHeight(30)
     ;

 // Collect all the tabs
 cp5.getTab("default")
     .activateEvent(true)
     .setLabel("SENDERS")
     .setId(1)
     .setWidth(125)
     .setHeight(30)
     .setColorBackground(color(0, 160, 100))
     .setColorLabel(color(255))
     .setColorActive(color(255,128,0))
     ;

 cp5.getTab("DRIVERS")
     .activateEvent(true)
     .setId(2)
```

```
    ;

  cp5.getTab("GROUPS")
    .activateEvent(true)
    .setId(3)
    ;

  cp5.getTab("STATUS")
    .activateEvent(true)
    .setId(4)
    ;
}

void checkTab(ControlEvent theControlEvent)
{
  if(theControlEvent.getTab().getName() == "default")
  {
    CurrentTab = "SENDERS";
  }
  else if(theControlEvent.getTab().getName() == "DRIVERS")
  {
    CurrentTab = "DRIVERS";
  }
  else if(theControlEvent.getTab().getName() == "GROUPS")
  {
    CurrentTab = "GROUPS";
  }
  else if(theControlEvent.getTab().getName() == "STATUS")
  {
    CurrentTab = "STATUS";
  }
}

void displaySendersTab()
{
  sl1.update();
  sl1.display();
  //if(senderDatabase.size() > 0)
  //{
    //detailbox_ptr.display();
  //}
}

void displayDriversTab()
{
  dl1.update();
  dl1.display();
  if(driverDatabase.size() > 0)
  {
    detailbox_ptr.update(driverDatabase.get(dl1.selectedField));
    detailbox_ptr.display();
  }
}

void displayGroupsTab()
{
```

```
}

void displayStatusTab()
{
  statuswin.display();
}

final byte ADMIN_CONNECT     = 0x00;
final byte ADMIN_DISCONNECT  = 0x01;
final byte REQ_SNDDATA       = 0x02;
final byte REQ_DRVDATA       = 0x03;
final byte SETT_CHANGE       = 0x04;

final byte CLASS_STRIP   = 0x00;
final byte CLASS_MATRIX  = 0x01;
final byte CLASS_RAW     = 0x03;
final byte TYPE_ADDR     = 0x02;
final byte TYPE_NONADDR  = 0x03;
final byte CONFIG_LINE   = 0x04;
final byte CONFIG_OTHER  = 0x05;
final byte COLOR_MONO    = 0x06;
final byte COLOR_RGB     = 0x07;

void connectToServer()
{
  byte[] msgUDP = new byte[3];
  msgUDP[0] = '@';
  msgUDP[1] = ADMIN_CONNECT;
  msgUDP[2] = '/';

  String ip       = SERV_ADDR;
  int port        = SERV_PORT;

  udp.send(msgUDP, ip, port);
  msgUDP = null;
}

void DisconnectServer()
{
  byte[] msgUDP = new byte[3];
  msgUDP[0] = '@';
  msgUDP[1] = ADMIN_DISCONNECT;
  msgUDP[2] = '/';

  String ip       = SERV_ADDR;
  int port        = SERV_PORT;

  udp.send(msgUDP, ip, port);
  msgUDP = null;
}

void requestSendersData()
{
  byte[] msgUDP = new byte[3];
  msgUDP[0] = '&';
```

```
  msgUDP[1] = REQ_SNDDATA;
  msgUDP[2] = '/';

  String ip       = SERV_ADDR;
  int port        = SERV_PORT;

  udp.send(msgUDP, ip, port);
  msgUDP = null;
}

void requestDriversData()
{
  byte[] msgUDP = new byte[3];
  msgUDP[0] = '&';
  msgUDP[1] = REQ_DRVDATA;
  msgUDP[2] = '/';

  String ip       = SERV_ADDR;
  int port        = SERV_PORT;

  udp.send(msgUDP, ip, port);
  msgUDP = null;
}

void changeDriverSettings()
{
  byte[] msgUDP = new byte[19];
  msgUDP[0] = '@';
  msgUDP[1] = SETT_CHANGE;
  msgUDP[2] = '/';
  msgUDP[3] = byte(dl1.selectedField>>8);
  msgUDP[4] = byte(dl1.selectedField);
  msgUDP[5] = '/';

  if(driverDatabase.get(dl1.selectedField).getClassType() == CLASS_STRIP)
  {
    msgUDP[6] = CLASS_STRIP;
    msgUDP[7] = '/';
    if(((Strip)driverDatabase.get(dl1.selectedField)).getType() == TYPE_ADDR)
    {
      msgUDP[13] = TYPE_ADDR;
      msgUDP[14] = '/';
    }
    else if(((Strip)driverDatabase.get(dl1.selectedField)).getType() == TYPE_NONADDR)
    {
      msgUDP[13] = TYPE_NONADDR;
      msgUDP[14] = '/';
    }

    if(((Strip)driverDatabase.get(dl1.selectedField)).getConfiguration() == CONFIG_LINE)
    {
      msgUDP[15] = CONFIG_LINE;
      msgUDP[16] = '/';
    }
    else if(((Strip)driverDatabase.get(dl1.selectedField)).getConfiguration() == CONFIG_OTHER)
    {
```

```
      msgUDP[15] = CONFIG_OTHER;
      msgUDP[16] = '/';
    }

    // length. only 16 bits used
    byte[] striplen = toBytes(((Strip)driverDatabase.get(dl1.selectedField)).getLength());
    msgUDP[17] = striplen[2];
    msgUDP[18] = striplen[3];
  }
  else if(driverDatabase.get(dl1.selectedField).getClassType() == CLASS_MATRIX)
  {
     msgUDP[6] = CLASS_MATRIX;
     msgUDP[7] = '/';

     // length. only 16 bits used
     byte[] widthmtx = toBytes(((Matrix)driverDatabase.get(dl1.selectedField)).getWidth());
     byte[] heightmtx = toBytes(((Matrix)driverDatabase.get(dl1.selectedField)).getHeight());
     msgUDP[13] = heightmtx[2];
     msgUDP[14] = heightmtx[3];
     msgUDP[15] = '/';
     msgUDP[16] = widthmtx[2];
     msgUDP[17] = widthmtx[3];
  }
  else if(driverDatabase.get(dl1.selectedField).getClassType() == CLASS_RAW)
  {
    msgUDP[6] = CLASS_RAW;
    msgUDP[7] = '/';
  }

  if(driverDatabase.get(dl1.selectedField).getColorMode() == COLOR_MONO)
  {
     msgUDP[8] = COLOR_MONO;
     msgUDP[9] = '/';
  }
  else if(driverDatabase.get(dl1.selectedField).getColorMode() == COLOR_RGB)
  {
     msgUDP[8] = COLOR_RGB;
     msgUDP[9] = '/';
  }
  //Group number
  msgUDP[10] = 0x00;
  msgUDP[11] = 0x01;
  msgUDP[12] = '/';

  String ip       = SERV_ADDR;
  int port        = SERV_PORT;

  udp.send(msgUDP, ip, port);
  msgUDP = null;
}

Box detailbox_ptr;
senderBoxStrip detail_strip_s;
senderBoxMatrix detail_matrix_s;
senderBoxRaw detail_raw_s;
driverBoxStrip detail_strip_d;
```

```
driverBoxMatrix detail_matrix_d;
driverBoxRaw detail_raw_d;

DropdownList classType, colorMode, ledType, config_dp;
Textfield length_st, width_mtx, height_mtx;

int windowWidth = 800;
int windowHeight = 200;

public abstract class Box
{
  public float xpos, ypos;
  public Box(float xp, float yp)
  {
    xpos = xp;
    ypos = yp;
    cp5.addButton("Apply")
       .setBroadcast(false)
       .setValue(0)
       .setPosition(540,650)
       .setSize(150,19)
       .setLabel("APPLY")
       .getCaptionLabel().align(CENTER,CENTER)

       ;

    cp5.getController("Apply").moveTo("DRIVERS");
    cp5.getController("Apply").setBroadcast(true);
  }
  public void display()
  {
    noStroke();
    fill(220);
    rect(xpos, ypos, windowWidth, windowHeight);
  }
  public abstract void update(LED l);
}

public abstract class senderBox extends Box
{
  public senderBox(float xp, float yp)
  {
    super(xp,yp);
    cp5.addButton("RefreshSender")
       .setBroadcast(false)
       .setValue(0)
       .setPosition(700,650)
       .setSize(150,19)
       .setLabel("REFRESH")
       .getCaptionLabel().align(CENTER,CENTER)
       ;

    cp5.getController("RefreshSender").moveTo("default");
    cp5.getController("RefreshSender").setBroadcast(true);
  }
  public abstract void update(LED l);
```

```
}

public abstract class driverBox extends Box
{
  public driverBox(float xp, float yp)
  {
    super(xp,yp);
    cp5.addButton("RefreshDriver")
       .setBroadcast(false)
       .setValue(0)
       .setPosition(700,650)
       .setSize(150,19)
       .setLabel("REFRESH")
       .getCaptionLabel().align(CENTER,CENTER)
       ;
    cp5.getController("RefreshDriver").moveTo("DRIVERS");
    cp5.getController("RefreshDriver").setBroadcast(true);
  }
  public abstract void update(LED l);
}

public class senderBoxStrip extends senderBox
{
  public senderBoxStrip(float xp, float yp)
  {
    super(xp,yp);
  }
  public void update(LED l)
  {
    //display sender strip info
  }
}

public class senderBoxMatrix extends senderBox
{
  public senderBoxMatrix(float xp, float yp)
  {
    super(xp,yp);
  }
  public void update(LED l)
  {
    // display sender matrix info
  }
}

public class senderBoxRaw extends senderBox
{
  public senderBoxRaw(float xp, float yp)
  {
    super(xp,yp);
  }
  public void update(LED l)
  {
    //display sender strip info
  }
}
```

```
public class driverBoxStrip extends driverBox
{
  public driverBoxStrip(float xp, float yp)
  {
    super(xp,yp);
    classType = cp5.addDropdownList("Class")
                   .setPosition(150,490)
                   .setSize(220, 300)
                   .setBarHeight(25)
                   .setItemHeight(25)
                   .moveTo("DRIVERS")
                   ;

    colorMode = cp5.addDropdownList("Color Mode")
                   .setPosition(400, 490)
                   .setSize(220, 300)
                   .setBarHeight(25)
                   .setItemHeight(25)
                   .moveTo("DRIVERS")
                   ;

    ledType = cp5.addDropdownList("LED type")
                  .setPosition(150, 590)
                  .setSize(220, 300)
                  .setBarHeight(25)
                  .setItemHeight(25)
                  .moveTo("DRIVERS")
                  ;

    config_dp = cp5.addDropdownList("Configuration")
                   .setPosition(400, 590)
                   .setSize(220, 300)
                   .setBarHeight(25)
                   .setItemHeight(25)
                   .moveTo("DRIVERS")
                   ;

    length_st = cp5.addTextfield("LENGTH")
                  .setPosition(650,500)
                  .setSize(150,40)
                  .setFocus(true)
                  .setColor(color(0,0,0))
                  .setColorBackground(color(255,255,255))
                  .setColorCursor(color(0,0,0))
                  .setColorCaptionLabel(color(0,0,0))
                  ;

    config_dp.addItem("line", 1);
    config_dp.addItem("other", 2);
    ledType.addItem("addressable", 1);
    ledType.addItem("non-addressable", 2);
    colorMode.addItem("mono", 1);
    colorMode.addItem("RGB", 2);
    classType.addItem("strip", 1);
    classType.addItem("matrix", 2);
```

```
    classType.addItem("raw",3);

    cp5.getController("LENGTH").moveTo("DRIVERS");
    config_dp.hide();
    ledType.hide();
    length_st.hide();
  }
  public void update(LED l)
  {
    config_dp.show();
    ledType.show();
    length_st.show();
    width_mtx.hide();
    height_mtx.hide();
    if(l.getClassType() == CLASS_STRIP)
    {
      classType.setIndex(0);
    }
    else if(l.getClassType() == CLASS_MATRIX)
    {
      classType.setIndex(1);
    }
    else //if(l.getClassType() == CLASS_RAW)
    {
      classType.setIndex(2);
    }

    if(((Strip)l).getType() == TYPE_ADDR) //<>//
    {
      ledType.setIndex(0);
    }
    else //if(((Strip)l).getType() == TYPE_NONADDR)
    {
      ledType.setIndex(1);
    }

    if(((Strip)l).getConfiguration() == CONFIG_LINE)
    {
      config_dp.setIndex(0);
    }
    else //if(((Strip)l).getConfiguration() == CONFIG_OTHER)
    {
      config_dp.setIndex(1);
    }

    if(l.getColorMode() == COLOR_MONO)
    {
      colorMode.setIndex(0);
    }
    else //if(l.getColorMode() == COLOR_RGB)
    {
      colorMode.setIndex(1);
    }
  }
}
```

```
public class driverBoxMatrix extends driverBox
{
  public driverBoxMatrix(float xp, float yp)
  {
    super(xp,yp);
    classType = cp5.addDropdownList("Class")
                .setPosition(150,490)
                .setSize(220, 300)
                .setBarHeight(25)
                .setItemHeight(25)
                .moveTo("DRIVERS")
                ;

    colorMode = cp5.addDropdownList("Color Mode")
                .setPosition(400, 490)
                .setSize(220, 300)
                .setBarHeight(25)
                .setItemHeight(25)
                .moveTo("DRIVERS")
                ;

    width_mtx = cp5.addTextfield("WIDTH")
                .setPosition(150,550)
                .setSize(150,40)
                .setFocus(true)
                .setColor(color(0,0,0))
                .setColorBackground(color(255,255,255))
                .setColorCursor(color(0,0,0))
                .setColorCaptionLabel(color(0,0,0))
                ;

    height_mtx = cp5.addTextfield("HEIGHT")
                 .setPosition(400,550)
                 .setSize(150,40)
                 .setFocus(true)
                 .setColor(color(0,0,0))
                 .setColorBackground(color(255,255,255))
                 .setColorCursor(color(0,0,0))
                 .setColorCaptionLabel(color(0,0,0))
                 ;

    colorMode.addItem("mono", 1);
    colorMode.addItem("RGB", 2);
    classType.addItem("strip", 1);
    classType.addItem("matrix", 2);
    classType.addItem("raw",3);

    cp5.getController("WIDTH").moveTo("DRIVERS");
    cp5.getController("HEIGHT").moveTo("DRIVERS");
    width_mtx.hide();
    height_mtx.hide();
    config_dp.hide();
    ledType.hide();
    length_st.hide();
  }
  public void update(LED l)
```

```
 {
   width_mtx.show();
   height_mtx.show();
   config_dp.hide();
   ledType.hide();
   length_st.hide();
   if(l.getClassType() == CLASS_STRIP)
   {
     classType.setIndex(0);
     if(((Strip)l).getType() == TYPE_ADDR)
     {
       ledType.setIndex(0);
     }
     else //if(((Strip)l).getType() == TYPE_NONADDR)
     {
       ledType.setIndex(1);
     }

     if(((Strip)l).getConfiguration() == CONFIG_LINE)
     {
       config_dp.setIndex(0);
     }
     else //if(((Strip)l).getConfiguration() == CONFIG_OTHER)
     {
       config_dp.setIndex(1);
     }
   }
   else if(l.getClassType() == CLASS_MATRIX)
   {
     classType.setIndex(1);
   }
   else //if(l.getClassType() == CLASS_RAW)
   {
     classType.setIndex(2);
   }

   if(l.getColorMode() == COLOR_MONO)
   {
     colorMode.setIndex(0);
   }
   else //if(l.getColorMode() == COLOR_RGB)
   {
     colorMode.setIndex(1);
   }
 }
}

public class driverBoxRaw extends driverBox
{
  public driverBoxRaw(float xp, float yp)
  {
    super(xp,yp);
    classType = cp5.addDropdownList("Class")
                   .setPosition(150,490)
                   .setSize(220, 300)
                   .setBarHeight(25)
```

```
                .setItemHeight(25)
                .moveTo("DRIVERS")
                ;

  colorMode = cp5.addDropdownList("Color Mode")
                .setPosition(400, 490)
                .setSize(220, 300)
                .setBarHeight(25)
                .setItemHeight(25)
                .moveTo("DRIVERS")
                ;

  colorMode.addItem("mono", 1);
  colorMode.addItem("RGB", 2);
  classType.addItem("strip", 1);
  classType.addItem("matrix", 2);
  classType.addItem("raw",3);

  cp5.getController("WIDTH").moveTo("DRIVERS");
  cp5.getController("HEIGHT").moveTo("DRIVERS");
  width_mtx.hide();
  height_mtx.hide();
  config_dp.hide();
  ledType.hide();
  length_st.hide();
}
public void update(LED l)
{
  width_mtx.hide();
  height_mtx.hide();
  config_dp.hide();
  ledType.hide();
  length_st.hide();
  if(l.getClassType() == CLASS_STRIP)
  {
    classType.setIndex(0);
    if(((Strip)l).getType() == TYPE_ADDR)
    {
      ledType.setIndex(0);
    }
    else //if(((Strip)l).getType() == TYPE_NONADDR)
    {
      ledType.setIndex(1);
    }

    if(((Strip)l).getConfiguration() == CONFIG_LINE)
    {
      config_dp.setIndex(0);
    }
    else //if(((Strip)l).getConfiguration() == CONFIG_OTHER)
    {
      config_dp.setIndex(1);
    }
  }
  else if(l.getClassType() == CLASS_MATRIX)
  {
```

```
      classType.setIndex(1);
    }
    else //if(l.getClassType() == CLASS_RAW)
    {
      classType.setIndex(2);
    }

    if(l.getColorMode() == COLOR_MONO)
    {
      colorMode.setIndex(0);
    }
    else //if(l.getColorMode() == COLOR_RGB)
    {
      colorMode.setIndex(1);
    }
  }
}

void setupDetailBox()
{
  detail_strip_s = new senderBoxStrip(50, 450);
  detail_matrix_s = new senderBoxMatrix(50, 450);
  detail_raw_s = new senderBoxRaw(50, 450);
  detail_strip_d = new driverBoxStrip(50, 450);
  detail_matrix_d = new driverBoxMatrix(50, 450);
  detail_raw_d = new driverBoxRaw(50, 450);
}

void checkdetailButton(ControlEvent theControlEvent)
{
  if(theControlEvent.controller().name()=="RefreshSender")
  {
    requestSendersData();
  }
  else if(theControlEvent.controller().name()=="RefreshDriver")
  {
    requestDriversData();
  }
  else if(theControlEvent.controller().name()=="Apply")
  {
    if(driverDatabase.get(dl1.selectedField).getClassType() == CLASS_STRIP)
    {

((Strip)driverDatabase.get(dl1.selectedField)).setLength(Integer.parseInt(cp5.get(Textfield.class,"LEN
GTH").getText()));
    }
    else if(driverDatabase.get(dl1.selectedField).getClassType() == CLASS_MATRIX)
    {

((Matrix)driverDatabase.get(dl1.selectedField)).setWidth(Integer.parseInt(cp5.get(Textfield.class,"WID
TH").getText()));

((Matrix)driverDatabase.get(dl1.selectedField)).setHeight(Integer.parseInt(cp5.get(Textfield.class,"HEI
GHT").getText()));
    }
    else //if(driverDatabase.get(dl1.selectedField).getClassType() == CLASS_RAW)
```

```
    {

    }

    changeDriverSettings();
    requestDriversData();
  }
}

void checkDropdownList(ControlEvent theControlEvent)
{
  if(theControlEvent.getGroup().getName() == "Class")
  {
    if(theControlEvent.getGroup().getValue() == 1.0)
    {
      if(driverDatabase.get(dl1.selectedField).getClassType() != CLASS_STRIP)
      {
        replacewithStrip();
      }
    }
    else if(theControlEvent.getGroup().getValue() == 2.0)
    {
      if(driverDatabase.get(dl1.selectedField).getClassType() != CLASS_MATRIX)
      {
        replacewithMatrix();
      }
    }
    else //if(theControlEvent.getGroup().getValue() == 3.0)
    {
      if(driverDatabase.get(dl1.selectedField).getClassType() != CLASS_RAW)
      {
        replacewithRaw();
      }
    }
  }
  else if(theControlEvent.getGroup().getName() == "Color Mode")
  {
    if(theControlEvent.getGroup().getValue() == 1.0)
    {
      (driverDatabase.get(dl1.selectedField)).setColorMode(COLOR_MONO);
    }
    else if(theControlEvent.getGroup().getValue() == 2.0)
    {
      (driverDatabase.get(dl1.selectedField)).setColorMode(COLOR_RGB);
    }
  }
  else if(theControlEvent.getGroup().getName() == "LED type")
  {
    if(theControlEvent.getGroup().getValue() == 1.0)
    {
      ((Strip)(driverDatabase.get(dl1.selectedField))).setType(TYPE_ADDR);
    }
    else if(theControlEvent.getGroup().getValue() == 2.0)
    {
      ((Strip)(driverDatabase.get(dl1.selectedField))).setType(TYPE_NONADDR);
    }
```

```
  }
  else if(theControlEvent.getGroup().getName() == "Configuration")
  {
    if(theControlEvent.getGroup().getValue() == 1.0)
    {
      ((Strip)(driverDatabase.get(dl1.selectedField))).setConfiguration(CONFIG_LINE);
    }
    else if(theControlEvent.getGroup().getValue() == 2.0)
    {
      ((Strip)(driverDatabase.get(dl1.selectedField))).setConfiguration(CONFIG_OTHER);
    }
  }
}


// Converts 2 bytes into an integer.
// Useful for recomposition of an integer sent through UDP.
int ByteToInt(byte MSB, byte LSB)
{
    return ((int)MSB<<8) | ((int)LSB&0xFF);
}

// Converts an interger into a byte.
// Byte is always signed
// int = b & 0xFF
byte[] toBytes(int i)
{
  byte[] result = new byte[4];

  result[0] = (byte) (i >> 24);
  result[1] = (byte) (i >> 16);
  result[2] = (byte) (i >> 8);
  result[3] = (byte) (i /*>> 0*/);

  return result;
}

void printFontsAvailable()
{
  // Print the fonts available to use
 String[] fontList = PFont.list();
 println(fontList);
}

void setLucidaGrandeFont()
{
  PFont LucidaGrande = createFont("LucidaGrande",20, true);
  cp5.setControlFont(LucidaGrande);
}

//Function to convert integer to string
/* Integer.toString(int) */

//Function to convert string to integer
/* Integer.parseInt(string) */
```

```
HashMap<Integer, LED> senderDatabase;
HashMap<Integer, LED> driverDatabase;

/*
final byte CLASS_STRIP   = 0x00;
final byte CLASS_MATRIX  = 0x01;
final byte TYPE_ADDR     = 0x02;
final byte TYPE_NONADDR  = 0x03;
final byte CONFIG_LINE   = 0x04;
final byte CONFIG_OTHER  = 0x05;
final byte COLOR_MONO    = 0x06;
final byte COLOR_RGB     = 0x07;
*/

void setupDatabases()
{
  senderDatabase = new HashMap<Integer, LED>();
  driverDatabase = new HashMap<Integer, LED>();
}

void addSenderDataToDatabase(String[][] info)
{
  senderDatabase.clear();
  int index = 0;
  for(int i=1; i<info.length; i=i+4)
  {
    if(info[i+3][1].equals("strip"))
    {
      senderDatabase.put(index, new Strip());
      senderDatabase.get(index).setledName(info[i][1]);
      senderDatabase.get(index).setSenderIP(info[i+1][1]);
      senderDatabase.get(index).setSenderPort(Integer.parseInt(info[i+2][1]));
      senderDatabase.get(index).setOther("other");
      senderDatabase.get(index).setClassType(CLASS_STRIP);
    }
    else if(info[i+3][1].equals("matrix"))
    {
      senderDatabase.put(index, new Matrix());
      senderDatabase.get(index).setledName(info[i][1]);
      senderDatabase.get(index).setSenderIP(info[i+1][1]);
      senderDatabase.get(index).setSenderPort(Integer.parseInt(info[i+2][1]));
      senderDatabase.get(index).setOther("other");
      senderDatabase.get(index).setClassType(CLASS_MATRIX);
    }
    else if(info[i+3][1].equals("raw"))
    {
      senderDatabase.put(index, new Raw());
      senderDatabase.get(index).setledName(info[i][1]);
      senderDatabase.get(index).setSenderIP(info[i+1][1]);
      senderDatabase.get(index).setSenderPort(Integer.parseInt(info[i+2][1]));
      senderDatabase.get(index).setOther("other");
      senderDatabase.get(index).setClassType(CLASS_RAW);
    }
    index++;
  }
}
```

```
void addDriverDataToDatabase(String[][] info)
{
  driverDatabase.clear();
  int index = 0;
  int i = 1;
//  for(int i=1; i<info.length; i=i)
//  {
  while(i<info.length)
  {
    if(info[i+3][1].equals("strip"))
    {
      driverDatabase.put(index, new Strip());
      driverDatabase.get(index).setledName(info[i][1]);
      driverDatabase.get(index).setDriverIP(info[i+1][1]);
      driverDatabase.get(index).setDriverPort(Integer.parseInt(info[i+2][1]));
      driverDatabase.get(index).setOther("other");
      driverDatabase.get(index).setClassType(CLASS_STRIP);

      if(info[i+4][1].equals("mono"))
      {
        driverDatabase.get(index).setColorMode(COLOR_MONO);
      }
      else //if(info[i+4][1].equals("RGB"))
      {
        driverDatabase.get(index).setColorMode(COLOR_RGB);
      }

      if(info[i+5][1].equals("addressable"))
      {
        ((Strip)driverDatabase.get(index)).setType(TYPE_ADDR);
      }
      else //if(info[i+5][1].equals("non-addressable"))
      {
        ((Strip)driverDatabase.get(index)).setType(TYPE_NONADDR);
      }

      if(info[i+6][1].equals("line"))
      {
        ((Strip)driverDatabase.get(index)).setConfiguration(CONFIG_LINE);
      }
      else //if(info[i+6][1].equals("other"))
      {
        ((Strip)driverDatabase.get(index)).setConfiguration(CONFIG_OTHER);
      }

      ((Strip)driverDatabase.get(index)).setLength(Integer.parseInt(info[i+7][1]));
      i=i+8;
    }
    else if(info[i+3][1].equals("matrix"))
    {
      driverDatabase.put(index, new Matrix());
      driverDatabase.get(index).setledName(info[i][1]);
      driverDatabase.get(index).setDriverIP(info[i+1][1]);
      driverDatabase.get(index).setDriverPort(Integer.parseInt(info[i+2][1]));
      driverDatabase.get(index).setOther("other");
```

```
    driverDatabase.get(index).setClassType(CLASS_MATRIX);

    if(info[i+4][1].equals("mono"))
    {
      driverDatabase.get(index).setColorMode(COLOR_MONO);
    }
    else //if(info[i+4][1].equals("RGB"))
    {
      driverDatabase.get(index).setColorMode(COLOR_RGB);
    }
    ((Matrix)driverDatabase.get(index)).setWidth(Integer.parseInt(info[i+5][1]));
    ((Matrix)driverDatabase.get(index)).setHeight(Integer.parseInt(info[i+6][1]));
    i=i+7;
   }
   else //if(info[i+3][1].equals("raw"))
   {
    driverDatabase.put(index, new Raw());
    driverDatabase.get(index).setledName(info[i][1]);
    driverDatabase.get(index).setDriverIP(info[i+1][1]);
    driverDatabase.get(index).setDriverPort(Integer.parseInt(info[i+2][1]));
    driverDatabase.get(index).setOther("other");
    driverDatabase.get(index).setClassType(CLASS_RAW);

    if(info[i+4][1].equals("mono"))
    {
      driverDatabase.get(index).setColorMode(COLOR_MONO);
    }
    else //if(info[i+4][1].equals("RGB"))
    {
      driverDatabase.get(index).setColorMode(COLOR_RGB);
    }
    i=i+5;
   }

  index++;
  }
}

public abstract class LED
{
  private String name;
  private String driverIP;
  private int driverport;
  private String senderIP;
  private int senderport;
  private String otherdata;
  private byte class_type;
  private byte color_mode;
  public void setledName(String n)
  {
    name = n;
  }
  public void setDriverIP(String ip)
  {
    driverIP = ip;
  }
```

```
public void setDriverPort(int port)
{
  driverport = port;
}
public void setSenderIP(String ip)
{
  senderIP = ip;
}
public void setSenderPort(int port)
{
  senderport = port;
}
public void setOther(String o)
{
  otherdata = o;
}
public void setClassType(byte c)
{
  class_type = c;
}
public void setColorMode(byte color_m)
{
  color_mode = color_m;
}
public String getledName()
{
  return name;
}
public String getDriverIP()
{
  return driverIP;
}
public int getDriverPort()
{
  return driverport;
}
public String getSenderIP()
{
  return senderIP;
}
public int getSenderPort()
{
  return senderport;
}
public String getOther()
{
  return otherdata;
}
public byte getClassType()
{
  return class_type;
}
public byte getColorMode()
{
  return color_mode;
}
```

```
}

public class Strip extends LED
{
  private int strip_length;
  private byte type;
  private byte configuration;
  public void setLength(int len)
  {
    strip_length = len;
  }
  public void setType(byte t)
  {
    type = t;
  }
  public void setConfiguration(byte config)
  {
    configuration = config;
  }
  public int getLength()
  {
    return strip_length;
  }
  public byte getType()
  {
    return type;
  }
  public byte getConfiguration()
  {
    return configuration;
  }
}

public class Matrix extends LED
{
  private int matrix_width;
  private int matrix_height;
  public void setWidth(int w)
  {
    matrix_width = w;
  }
  public void setHeight(int h)
  {
    matrix_height = h;
  }
  public int getWidth()
  {
    return matrix_width;
  }
  public int getHeight()
  {
    return matrix_height;
  }
}

public class Raw extends LED
```

```
{

}

public void replacewithStrip()
{
  LED m = driverDatabase.get(dl1.selectedField);
  String name = m.getledName();
  String ip = m.getDriverIP();
  int port = m.getDriverPort();
  String other = m.getOther();
  byte color_m = m.getColorMode();

  driverDatabase.put(dl1.selectedField, new Strip());
  driverDatabase.get(dl1.selectedField).setledName(name);
  driverDatabase.get(dl1.selectedField).setDriverIP(ip);
  driverDatabase.get(dl1.selectedField).setDriverPort(port);
  driverDatabase.get(dl1.selectedField).setOther(other);
  driverDatabase.get(dl1.selectedField).setClassType(CLASS_STRIP);
  driverDatabase.get(dl1.selectedField).setColorMode(color_m);
}

public void replacewithMatrix()
{
  LED s = driverDatabase.get(dl1.selectedField);
  String name = s.getledName();
  String ip = s.getDriverIP();
  int port = s.getDriverPort();
  String other = s.getOther();
  byte color_m = s.getColorMode();

  driverDatabase.put(dl1.selectedField, new Matrix());
  driverDatabase.get(dl1.selectedField).setledName(name);
  driverDatabase.get(dl1.selectedField).setDriverIP(ip);
  driverDatabase.get(dl1.selectedField).setDriverPort(port);
  driverDatabase.get(dl1.selectedField).setOther(other);
  driverDatabase.get(dl1.selectedField).setClassType(CLASS_MATRIX);
  driverDatabase.get(dl1.selectedField).setColorMode(color_m);
}

public void replacewithRaw()
{
  LED s = driverDatabase.get(dl1.selectedField);
  String name = s.getledName();
  String ip = s.getDriverIP();
  int port = s.getDriverPort();
  String other = s.getOther();
  byte color_m = s.getColorMode();

  driverDatabase.put(dl1.selectedField, new Raw());
  driverDatabase.get(dl1.selectedField).setledName(name);
  driverDatabase.get(dl1.selectedField).setDriverIP(ip);
  driverDatabase.get(dl1.selectedField).setDriverPort(port);
  driverDatabase.get(dl1.selectedField).setOther(other);
  driverDatabase.get(dl1.selectedField).setClassType(CLASS_RAW);
  driverDatabase.get(dl1.selectedField).setColorMode(color_m);
```

```
}public abstract class LED
{
  private String name;
  private String driverIP;
  private int driverport;
  private String senderIP;
  private int senderport;
  private String otherdata;
  private byte class_type;
  private byte color_mode;
  public void setledName(String n)
  {
    name = n;
  }
  public void setDriverIP(String ip)
  {
    driverIP = ip;
  }
  public void setDriverPort(int port)
  {
    driverport = port;
  }
  public void setSenderIP(String ip)
  {
    senderIP = ip;
  }
  public void setSenderPort(int port)
  {
    senderport = port;
  }
  public void setOther(String o)
  {
    otherdata = o;
  }
  public void setClassType(byte c)
  {
    class_type = c;
  }
  public void setColorMode(byte color_m)
  {
    color_mode = color_m;
  }
  public String getledName()
  {
    return name;
  }
  public String getDriverIP()
  {
    return driverIP;
  }
  public int getDriverPort()
  {
    return driverport;
  }
  public String getSenderIP()
  {
```

```
    return senderIP;
  }
  public int getSenderPort()
  {
    return senderport;
  }
  public String getOther()
  {
    return otherdata;
  }
  public byte getClassType()
  {
    return class_type;
  }
  public byte getColorMode()
  {
    return color_mode;
  }
}

public class Strip extends LED
{
  private int strip_length;
  private byte type;
  private byte configuration;
  public void setLength(int len)
  {
    strip_length = len;
  }
  public void setType(byte t)
  {
    type = t;
  }
  public void setConfiguration(byte config)
  {
    configuration = config;
  }
  public int getLength()
  {
    return strip_length;
  }
  public byte getType()
  {
    return type;
  }
  public byte getConfiguration()
  {
    return configuration;
  }
}

public class Matrix extends LED
{
  private int matrix_width;
  private int matrix_height;
  public void setWidth(int w)
```

```
  {
    matrix_width = w;
  }
  public void setHeight(int h)
  {
    matrix_height = h;
  }
  public int getWidth()
  {
    return matrix_width;
  }
  public int getHeight()
  {
    return matrix_height;
  }
}

public class Raw extends LED
{

}

public void replacewithStrip()
{
  LED m = driverDatabase.get(dl1.selectedField);
  String name = m.getledName();
  String ip = m.getDriverIP();
  int port = m.getDriverPort();
  String other = m.getOther();
  byte color_m = m.getColorMode();

  driverDatabase.put(dl1.selectedField, new Strip());
  driverDatabase.get(dl1.selectedField).setledName(name);
  driverDatabase.get(dl1.selectedField).setDriverIP(ip);
  driverDatabase.get(dl1.selectedField).setDriverPort(port);
  driverDatabase.get(dl1.selectedField).setOther(other);
  driverDatabase.get(dl1.selectedField).setClassType(CLASS_STRIP);
  driverDatabase.get(dl1.selectedField).setColorMode(color_m);
}

public void replacewithMatrix()
{
  LED s = driverDatabase.get(dl1.selectedField);
  String name = s.getledName();
  String ip = s.getDriverIP();
  int port = s.getDriverPort();
  String other = s.getOther();
  byte color_m = s.getColorMode();

  driverDatabase.put(dl1.selectedField, new Matrix());
  driverDatabase.get(dl1.selectedField).setledName(name);
  driverDatabase.get(dl1.selectedField).setDriverIP(ip);
  driverDatabase.get(dl1.selectedField).setDriverPort(port);
  driverDatabase.get(dl1.selectedField).setOther(other);
  driverDatabase.get(dl1.selectedField).setClassType(CLASS_MATRIX);
  driverDatabase.get(dl1.selectedField).setColorMode(color_m);
```

```
}

public void replacewithRaw()
{
  LED s = driverDatabase.get(dl1.selectedField);
  String name = s.getledName();
  String ip = s.getDriverIP();
  int port = s.getDriverPort();
  String other = s.getOther();
  byte color_m = s.getColorMode();

  driverDatabase.put(dl1.selectedField, new Raw());
  driverDatabase.get(dl1.selectedField).setledName(name);
  driverDatabase.get(dl1.selectedField).setDriverIP(ip);
  driverDatabase.get(dl1.selectedField).setDriverPort(port);
  driverDatabase.get(dl1.selectedField).setOther(other);
  driverDatabase.get(dl1.selectedField).setClassType(CLASS_RAW);
  driverDatabase.get(dl1.selectedField).setColorMode(color_m);
}

StatusWindow statuswin;

public class StatusWindow
{
  public int connectionState;
  public StatusWindow()
  {
    cp5.addButton("CONNECT")
       .setBroadcast(false)
       .setValue(0)
       .setPosition(100,140)
       .setSize(150,19)
       .getCaptionLabel().align(CENTER,CENTER)
       ;

    cp5.addButton("DISCONNECT")
       .setBroadcast(false)
       .setValue(0)
       .setPosition(300,140)
       .setSize(150,19)
       .getCaptionLabel().align(CENTER,CENTER)
       ;

    cp5.getController("CONNECT").moveTo("STATUS");
    cp5.getController("DISCONNECT").moveTo("STATUS");
    cp5.getController("CONNECT").setBroadcast(true);
    cp5.getController("DISCONNECT").setBroadcast(true);
  }

  public void display()
  {
    noStroke();
    fill(220);
    rect(50, 100, 515, 200);
    if(connectionState == 1)
    {
```

```
      noStroke();
      textSize(32);
      fill(132,170,247);
      text("Connected to server", 100, 250);
    }
    else
    {
      noStroke();
      textSize(32);
      fill(132,170,247);
      text("Not connected to server", 100, 250);
    }
  }
}

public void setupStatusWindow()
{
  statuswin = new StatusWindow();
}

void checkstatusButton(ControlEvent theControlEvent)
{
  if(theControlEvent.controller().name()=="CONNECT")
  {
    connectToServer();
    requestDriversData();
    requestSendersData();
  }
  else if(theControlEvent.controller().name()=="DISCONNECT")
  {
    DisconnectServer();
  }
}
```

## Appendix 14: API test source code

\\psf\Home\Documents\Visual Studio 2013\Projects\LEDSenderCtrl_DLL\API test\API test.cpp 1

```
// API test.cpp : Defines the entry point for the console application.
// This test program displays three colors: RED, GREEN and BLUE.
// This program creates a pixel shifting effect.

#include "stdafx.h"
#include "LEDSenderCtrl_DLL.h"
#include <Windows.h>

using namespace std;

unsigned char* RED();
unsigned char* GREEN();
unsigned char* BLUE();

int main()
{
    API::Functions::SetMyAddress("192.168.1.27", 6500);
    API::Functions::SetServerAddress("192.168.1.27", 21234);
    API::StripChannelParameters scp;
    scp.Class = STRIP;
    scp.ChannelNumber = 1;
    scp.ColorMode = RGB;
    scp.Configuration = LINE;
    scp.StripType = ADDRESSABLE;
    scp.Length = 256;

    API::Functions::ConnectChannelToServer(&scp);
    int color_order = 0;
    unsigned char data[768] = { 0 };
    unsigned char* rgb;
    while (1)
    {

        for (int i = 0; i < (scp.Length * 3); i += 9)
        {
            if (color_order == 0)
            {
                rgb = RED();
            }
            else if (color_order == 1)
            {
                rgb = GREEN();
            }
            else //if (color_order == 2)
            {
                rgb = BLUE();
            }

            data[i] = *rgb;
            data[i + 1] = *++rgb;
            data[i + 2] = *++rgb;
        }
        for (int i = 3; i < (scp.Length * 3); i += 9)
        {
            if (color_order == 0)
            {
                rgb = GREEN();
            }
            else if (color_order == 1)
            {
                rgb = BLUE();
            }
            else //if (color_order == 2)
            {
                rgb = RED();
            }
```

\\psf\Home\Documents\Visual Studio 2013\Projects\LEDSenderCtrl_DLL\API test\API test.cpp 2

```
            data[i] = *rgb;
            data[i + 1] = *++rgb;
            data[i + 2] = *++rgb;
        }
        for (int i = 6; i < (scp.Length * 3); i += 9)
        {
            if (color_order == 0)
            {
                rgb = BLUE();
            }
            else if (color_order == 1)
            {
                rgb = RED();
            }
            else //if (color_order == 2)
            {
                rgb = GREEN();
            }

            data[i] = *rgb;
            data[i + 1] = *++rgb;
            data[i + 2] = *++rgb;
        }
        API::Functions::DrawAll(&scp, data);
        if (color_order < 2)
        {
            color_order++;
        }
        else color_order = 0;

        Sleep(500);
    }

    return 0;
}

unsigned char* RED()
{
    unsigned char rgb[3];
    rgb[0] = 0xFF;
    rgb[1] = 0x00;
    rgb[2] = 0x00;
    return rgb;
}

unsigned char* GREEN()
{
    unsigned char rgb[3];
    rgb[0] = 0x00;
    rgb[1] = 0xFF;
    rgb[2] = 0x00;
    return rgb;
}
unsigned char* BLUE()
{
    unsigned char rgb[3];
    rgb[0] = 0x00;
    rgb[1] = 0x00;
    rgb[2] = 0xFF;
    return rgb;
}
```

# Appendix 15: API library source code

\\psf\Home\Documents\Visual Studio 2013\Projects\LEDSenderCtrl_DLL\LEDSenderCtrl_DLL\LEDSenderCtrl_DLL.h 1

```
// LEDSenderCtrl_DLL.h

#ifdef LEDSENDERCTRL_DLL_EXPORTS
#define LEDSENDERCTRL_DLL_API __declspec(dllexport)
#else
#define LEDSENDERCTRL_DLL_API __declspec(dllimport)
#endif

#define _WINSOCK_DEPRECATED_NO_WARNINGS

enum led_t
{
    ADDRESSABLE,
    NONADDRESSABLE,
};

enum color_mode
{
    MONO,
    RGB,
};

enum config_t
{
    LINE,
    OTHER,
};

enum class_t
{
    STRIP,
    MATRIX,
};

namespace API
{
    class LEDSENDERCTRL_DLL_API ChannelParameters
    {
    public:
        virtual ~ChannelParameters();
        int ChannelNumber;
        color_mode ColorMode;
        class_t Class;
    };

    class LEDSENDERCTRL_DLL_API StripChannelParameters : public ChannelParameters
    {
    public:
        config_t Configuration;
        led_t StripType;
        int Length;
    };

    class LEDSENDERCTRL_DLL_API MatrixChannelParameters : public ChannelParameters
    {
    public:
        int Width;
        int Height;
    };

    class Functions
    {
    public:
        static LEDSENDERCTRL_DLL_API void SetServerAddress(char* IP, int port);
        static LEDSENDERCTRL_DLL_API void SetMyAddress(char* IP, int port);
        static LEDSENDERCTRL_DLL_API void ConnectChannelToServer(ChannelParameters* param);
```

\\psf\Home\Documents\Visual Studio 2013\Projects\LEDSenderCtrl_DLL\LEDSenderCtrl_DLL\LEDSenderCtrl_DLL.h 2

```
        static LEDSENDERCTRL_DLL_API void DisconnectChannelFromServer(const ChannelParameters& param);
        static LEDSENDERCTRL_DLL_API void DrawAll(ChannelParameters* param, unsigned char data[]);
        static LEDSENDERCTRL_DLL_API void DrawSingle(ChannelParameters* param, int index[], unsigned char
    data[]);
        static LEDSENDERCTRL_DLL_API void DrawMultiple(ChannelParameters* param, unsigned char pixels, int
    indexes[], unsigned char data[]);
    private:
        static void SendMessageToServer(unsigned char message[], int message_length);
    };
}
```

\\psf\Home\Documents\Visual Studio 2013\Projects\LEDSenderCtrl_DLL\LEDSenderCtrl_DLL\LEDSenderCtrl_DLL.cpp 1

```
// LEDSenderCtrl_DLL.cpp : Defines the exported functions for the DLL application.

#include "stdafx.h"
#include "LEDSenderCtrl_DLL.h"
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdlib.h>
#include <stdio.h>
#pragma comment(lib,"ws2_32.lib") //Winsock Library

#include <iostream>

using namespace std;

/* SENDERCMD definitions */
const unsigned char SENDER_CONNECT = 0x00;
const unsigned char SENDER_DISCONNECT = 0x02;
/* SENDERDATA data range identification */
const unsigned char ALL = 0x00;
const unsigned char SINGLE = 0x01;
const unsigned char MULTIPLE = 0x02;
/* LED object parameters */
const unsigned char CLASS_STRIP = 0x00;
const unsigned char CLASS_MATRIX = 0x01;
const unsigned char TYPE_ADDRESSABLE = 0x02;
const unsigned char TYPE_NONADDRESSABLE = 0x03;
const unsigned char CONFIG_LINE = 0x04;
const unsigned char CONFIG_OTHER = 0x05;
const unsigned char COLOR_MONO = 0x06;
const unsigned char COLOR_RGB = 0x07;

const char* MY_ADDR;
int MY_PORT;
const char* SERVER_ADDR;
int SERVER_PORT;

namespace API
{
    ChannelParameters::~ChannelParameters(){}

    void Functions::SetServerAddress(char* IP, int port)
    {
        SERVER_ADDR = IP;
        SERVER_PORT = port;
    }

    void Functions::SetMyAddress(char* IP, int port)
    {
        MY_ADDR = IP;
        MY_PORT = port;
    }

    void Functions::ConnectChannelToServer(ChannelParameters* param)
    {
        if (param->Class == STRIP)
        {
            StripChannelParameters* strip_param = dynamic_cast<StripChannelParameters*>(param);
            unsigned char message[16];
            message[0] = '*';
            message[1] = SENDER_CONNECT;
            message[2] = '/';
            message[3] = (strip_param->ChannelNumber >> 8) & 0xFF;
            message[4] = (strip_param->ChannelNumber & 0xFF);
            message[5] = '/';

            if (strip_param->ColorMode == MONO)
```

\\psf\Home\Documents\Visual Studio 2013\Projects\LEDSenderCtrl_DLL\LEDSenderCtrl_DLL\LEDSenderCtrl_DLL.cpp 2

```
            {
                message[6] = COLOR_MONO;
            }
            else //if (strip_param->ColorMode == RGB)
            {
                message[6] = COLOR_RGB;
            }

            message[7] = '/';
            message[8] = CLASS_STRIP;
            message[9] = '/';
            message[10] = (strip_param->Length >> 8) & 0xFF;
            message[11] = (strip_param->Length & 0xFF);
            message[12] = '/';

            if (strip_param->StripType == NONADDRESSABLE)
            {
                message[13] = TYPE_NONADDRESSABLE;
            }
            else //if (strip_param->StripType == ADDRESSABLE)
            {
                message[13] = TYPE_ADDRESSABLE;
            }

            message[14] = '/';

            if (strip_param->Configuration == LINE)
            {
                message[15] = CONFIG_LINE;
            }
            else //if (strip_param->StripType == OTHER)
            {
                message[15] = CONFIG_OTHER;
            }

            SendMessageToServer(message, 16);
        }
        else //if (param->Class == MATRIX)
        {
            MatrixChannelParameters* matrix_param = dynamic_cast<MatrixChannelParameters*>(param);
            unsigned char message[15];
            message[0] = '*';
            message[1] = SENDER_CONNECT;
            message[2] = '/';
            message[3] = (matrix_param->ChannelNumber >> 8) & 0xFF;
            message[4] = (matrix_param->ChannelNumber & 0xFF);
            message[5] = '/';

            if (matrix_param->ColorMode == MONO)
            {
                message[6] = COLOR_MONO;
            }
            else //if (matrix_param->ColorMode == RGB)
            {
                message[6] = COLOR_RGB;
            }

            message[7] = '/';
            message[8] = CLASS_MATRIX;
            message[9] = '/';
            message[10] = (matrix_param->Width >> 8) & 0xFF;
            message[11] = (matrix_param->Width & 0xFF);
            message[12] = '/';
            message[13] = (matrix_param->Height >> 8) & 0xFF;
            message[14] = (matrix_param->Height & 0xFF);
```

\\psf\Home\Documents\Visual Studio 2013\Projects\LEDSenderCtrl_DLL\LEDSenderCtrl_DLL\LEDSenderCtrl_DLL.cpp 3

```
            SendMessageToServer(message, 15);
        }
    }

    void Functions::DisconnectChannelFromServer(const ChannelParameters& param)
    {
        unsigned char message[5];
        message[0] = '*';
        message[1] = SENDER_DISCONNECT;
        message[2] = '/';
        message[3] = (param.ChannelNumber >> 8) & 0xFF;
        message[4] = param.ChannelNumber & 0xFF;

        SendMessageToServer(message, 5);
    }

    void Functions::DrawAll(ChannelParameters* param, unsigned char data[])
    {
        if (param->Class == STRIP)
        {
            StripChannelParameters* strip_param = dynamic_cast<StripChannelParameters*>(param);
            if (strip_param->StripType == ADDRESSABLE)
            {
                if (param->ColorMode == RGB)
                {
                    int message_length = strip_param->Length * 3 + 6;
                    unsigned char* message = new unsigned char[message_length];
                    int temp = 6;
                    message[0] = '>';
                    message[1] = (param->ChannelNumber >> 8) & 0xFF;
                    message[2] = param->ChannelNumber & 0xFF;
                    message[3] = '/';
                    message[4] = ALL;
                    message[5] = '/';
                    for (int it = 0; it < strip_param->Length*3; it++)
                    {
                        message[temp++] = data[it];
                    }

                    SendMessageToServer(message, message_length);
                    delete message;
                }
                else //if (param->ColorMode == MONO)
                {
                    int message_length = strip_param->Length + 6;
                    unsigned char* message = new unsigned char[message_length];
                    int temp = 6;
                    message[0] = '>';
                    message[1] = (param->ChannelNumber >> 8) & 0xFF;
                    message[2] = param->ChannelNumber & 0xFF;
                    message[3] = '/';
                    message[4] = ALL;
                    message[5] = '/';
                    for (int it = 0; it < strip_param->Length; it++)
                    {
                        message[temp++] = data[it];
                    }

                    SendMessageToServer(message, message_length);
                    delete message;
                }
            }
            else //if (strip_param->StripType == NONADDRESSABLE)
            {
                if (param->ColorMode == RGB)
                {
```

\\psf\Home\Documents\Visual Studio 2013\Projects\LEDSenderCtrl_DLL\LEDSenderCtrl_DLL\LEDSenderCtrl_DLL.cpp 4

```
                unsigned char message[7];
                message[0] = '>';
                message[1] = (param->ChannelNumber >> 8) & 0xFF;
                message[2] = param->ChannelNumber & 0xFF;
                message[3] = '/';
                message[4] = data[0];
                message[5] = data[1];
                message[6] = data[2];

                SendMessageToServer(message, 7);
            }
            else //if (param->ColorMode == MONO)
            {
                unsigned char message[5];
                message[0] = '>';
                message[1] = (param->ChannelNumber >> 8) & 0xFF;
                message[2] = param->ChannelNumber & 0xFF;
                message[3] = '/';
                message[4] = data[0];

                SendMessageToServer(message, 5);
            }
        }
    }
    else //if (param->Class == MATRIX)
    {
        MatrixChannelParameters* matrix_param = dynamic_cast<MatrixChannelParameters*>(param);
        if (param->ColorMode == RGB)
        {
            int message_length = matrix_param->Width * matrix_param->Height * 3 + 6;
            unsigned char* message = new unsigned char[message_length];
            int temp = 6;
            message[0] = '>';
            message[1] = (param->ChannelNumber >> 8) & 0xFF;
            message[2] = param->ChannelNumber & 0xFF;
            message[3] = '/';
            message[4] = ALL;
            message[5] = '/';
            for (int it = 0; it < matrix_param->Width * matrix_param->Height * 3; it++)
            {
                message[temp++] = data[it];
            }

            SendMessageToServer(message, message_length);
            delete message;
        }
        else //if (param->ColorMode == MONO)
        {
            int message_length = matrix_param->Width * matrix_param->Height + 6;
            unsigned char* message = new unsigned char[message_length];
            int temp = 6;
            message[0] = '>';
            message[1] = (param->ChannelNumber >> 8) & 0xFF;
            message[2] = param->ChannelNumber & 0xFF;
            message[3] = '/';
            message[4] = ALL;
            message[5] = '/';
            for (int it = 0; it < matrix_param->Width * matrix_param->Height; it++)
            {
                message[temp++] = data[it];
            }

            SendMessageToServer(message, message_length);
            delete message;
        }
    }
```

\\psf\Home\Documents\Visual Studio 2013\Projects\LEDSenderCtrl_DLL\LEDSenderCtrl_DLL\LEDSenderCtrl_DLL.cpp 5

```
}

void Functions::DrawSingle(ChannelParameters* param, int index[], unsigned char data[])
{
    if (param->Class == STRIP)
    {
        StripChannelParameters* strip_param = dynamic_cast<StripChannelParameters*>(param);
        if (strip_param->StripType == ADDRESSABLE)
        {
            if (param->ColorMode == RGB)
            {
                unsigned char message[12];
                message[0] = '>';
                message[1] = (param->ChannelNumber >> 8) & 0xFF;
                message[2] = param->ChannelNumber & 0xFF;
                message[3] = '/';
                message[4] = SINGLE;
                message[5] = '/';
                message[6] = (index[0] >> 8) & 0xFF;
                message[7] = index[0] & 0xFF;
                message[8] = '/';
                message[9] = data[0];
                message[10] = data[1];
                message[11] = data[2];

                SendMessageToServer(message, 12);
            }
            else //if (param->ColorMode == MONO)
            {
                unsigned char message[10];
                message[0] = '>';
                message[1] = (param->ChannelNumber >> 8) & 0xFF;
                message[2] = param->ChannelNumber & 0xFF;
                message[3] = '/';
                message[4] = SINGLE;
                message[5] = '/';
                message[6] = (index[0] >> 8) & 0xFF;
                message[7] = index[0] & 0xFF;
                message[8] = '/';
                message[9] = data[0];

                SendMessageToServer(message, 10);
            }
        }
    }
    else //if (param->Class == MATRIX)
    {
        MatrixChannelParameters* matrix_param = dynamic_cast<MatrixChannelParameters*>(param);
        if (param->ColorMode == RGB)
        {
            unsigned char message[15];
            message[0] = '>';
            message[1] = (param->ChannelNumber >> 8) & 0xFF;
            message[2] = param->ChannelNumber & 0xFF;
            message[3] = '/';
            message[4] = SINGLE;
            message[5] = '/';
            message[6] = (index[0] >> 8) & 0xFF;
            message[7] = index[0] & 0xFF;
            message[8] = '/';
            message[9] = (index[1] >> 8) & 0xFF;
            message[10] = index[1] & 0xFF;
            message[11] = '/';
            message[12] = data[0];
            message[13] = data[1];
            message[14] = data[2];
```

\\psf\Home\Documents\Visual Studio 2013\Projects\LEDSenderCtrl_DLL\LEDSenderCtrl_DLL\LEDSenderCtrl_DLL.cpp 7

```
                unsigned char* message = new unsigned char[message_length];
                int msgdata = (8 + pixels * 2);
                message[0] = '>';
                message[1] = (param->ChannelNumber >> 8) & 0xFF;
                message[2] = param->ChannelNumber & 0xFF;
                message[3] = '/';
                message[4] = MULTIPLE;
                message[5] = '/';
                message[6] = pixels;
                message[7] = '/';

                for (int it = 8; it < msgdata; it = it + 2)
                {
                    message[it] = (*indexes >> 8) & 0xFF;
                    message[it + 1] = *indexes++ & 0xFF;
                }

                message[msgdata] = '/';

                for (int it = msgdata + 1; it <(msgdata + 1 + pixels); it++)
                {
                    message[it] = *data++;
                }

                SendMessageToServer(message, message_length);
                delete message;
            }
        }
    }
    else //if (param->Class == MATRIX)
    {
        MatrixChannelParameters* matrix_param = dynamic_cast<MatrixChannelParameters*>(param);
        if (param->ColorMode == RGB)
        {
            int message_length = 9 + pixels * 7;
            unsigned char* message = new unsigned char[message_length];
            int RGBdata = (8 + pixels * 4);
            message[0] = '>';
            message[1] = (param->ChannelNumber >> 8) & 0xFF;
            message[2] = param->ChannelNumber & 0xFF;
            message[3] = '/';
            message[4] = MULTIPLE;
            message[5] = '/';
            message[6] = pixels;
            message[7] = '/';

            for (int it = 8; it < RGBdata; it = it + 2)
            {
                message[it] = (*indexes >> 8) & 0xFF;
                message[it + 1] = *indexes++ & 0xFF;
            }

            message[RGBdata] = '/';

            for (int it = RGBdata + 1; it <(RGBdata + 1 + pixels*3); it++)
            {
                message[it] = *data++;
            }

            SendMessageToServer(message, message_length);
            delete message;
        }
        else //if (param->ColorMode == MONO)
        {
            int message_length = 9 + sizeof(indexes) * 5;
            unsigned char* message = new unsigned char[message_length];
```

\\psf\Home\Documents\Visual Studio 2013\Projects\LEDSenderCtrl_DLL\LEDSenderCtrl_DLL\LEDSenderCtrl_DLL.cpp 8

```
                int msgdata = (8 + pixels * 4);
                message[0] = '>';
                message[1] = (param->ChannelNumber >> 8) & 0xFF;
                message[2] = param->ChannelNumber & 0xFF;
                message[3] = '/';
                message[4] = MULTIPLE;
                message[5] = '/';
                message[6] = pixels;
                message[7] = '/';

                for (int it = 8; it < msgdata; it = it + 2)
                {
                    message[it] = (*indexes >> 8) & 0xFF;
                    message[it + 1] = *indexes++ & 0xFF;
                }

                message[msgdata] = '/';

                for (int it = msgdata + 1; it <(msgdata + 1 + pixels); it++)
                {
                    message[it] = *data++;
                }

                SendMessageToServer(message, message_length);
                delete message;
            }
        }
    }

    void Functions::SendMessageToServer(unsigned char message[], int message_length)
    {
        static struct sockaddr_in myaddr;
        static struct sockaddr_in serveraddr;
        static SOCKET s = INVALID_SOCKET;
        static WSADATA wsa;

        if (s == INVALID_SOCKET)
        {
            //Initialise winsock
            printf("\nInitialising Winsock...");
            if (WSAStartup(MAKEWORD(2, 2), &wsa) != 0)
            {
                printf("Failed. Error Code : %d", WSAGetLastError());
                exit(EXIT_FAILURE);
            }
            printf("Initialised.\n");

            //Create a socket
            if ((s = socket(AF_INET, SOCK_DGRAM, 0)) == INVALID_SOCKET)
            {
                printf("Could not create socket : %d", WSAGetLastError());
            }
            printf("Socket created.\n");

            memset((char*)&myaddr, 0, sizeof(myaddr));
            myaddr.sin_family = AF_INET;
            myaddr.sin_addr.s_addr = inet_addr(MY_ADDR);
            myaddr.sin_port = htons(MY_PORT);

            //Bind
            if (bind(s, (struct sockaddr *)&myaddr, sizeof(myaddr)) == SOCKET_ERROR)
            {
                printf("Bind failed with error code : %d", WSAGetLastError());
                exit(EXIT_FAILURE);
            }
            puts("Bind done");
```

\\psf\Home\Documents\Visual Studio 2013\Projects\LEDSenderCtrl_DLL\LEDSenderCtrl_DLL\LEDSenderCtrl_DLL.cpp 9

```
            memset((char*)&serveraddr, 0, sizeof(serveraddr));
            serveraddr.sin_family = AF_INET;
            serveraddr.sin_addr.s_addr = inet_addr(SERVER_ADDR);
            serveraddr.sin_port = htons(SERVER_PORT);
        }

        if (sendto(s, reinterpret_cast<char*>(message), message_length, 0, (struct sockaddr*)&serveraddr,  ↙
    sizeof(serveraddr)) < 0)
        {
            perror("sendto failed");
        }
    }
}
```