Product report

on

NETWORKED LED DRIVER SYSTEM AND API

by

Winer Bao

11112514

Electrical & Electronics Engineering

The Hague University of Applied Sciences

Company: Tekt Industries Pty. Ltd.

Supervisor: Matthew Adams

May 26th 2015

Preface

This project is part of the graduation thesis for a final year electrical & electronics-engineering student. This is my third and last internship and is part of my curriculum at The Hague University of Applied Sciences. To conclude my studies, I have to successfully execute my graduation project and defend my thesis. During my studies, I tried to focus on sociocultural globalization as I think this experience/knowledge can be a personal benefit in the future. I decided to do my thesis in a foreign country, specifically an English-speaking country. During my search for the perfect company with my desired project, I found a company Tekt Industries Pty. Ltd located in Melbourne, Australia.

For five months from February 2015 until June 2015, I did an internship at Tekt Industries Pty. Ltd. I worked on an assignment project in which I create a server software in programming language C++ to interface with hardware prototypes the company designed and fabricated. This topic suits my future major in embedded systems. Through the assignment, I did not only gain a lot of knowledge but more importantly, I also had a great chance to sharpen my skills in a professional working environment. Not less important than the knowledge gain that I have learnt is the communication skills that I have been trained and practiced through giving presentations, discussing with the supervisors, experts in the field and other staffs within the company.

I am very appreciative to Mr. M. Adams; my supervisor at Tekt Industries Pty. Ltd. Matthew gave me very valuable instructions at critical moments during the project.

Throughout the internship, I have also learnt many things about circuitry design, PCB assembly and working in an English-speaking environment, whose benefits are far beyond what I could learn in a normal project. In short, I would like to thank Matthew and The Hague University of Applied Sciences for introducing me to this great opportunity in which I have developed myself both academically, professionally and socially.

May 26th 2015

Executive summary

The company, Tekt Industries, is currently developing two LED driver boards which can be connected to the network or by a USB connection. Each device connected to a local network has a unique IP address and a port number to where data can be sent. The user must remember the IP address and port number of a driver board to be able to send data to it. Remembering this information can be tedious.

The solution to the problem is to run a server between the user's program and the driver boards. This solution is recommended and requested by the project provider. The main purpose of the server is to provide a level of abstraction. The user only needs to remember the IP address and port number of the server to send data to any driver boards connected to the network.

A communication protocol is set up through which all programs can interact with the server software. The communication protocol is a UDP based protocol because of the time sensitivity of application. All messages start with a Source ID. This ID indicates the type of source and the type of message. This is followed by parameters, which are separated by forward-slashes. This simple, yet clear protocol is chosen because it is easily understandable to the user and the software.

The C++ server software is written for its functionality rather than speed, graphical User Interface (UI) and such. The software is also written with future modification and expansion in mind. The server software can be divided in multiple levels. Each level has its own purpose and the structure of the server is depend on each one of the levels. Some level handles the message reception and delivery while others create object to store channel settings.

A total of seven test and sample programs are created to test the performance of the server. Each sender program sends LED color data. Each data output varies slightly in types of data format. The server handled multiple data message and commands without any interruption and/ or problems. The output was able to display on actual LED hardware as well as on the driver simulation program.

An API library is written to create a simple environment which software developers can use without diving deeply into the communication protocol, socket setup and other complex processes.

Contents

Appendices

Pictures

Figures

Tables

List of Abbreviations

LED	light emitting diode
LANs	local area networks
PCBs	printed circuit boards
CAD	computer aided design
DSP	digital signal processing

VHDL	VHSIC Hardware Description Language
FPGA	field-programmable gate arrays
CPLDs	complex programmable logic devices
USB	universal serial bus
UDP	User Datagram Protocol
CPU	central processing unit
MCU	memory control unit
API	application programming interface
OS	operating system
UI	user interface
IDE	integrated development environment

1. Introduction

Light emitting diode (LED) strip lights are becoming more and more popular and it is understandable why – they provide an endless source of practical light and are aesthetically appealing in interior and outdoor design. They offer plenty of home décor possibilities due to their wide range of applications and designs. (Dési, 2011)

LED lighting is not only seen in homes but also galleries, theaters, venues and many more places use them to enhance or create a unique visual experience. The gadgets are only possible thanks to advancements in LED technologies and computer sciences. New researches and advancements in LED technologies lowered production costs of LED lighting and increased their efficiency by a substantial amount. As prices of LEDs falls and options abound, LED products are becoming more accessible to the general consumer. (Tweed, 2013) LEDs are very efficient compared to other types of light sources, can have practically any color you want and have a long lifetime. This, along with many other advantages, makes LED lighting almost a no-brainer. These LEDs combined with software can create spectacular displays and décor.

Computers process data under control of sets of instructions called computer programs. These computer programs that run on a computer are also referred to as software. High-level languages were developed to speed up the programming process. C and C++ are among the most powerful and most widely used high-level programming languages. As computers became more powerful, it was thought that many tasks could be made to share the resources of the computer to achieve better utilization. This is called multiprogramming. These machines could in turn be linked together in computer networks, such as in Local Area Networks (LANs), to increase its total performance. This led to the phenomenon of distributed computing, where different portions of a task can be performed on different computers. Information is easily shared across computer networks where some computers called servers can store programs and data that may be used by client computers distributed throughout the network, hence the term client/server computing. C and C++ have become the programming language of choice for writing software for operating systems, for computer networking and for distributed client/server applications. (Deitel, 1997)

Tekt Industries Pty. Ltd, provider of this project, is currently developing two LED driver boards capable of controlling multiple LED strips and/or LED matrixes. The driven LED can be of any type, color, length, dimensions and can be in any position or location. This offer limitless combinations and opens the door to many interesting applications. The small size of the driver boards makes it easy to embed them in e.g. furniture. Data is send by programs created by the user to a server/hub. The server decodes the data and sends it to the individual driver boards.

The purpose of this report is to document the technical details of the software created in this project. The design and the implementation of the software are carefully explained in detail. Nonetheless, the design idea is brought to light in this report.

Chapter 2 gives a brief background of the company and the project. Chapter 3 defines the problem which this project is trying to solve. The requirement, project boundaries and deliverables are also stated in this chapter. A broad description of the server software is presented in chapter 4. Chapter 5 will dissect the project into key components and each component is described and explained in details. The testing and outcome of the product development is documented in chapter 6. A conclusion is made and some recommendations are given in chapter 7 and 8.

2. Project background

Tekt Industries Pty. Ltd. is a company located in Melbourne, Australia. It is a small and healthy company offering engineering services like circuit design, embedded programming, test and measurement and prototype fabrication to its clients. To get a better understanding of the context of this project, brief background information about the company is given in this chapter.

2.1. Company profile

Tekt Industries Pty. Ltd. has been active in the area of PCB design, embedded programming, test and measurement and prototype fabrication. The company worked with clients in many fields from entertainment to healthcare. They have clients such as Intel Australia & Sensing City, Melbourne Zoo and RMIT – School of Fashion and Textiles.

They have a dedicated team specializing in the design and layout of rigid and flexible printed circuit boards (PCBs) which incorporate a range of technologies such as analog filters and amplifiers, low data rate digital RF transceiver implementations, and embedded microcontroller systems. They use a complete suite of computer aided design (CAD) software including Solidworks 2015 and Altium Designer 15 so that any design, which involves tight integration between complex mechanical and electrical sub-assemblies, can be developed with speed and confidence. (Tekt Industries Pty. Ltd., 2015)

No embedded system is complete without the software and so they develop using standard C/C++ programming languages alongside C# and even a bit of Processing/Java to implement application specific functionality such as control, digital signal processing (DSP), and sensing. They have worked with different device brands such as Microchip, Atmel, Freescale, ARM and STMicroelectronics. This does not mean that a bit of VHSIC Hardware Description Language (VHDL) on Xilinx, Altera, or MicroSemi field-programmable gate arrays (FPGAs) and complex programmable logic devices (CPLDs) is out of the question. (Tekt Industries Pty. Ltd., 2015)

Tekt Industries has invested heavily in test equipment such as high-resolution thermal imaging cameras (FLIR), state of the art Oscilloscopes (Rhode & Schwarz), Spectrum Analyzers (Agilent), and specialist audio test equipment (Audio Precision). This equipment allows them to see deeper into any design during prototyping and provide their clients with better metrics to gauge design performance. It also helps then prepare their client's product for both local and overseas compliance testing so that this process may be streamlined. (Tekt Industries Pty. Ltd., 2015)

Initial PCB prototypes can often be assembled in-house to accelerate the design validation process and gain further understanding of assembly constrains. They also routinely organize manufacture and assembly of PCBs through local and overseas providers for more complex designs and higher volume orders. With an in-house CNC laser cutter, 3D printer, and CNC milling machine, small proof of concept prototypes can be fabricated with much faster turnaround than many offshore fabricators depending on desired finish. (Tekt Industries Pty. Ltd., 2015)

2.2. Company brands

Apart from providing engineering services to contractors, Tekt Industries also has its own product line. The company recently launched the Tektyte brand.

“Founded in December 2014, Tektyte focuses on creating high quality tools and development platforms for electronics professionals and enthusiasts.” (Tekt Industries Pty. Ltd., 2015)

Kickstarter is place where backers can pledge their money to fund creative projects. It is home for everything from films, games, and music to art, design and technology. (Kickstarter Inc., n.d.)

With its recent successfully funded kickstarter project for a circuit tester called the LogIt, the company is looking forward to expand its product line by creating new development boards and tools. One of the way the company is trying to make its yet-to-be-released product line stand out from the crowd by using a unique hexagonal PCB shape.

Two of such development boards are network connected or Universal Serial Bus (USB) connected LED driver board (LED-PR-001 and LED-LDR-001), each with its specific application and features. These two boards are called: LED Point Runner and LED Line Driver. The LED Point Runner LED-PR-001 board has 40 channels, capable of driving up to 1000 LEDs per channel. The board can drive LED strips and/or LED matrixes and each LED are individually addressable. The color data is sent as 24-bit value; each LED can therefore display 16777216 different color. The brain of the system is an Altera Max 10 FPGA, which can interpret USB data and decode them to drive the LEDs connected to its channels. The device can be connected to a computer via the micro-USB to receive data from the server. An Intel Edison (optional) can be plugged in to act as an User Datagram Protocol (UDP) to USB bridge. It will receive data sent by the server over a wireless connection, convert it to USB data and sends it to the FPGA. The Intel Edison is a small, powerful computer with high performance, dual-core central processing unit (CPU) and a single core micro-controller to support complex data collections in a low power package. (Intel Corporation, n.d.)

The LED Line Driver LED-LDR-001 has 16 channels, capable of driving up to 1000 LEDs per channel. The board can drive any non-addressable LED strips and it is connected to a network via the Ethernet connection to receive data from the server. The color data is sent as 24-bit value. The brain of the system is an Atmel SAM4E Memory Control Unit (MCU), which can interpret UDP data and decode them to drive the LEDs connected to its channels. An Intel Edison (optional) is onboard to act as a wireless UDP receiver. It will receive data sent by the server over a wireless connection and redirects it to the MCU. Figure 2.1 and 2.2 shows the two Tektyte driver boards that the company designed.

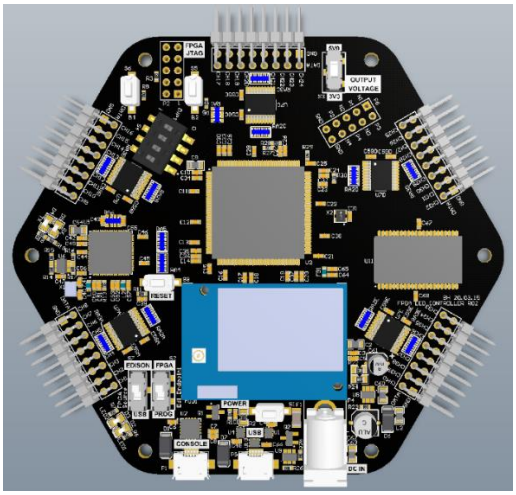


Figure 2.1 LED Point Runner

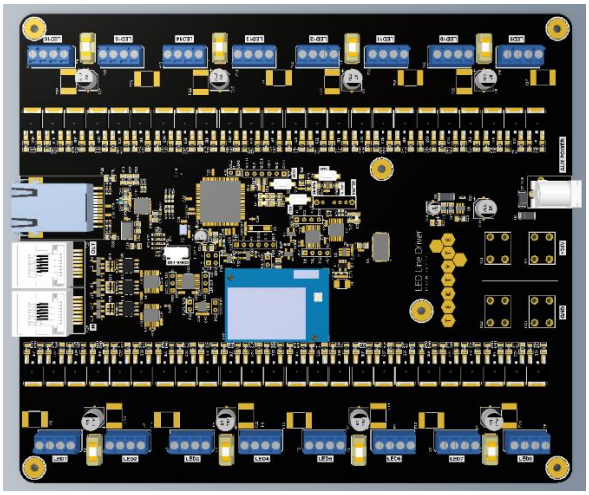


Figure 2.2 LED Line Driver

These driver boards are still in development phase. Therefore, the embedded software for these boards are limited in capabilities and the application programming interface (API) is still evolving. The Point Runner boards can only receive one command (set length) and receive data designed for LED strips. At the time of writing, the Point Runner cannot receive any data by the UDP protocol/ internet network. The only input port is the USB interface of the device. The Line Driver boards receive data by hard-wire Ethernet using an old data protocol. The data protocol is bare-bone and designed for minimal to none data manipulation on the microcontroller. The company wishes to change this protocol in the future.

3. Project definition

The Tektyte Point Runner board connects to a computer via a USB connection to receive color data. This color data is needed to display colors on the LEDs connected to its channel port. When an Intel Edison is plugged in the board, the Edison can process UDP packets received via the integrated Wi-Fi and feed this data to the USB data line connected to the FPGA. The Line Driver board connects only via an Ethernet port to a local network. Thus, the color data is sent via an internet protocol. To make the driver board truly wireless (in terms of data connection), an Intel Edison can be added to utilize its Wi-Fi capability.

3.1. Problem statement

Each device connected to a local network has a unique IP address and a port number to where data can be sent. The user must remember the IP address and port number of a driver board to be able to send data to it. Remembering this information can be tedious. However, if the user has multiple boards deployed in the network, it is almost impossible to remember the IP addresses and port numbers of all the different boards. The company does not have a solution to this problem yet.

3.2. Project requirements and boundaries

The start date of this project is on February 9th 2015. The project assignment lasts 17 weeks and ends on June 5th 2015.

The project executor is only working on the software development. The hardware of the prototype is in charge of the client and any changes in the hardware are reported to the project executor.

The solution must be written in C++ programming language and run on a Windows operating system (OS). Before creating the solution, a communication protocols must be defined. The solution must be able to handle incoming data packets from multiple senders. Important information is stored in the server for future use and the data is then processed and forwarded to the correct driver boards. An administrator program connects to the solution to view and change various properties of the solution. The solution is able to send data to the Point Runner driver board by USB or by the UDP protocol. Creating an API is also part of the project. The API is written in C++ and its main purpose is provide an easier and faster programming environment for the end-user. Some sample programs are created in Processing and also serve as programs to test the solution's performance.

Processing is a programming language, development environment, and online community. Since 2001, Processing has promoted software literacy within the visual arts and visual literacy within technology. Initially created to serve as a software sketchbook and to teach computer programming fundamentals within a visual context, Processing evolved into a development tool for professionals. Today, there are tens of thousands of students, artists, designers, researchers, and hobbyists who use Processing for learning, prototyping, and production. (Processing Foundation, n.d.)

3.3. Project goal and deliverables

The goal of this project is to research and implement a solution that meets the project requirements. During 17 weeks, a software is created to interface with the LED driver boards, which implements various features requested by the project provider. At the end of the project, the intern submits a final report of the project product. In this report all technical details of the product are documented.

The deliverables of the project are:

1 Solution communication protocol documentation
2 Solution software Visual Studio code
3 API library Visual Studio code
4 Test/ Sample sender programs Processing code
5 Test/ Sample driver program Processing code
6 Test/ Sample administrator program Processing code
7 Final product report

4. Basic of design

The solution to the problem is to run a server between the user's program and the driver boards. This solution is recommended and requested by the project provider. The main purpose of the server is to provide a level of abstraction. The user sends data to the server and the server processes and routes the data to the correct address of the driver boards. This is called network address translation. The user only needs to remember the IP address and port number of the server to send data to any driver boards connected to the network. For the user, this means a cleaner and easier interface to work with. The server also decodes and checks the data send by the user for any error. Therefore, it acts like a data format error filter. An API is created to further simplify the interface with the server. The API hides the protocol between the user program and the server software and provides a unified data handling. Figure 4.1 shows the visual representation of the Network LED Driver System.

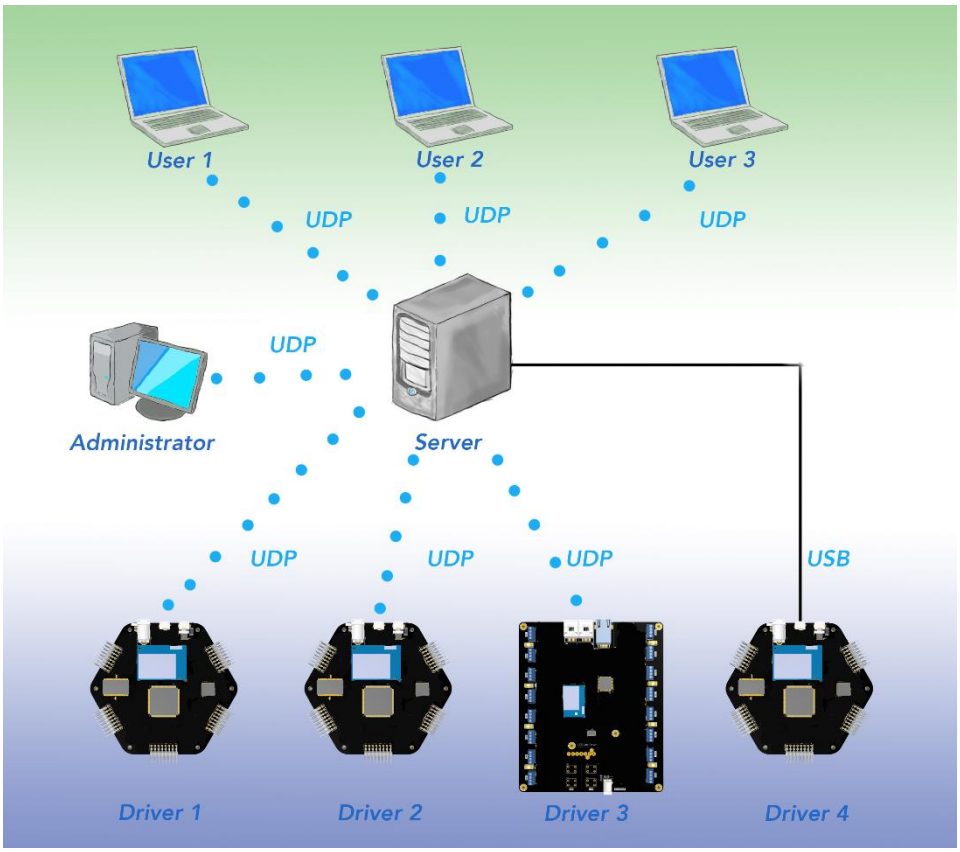


Figure 4.1 Networked LED Driver System overview

The server makes use of the thread library to benefit from the multithreading capabilities. Multiple threads can run simultaneously to increase the server's performance. For example, the server can process messages while it waits for the other messages to send.

The server activities can be divided into 3 categories/ phases: Receive, Process and Send. (Figure 4.2)

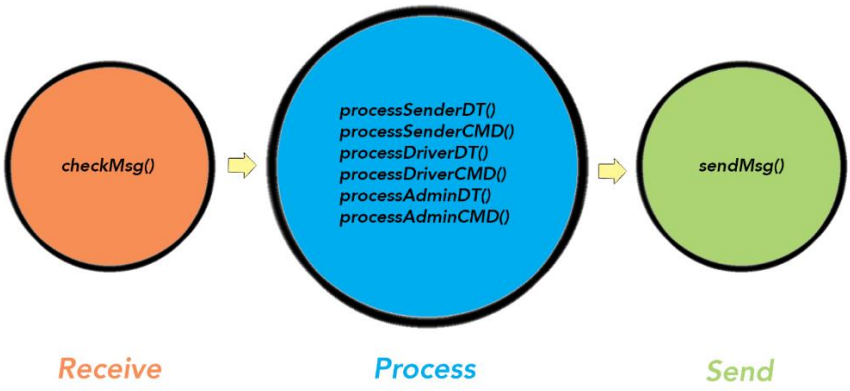


Figure 4.2 Server phases

The first phase Receive waits until an UDP message arrives at the server socket. When a message is received, it sorts and adds the messages safely in the appropriate inbox. The messages are sorted based on the send source and message type. The message can be send by a user program (Sender), a driver program (Driver) or an administrator program (Admin). Each program can send two types of messages: data or command.

The second phase Process contains multiple inboxes. A message is removed from the inbox when Process thread becomes available. The message is then processed according to a predetermined method. The processing method varies depending on the type of message, the send source and the instruction of the message. A simplified and universal method to process a data message type can be seen in figure 4.3. The function checks and waits for notification that new messages are available. When the function is notified to continue, it will process and decode the message. If the driver board is connected by an Internet protocol, the decoded message is added to the outbox. If the outbox is already in use, the function will wait for its turn. After the function added the message to the outbox, it will start from the beginning again. If the driver board is connected by a USB connection, the decoded message is sent forwarded to a USB driver program.

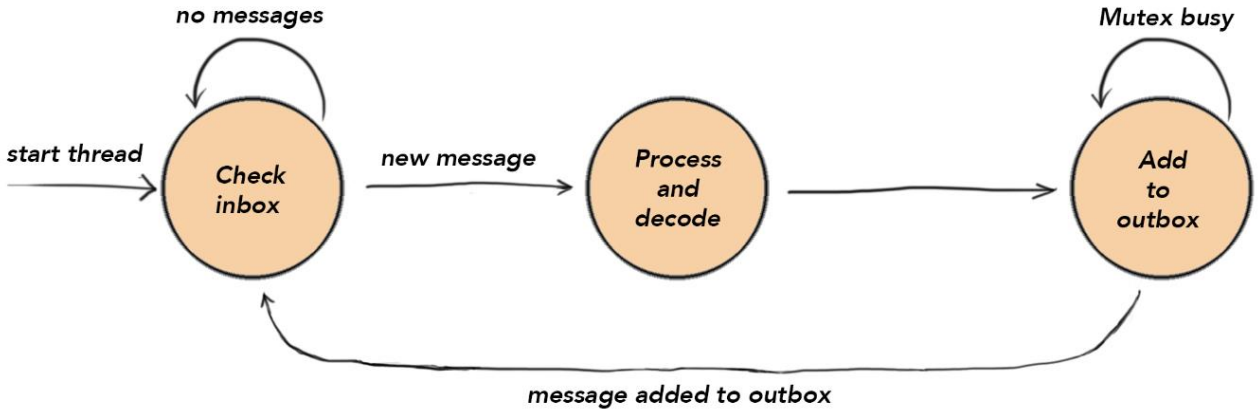


Figure 4.3 Data processing overview

The processing of a command message type looks similar to the one from a data type except no message is added to the outbox. A command often gives an instruction to change one or more configuration of the server. Figure 4.4 illustrates a simplified and universal method to process a command message type.

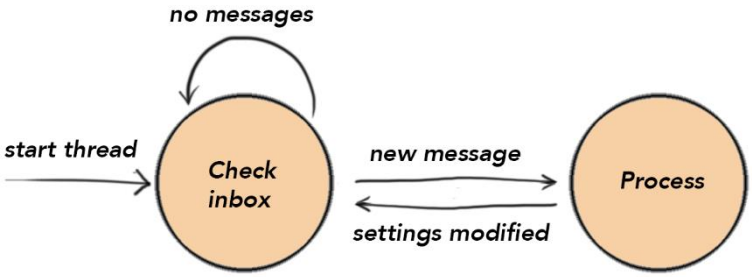


Figure 4.4 Command processing overview

The newly received message is always checked for the correct format before being decoded. An error message is displayed when it fails the test. After successfully passing this test, the message is processed accordingly.

The third phase Send removes a message from the outbox and sends the message to the correspondent using UDP or USB. If there is no message available in the outbox, the function will wait until notified to continue.

Two sender sample programs are created using Processing. These serve as programs to tests the proper working of the server software.

Processing is a programming language, development environment, and online community. Since 2001, Processing has promoted software literacy within the visual arts and visual literacy within technology. Initially created to serve as a software sketchbook and to teach computer programming fundamentals within a visual context, Processing evolved into a development tool for professionals. Today, there are tens of thousands of students, artists, designers, researchers, and hobbyists who use Processing for learning, prototyping, and production. (Processing Foundation, n.d.)

The first sample software gathers a video stream of an webcam and down samples it to a lower resolution. Each row is sent to the server as a strip. Before any data can be sent to the server, the user must connect to the server To stop the data stream, the user disconnect the program from the server.

The second sample program the user can control each LED pixel individually. The color of the pixel can be set by selecting a LED pixel and turning knobs to change its red, green and blue values. The amount of channel and pixels per channel can be changed in the program. The program must connected to the server before sending data. When the program is connected to the server the indicator on the top right changes from black to green and vice versa.

The driver sample program is created to visualize the output of the server software This program simulates the functions of a driver board and LED strips. It takes individual channel information and displays it on screen in circular dots.

The server administrator can use the administrator program to review and change channel information. The program has four tabs. The senders tab shows all senders currently registered on the server network. Their IP, port number, name and such is shown in a list. The user can click on a channel to show more information about a channel. In the drivers tab the user can see all the drivers registered to the server network. The user can change some channel settings in this tab.

When a channel linked to a strip object is selected, the user can change the class type to matrix, color mode, LED type, configuration and the length of the object. The length of a strip is the amount of LED pixels on the strip. When a channel linked to a matrix object is selected, the user can change the class type to strip, color mode, the width and the height of the object. The width of a matrix is the amount of LED pixel on the X-axis of the matrix. The height of a matrix is the amount of LED pixel on the Y-axis of the matrix. A channel can also be linked to a raw object. This object contains no changeable parameters and is intended for compatibility with previously created sender programs.

5. Product development

Tekt Industries Pty. Ltd. is developing two LED driver boards with model number LED-PR-001 and LED-LDR-001. The intern is not responsible for the hardware design/ manufacturing and embedded software of these boards. The intern is only responsible for creating the server software and sample programs. In this chapter, the communication protocol, the server software, test/sample software and the API library are explained in detail.

5.1. Communication protocol

Before the server software can be created, a communication protocol must be set. The driver boards are primarily intended for art displays. A UDP based protocol is chosen because of the time sensitivity of application. Time-sensitive applications often use UDP because dropping packets is preferable to waiting for delayed packets.

A simple, yet clear protocol is chosen that the user and the software can easily understand. Figure 5.1 shows the general format of the communication protocol.

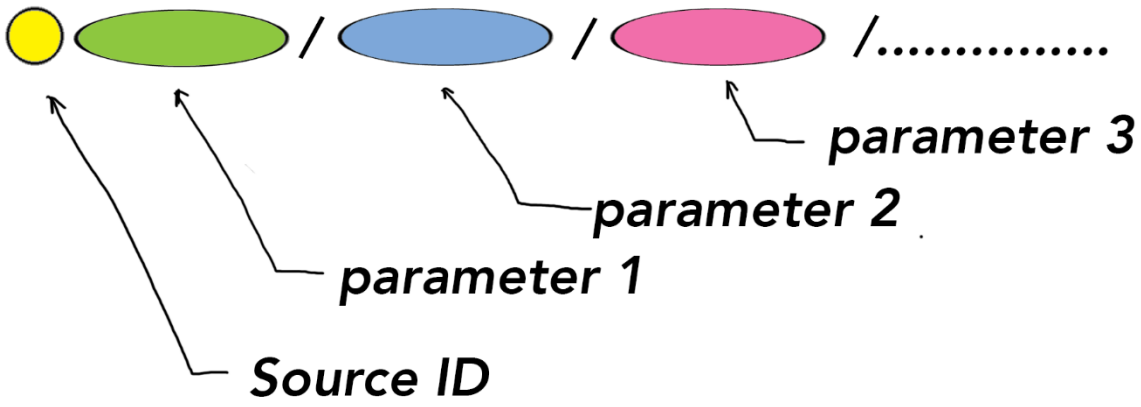


Figure 5.1 General communication protocol format

All messages start with a Source ID. This ID indicates the type of source and the type of message. For example, a type of source can be a sender and the type of the message can be a command; the source ID of this message is “*” (see Table 5.1).

Sender data	>
Sender command	*
Driver data	/
Driver command	#
Administrator data	&
Administrator command	@

Table 5.1 Communication protocol - Source IDs

After the source ID comes the first parameter. Parameters are separated by a forward-slash symbol (“/”). The content of the parameters varies and can be one or two bytes long. Command messages generally contain settings information of an channel while

data message generally contain a channel number and color data. The communication protocol is documented and can be found in Appendix 1.

5.2. Server software

It is important to note that this is a first generation software. That means the software is written for its functionality rather than speed, graphical user interface (UI) and such. The software is also written with future modification and expansion in mind. The integrated development environment (IDE) used in this project is Microsoft Visual Studio Express 2013 for Windows Desktop Version 12.0.31101.00 Update 4. The entire server software is written in C++ programming language.

As programs grow larger and larger (and include more files), it becomes increasingly tedious to have to forward declare every function you want to use that lives in a different file. Wouldn't it be nice if you could put all your declarations in one place? C++ code files (with a .cpp extension) are not the only files commonly seen in C++ programs. The other type of file is called a header file, sometimes known as an include file. Header files usually have a .h extension, but you will sometimes see them with a .hpp extension or no extension at all. The purpose of a header file is to hold declarations for other files to use. (Alex, 2007)

The software is split into seven header files and six cpp files: Definitions.h, main.cpp, MessageHandler.h, MessageHandler.cpp, Database.h, Database.cpp, Channel-Object.h, ChannelObject.cpp, USBdriver.h, USBdriver.cpp, stdafx.h, stdafx.cpp and targetver.h. The header file stdafx.h, cpp file stdafx.cpp and header file targetver.h are needed for proper working in Windows and do not contribute any significance to the features of the server. These files are automatically generated by the IDE. They are generic code and are, therefore, not explained.

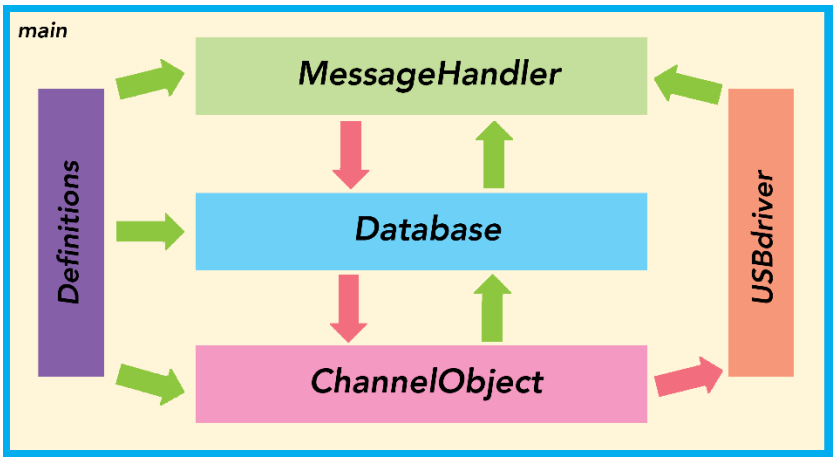


Figure 5.2 Server structure and dataflow

Figure 5.1 shows the structure and dataflow of the server. Pink arrows show when a function is called and the green arrows show that a value is returned. Functions and/or terms from the lower level files are often used in the higher level files. The Definition header files are used by all other header and cpp files. Based on how the different files interact with one another, it is best to start the explanation at the Definition header file.

From there, the next lowest level file (ChannelObject) is explained next. Going up the level, Database, MessageHandler, USBdriver and main are explained.

5.2.1. Definitions header file

This header file declares many constants used throughout the entire program.

```
const int TOTAL_SENDERDATAPROCESSINGTHREADS        = 50;
const int TOTAL_SENDERCOMMANDPROCESSINGTHREADS     = 1;
const int TOTAL_DRIVERDATAPROCESSINGTHREADS        = 1;
const int TOTAL_DRIVERCOMMANDPROCESSINGTHREADS     = 1;
const int TOTAL_ADMINDATAPROCESSINGTHREADS         = 1;
const int TOTAL_ADMINCOMMANDPROCESSINGTHREADS      = 1;
```

The above shown constants are used to specify the amount of threads to be started at the beginning of the program. There are six different functions running in parallel. Multiple threads of the function that processes sender data are created to handle the large amount of data coming into the program by UDP.

```
static const char* SERVER_ADDR = "127.0.0.1";
const int SERVER_PORT = 21234;
```

The above shown constants declare the IP address and port number of the server.

```
const char SENDERDATA                  = '>';
const char SENDERCMD                   = '*';
const char DRIVERDATA                  = '/';
const char DRIVERCMD                   = '#';
const char ADMINDATA                   = '&';
const char ADMINCMD                    = '@';
const unsigned char ALL                = 0x00;
const unsigned char SINGLE             = 0x01;
const unsigned char MULTIPLE           = 0x02;

/* SENDERCMD definitions */
const unsigned char SENDER_CONNECT     = 0x00;
const unsigned char SENDER_DISCONNECT  = 0x02;

const unsigned char DRIVER_CONNECT     = 0x00;
const unsigned char DRIVER_DISCONNECT  = 0x01;

const unsigned char REQUEST_SENDERDATA = 0x02;
const unsigned char REQUEST_DRIVERDATA = 0x03;

const unsigned char ADMIN_CONNECT      = 0x00;
const unsigned char ADMIN_DISCONNECT   = 0x01;
const unsigned char CHANNELMOD         = 0x04;
const unsigned char SET_LENGTH         = 0x04;
```

```
const unsigned char GET_LENGTH                  = 0x05;

const unsigned char CLASS_STRIP                 = 0x00;
const unsigned char CLASS_MATRIX                = 0x01;
const unsigned char CLASS_RAW                   = 0x03;
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;
```

The above shown constants are used to specify the source and type of the message. These symbols are set in the communication protocol.

```
enum led_t
{
        ADDRESSABLE,
        NONADDRESSABLE,
};

enum color_mode

{
        MONO,
        RGB,
};

enum config_t
{
        LINE,
        OTHER,
};

enum class_t
{
        STRIP,
        MATRIX,
        RAW,
};

enum connection_t
{
        USB,
        UDP,
};
```

These five enumeration declaration (shown above) are defined to restrict to one of several explicitly named constants. This way, input error can be limited significantly. New constants are easily added if needed in future iterations of the server software.

extern int ByteToInt(unsigned char MSB, unsigned char LSB);

This declares a helper function used often in the program. It converts two bytes into a 16-bit integer.

5.2.2. ChannelObject header and cpp file

The ChannelObject header file declares twelve classes and its member functions and attributes. Each class has functions that saves and returns a value of one of the attributes important to the proper functioning of the server.

A struct is defined of custom exceptions. When a “MyException” is thrown, it prints the error message on the console.

Figure 5.3 shows a detailed hierarchy of the different classes. This hierarchy can be easily expanded in the future. Using classes, inheritance and polymorphism, complex software with great deal of flexibility can be achieved. More flexibility means a greater degree of freedom to create a logical, highly upgradable and easy-to-understand software. All classes contain private attributes which can be changed using public member functions. This is called data encapsulation. It creates a great level of data abstraction. Class private attributes are protected from inadvertent user-level errors, which might corrupt the state of the object. The levels of classes are created to give the future developer of the server more control over which objects he/she wishes to modify. For example: if the developer wishes control all Strip objects, he/she can call a virtual function of all Strip objects. Thanks to polymorphism, the program will execute the correct function of any derived object from Strip.

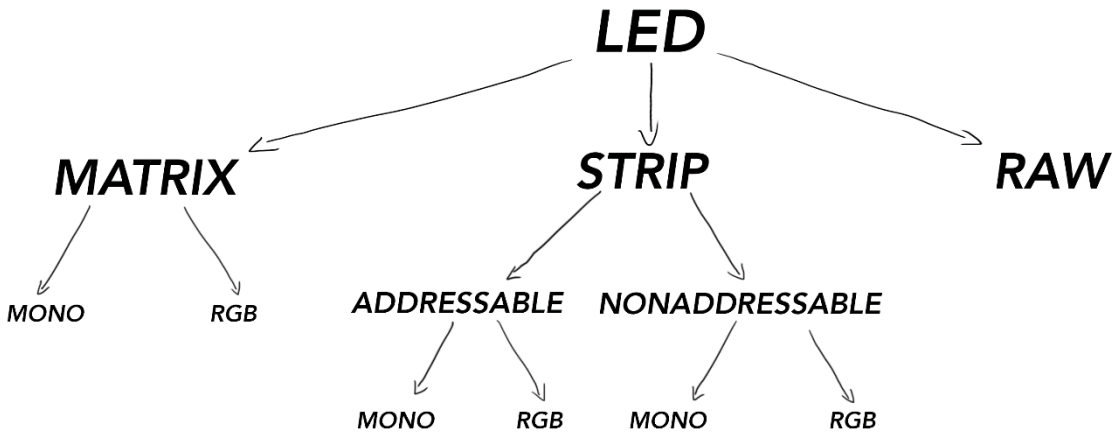


Figure 5.3 Class hierarchy

Class LED is an abstract base class and stores basic information all of its derived classes must contain. The SET-functions takes call by reference arguments to avoid

unnecessary coping of data. The GET-functions are declared as constant to prevent any modification to the private variables.

The LED class has ten private variables:

1. ***char* channelName***
2. ***int external_channel***
3. ***int internal_channel***
4. ***color_mode color***
5. ***int group***
6. ***class_t classType***
7. ***std::string DriverIP***
8. ***int DriverPort***
9. ***std::string SenderIP***
10. ***int SenderPort***
11. ***connection_t connectType***

These private attributes are self-explanatory and can be changed or returned with the public member functions in its class. The variable “internal_channel” is the channel number the driver uses to identify its channel. The variable “external_channel” is the channel number by which the sender can access the object. Each external channel number is unique and mapped to the internal channel number.

The LED class has twenty-three public member functions:

1. ***LED(const char* name, const int& Ext_ch, const int& Int_ch, const color_mode& c, const class_t& ct)***

   This is the constructor of this class. The first argument is a char pointer to a string. The string contains the name associated to the new created LED object. The name can be used by the user to identify the channel faster. The second argument is the external channel number coupled to the object. This number is also used for reference purposes for the sender. The third argument is the internal channel number. The fourth argument holds the color mode of the objects. The most common LEDs can either display one color or a RGB color. The final argument declares the LED object to be a certain class. Some LEDs come in strip or matrix form.

2. ***virtual ~LED()***

   This is the default destructor of this class. It is declare virtual to benefit of the polymorphism capabilities of the C++ language. The function deletes the memory content pointer by the name pointer before the object destruction to prevent memory leaks.

3. ***void setExternnalChannel(const int& ch)***

   changes the private variable external_ channel to the value of argument “ch”.

4. ***void setInternalChannel(const int& ch)***

   changes the private variable internal_ channel to the value of argument “ch”.

5. ***void setColorMode(const color_mode& c)***

   changes the private variable color to the value of argument “c”.

6. ***void setGroup(const int& g)***

   changes the private variable group to the value of argument “g”.

7. ***void setClassType(const class_t& ct)***

   changes the private variable classType to the value of argument “ct”.

8. ***void setDriverInfo(const std::string& IP, const int& port)***

   changes the private variable DriverIP and DriverPort to the value of argument “IP” and “port” respectively.

9. ***void setSenderInfo(const std::string& IP, const int& port)***

   changes the private variable SenderIP and SenderPort to the value of argument “IP” and “port” respectively.

10. ***void setConnectionType(const connection_t& con_t)***

    changes the private variable connectType to the value of argument “con_t”.

11. ***char* getName() const***

    returns a char pointer of the memory where the name of the channel is stored.

12. ***int getChannel() const***

    returns the channel number of the object.

13. ***color_mode getColorMode() const***

    returns the color mode of the object.

14. ***int getGroup() const***

    returns the group number of the object.

15. ***class_t getClassType() const***

    returns the class of the object.

16. ***std::string getDriverIP() const***

    returns the IP address of the driver board associated to this object.

17. ***int getDriverPort() const***

    returns the port number of the driver board associated to this object.

18. ***std::string getSenderIP() const***

    returns the IP address of the sender associated to this object.

19. ***int getSenderPort() const***

    returns the port number of the sender associated to this object.

20. ***connection_t getConnectionType()***

    returns the connection type of the driver associated to this object.

21. ***virtual bool checkSenderDTformat(const std::vector<unsigned char>& bufIn) const = 0***

    This function is declared pure virtual. Derived classes of LED must define this function.

22. ***virtual void decodeUDP(std::vector<unsigned char>& TXbuf, const std::vector<unsigned char>& bufIn) = 0***

    This function is declared pure virtual. Derived classes of LED must define this function.

23. ***virtual void decodeUSB(const std::vector<unsigned char>& bufIn) = 0***

    This function is declared pure virtual. Derived classes of LED must define this function.

Class Matrix is a derived class of the base class LED. It has two unique properties; it has a height and a width. The inherited virtual functions are not defined in this class. Therefore, this class is also an abstract class. The SET-functions takes call by reference arguments to avoid unnecessary coping of data. The GET-functions are declared as constant to prevent any modification to the private variables.

The Matrix class has two private (non-inherited) variables:

1. ***int width***
2. ***int height***

These private attributes are self-explanatory and can be changed or returned with the public member functions in its class.

The Matrix class has seven public (non-inherited) member functions:

1. ***Matrix(const char* name, const int& Ext_ch, const int& Int_ch, const int& wd, const int& ht, const color_mode& c)***

   This is the constructor of a Matrix object. It calls the LED constructor and passes the arguments “name”, “Ext_ch”, “Int_ch”, “c” and MATRIX. This automatically creates an LED object with the new attribues. The function also sets the width and the height of the object.

2. ***virtual ~Matrix()***

   This is the default destructor. This function is declared as virtual ensure proper polymorphic destruction of the object at the end of its life.

3. ***void setWidth(const int& wd)***

   changes the private variable width to the value of the argument “wd”.

4. ***void setHeigth(const int& ht)***

   changes the private variable height to the value of the argument “ht”.

5. ***void setDimensions(const int& wd, const int& ht)***

   changes both the private variable width and height to the value of the argument “wd” and “ht” respectively.

6. ***int getWidth() const***

   returns the value of the private variable width.

7. ***int getHeight() const***

   returns the value of the private variable height.

Class Strip is a derived class of the base class LED. The inherited virtual functions are not defined in this class. Therefore, this class is also an abstract class. This class has a unique property; it has a length. The SET-functions takes call by reference arguments to avoid unnecessary coping of data. The GET-functions are declared as constant to prevent any modification to the private variables.

The Strip class has three private (non-inherited) variables:

1. ***int pixelLength;***
2. ***led_t ledtype;***
3. ***config_t configuration;***

These private attributes are self-explanatory and can be changed or returned with the public member functions in its class.

The Strip class has eight public (non-inherited) member functions:

1. ***Strip(const char* name, const int& Ext_ch, const int& Int_ch, const int& len, const led_t& type, const color_mode& c, const config_t& config);***

   This is the constructor of a Strip object. It calls the LED constructor and passes the arguments “name”, “Ext_ch”, “Int_ch”, “c” and STRIP. This automatically creates an LED object with the new attribues. The function also sets the length, the LED type and the configuration type of the object.

2. ***virtual ~Strip();***

   This is the default destructor. This function is declared as virtual ensure proper polymorphic destruction of the object at the end of its life.

3. ***void setLength(const int& len);***

   sets the length of the LED strip to the value of argument “len”.

4. ***void setType(const led_t& type);***

   sets the type of LED to the value of argument “type”.

5. ***void setConfig(const config_t& config);***

   sets the configuration in which the LED strip is. The configuration variable has no meaningful function and is created for future expansion of the server software.

6. ***int getLength() const;***

   returns the length of the LED strip.

7. ***led_t getType() const;***

   returns the type of the LED strip

8. ***config_t getConfig() const;***

   returns the configuration of the LED strip.

Class Raw is a derived class of the base class LED. The inherited functions are defined. Therefore this class is not an abstract class. This class is created for the sole purpose of backwards compatibility with the older programs the project provider created prior this project. This class only stores the most essential information and has no unique property. Therefore, this class is a wrapper class to package the data into a message the server software can accept. This class should be phased out in future iterations of the server software.

The class contains no private variable and has five public (non-inherited) member functions:

1. ***Raw(const char* name, const int& Ext_ch, const int& Int_ch)***

   This is the constructor of a Raw object. It calls the LED constructor and passes the arguments “name”, “Ext_ch”, “Int_ch”, “c” and RAW. This automatically creates an LED object with the new attribues.

2. ***virtual ~Raw()***

   This is the default destructor. This function is declared as virtual ensure proper polymorphic destruction of the object at the end of its life.

3. ***virtual bool checkSenderDTformat(const std::vector<unsigned char>& bufIn) const override***

   This function inspects if the incoming message (bufIn) complies with the communication protocol. The function returns a Boolean; true return value means the message does conform to the protocol and false means the message does not conform to the protocol. This function defines the pure virtual function declared in the LED class.

4. ***virtual void decodeUDP(std::vector<unsigned char>& TXbuf, const std::vector<unsigned char>& bufIn) override***

   This function is used when the driver board associated to this object is using the UDP protocol to send and receive data. The color data is extracted from the message (bufIn) by removing all the unnecessary information. This color data is then copied to the outgoing message container/vector TXbuf. In other words, the message is stripped to its bare minimum (raw data). This function defines the pure virtual function declared in the LED class.

5. ***virtual void decodeUSB(const std::vector<unsigned char>& bufIn) override***

   This function is used when the driver board associated to this object is connected to the server by USB. The function decodes the incoming message (bufIn). The color data is extracted for the message and transferred to a USB driver software (created by the company). This function defines the pure virtual function declared in the LED class.

Class AddressableStrip is a derived class of the class Strip. The inherited virtual functions are not defined in this class. Therefore, this class is also an abstract classes. As explained at the beginning of this chapter, this class is created to give the developer more creative control over groups of subclasses.

The class contains no private variable and has two public (non-inherited) member functions:

1. ***AddressableStrip(const char* name, const int& Ext_ch, const int& Int_ch, const int& len, const color_mode& c, const config_t config)***

   This is the constructor of an AddressableStrip object. It calls the Strip constructor and passes the arguments “name”, “Ext_ch”, “Int_ch”, “len”, ADDRESSABLE, “c” and “config”. This automatically creates a Strip object with the new attribues.

2. ***virtual ~AddressableStrip()***

   This is the default destructor. This function is declared as virtual ensure proper polymorphic destruction of the object at the end of its life.

Class NonAddressableStrip is a derived class of the class Strip. The inherited virtual functions are not defined in this class. Therefore, this class is also an abstract classes. As explained at the beginning of this chapter, this class is created to give the developer more creative control over groups of subclasses.

The class contains no private variable and has two public (non-inherited) member functions:

1. ***NonAddressableStrip(const char* name, const int& Ext_ch, const int& Int_ch, const int& len, const color_mode& c, const config_t config)***

   This is the constructor of an NonAddressableStrip object. It calls the Strip constructor and passes the arguments “name”, “Ext_ch”, “Int_ch”, “len”,

NONADDRESSABLE, “c” and “config”. This automatically creates a Strip object with the new attribues.

2. ***virtual ~AddressableStrip()***

   This is the default destructor. This function is declared as virtual ensure proper polymorphic destruction of the object at the end of its life.

Derived classes from the AddressableStrip, NonAddressableStrip and Matrix classes are created because each class, type and color mode combination requires a different method to process/decode data. Each derived class contains three member functions which has its own definition: checkSenderDTformat(…), decodeUDP(…) and decodeUSB(…). Each class has its own message format and its own way of processing the data. The processing sequence is determined by the communication protocol described in chapter 5.1.

Class AddressableStripMono is a derived class of the base class AddressableStrip. The inherited functions are defined. Therefore this class is not an abstract class.

The class contains no private variable and has six public (non-inherited) member functions:

1. ***AddressableStripMono(const char* name, const int& Ext_ch, const int& Int_ch)***

   This is a constructor of an AddressableStripMono object. This function is used when a driver connects to the server. It calls the AddressableStrip constructor and passes the arguments “name”, “Ext_ch”, “Int_ch”, 1, ADDRESSABLE, MONO and OTHER. This automatically creates the AddressableStrip object with the new attribues. The channel settings are unknown and therefore set to a default.

2. ***AddressableStripMono(const char* name, const int& Ext_ch, const int& Int_ch, const int& len, const config_t config)***

   This is a constructor of an AddressableStripMono object. It calls the AddressableStrip constructor and passes the arguments “name”, “Ext_ch”, “Int_ch”, “len”, ADDRESSABLE, MONO and “config”. This automatically creates an AddressableStrip object with the new attribues. This function is used when the channel settings become known or is set by the administrator.

3. ***virtual ~AddressableStripMono()***

   This is the default destructor. This function is declared as virtual ensure proper polymorphic destruction of the object at the end of its life.

4. ***virtual bool checkSenderDTformat(const std::vector<unsigned char>& bufIn) const override***

   checks the message size and byte sequence of the message to determine if the message complies with the communication protocol. The message size must equal to 6 + (length of the strip) bytes.

5. ***virtual void decodeUDP(std::vector<unsigned char>& TXbuf, const std::vector<unsigned char>& bufIn) override***

   decodes and transform the incoming message (bufIn) into an outgoing message (TXbuf) that conforms to the communication protocol. The external channel number in bufIn is replaced with the internal channel number. Refer to the communication protocol in Appendix 1 to see the exact format of the message.

6. ***virtual void decodeUSB(std::vector<unsigned char>& TXbuf, const std::vector<unsigned char>& bufIn) override***

   This function has not be implemented yet because no USB driver software for this class has been written by the company yet.

Class AddressableStripRGB is a derived class of the base class AddressableStrip. The inherited functions are defined. Therefore this class is not an abstract class.

The class contains no private variable and has six public (non-inherited) member functions:

1. ***AddressableStripRGB(const char* name, const int& Ext_ch, const int& Int_ch)***

   This is a constructor of an AddressableStripRGB object. This function is used when a driver connects to the server. It calls the AddressableStrip constructor and passes the arguments “name”, “Ext_ch”, “Int_ch”, 1, ADDRESSABLE, RGB and OTHER. This automatically creates an AddressableStrip object with the new attribues. The channel settings are unknown and therefore set to a default.

2. ***AddressableStripRGB(const char* name, const int& Ext_ch, const int& Int_ch, const int& len, const config_t config)***

   This is a constructor of an AddressableStripRGB object. It calls the AddressableStrip constructor and passes the arguments “name”, “Ext_ch”, “Int_ch”, “len”, ADDRESSABLE, RGB and “config”. This automatically creates an AddressableStrip object with the new attribues. This function is used when the channel settings become known or is set by the administrator.

3. ***virtual ~AddressableStripRGB()***

   This is the default destructor. This function is declared as virtual ensure proper polymorphic destruction of the object at the end of its life.

4. ***virtual bool checkSenderDTformat(const std::vector<unsigned char>& bufIn) const override***

   checks the message size and byte sequence of the message to determine if the message complies with the communication protocol. The message size must equal to 6 + (length of the strip x 3) bytes.

5. ***virtual void decodeUDP(std::vector<unsigned char>& TXbuf, const std::vector<unsigned char>& bufIn) override***

   decodes and transform the incoming message (bufIn) into an outgoing message (TXbuf) that conforms to the communication protocol. The external channel number in bufIn is replaced with the internal channel number. Refer to the communication protocol in Appendix 1 to see the exact format of the message.

6. ***virtual void decodeUSB(std::vector<unsigned char>& TXbuf, const std::vector<unsigned char>& bufIn) override***

   The color data is extracted from the incoming message (bufIn). This data is then passed to the USB driver function std::vector<unsigned char> cmd_load_data(unsigned int ch_num, unsigned int len, const std::vector<unsigned char>& RGBdata) to process a message suitable for the USB driver software. This USB driver function output the message to the outgoing message (TXbuf). The USB protocol is previously defined by the company and the API is created by a member of the company. The executer of this project did not participate in the USB protocol definition and API creation.

Class NonAddressableStripMono is a derived class of the base class NonAddressableStrip. The inherited functions are defined.

The class contains no private variable and has six public (non-inherited) member functions:

1. ***NonAddressableStripMono(const char* name, const int& Ext_ch, const int& Int_ch)***

   This is a constructor of a NonAddressableStripMono object. This function is used when a driver connects to the server. It calls the NonAddressableStrip constructor and passes the arguments “name”, “Ext_ch”, “Int_ch”, 1, NONADDRESSABLE, MONO and OTHER. This automatically creates a NonAddressableStrip object with the new attribues. The channel settings are unknown and therefore set to a default.

2. ***NonAddressableStripMono(const char* name, const int& Ext_ch, const int& Int_ch, const int& len, const config_t config)***

   This is a constructor of a NonAddressableStripMono object. It calls the NonAddressableStrip constructor and passes the arguments “name”, “Ext_ch”, “Int_ch”, “len”, NONADDRESSABLE, MONO and “config”. This automatically creates a NonAddressableStrip object with the new attribues. This function is used when the channel settings become known or is set by the administrator.

3. ***virtual ~NonAddressableStripMono()***

   This is the default destructor. This function is declared as virtual ensure proper polymorphic destruction of the object at the end of its life.

4. ***virtual bool checkSenderDTformat(const std::vector<unsigned char>& bufIn) const override***

   checks the message size and byte sequence of the message to determine if the message complies with the communication protocol. The message size must equal to 5 bytes.

5. ***virtual void decodeUDP(std::vector<unsigned char>& TXbuf, const std::vector<unsigned char>& bufIn) override***

   decodes and transform the incoming message (bufIn) into an outgoing message (TXbuf) that conforms to the communication protocol. The external channel number in bufIn is replaced with the internal channel number. Refer to the communication protocol in Appendix 1 to see the exact format of the message.

6. ***virtual void decodeUSB(std::vector<unsigned char>& TXbuf, const std::vector<unsigned char>& bufIn) override***

   This function has not be implemented yet because no USB driver software for this class has been written by the company yet.

Class NonAddressableStripRGB is a derived class of the base class NonAddressableStrip. The inherited functions are defined. Therefore this class is not an abstract class.

The class contains no private variable and has six public (non-inherited) member functions:

1. ***NonAddressableStripRGB(const char* name, const int& Ext_ch, const int& Int_ch)***

   This is a constructor of a NonAddressableStripRGB object. This function is used when a driver connects to the server. It calls the NonAddressableStrip constructor and passes the arguments “name”, “Ext_ch”, “Int_ch”, 1, NONADDRESSABLE, RGB and OTHER. This automatically creates a NonAddressableStrip object with the new attribues. The channel settings are unknown and therefore set to a default.

2. ***NonAddressableStripRGB(const char* name, const int& Ext_ch, const int& Int_ch, const int& len, const config_t config)***

   This is a constructor of a NonAddressableStripMono object. It calls the NonAddressableStrip constructor and passes the arguments “name”, “Ext_ch”, “Int_ch”, “len”, NONADDRESSABLE, RGB and “config”. This automatically creates a NonAddressableStrip object with the new attribues. This function is used when the channel settings become known or is set by the administrator.

3. ***virtual ~NonAddressableStripRGB()***

   This is the default destructor. This function is declared as virtual ensure proper polymorphic destruction of the object at the end of its life.

4. ***virtual bool checkSenderDTformat(const std::vector<unsigned char>& bufIn) const override***

   checks the message size and byte sequence of the message to determine if the message complies with the communication protocol. The message size must equal to 7 bytes.

5. ***virtual void decodeUDP(std::vector<unsigned char>& TXbuf, const std::vector<unsigned char>& bufIn) override***

   decodes and transform the incoming message (bufIn) into an outgoing message (TXbuf) that conforms to the communication protocol. The external channel number in bufIn is replaced with the internal channel number. Refer to the communication protocol in Appendix 1 to see the exact format of the message.

6. ***virtual void decodeUSB(std::vector<unsigned char>& TXbuf, const std::vector<unsigned char>& bufIn) override***

   This function has not be implemented yet because no USB driver software for this class has been written by the company yet.

Class MatrixMono is a derived class of the base class Matrix. The inherited functions are defined. Therefore this class is not an abstract class.

The class contains no private variable and has six public (non-inherited) member functions:

1. ***MatrixMono(const char* name, const int& Ext_ch, const int& Int_ch)***

   This is a constructor of a MatrixMono object. This function is used when a driver connects to the server. It calls the Matrix constructor and passes the arguments “name”, “Ext_ch”, “Int_ch”, 1, 1 and MONO. This automatically creates a Matrix object with the new attribues. The channel settings are unknown and therefore set to a default.

2. ***MatrixMono(const char* name, const int& Ext_ch, const int& Int_ch, const int& wd, const int& ht)***

   This is a constructor of a MatrixMono object. It calls the Matrix constructor and passes the arguments “name”, “Ext_ch”, “Int_ch”, “wd”, “ht” and MONO. This automatically creates a Matrix object with the new attribues. This function is used when the channel settings become known or is set by the administrator.

3. ***virtual ~MatrixMono()***

   This is the default destructor. This function is declared as virtual ensure proper polymorphic destruction of the object at the end of its life.

4. ***virtual bool checkSenderDTformat(const std::vector<unsigned char>& bufIn) const override***

   checks the message size and byte sequence of the message to determine if the message complies with the communication protocol. The message size must equal to 5 bytes.

5. ***virtual void decodeUDP(std::vector<unsigned char>& TXbuf, const std::vector<unsigned char>& bufIn) override***

   decodes and transform the incoming message (bufIn) into an outgoing message (TXbuf) that conforms to the communication protocol. The external channel number in bufIn is replaced with the internal channel number. Refer to the communication protocol in Appendix 1 to see the exact format of the message.

6. ***virtual void decodeUSB(std::vector<unsigned char>& TXbuf, const std::vector<unsigned char>& bufIn) override***

   This function has not be implemented yet because no USB driver software for this class has been written by the company yet.

Class MatrixRGB is a derived class of the base class Matrix. The inherited functions are defined. Therefore this class is not an abstract class.

The class contains no private variable and has six public (non-inherited) member functions:

1. ***MatrixRGB(const char* name, const int& Ext_ch, const int& Int_ch)***

   This is a constructor of a MatrixMono object. This function is used when a driver connects to the server. It calls the Matrix constructor and passes the arguments “name”, “Ext_ch”, “Int_ch”, 1, 1 and RGB. This automatically creates a Matrix object with the new attribues. The channel settings are unknown and therefore set to a default.

2. ***MatrixRGB(const char* name, const int& Ext_ch, const int& Int_ch, const int& wd, const int& ht)***

   This is a constructor of a MatrixRGB object. It calls the Matrix constructor and passes the arguments “name”, “Ext_ch”, “Int_ch”, “wd”, “ht” and RGB. This automatically creates a Matrix object with the new attribues. This function is used when the channel settings become known or is set by the administrator.

3. ***virtual ~MatrixRGB()***

   This is the default destructor. This function is declared as virtual ensure proper polymorphic destruction of the object at the end of its life.

4. ***virtual bool checkSenderDTformat(const std::vector<unsigned char>& bufIn) const override***

   checks the message size and byte sequence of the message to determine if the message complies with the communication protocol. The message size must equal to 5 bytes.

5. ***virtual void decodeUDP(std::vector<unsigned char>& TXbuf, const std::vector<unsigned char>& bufIn) override***

   decodes and transform the incoming message (bufIn) into an outgoing message (TXbuf) that conforms to the communication protocol. The external channel number in bufIn is replaced with the internal channel number. Refer to the communication protocol in Appendix 1 to see the exact format of the message.

6. ***virtual void decodeUSB(std::vector<unsigned char>& TXbuf, const std::vector<unsigned char>& bufIn) override***

   This function has not be implemented yet because no USB driver software for this class has been written by the company yet.

5.2.3. Database header and cpp file

The Database header file declares one class and its member functions and attributes. The class has functions that saves and returns a value of one of the attributes important to the proper functioning of the server. The header file also define a template function.

Class Mapping has a private data structure (databaseAdmin) to hold the IP address and port number of the administrator. It also has a private storage container “database” that holds all the objects associated with the driver channels connected to the server.

The Mapping class has thirty-two public member functions and three private member functions:

1. ***void registerDriver(const int& channel, const std::string& IP, const int& port, connection_t con_t)***

   This function creates a default Raw object when a driver channel connects to the server. This new object is added to the server database. Its channel number, IP address, port number and the connection type is stored.

2. ***void deregisterDriver(const int& channel)***

   This function is called when a driver channel disconnects from the server. It removes the object associated with the driver channel from the database.

3. ***void registerSender(const int& channel, const std::string& IP, const int& port)***

   This function is called when a sender connects to a channel. The IP address and port number of the sender is saved into the object with the same channel number.

4. ***void deregisterSender(const int& channel)***

   When a sender disconnects from a channel, the sender IP address and sender port number of the object is set back to “0.0.0.0” and 0 respectively.

5. ***void registerAdmin(const std::string& IP, const int& port)***

   This function is called when an administrator connects to the server. The IP address and port number of the administrator is stored in the struct “databaseAdmin”.

6. ***void deregisterAdmin()***

   This function is called when the administrator disconnects form the server The IP address and port number of the administrator is reset to “0.0.0.0” and 0 respectively.

7. ***void changeStrip(const int& channel, const int& length, const led_t& msgType, const config_t& msgConfig, const color_mode& color, const int& group)***

   This function retrieves the driver and sender IP addresses and port number of the object stored in the database. This old object is destroyed and a new derived Strip object is constructed. The IP addresses, port numbers and other parameters, passed to the function, is then stored into the newly created object.

8. ***void changeMatrix(const int& channel, const int& width, const int& height, const color_mode& color, const int& group)***

   This function retrieves the driver and sender IP addresses and port number of the object stored in the database. This old object is destroyed and a new derived Matrix object is constructed. The IP addresses, port numbers and other parameters, passed to the function, is then stored into the newly created object.

9. ***void changeRaw(const int& channel, const int& group)***

   This function retrieves the driver and sender IP addresses and port number of the object stored in the database. This old object is destroyed and a new derived Raw object is constructed. The IP addresses, port numbers and other parameter, passed to the function, is then stored into the newly created object.

10. ***std::string printdatabaseChannelDriver()***

    assembles a string containing all attribute information of all the driver channels stored in the database. The message format is defined in the communication protocol.

11. ***std::string printdatabaseChannelSender()***

    assembles a string containing all attribute information of all the sender stored in the database. The message format is defined in the communication protocol.

12. ***bool doesChannelExist(const int& channel) const***

    returns true when a channel is linked to a driver board. If no driver board is linked to this channel, a false is returned.

13. ***bool doesChannelBelongToSender(const int& channel, const std::string& IP, const int& port)***

    returns true when the sender IP and port number matches the IP and port number stored in the database. If no match is found, a false is returned.

14. ***bool doesChannelBelongToDriver(int channel, std::string IP, int port)***

    returns true when the driver IP and port number matches the IP and port number stored in the database. If no match is found, a false is returned.

15. ***bool isChannelOccupied(const int& channel)***

    checks if the channel is already used by another sender. If so, the function returns a true and vice versa.

16. ***bool isDriverOccupied(const int& channel) const***

    checks if a driver is already using the channel number. If so, the function returns a true and vice versa.

17. ***bool doesAdminExist() const***

    checks if an administrator is already connected to the server.

18. ***bool doesBelongToAdmin(const std::string& IP, const int& port) const***

    returns true when the administrator IP and port number matches the IP and port number stored in the database. If no match is found, a false is returned.

19. ***bool checkSenderDTformat(const int& channel, const std::vector<unsigned char>& bufIn)***

    searches for the object associated to the channel number (channel) in the database. The function then calls the checkSenderDTformat(…) function of that object.

20. ***void decodeUDP(const int& channel, std::vector<unsigned char>& TXbuf, const std::vector<unsigned char>& bufIn)***

    searches for the object associated to the channel number (channel) in the database. The function then calls the decodeUDP(…) function of that object.

21. ***void decodeUSB(const int& channel, const std::vector<unsigned char>& bufIn)***

    searches for the object associated to the channel number (channel) in the database. The function then calls the decodeUSB(…) function of that object.

22. ***std::string getDriverIPaddr(const int& channel)***

    searches for the object associated to the channel number (channel) in the database. The function then calls the getDriverIPaddr(…) function of that object.

23. ***int getDriverPort(const int& channel)***

    searches for the object associated to the channel number (channel) in the database. The function then calls the getDriverPort(…) function of that object.

24. ***class_t getClass(const int& channel)***

    searches for the object associated to the channel number (channel) in the database. The function then calls the getClass(…) function of that object.

25. ***color_mode getColorMode(const int& channel)***

    searches for the object associated to the channel number (channel) in the database. The function then calls the getColorMode (…) function of that object.

26. ***connection_t getConnectionType(const int& channel)***

    searches for the object associated to the channel number (channel) in the database. The function then calls the getConnectionType(…) function of that object.

27. ***int getLength(const int& channel)***

    searches for the Strip object associated to the channel number (channel) in the database. The function then calls the getLength(…) function of that Strip object.

28. ***led_t getType(const int& channel)***

    searches for the Strip object associated to the channel number (channel) in the database. The function then calls the getType(…) function of that Strip object.

29. ***config_t getConfig(const int& channel)***

    searches for the Strip object associated to the channel number (channel) in the database. The function then calls the getConfig(…) function of that Strip object.

30. ***int getWidth(const int& channel)***

    searches for the Matrix object associated to the channel number (channel) in the database. The function then calls the getWidth(…) function of that Matrix object.

31. ***int getHeight(const int& channel)***

    searches for the Matrix object associated to the channel number (channel) in the database. The function then calls the getHeight(…) function of that Matrix object.

32. ***LED* createDerivedStrip(const int& channel, const std::string& IP, const int& port, const int& length, const led_t& msgType, const config_t& msgConfig, const color_mode& colorMode)***

    The function determines the class of the Strip object needed based on the parameter passed to the arguments and uses the createObject template to create the required Strip object.

33. ***LED* createDerivedMatrix(const int& channel, const std::string& IP, const int& port, const color_mode& colorMode, const int& width, const int& height)***

    The function determines the class of the Matrix object needed based on the parameter passed to the arguments and uses the createObject template to create the required Matrix object.

34. ***LED* createDerivedRaw(const int& channel, const std::string& IP, const int& port)***

    uses the createObject template to create the required Matrix object.

35. ***template<typename T>***
    T* createObject(int Ext_channel, int Int_channel) const

    The function template calls the constructor with three argument of the T class. The first argument is the channel number converted to a string.

5.2.4. MessageHander header and cpp file

The MessageHandler program receives, sorts, processes and sends message that arrives at the UDP socket of the server. Based on the source and type of message, the message is redirected to different location/paths.

The header file defines a constant BUFSIZE as 65507. This is the maximum size an incoming UDP message can be. Any message sizes larger than this number returns an error. There is also an enumeration declaration Response in this header file. This used to generate generic feedback message to the sender of a message.

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

The above shown code contains the different keys. More key can be added in the future.

The Socket class has ten private variables:

1. ***struct sockaddr_in myaddr;***

   stores the IP address, port number and other network setting in a struct sockaddr_in. This struct is needed to access the UDP socket.

2. ***struct sockaddr_in remaddr;***

   stores the IP address, port number and other network setting of the receiver of the outgoing message.

3. ***int addrlen = sizeof(remaddr);***

   stores the address length of the remote address.

4. ***int recvlen;***

   stores the byte length of the incoming message.

5. ***SOCKET s;***

   This is the socket used for receiving and sending data

6. ***WSADATA wsa;***

   Stores information about the socket

7. ***char buf[BUFSIZE];***

   This is the buffer where the incoming message is temporarily stored before being routed to the appropriate message storage container.

8. ***bool Server_isRunning;***

   is false when the server must shut down. The server continues to run when the variable is true.

The Socket class has eleven public member functions and six private member functions:

1. ***Socket(Mapping& m);***

   This is the constructor of an Socket object. It calls the setupUDP() function and initializes threads. These threads are collected in a storage container called “thread”. There are seven different functions called in the threads: processSenderDT(…), processSenderCMD(…), processDriverDT(…), process-DriverCMD(…), processAdmin-DT(…), processAdminCMD(…) and sendUDP-msg(…).

2. ***~Socket();***

   This is the object default destructor. It sets the variable Server_isRunning to false to indicate a sever shutdown and wakes all threads up from their blocked state. A

command is send is sent to close the UDP socket used by the server. Finally, all threads will be joined before the server is shutdown.

3. *int setupUDPsocket();*

This creates a UDP socket where messages can be sent to and from. If the socket initialization cannot be completed, the program will terminated.

4. *void checkUDPmsg();*

This function will run continuously if variable “Server_isRunning” equals true. The recvfrom() function copies any messages received at the UDP socket to a char array (buf) and returns the byte size of the received message. The size number is saved in variable “recvlen”. The recvfrom blocks until a message is received or when the socket is closed. A function is said to block when the process cannot continue because the function is not exiting. The server can start shutting down while the revcfrom() function is blocking. When the socket is closed, the recfrom() function exits. The variable “Server_isRunning is tested immediately. If the variable equals false, the checkUDPmsg() returns.

If variable recvlen does not equal -1 (SOCKET_ERROR), the message stored in buf is copied to a vector storage container. This removes all the used space allocated in the buf array and the software developer can use all its advanced member functions such as iterators. The message and source information are stored in the correct inbox/message queue based on the source and type of the message. Since the inboxes are shared with other threads, mutexes are used to ensure safe access to the inboxes.

5. *void processSenderDT(Mapping& map);*

This function will run continuously if variable “Server_isRunning” equals true. The function tries to lock the mutex “mtxSenderDT”. If it fails, the thread blocks. The server can start shutting down while the thread is blocked. When the conditional variable is notified, the thread is unblocked. When the thread wakes up, it locks the mutex if the inbox “queueSenderDT” contains messages or the “Server_isRunning” equals false. The thread continues to block if none of the conditions are true. The variable “Server_isRunning is tested immediately. If the variable equals false, the processSenderDT() returns.

If the server is still running, a message and its source information are removed from the inboxes. The message must pass the message format test. This is done by calling the checkSenderDTformat() function from the Mapping class. The channel number and the message itself is passed to the function. The next test (doesChannelExist() function from the Mapping class) checks if a driver is connected to drive the requested channel. The next test (isChannelOccupied() function from the Mapping class) checks if a sender is already connected to the server. All senders must be connected to the server before sending data message to the server. The final test (doesChannelBelongToSender() function from the Mapping class) checks if the sender is the owner/ connected to this channel. If the channel is connected to the driver by UDP, the decodeUDP() function is called.

The decoded message is redirected to the outbox (OutgoingMessage-Queue_UDP). If the channel is connected to the driver by USB, the decodeUSB() function is called.

6. *void processSenderCMD(Mapping& map);*

This function will run continuously if variable “Server_isRunning” equals true. The function tries to lock the mutex “mtxSenderCMD”. If it fails, the thread blocks. The server can start shutting down while the thread is blocked. When the conditional variable is notified, the thread is unblocked. When the thread wakes up, it locks the mutex if the inbox “queueSenderCMD” contains messages or the “Server_isRunning” equals false. The thread continues to block if none of the conditions are true. The variable “Server_isRunning is tested immediately. If the variable equals false, the processSender-CMD() returns.

If the server is still running, a message and its source information are removed from the inboxes. The message must pass the message format test. This is done by calling the checkSenderCMDformat() function from the Mapping class. The channel number and the message itself is passed to the function. The message contains a command identifier.

If the identifier equals to SENDER_CONNECT, the first test (doesChannelExist() function from the Mapping class) checks if a driver is connected to drive the requested channel. The next test (isChannelOccupied() function from the Mapping class) checks if a sender is already connected to the server. Only when all parameters matches the settings saved in the server, the registerSender() function is called.

If the identifier equals to SENER_DISCONNECT, the program checks if the sender is indeed who he claims to be before disconnecting the sender from the server.

7. *void processDriverDT(Mapping& map);*

This function is a framework and is to be implement in the future. At the time of writing, the driver boards cannot send any data to the server.

8. *void processDriverCMD(Mapping& map);*

This function will run continuously if variable “Server_isRunning” equals true. The function tries to lock the mutex “mtxDriverCMD”. If it fails, the thread blocks. The server can start shutting down while the thread is blocked. When the conditional variable is notified, the thread is unblocked. When the thread wakes up, it locks the mutex if the inbox “queueDriverCMD” contains messages or the “Server_isRunning” equals false. The thread continues to block if none of the conditions are true. The variable “Server_isRunning is tested immediately. If the variable equals false, the processDriver-CMD() returns.

If the server is still running, a message and its source information are removed from the inboxes. The message must pass the message format test. This is done by calling the checkDriverCMDformat() function from the Mapping class. The

channel number and the message itself is passed to the function. The message contains a command identifier.

If the identifier equals to DRIVER_CONNECT, the isChannelOccupied() function from the Mapping class checks if a driver is already connected to the server. The driver is registered to the server by calling the registerDriver() function.

If the identifier equals to DRIVER_DISCONNECT, the program checks if the driver is indeed who he claims to be before disconnecting the driver from the server.

9. *void processAdminDT(Mapping& map);*

This function will run continuously if variable “Server_isRunning” equals true. The function tries to lock the mutex “mtxAdminDT”. If it fails, the thread blocks. The server can start shutting down while the thread is blocked. When the conditional variable is notified, the thread is unblocked. When the thread wakes up, it locks the mutex if the inbox “queueAdminDT” contains messages or the “Server_isRunning” equals false. The thread continues to block if none of the conditions are true. The variable “Server_isRunning is tested immediately. If the variable equals false, the processAdminDT() returns.

If the server is still running, a message and its source information are removed from the inboxes. The message must pass the message format test. This is done by calling the checkAdminDTformat() function from the Mapping class. The channel number and the message itself is passed to the function. The message contains a request identifier.

The administrator must be registered to the server before it can request data. The function “doesBelongToAdmin()” is called to check if the sender of the request is indeed the registered administrator. If the test is passed, the request identifier “REQUEST_SENDERDATA” will call the function getDatabase_SenderInfo() or “REQUEST_DRIVERDATA” will call the function getDatabase_DriverInfo().

10. *void processAdminCMD(Mapping& map);*

This function will run continuously if variable “Server_isRunning” equals true. The function tries to lock the mutex “mtxAdminCMD”. If it fails, the thread blocks. The server can start shutting down while the thread is blocked. When the conditional variable is notified, the thread is unblocked. When the thread wakes up, it locks the mutex if the inbox “queueAdminCMD” contains messages or the “Server_isRunning” equals false. The thread continues to block if none of the conditions are true. The variable “Server_isRunning is tested immediately. If the variable equals false, the processAdminCMD() returns.

If the server is still running, a message and its source information are removed from the inboxes. The message must pass the message format test. This is done by calling the checkAdminCMDformat() function from the Mapping class. The channel number and the message itself is passed to the function. The message contains a command identifier.

If the identifier equals to ADMIN_CONNECT, the doesAdminExist() function from the Mapping class checks if a administrator is already connected to the server. The administrator is registered to the server by calling the registerAdmin() function.

If the identifier equals to ADMIN_DISCONNECT, the program checks if the administrator is indeed who he claims to be before disconnecting the administrator from the server.

If the identifier equals to CHANNELMOD, the program changes the channel object by calling “changeStrip()”, “changeMatrix() or changeRaw()” function from the Mapping class. Before this is done, an identity test is done to ensure the right administrator is sending this command.

11. *void sendUDPmsg(Mapping& map);*

This function will run continuously if variable “Server_isRunning” equals true. The function tries to lock the mutex “mtxOut_UDP”. If it fails, the thread blocks. The server can start shutting down while the thread is blocked. When the conditional variable is notified, the thread is unblocked. When the thread wakes up, it locks the mutex if the inbox “OutgoingMessageQueue_UDP” contains messages or the “Server_isRunning” equals false. The thread continues to block if none of the conditions are true. The variable “Server_isRunning is tested immediately. If the variable equals false, the sendUDPmsg() returns.

If the server is still running, a message and its source information are removed from the outbox. The message is send to the destination using the function “sendto(…).

12. *void bool isSenderCMDMessageFormatCorrect(const std::vector<unsigned char>& bufIn) const;*

checks the message size and byte sequence of the message to determine if the message complies with the communication protocol. The format varies per type of command. Refer to the communication protocol in Appendix 1 for more detail on the format.

13. *bool isDriverDTMessageFormatCorrect(const std::vector<unsigned char>& bufIn) const;*

checks the message size and byte sequence of the message to determine if the message complies with the communication protocol. The format varies per type of request. Refer to the communication protocol in Appendix 1 for more detail on the format.

14. *bool isDriverCMDMessageFormatCorrect(const std::vector<unsigned char>& bufIn) const;*

checks the message size and byte sequence of the message to determine if the message complies with the communication protocol. The format varies per type of command. Refer to the communication protocol in Appendix 1 for more detail on the format.

15. ***bool isAdminDTMessageFormatCorrect(const std::vector<unsigned char>& bufIn) const;***

    checks the message size and byte sequence of the message to determine if the message complies with the communication protocol. The format varies per type of request. Refer to the communication protocol in Appendix 1 for more detail on the format.

16. ***bool isAdminCMDMessageFormatCorrect(const std::vector<unsigned char>& bufIn) const;***

    checks the message size and byte sequence of the message to determine if the message complies with the communication protocol. The format varies per type of command. Refer to the communication protocol in Appendix 1 for more detail on the format.

17. ***void feedbackMessage(const Response& response, const sockaddr_in& msgaddr);***

    creates a string message intended to provide the sender of the message some feedback. The sender can used this feedback to track the state of the connection with the server.

The Socket class also has fourteen storage container, eight mutexes and seven conditional variables:

```
/* 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;
```

5.2.5. USBdriver header and cpp file

These files are part of the API written by another member of the company. The executer of this project created new useful functions to streamline the API to the server. The API functions not created by the project executer will be explain in a clear but not detailed manner.

The enumeration exitCodes are used to provide the programmer some feedback when the program is terminated by an error. When a driver board is connected by USB to the server, a Device object is created which stores information such as the device description.

The header file defines ten functions:

1. ***void scanForUSBdevices()***

   calls the openDevices(), purgeRxTxBuffers(…) and setBitMode(…) functions. This function is added by the project executer.

2. ***void setUSBChannelLength(unsigned int chNum, unsigned int len)***

   calls cmd_set_length(…) functions and sends the message stored in TXbuffer to the driver board. This function is added by the project executer.

3. ***unsigned int cmd_set_length(unsigned int ch_num, unsigned int len)***

   assembles a set length command in TXbuffer

4. ***std::vector<unsigned char> cmd_load_data(unsigned int ch_num, unsigned int len, const std::vector<unsigned char>& RGBdata)***

   assembles and returns a color data message. The message follows a protocol created by the company. This function is added by the project executer.

5. ***void sendUSB_RGBStripData(std::vector<unsigned char>& RGBdata)***

   sends the message created by cmd_load_data to the driver board. This function is added by the project executer.

6. ***int openDevices()***

   scans for any driver boards connected to the server by USB. When the found device are save in a list called “deviceList”.

7. ***void purgeRxTxBuffers(FT_HANDLE ftHandle)***

   clears the RX and TX buffer of the driver boards.

8. ***void setBitMode(FT_HANDLE ftHandle, UCHAR ucMask, UCHAR ucMode)***

   sets the mode of the USB device

9. ***Device* get_deviceList()***

   returns a pointer to the “deviceList”. The program can gain access to this list. This function is added by the project executer.

10. ***DWORD get_numDevs()***

    Returns the number for connected driver boards. This function is added by the project executer.

5.2.6. Main cpp file

This is the highest level program; the program start here. A Mapping object is created to store LED objects. A Socket object is created which will open an UDP socket when other programs can send messages to. Creating the Socket object will also start the various threads described in chapter 5.2.4. The program searches for USB connected driver boards. If there are USB boards connected, the new channels are registered.

New UDP channels are automatically created because the driver boards cannot send any data to the server as of yet. When “shutdown” is typed in the console, the program arrives at the end of the program. The destructor of the Socket object will close the UDP socket and join all the threads. The program closes with a “server shutdown” message.

5.3. Test/ Sample programs

The primary purpose of these programs is to test the performance of the server software. They can also be used as sample programs because part of the sample program can be reused. A total of six test Processing programs are made: Sender_Knobs, Sender_Webcam, Sender_Rainbow, Sender_GIF, Driver_Receive and Admin_Control. Another test program is created in Visual Studio that uses the API library describe in chapter 5.4.

The **Sender_Knob** program displays four virtual LED strip. They are 1, 5, 8 and 10 LEDs long and each virtual LED can be controlled. The user clicks on one of the LEDs and can changed the color of the LED but adjusting the knob on the left (see figure 5.4). Before the user can send data to the server, the user must press “s” on their keyboard to allow the program to connect to the server. Pressing “q’ will disconnect the program from the server.

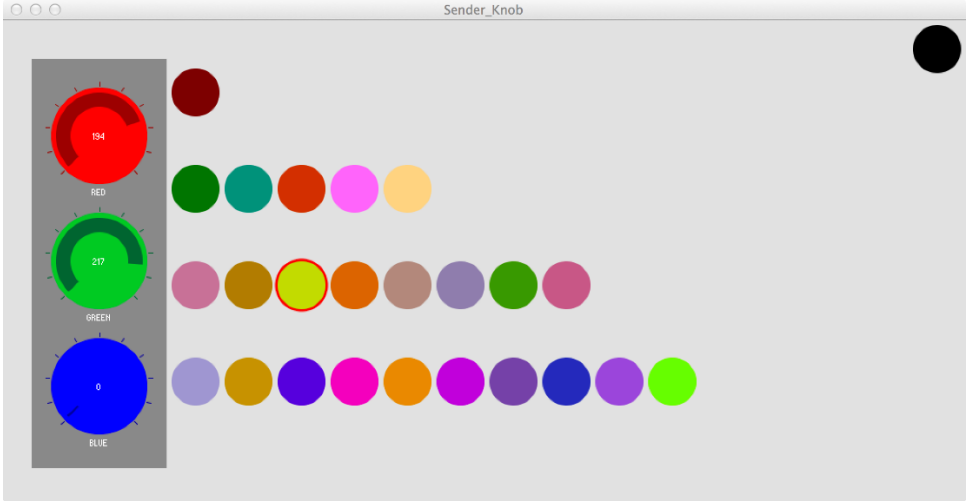


Figure 5.4 Sender_Knob user interface

The **Sender_Webcam** program runs on a macintosh operation system and loads the data stream of a built-in webcam. The video feed has a dimension of 1280 by 720 pixels and is downsampled to 40 by 23 pixels. This downsampled video feed is then processed as 23 individual LED strips of 40 LEDs each (figure 5.5). Before the user can send data to the server, the user must press “s” on their keyboard to allow the program to connect to the server. Pressing “q’ will disconnect the program from the server.

The **Sender_Rainbow** program has no user interface (UI). The user presses “s” to connect and start the program. Data for one LED strip with a length of 256 is send to the server. The output of the program is a scroll of rainbow colors across the LED strip.

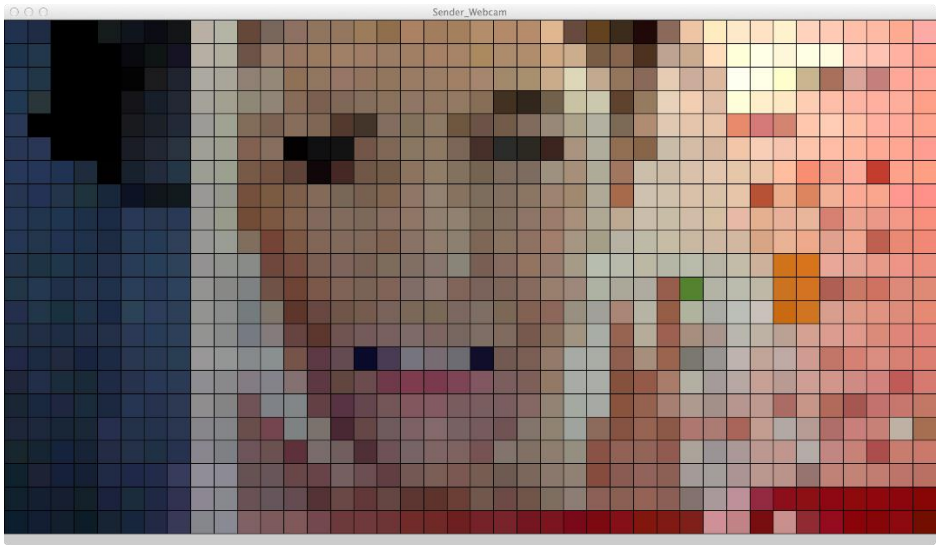


Figure 5.5 Sender_Webcam user interface

The **Sender_GIF** program can drive one or more LED matrixes. On matrix panel is set to be 32 by 8 pixels. The size of the panels can be changed in the code. The program sends data to three panels of 32 by 8 pixels. A total of 768 points are sampled for the GIF file and send to the server. Before the user can send data to the server, the user

must press “s” on their keyboard to allow the program to connect to the server. Pressing “q’ will disconnect the program from the server. Figure 5.6 shows a loaded GIF file in the UI of the program.

Figure 5.6 Sender_GIF user interface

At the time of writing, no UDP software support has been implemented on the Point Runner driver board yet. To test the UDP functions of the server, a receiver program is created. The **Driver_Recieve** program is supposed to replicate the same function of the Point Runner driver boards. The program has predefined amount of virtual LED strip with a predefined length. The program decodes the output message of the server and put the color on the display.

The **Admin_Control** program is used to control the channel settings of all the driver boards connected to the server in the DRIVERS tab. IP address and port numbers of all the senders can also be reviewed in the SENDERS tab. The GROUP does not contain any interface and will be implemented in the future. The user connects and disconnect the program to/from the server in the STATUS tab. Figure 5.7 shows the DRIVER tab where the user can change the channel’s settings.

The **API test** program uses the API library to send data of one strip with a length of 256. The color alternates between red, green and blue. The colors are shifted to the left after a predefined interval.

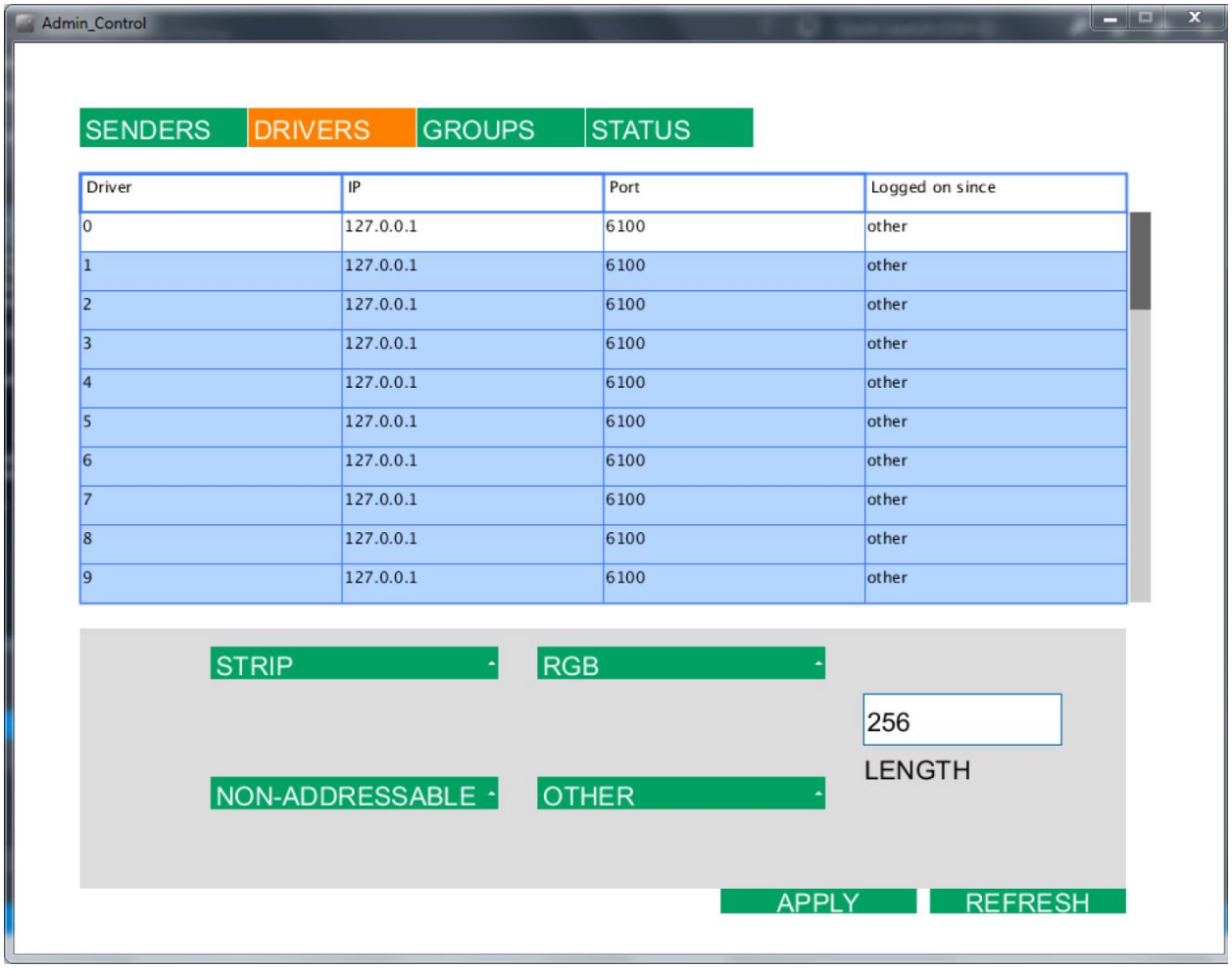


Figure 5.7 Admin_Control user interface

5.4. API library

The purpose of an API is to simplify the interface, which the programmer use to create their own program. The idea behind this API library is to create a minimal interface that can process multiple types of data. The programmer must create a “StripChannelParameter” or “MatrixChannelParameter” object. This object hold the settings of a channel message. The API uses this object to determine how to process a message

When a StripChannelParameter object is created, its member variable must be set. This object contain the following variables: Int ChannelNumber, color_mode ColorMode, class_t Class, config_t Configuration, led_t StripType and int Length.

When a MatrixChannelParameter object is created, its member variable must be set. This object contain the following variables: Int ChannelNumber, color_mode ColorMode, class_t Class, int Width and int Height.

To define some of the variable the following enumrations must be used:

```
enum led_t
{
        ADDRESSABLE,
        NONADDRESSABLE,
};

enum color_mode
{
        MONO,
        RGB,
};

enum config_t
{
        LINE,
        OTHER,
};

enum class_t
{
        STRIP,
        MATRIX,
};
```

The programmer can use the seven API functions:

1. ***void SetServerAddress(char* IP, int port);***

   sets the IP address and port number to which messages are sent.

2. ***void SetMyAddress(char* IP, int port);***

   sets the IP address and port number of the program. This is needed to create a UDP socket, which is used to send the UDP message.

3. ***void ConnectChannelToServer(ChannelParameters* param);***

   sends a connect command to the server.

4. ***void DisconnectChannelFromServer(const ChannelParameters& param);***

   sends a disconnect command to the server.

5. ***void DrawAll(ChannelParameters* param, unsigned char data[]);***

   This function update all LED of the channel. The programmer must ensure the data[] is the correct size.

6. ***void DrawSingle(ChannelParameters* param, int index[], unsigned char data[]);***

   This function update only one LED of the channel. The programmer must ensure the data[] is the correct size.

7. ***void DrawMultiple(ChannelParameters* param, unsigned char pixels, int indexes[], unsigned char data[]);***

   This function updates multiple LEDs of the channel. This is useful when a couple of LEDs must be changed. However if a larger amount of LEDs needs to be changed, using the DrawAll() function is more efficient. The programmer must ensure the data[] is the correct size.

6. Testing

To test the performance of the server software, all the test programs are run with the Driver_Receive program and with a USB connected Point Runner. All test were successfully completed.

Test 1:

The Sender_Knob program sends data to the server and this data is reflected on the Driver_Receive program and on the USB connected driver board. Figure 6.1 shows the program sending the data and figure 6.2 shows the output of the Driver_Receive program.

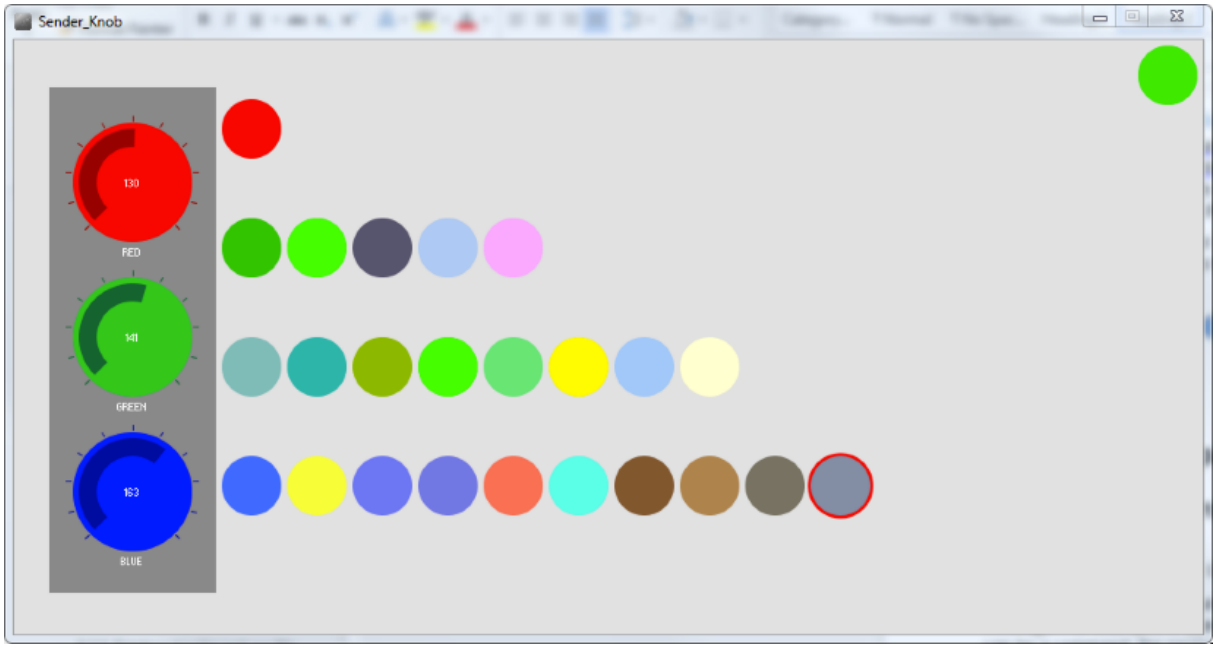


Figure 6.1 Sender_Knob data output

Figure 6.2 Sender_Knob data display

Test 2:

A webcam stream is downsampled and sent as output. Figure 6.3 shows the program sending the data and figure 6.4 shows the output of the Driver_Receive program. The display are updating at 30 frames per second.

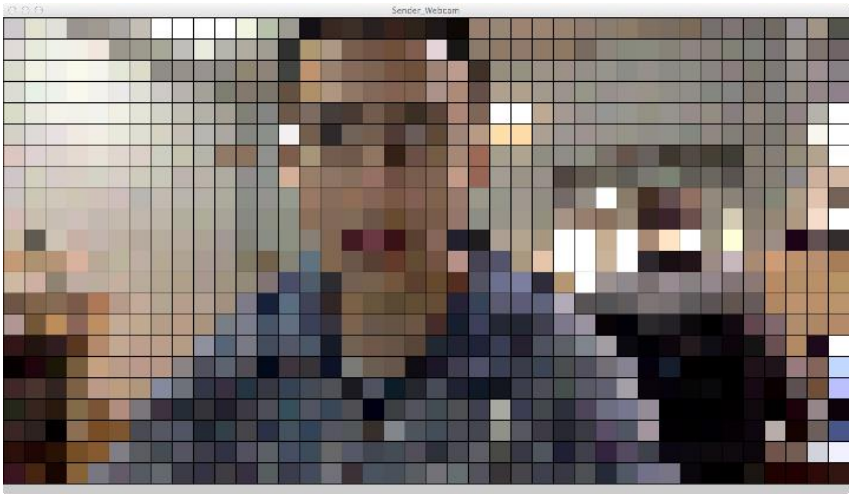


Figure 6.3 Sender_Webcam data output

Figure 6.4 Sender_Webcam data display

Test 3:

The Sender_Rainbow program sends three strip of data. The data is decoded and dispayed in the Driver_Receive program (figure 6.5). The display are updating at 60 frames per second.

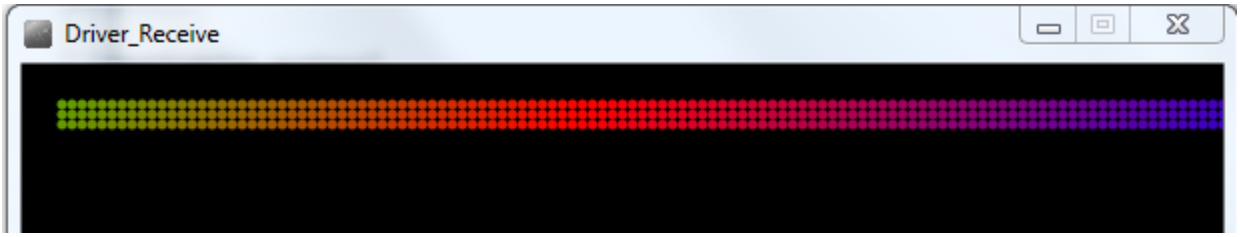
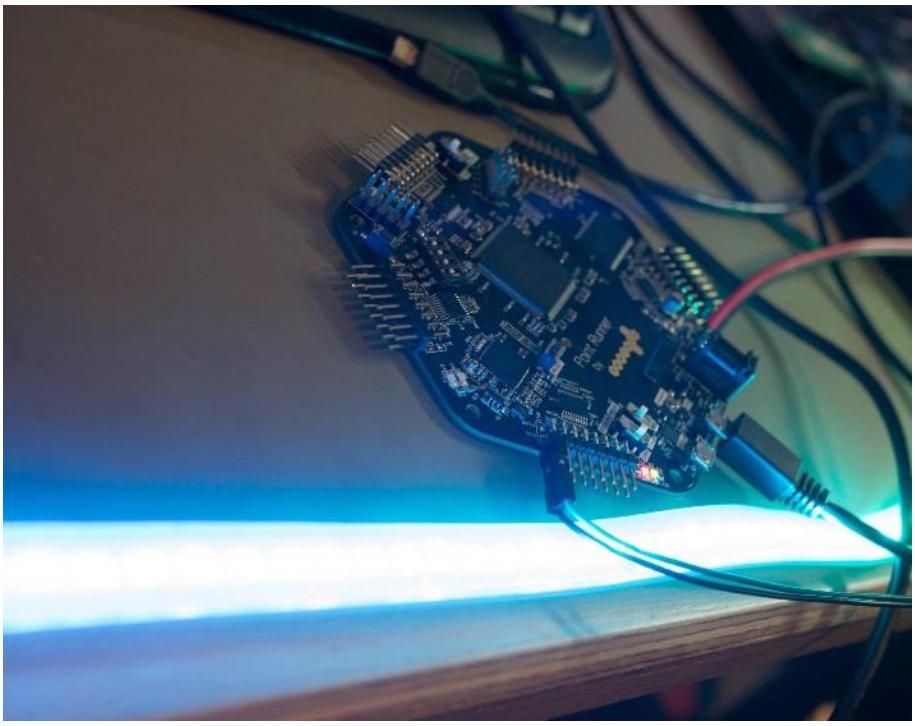


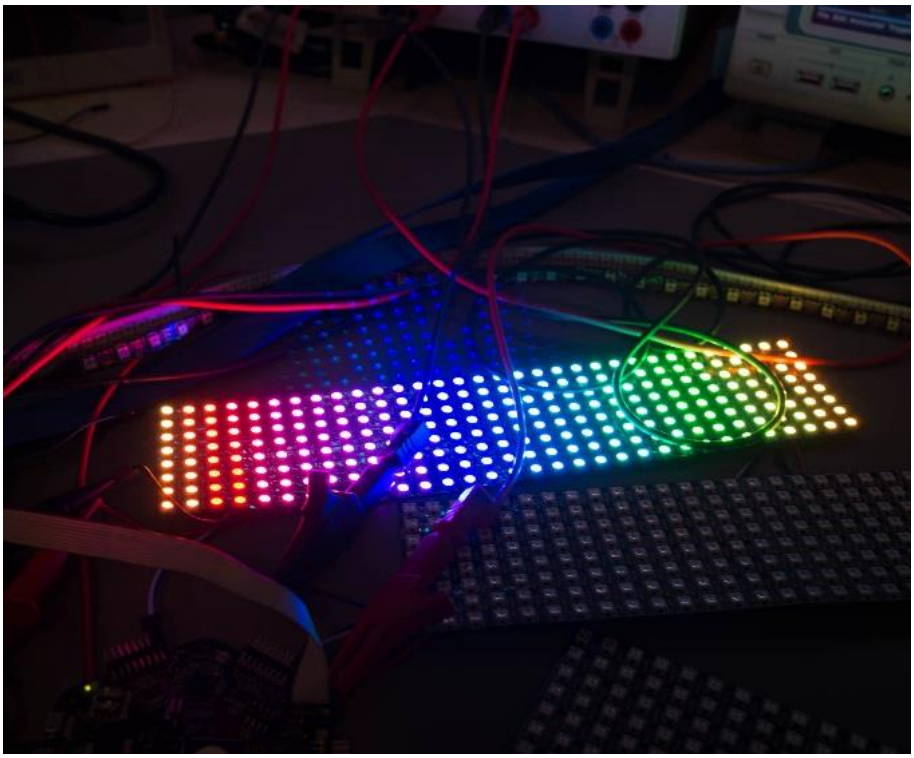
Figure 6.5 Sender_Rainbow data display

Test 4:

The Sender_Rainbow program sends data to different types of LED strips connected to the Point Runner. Picture 6.1 and 6.2 shows the LED strips being driven by the Point Runner driver board. The display are updating at 60 frames per second.



Picture 6.1 Sender_Rainbow on LED strip

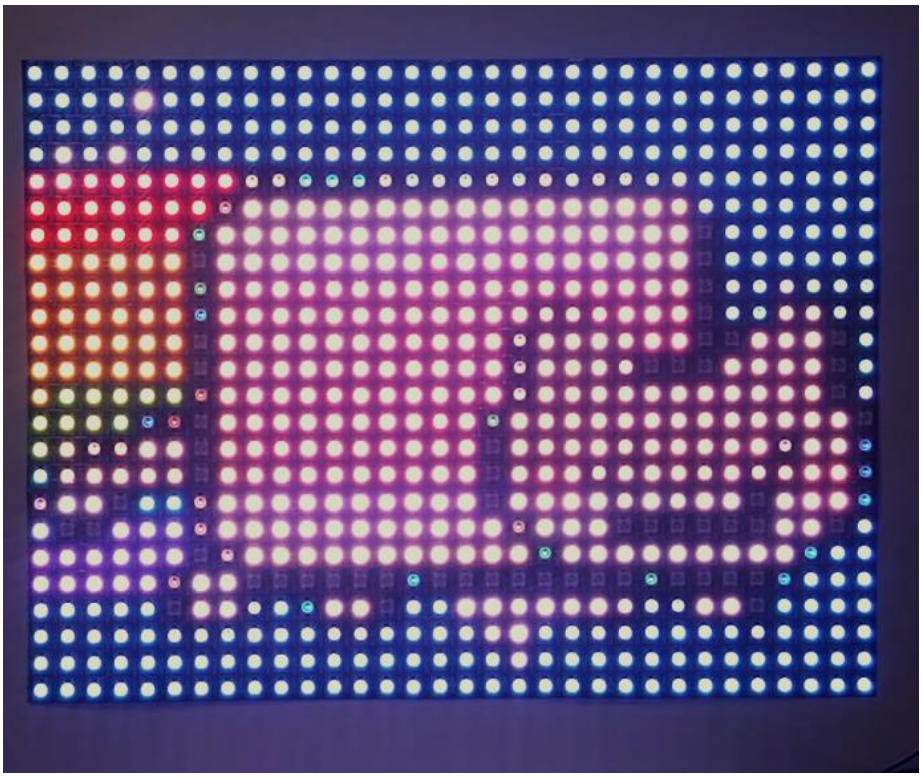


Picture 6.2 Sender_Rainbow on Matrix-style strips

Test 5:

The Sender_GIF program samples a GIF file and sends data to the server. The driver boards drives three LED strips. The LED strips are arranged in a Matrix format. Each of the panels has a width of 32 pixels and a height of 8 pixels. The total frame is therefore 32 pixels wide and 24 pixels high. The display are updating at 30 frames per second. The animation is shown on the Sender_GIF window (figure 6.6). In picture 6.3, the LED matrixes driven by the driver board is shown.

Figure 6.6 Sender_GIF data output



Picture 6.3 Sender_GIF data display

Test 6:

The API test program sends alternating red, green and blue color to the server. The Driver_Receive program displays the color data in figure 6.7.

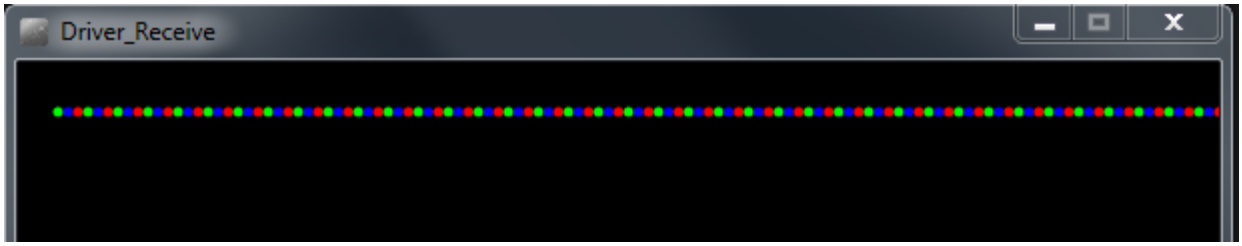


Figure 6.7 API test program data display

7. Conclusion

The goal of this project is to create a solution, which will eliminate the need for the user to remember multiple IP address and port number of driver boards connected to the network. The chosen solution is to implement a server, which will automatically routes messages sent by the user to the correct driver board. The IP address and port number of the server is the only information the user needs to remember.

Before the server software can be defined, a communication protocol is set. In this protocol, a set of instructions and message format is defined. Any program that interacts with the server must follow this protocol. Otherwise, its messages will be discarded. The server can decode incoming UDP message and the processed message to the driver board either by UDP of by USB. Various settings can be changed with an administrator program. Every incoming message is checked for its message format, source identity and settings before being processed. Various test programs are created to test the performance of the server. There were no problems detected during testing. An API library is created to provide future software developer a simple interface to create their own software. The API library hides complex process needed to send message to the server. This allows the software developer to focus more on creating great software rather on the protocol.

8. Recommendations

The data transfer between the server and driver boards are unidirectional. The driver boards can only receive data from the server but not send data to the server. Expanding the embedded software to support bi-directional data transfer on the driver boards offers better device identification, settings synchronization and more.

The administrator program can be improved to include better data processing and manipulation. Some yet-to-be implemented features should be added in the future.

Bibliography

Alex, 2007. *1.9 - Header files << Learn C++.* [Online]
Available at: http://www.learncpp.com/cpp-tutorial/19-header-files/
[Accessed 13 May 2015].

Allain, A., n.d. *Enumerated Types - Enums in C++- Cprogramming.com.* [Online]
Available at: http://www.cprogramming.com/tutorial/enum.html
[Accessed 24 February 2015].

Anon., n.d. *http://ubaa.net/shared/processing/udp/index.htm.* [Online]
Available at: http://ubaa.net/shared/processing/udp/index.htm
[Accessed 10 February 2015].

Arduino, n.d. *Arduino - APIStyleGuide.* [Online]
Available at: http://arduino.cc/en/Reference/APIStyleGuide
[Accessed 27 February 2015].

Broeders, H., 2014. *Informatie voor studenten van Harry Broeders.* [Online]
Available at: http://bd.eduweb.hhs.nl/studinfo.htm
[Accessed 6 January 2015].

Centers for Medicare & Medicaid Services - United States Department of Health and Human Services, 2008. *Selecting a development approach,* s.l.: Office of Information Service.

cplusplus.com, n.d. *<regex> - C++ Reference.* [Online]
Available at: http://www.cplusplus.com/reference/regex/
[Accessed 18 February 2015].

cplusplus.com, n.d. *shared_ptr - C++ Reference.* [Online]
Available at: http://www.cplusplus.com/reference/memory/shared_ptr/
[Accessed 26 March 2015].

Deitel, D. &., 1997. C++ How to program Second Edition. In: L. Steele, C. Trentacoste, M. Schiaparelli & L. J. Clark, eds. *C++ How to program Second Edition.* New Jersey: Prentice Hall, p. PREFACE XXXIII.

Deitel, D. &., 1997. C++ How to program Second Edition. In: L. Steele, C. Trentacoste, M. Schiaparelli & L. J. Clark, eds. *C++ How to program Second Edition.* New Jersey: Prentice Hall, p. 365.

Deitel, D. &., 1997. C++ How to program Second Edition. In: L. Steele, C. Trentacoste, M. Schiaparelli & L. J. Clark, eds. *C++ How to program Second Edition.* New Jersey: Prentice Hall, p. 520.

Deitel, D. &., 1997. C++ How to Program Second Edition. In: L. Steele, C. Trentacoste, M. Schiaparelli & L. J. Clark, eds. *C++ How to Program Second Edition.* New Jersey: Prentice Hall, pp. 5-45.

Dési, A., 2011. *LED strip lights are all the rage.* [Online]
Available at: http://www.property24.com/articles/led-strip-lights-are-all-the-

rage/13656
[Accessed 07 April 2015].

Goyvaerts, J., 2013. *Regular Expressions Quick Start.* [Online]
Available at: http://www.regular-expressions.info/quickstart.html
[Accessed 16 February 2015].

HectorLasso, 2002. *UNIX Socket FAQ (Page 1)/ UNIX Socket FAQ.* [Online]
Available at: http://developerweb.net/viewforum.php?id=70
[Accessed 23 February 2015].

Intel Corporation, n.d. *Intel Edison - One tiny platform, endless possibilities.* [Online]
Available at: http://www.intel.com.au/content/www/au/en/do-it-yourself/edison.html
[Accessed 12 May 2015].

Kickstarter Inc., n.d. *What is kickstarter - Kickstarter.* [Online]
Available at: https://www.kickstarter.com/hello?ref=footer
[Accessed 12 May 2015].

Krzyzanowski, P., 2015. *CS 417 Documents.* [Online]
Available at: https://www.cs.rutgers.edu/~pxk/417/notes/sockets/udp.html
[Accessed 17 February 2015].

Maneas, S.-E., 2014. *Java Map Example.* [Online]
Available at: http://examples.javacodegeeks.com/java-basics/java-map-example/
[Accessed 13 February 2015].

Oracle, 2011. *HashMap (Java Platform SE 6).* [Online]
Available at:
http://docs.oracle.com/javase/6/docs/api/java/util/HashMap.html#keySet()
[Accessed 13 February 2015].

O'Steen, P., n.d.
http://cs.baylor.edu/~donahoo/practical/CSockets/WindowsSockets.pdf. [Online]
Available at: http://cs.baylor.edu/~donahoo/practical/CSockets/WindowsSockets.pdf
[Accessed 26 February 2015].

Overland, B., 2013. *Be Smart About C++11 Smart Pointers.* [Online]
Available at: http://www.informit.com/articles/article.aspx?p=2085179
[Accessed 06 March 2015].

Paul, J., 2012. *How to convert Char to String in Java.* [Online]
Available at: http://javarevisited.blogspot.com.au/2012/02/how-to-convert-char-to-string-in-java.html
[Accessed 10 February 2015].

Pohl, I. & Kelley, A., 2009. De programmeertaal C. In: L. Schoofs, ed. *De programmeertaal C.* Renewed 4th ed. Amsterdam: Addison Wesley Longman, pp. 2-3.

Processing Foundation, n.d. *Processing.org.* [Online]
Available at: https://processing.org/
[Accessed 10 April 2015].

Schlegel, A., n.d. *controlP5 (Javadocs: controlP5).* [Online]
Available at: http://www.sojamo.de/libraries/controlP5/reference/index.html
[Accessed 10 February 2015].

Shiffman, D., 2011. *Daniel Shiffman.* [Online]
Available at: http://shiffman.net/2011/12/22/night-3-regular-expressions-in-processing/
[Accessed 25 March 2015].

Skinner, G., n.d. *RegExr: Learn, Build, & Test RegEx.* [Online]
Available at: http://www.regexr.com/
[Accessed 18 February 2015].

Stroustrup, B., 2000. *The C++ Programming Language.* Special ed. New Jersey: Addison-Wesley Professional.

Tekt Industries Pty. Ltd., 2015. *Tekt Industries Pty. Ltd. >> Brands.* [Online]
Available at: www.tektindustries.com/brands/
[Accessed 12 May 2015].

Tekt Industries Pty. Ltd., 2015. *Tekt Industries Pty. Ltd. >> Engineering Services.* [Online]
Available at: www.tektindustries.com/engineeringservices/
[Accessed 12 May 2015].

Tweed, K., 2013. *$10 LED Price War Heats Up the Lighting Market.* [Online]
Available at: https://www.greentechmedia.com/articles/read/10-LED-Price-War-Heats-Up-The-Lighting-Market
[Accessed 07 April 2015].

Appendix 1: Communication protocol

NETWORKED LED DRIVER SYSTEM PROTOCOLS

SENDER

This is documentation of the sender to server protocols used for the networked LED driver system.

SENDER TO SERVER DATA PROTOCOL

Strip

Addressable Mono:

BYTE																			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	(8+n*2)	(8+n*2)+1	(8+n*2)+2	(8+n*2)+3	(8+n*2)+4	(8+n*2)+...
>	(Channel)		/	ALL	/	V0	V1	V2	V3	V4	V5	V6	V7	V8	...					
>	(Channel)		/	SINGLE	/	INDEX		/	V											
>	(Channel)		/	MULTIPLE	/	PIXELS	/	INDEX0		INDEX1		INDEX2		...	INDEXn		/	V0	V1	...

(Channel)	Channel number. 16-bit (0-65535)
ALL	0x00
SINGLE	0x01
MULTIPLE	0x02
Vn	8-bit brightness value
INDEX	16-bit value to indicate the position of the LED pixel
PIXELS	8-bit. Number of pixels to be changed
V	8-bit brightness value
INDEXn	16- bit value to indicate the position of the LED pixel

Addressable RGB:

BYTE																			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	(8+n*2)	(8+n*2)+1	(8+n*2)+2	(8+n*2)+3	(8+n*2)+4	(8+n*2)+...
>	(Channel)		/	ALL	/	R0	G0	B0	R1	G1	B1	R2	G2	B2	...					
>	(Channel)		/	SINGLE	/	INDEX		/	R	G	B									
>	(Channel)		/	MULTIPLE	/	PIXELS	/	INDEX0		INDEX1		INDEX2		...	INDEXn	/	R0	G0	B0	...

(Channel)	Channel number. 16-bit (0-65535)
ALL	0x00
SINGLE	0x01
MULTIPLE	0x02
Rn	8-bit red color value
Gn	8-bit green color value
Bn	8-bit blue color value
INDEX	16-bit value to indicate the position of the LED pixel
PIXELS	8-bit. Number of pixels to be changed
R	8-bit red color value
G	8-bit green color value
B	8-bit blue color value
INDEXn	16- bit value to indicate the position of the LED pixel

Non-addressable mono:

BYTE				
0	1	2	3	4
>	(Channel)		/	V

(Channel)	Channel number. 16-bit (0-65535)
V	8-bit brightness value

Non-addressable RGB:

BYTE						
0	1	2	3	4	5	6
>	(Channel)		/	R	G	B

(Channel)	Channel number. 16-bit (0-65535)
R	8-bit red color value
G	8-bit green color value
B	8-bit blue color value

Matrix

Mono:

BYTE																									
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	(8+n*4)	(8+n*4)+1	(8+n*4)+2	(8+n*4)+3	(8+n*4)+4	(8+n*4)+5	(8+n*4)+6	(8+n*4)+...
>	(Channel)		/	ALL	/	V0	V1	V2	V3	V4	V5	V6	V7	V8	...									
>	(Channel)		/	SINGLE	/	ROW		/	COLUMN		/	V												
>	(Channel)		/	MULTIPLE	/	PIXELS	/	ROW0		COLUMN0		ROW1		COLUMN1		...	ROWn		COLUMNn		/	V0	V1	...

(Channel)	16-bit (0-65535)
ALL	0x00
SINGLE	0x01
MULTIPLE	0x02
Vn	8-bit brightness value
ROW	16-bit value to indicate the X position of the LED pixel
COLUMN	16-bit value to indicate the Y position of the LED pixel
PIXELS	8-bit. Number of pixels to be changed
V	8-bit brightness value
ROWn	16-bit value to indicate the X position of the LED pixel
COLUMNn	16-bit value to indicate the Y position of the LED pixel

RGB:

BYTE																									
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	(8+n*4)	(8+n*4)+1	(8+n*4)+2	(8+n*4)+3	(8+n*4)+4	(8+n*4)+5	(8+n*4)+6	(8+n*4)+...
>	(Channel)		/	ALL	/	R0	G0	B0	R1	G1	B1	R2	G2	B2	...									
>	(Channel)		/	SINGLE	/	ROW		/	COLUMN		/	R	G	B										
>	(Channel)		/	MULTIPLE	/	PIXELS	/	ROW0		COLUMN0		ROW1		COLUMN1		...	ROWn		COLUMNn		/	R0	G0	...

(Channel)	16-bit (0-65535)
ALL	0x00
SINGLE	0x01
MULTIPLE	0x02
Rn	8-bit red color value
Gn	8-bit green color value
Bn	8-bit blue color value
ROW	16-bit value to indicate the X position of the LED pixel
COLUMN	16-bit value to indicate the Y position of the LED pixel
PIXELS	8-bit. Number of pixels to be changed
ROWn	16-bit value to indicate the X position of the LED pixel
COLUMNn	16-bit value to indicate the Y position of the LED pixel

SENDER TO SERVER COMMAND PROTOCOL

BYTE														
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
*	SENDER_CONNECT	/	(Channel)		/	(Color mode)	/	CLASS_STRIP	/	(Length)		/	(Type)	/	(Configuration)
*	SENDER_CONNECT	/	(Channel)		/	(Color mode)	/	CLASS_MATRIX	/	(Width)		/	(Height)		
*	SENDER_DISCONNECT	/	(Channel)												

SENDER_CONNECT	0x00
SENDER_DISCONNECT	0x02
SENDER_RECONFIG	0x01
(Channel)	16-bit (0 – 65535)
COLOR_MONO	0x06
COLOR_RGB	0x07
CLASS_STRIP	0x00
CLASS_MATRIX	0x01
(Length)	16-bit (0-65535)
(Width)	16-bit (0-65535)
TYPE_ADDRESSABLE	0x02
TYPE_NONADDRESSABLE	0x03
(Height)	16-bit (0-65535)
CONFIG_LINE	0x04
CONFIG_OTHER	0x05

SERVER TO SENDER PROTOCOL

The server can send nine types of feedback messages back to the sender:

"Wrong format"
“Access denied”
“Not registered”
"No driver connected to this channel"
"Connect successful"
"You're already connected to this channel"
"Channel is already taken"
"Reconfiguration successful"
"Disconnect successful"

DRIVER

This is documentation of the driver to server protocols used for the networked LED driver system.

DRIVER TO SERVER DATA PROTOCOL

BYTE					
0	1	2	3	4	5
/	(Channel)		/	(Length)	

(Channel)	Channel number. 16-bit
(Length)	Length of the strip. 16-bit

DRIVER TO SERVER COMMAND PROTOCOL

BYTE				
0	1	2	3	4
#	DRIVER_CONNECT	/	(Channel)	
#	DRIVER_DISCONNECT	/	(Channel)	

DRIVER_CONNECT	0x00
DRIVER_DISCONNECT	0x01
(Channel)	16-bit (0-65535)

SERVER TO DRIVER PROTOCOL

The server can send nine types of feedback messages back to the driver:

"Wrong format"
“Access denied”
“Not registered”
"No driver connected to this channel"
"Connect successful"
"You're already connected to this channel"
"Channel is already taken"
"Reconfiguration successful"
"Disconnect successful"

BYTE							
0	1	2	3	4	5	6	7
/	(Channel)		/	GET_LENGTH			
/	(Channel)		/	SET_LENGTH	/	(Length)	

GET_LENGTH	0x04
SET_LENGTH	0x05
(Channel)	Channel number. 16-bit
(Length)	Strip length. 16-bit

Strip

Addressable Mono:

BYTE																			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	(8+n*2)	(8+n*2)+1	(8+n*2)+2	(8+n*2)+3	(8+n*2)+4	(8+n*2)+...
/	(Channel)		/	ALL	/	V0	V1	V2	V3	V4	V5	V6	V7	V8	...					
/	(Channel)		/	SINGLE	/	INDEX		/	V											
/	(Channel)		/	MULTIPLE	/	PIXELS	/	INDEX0		INDEX1		INDEX2		...	INDEXn		/	V0	V1	...

(Channel)	Channel number. 16-bit (0-65535)
ALL	0x00
SINGLE	0x01
MULTIPLE	0x02
Vn	8-bit brightness value
INDEX	16-bit value to indicate the position of the LED pixel
PIXELS	8-bit. Number of pixels to be changed
V	8-bit brightness value
INDEXn	16- bit value to indicate the position of the LED pixel

Addressable RGB:

BYTE																			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	(8+n*2)	(8+n*2)+1	(8+n*2)+2	(8+n*2)+3	(8+n*2)+4	(8+n*2)+...
/	(Channel)		/	ALL	/	R0	G0	B0	R1	G1	B1	R2	G2	B2	...					
/	(Channel)		/	SINGLE	/	INDEX		/	R	G	B									
/	(Channel)		/	MULTIPLE	/	PIXELS	/	INDEX0		INDEX1		INDEX2		...	INDEXn	/	R0	G0	B0	...

(Channel)	Channel number. 16-bit (0-65535)
ALL	0x00
SINGLE	0x01

MULTIPLE	0x02
Rn	8-bit red color value
Gn	8-bit green color value
Bn	8-bit blue color value
INDEX	16-bit value to indicate the position of the LED pixel
PIXELS	8-bit. Number of pixels to be changed
R	8-bit red color value
G	8-bit green color value
B	8-bit blue color value
INDEXn	16- bit value to indicate the position of the LED pixel

Non-addressable Mono:

BYTE				
0	1	2	3	4
/	(Channel)		/	V

(Channel)	Channel number. 16-bit (0-65535)
V	8-bit brightness value

Non-addressable RGB:

BYTE						
0	1	2	3	4	5	6
/	(Channel)		/	R	G	B

(Channel)	Channel number. 16-bit (0-65535)
R	8-bit red color value
G	8-bit green color value
B	8-bit blue color value

Matrix

Mono:

BYTE																							
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	(8+n*4)	(8+n*4)+1	(8+n*4)+2	(8+n*4)+3	(8+n*4)+4	(8+n*4)+5	(8+n*4)+6	(8+n*4)+...
/	(Channel)		/	ALL	/	V0	V1	V2	V3	V4	V5	V6	V7	V8	...									
/	(Channel)		/	SINGLE	/	ROW		/	COLUMN		/	V												
/	(Channel)		/	MULTIPLE	/	PIXELS	/	ROW0		COLUMN0		ROW1		COLUMN1		...	ROWn		COLUMNn		/	V0	V1	...

(Channel)	16-bit (0-65535)
ALL	0x00
SINGLE	0x01
MULTIPLE	0x02
Vn	8-bit brightness value
ROW	16-bit value to indicate the X position of the LED pixel
COLUMN	16-bit value to indicate the Y position of the LED pixel
PIXELS	8-bit. Number of pixels to be changed
V	8-bit brightness value
ROWn	16-bit value to indicate the X position of the LED pixel
COLUMNn	16-bit value to indicate the Y position of the LED pixel

RGB:

BYTE																							
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	(8+n*4)	(8+n*4)+1	(8+n*4)+2	(8+n*4)+3	(8+n*4)+4	(8+n*4)+5	(8+n*4)+6	(8+n*4)+...
/	(Channel)		/	ALL	/	R0	G0	B0	R1	G1	B1	R2	G2	B2	...									
/	(Channel)		/	SINGLE	/	ROW		/	COLUMN		/	R	G	B										
/	(Channel)		/	MULTIPLE	/	PIXELS	/	ROW0		COLUMN0		ROW1		COLUMN1		...	ROWn		COLUMNn		/	R0	G0	...

(Channel)	16-bit (0-65535)
ALL	0x00
SINGLE	0x01
MULTIPLE	0x02

Rn	8-bit red color value
Gn	8-bit green color value
Bn	8-bit blue color value
ROW	16-bit value to indicate the X position of the LED pixel
COLUMN	16-bit value to indicate the Y position of the LED pixel
PIXELS	8-bit. Number of pixels to be changed
ROWn	16-bit value to indicate the X position of the LED pixel
COLUMNn	16-bit value to indicate the Y position of the LED pixel

ADMIN

This is documentation of the administrator to server protocols used for the networked LED driver system.

ADMINISTRATOR TO SERVER DATA PROTOCOL

BYTE		
0	1	2
&	REQUEST_SENDERDATA	/
&	REQUEST_DRIVERDATA	/

REQUEST_SENDERDATA	0x02
REQUEST_DRIVERDATA	0x03

ADMINISTRATOR TO SERVER COMMAND PROTOCOL

BYTE																		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
@	ADMIN_CONNECT	/																
@	ADMIN_DISCONNECT	/																
@	CHANNELMOD	/	(Channel)		/	CLASS_STRIP	/	(Color mode)	/	(Group)		/	(Type)	/	(Config)	/	(Length)	
@	CHANNELMOD	/	(Channel)		/	CLASS_MATRIX	/	(Color mode)	/	(Group)		/	(Width)		/	(Height)		

ADMIN_CONNECT	0x00
ADMIN_DISCONNECT	0x01
CHANNELMOD	0x04
(Channel)	Channel number. 16-bit (0-65535)
CLASS_STRIP	0x00
CLASS_MATRIX	0x01

COLOR_MONO	0x06
COLOR_RGB	0x07
(Group)	Group number. 16-bit (0-65535)
TYPE_ADDRESSABLE	0x02
TYPE_NONADDRESSABLE	0x03
CONFIG_LINE	0x04
CONFIG_OTHER	0x05
(Width)	Width of the matrix. 16-bit
(Height)	Height of the matrix. 16-bit
(Length)	Length of the strip. 16-bit

SERVER TO ADMINISTRATOR PROTOCOL

The server can send seven types of feedback messages back to the administrator:

"Wrong format"
“Access denied”
"Connect successful"
"You're already connected to this channel"
"Channel is already taken"
"Reconfiguration successful"
"Disconnect successful"

All messages sent from the server to the administrator are in string format.

STRING
& DRIVER / (NAMEn) / (DRIVERn IP) / (DRIVERn Port) / strip / (COLORMODEn) / (TYPEn) / (CONFIGn) / (LENGTHn) / (NAMEn+1) / ...
& DRIVER / (NAMEn) / (DRIVERn IP) / (DRIVERn Port) / matrix / (COLORMODEn) / (WIDTHn) / (HEIGHTn) / (NAMEn+1) / (DRIVERn+1 IP) / ...
& SENDER / (NAMEn) / (SENDERn IP) / (SENDERn Port) / (CLASSn) / (NAMEn+1) / (SENDERn+1 IP) / (SENDERn+1 Port) / (CLASSn+1) / (NAMEn+2) / ...

(NAMEn)	Name of the channel in string format
(DRIVERn IP)	IP address of the driver in string format
(DRIVERn Port)	Port number of the driver in string format

(SENDERn IP)	IP address of the sender in string format
(SENDERn Port)	Port number of the sender in string format
strip	The class of the driver is strip
Matrix	The class of the driver is strip
(COLORMODEn)	The color mode used by the driver: mono RGB
(TYPEn)	The type used by the driver: addressable non-addressable
(CONFIGn)	The configuration used by the driver: line other
(LENGTHn)	The length of the strip in string format
(WIDTHn)	The width of the matrix in string format
(HEIGHTn)	The height of the matrix in string format

Appendix 2: Definitions.h source code

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

```
//
//  Definitions.h
//  Networked LED Driver System server
//
//  Created by Winer Bao on 24/02/15.
//  for Tekt Industries Pty. Ltd.
//  Copyright (c) 2015 Winer Bao. All rights reserved.
//

#ifndef __Networked_LED_Driver_System_Server_Definitions__
#define __Networked_LED_Driver_System_Server_Definitions__

#include <stdio.h>
#include <queue>
#include <vector>

/* Number of Processing threads */
const int TOTAL_SENDERDATAPROCESSINGTHREADS     = 50;
const int TOTAL_SENDERCOMMANDPROCESSINGTHREADS  = 1;
const int TOTAL_DRIVERDATAPROCESSINGTHREADS     = 1;
const int TOTAL_DRIVERCOMMANDPROCESSINGTHREADS  = 1;
const int TOTAL_ADMINDATAPROCESSINGTHREADS      = 1;
const int TOTAL_ADMINCOMMANDPROCESSINGTHREADS   = 1;

/* Server address and port number */
static const char* SERVER_ADDR = "192.168.1.27";
const int SERVER_PORT = 21234;

/* Amount of drivers connected to the server */
const int TOTAL_DRIVERS = 40;
const int STRIP_LENGTH  = 256;

/* Driver address and port number */
static const char* DRIVER_ADDR = "192.168.1.27";
const int DRIVER_PORT = 6100;

/* Message source and type identification symbol */
const char SENDERDATA   = '>';
const char SENDERCMD    = '*';
const char DRIVERDATA   = '/';
const char DRIVERCMD    = '#';
const char ADMINDATA    = '&';
const char ADMINCMD     = '@';

/* SENDERDATA data range identification */
const unsigned char ALL         = 0x00;
const unsigned char SINGLE      = 0x01;
const unsigned char MULTIPLE    = 0x02;

/* SENDERCMD definitions */
const unsigned char SENDER_CONNECT      = 0x00;
const unsigned char SENDER_DISCONNECT   = 0x02;

/* DRIVERCMD definitions */
const unsigned char DRIVER_CONNECT      = 0x00;
const unsigned char DRIVER_DISCONNECT   = 0x01;

/* ADMINDATA definitions */
const unsigned char REQUEST_SENDERDATA  = 0x02;
const unsigned char REQUEST_DRIVERDATA  = 0x03;

/* ADMINCMD definitions */
const unsigned char ADMIN_CONNECT       = 0x00;
const unsigned char ADMIN_DISCONNECT    = 0x01;
const unsigned char CHANNELMOD          = 0x04;
const unsigned char SET_LENGTH          = 0x04;
const unsigned char GET_LENGTH          = 0x05;

/* LED object parameters */
const unsigned char CLASS_STRIP         = 0x00;
const unsigned char CLASS_MATRIX        = 0x01;
const unsigned char CLASS_RAW           = 0x03;
```

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

```
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;

/* Custom types. Used for LED class and its derived classes*/
enum led_t
{
    ADDRESSABLE,
    NONADDRESSABLE,
};

enum color_mode
{
    MONO,
    RGB,
};

enum config_t
{
    LINE,
    OTHER,
};

enum class_t
{
    STRIP,
    MATRIX,
    RAW,
};

enum connection_t
{
    USB,
    UDP,
};

/* Helper functions */
extern int ByteToInt(unsigned char MSB, unsigned char LSB);

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

Appendix 3: ChannelObject.h and ChannelObject.cpp source code

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

```
//
//  ChannelObject.h
//  Networked LED Driver System server
//
//  Created by Winer Bao on 24/02/15.
//  for Tekt Industries Pty. Ltd.
//  Copyright (c) 2015 Winer Bao. All rights reserved.
//

#ifndef __Networked_LED_Driver_System_Server_ChannelObject__
#define __Networked_LED_Driver_System_Server_ChannelObject__

#define _CRT_SECURE_NO_WARNINGS

#include <stdio.h>
#include <vector>
#include "Definitions.h"
#include "USBdriver.h"

struct MyException : public std::exception
{
    std::string s;
    MyException(std::string ss) : s(ss) {}
    ~MyException() throw () {} // Updated
    const char* what() const throw() { return s.c_str(); }
};

class LED
{
public:
    LED(const char* name, const int& Ext_ch, const int& Int_ch, const color_mode& c, const class_t& ct);
    virtual ~LED();
    void setName(const char* n);
    void setExternalChannel(const int& ch);
    void setInternalChannel(const int& ch);
    void setColorMode(const color_mode& c);
    void setGroup(const int& g);
    void setClassType(const class_t& ct);
    void setDriverInfo(const std::string& IP, const int& port);
    void setSenderInfo(const std::string& IP, const int& port);
    void setConnectionType(const connection_t& con_t);
    char* getName() const;
    int getExternalChannel() const;
    int getInternalChannel() const;
    color_mode getColorMode() const;
    int getGroup() const;
    class_t getClassType() const;
    std::string getDriverIP() const;
    int getDriverPort() const;
    std::string getSenderIP() const;
    int getSenderPort() const;
    connection_t getConnectionType() const;
    virtual bool checkSenderDTformat(const std::vector<unsigned char>& bufIn) const = 0;
    virtual void decodeUDP(std::vector<unsigned char>& TXbuf, const std::vector<unsigned char>& bufIn) =↙
     0;
    virtual void decodeUSB(std::vector<unsigned char>& TXbuf, const std::vector<unsigned char>& bufIn) =↙
     0;

private:
    char* channelName;
    int external_channel;
    int internal_channel;
    color_mode color;
    int group;
    class_t classType;
    std::string DriverIP;
    int DriverPort;
    std::string SenderIP;
    int SenderPort;
    connection_t connectType;
};
```

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

```
class Strip : public LED
{
public:
    Strip(const char* name, const int& Ext_ch, const int& Int_ch, const int& len, const led_t& type,
    const color_mode& c, const config_t config);
    virtual ~Strip();
    void setLength(const int& len);
    void setType(const led_t& type);
    void setConfig(const config_t& config);
    int getLength() const;
    led_t getType() const;
    config_t getConfig() const;

private:
    int pixelLength;
    led_t ledtype;
    config_t configuration;
};


class Matrix : public LED
{
public:
    Matrix(const char* name, const int& Ext_ch, const int& Int_ch, const int& wd, const int& ht, const
    color_mode& c);
    virtual ~Matrix();
    void setWidth(const int& wd);
    void setHeigth(const int& ht);
    void setDimensions(const int& wd, const int& ht);
    int getWidth() const;
    int getHeight() const;

private:
    int width;
    int height;
};

class Raw : public LED
{
public:
    Raw(const char* name, const int& Ext_ch, const int& Int_ch);
    virtual ~Raw();
    virtual bool checkSenderDTformat(const std::vector<unsigned char>& bufIn) const override;
    virtual void decodeUDP(std::vector<unsigned char>& TXbuf, const std::vector<unsigned char>& bufIn)
    override;
    virtual void decodeUSB(std::vector<unsigned char>& TXbuf, const std::vector<unsigned char>& bufIn)
    override;

private:
};

class AddressableStrip : public Strip
{
public:
    AddressableStrip(const char* name, const int& Ext_ch, const int& Int_ch, const int& len, const
    color_mode& c, const config_t config);
    virtual ~AddressableStrip();

private:
};

class NonAddressableStrip : public Strip
{
public:
    NonAddressableStrip(const char* name, const int& Ext_ch, const int& Int_ch, const int& len, const
    color_mode& c, const config_t config);
    virtual ~NonAddressableStrip();

private:
};

class AddressableStripMono : public AddressableStrip
```

```
{
public:
    AddressableStripMono(const char* name, const int& Ext_ch, const int& Int_ch);
    AddressableStripMono(const char* name, const int& Ext_ch, const int& Int_ch, const int& len, const
    config_t config);
    virtual ~AddressableStripMono();
    virtual bool checkSenderDTformat(const std::vector<unsigned char>& bufIn) const override;
    virtual void decodeUDP(std::vector<unsigned char>& TXbuf, const std::vector<unsigned char>& bufIn)
    override;
    virtual void decodeUSB(std::vector<unsigned char>& TXbuf, const std::vector<unsigned char>& bufIn)
    override;

private:
};

class AddressableStripRGB : public AddressableStrip
{
public:
    AddressableStripRGB(const char* name, const int& Ext_ch, const int& Int_ch);
    AddressableStripRGB(const char* name, const int& Ext_ch, const int& Int_ch, const int& len, const
    config_t config);
    virtual ~AddressableStripRGB();
    virtual bool checkSenderDTformat(const std::vector<unsigned char>& bufIn) const override;
    virtual void decodeUDP(std::vector<unsigned char>& TXbuf, const std::vector<unsigned char>& bufIn)
    override;
    virtual void decodeUSB(std::vector<unsigned char>& TXbuf, const std::vector<unsigned char>& bufIn)
    override;

private:
};

class NonAddressableStripMono : public NonAddressableStrip
{
public:
    NonAddressableStripMono(const char* name, const int& Ext_ch, const int& Int_ch);
    NonAddressableStripMono(const char* name, const int& Ext_ch, const int& Int_ch, const int& len,
    const config_t config);
    virtual ~NonAddressableStripMono();
    virtual bool checkSenderDTformat(const std::vector<unsigned char>& bufIn) const override;
    virtual void decodeUDP(std::vector<unsigned char>& TXbuf, const std::vector<unsigned char>& bufIn)
    override;
    virtual void decodeUSB(std::vector<unsigned char>& TXbuf, const std::vector<unsigned char>& bufIn)
    override;

private:
};

class NonAddressableStripRGB : public NonAddressableStrip
{
public:
    NonAddressableStripRGB(const char* name, const int& Ext_ch, const int& Int_ch);
    NonAddressableStripRGB(const char* name, const int& Ext_ch, const int& Int_ch, const int& len, const
     config_t config);
    virtual ~NonAddressableStripRGB();
    virtual bool checkSenderDTformat(const std::vector<unsigned char>& bufIn) const override;
    virtual void decodeUDP(std::vector<unsigned char>& TXbuf, const std::vector<unsigned char>& bufIn)
    override;
    virtual void decodeUSB(std::vector<unsigned char>& TXbuf, const std::vector<unsigned char>& bufIn)
    override;

private:
};

class MatrixMono : public Matrix
{
public:
    MatrixMono(const char* name, const int& Ext_ch, const int& Int_ch);
    MatrixMono(const char* name, const int& Ext_ch, const int& Int_ch, const int& wd, const int& ht);
    virtual ~MatrixMono();
    virtual bool checkSenderDTformat(const std::vector<unsigned char>& bufIn) const override;
    virtual void decodeUDP(std::vector<unsigned char>& TXbuf, const std::vector<unsigned char>& bufIn)
    override;
```

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

```
    virtual void decodeUSB(std::vector<unsigned char>& TXbuf, const std::vector<unsigned char>& bufIn) ↙
    override;

private:
};

class MatrixRGB : public Matrix
{
public:
    MatrixRGB(const char* name, const int& Ext_ch, const int& Int_ch);
    MatrixRGB(const char* name, const int& Ext_ch, const int& Int_ch, const int& wd, const int& ht);
    virtual ~MatrixRGB();
    virtual bool checkSenderDTformat(const std::vector<unsigned char>& bufIn) const override;
    virtual void decodeUDP(std::vector<unsigned char>& TXbuf, const std::vector<unsigned char>& bufIn) ↙
    override;
    virtual void decodeUSB(std::vector<unsigned char>& TXbuf, const std::vector<unsigned char>& bufIn) ↙
    override;

private:
};


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

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

```
//
//  ChannelObject.cpp
//  Networked LED Driver System server
//
//  Created by Winer Bao on 24/02/15.
//  for Tekt Industries Pty. Ltd.
//  Copyright (c) 2015 Winer Bao. All rights reserved.
//

#include "ChannelObject.h"
#include <assert.h>

using namespace std;

/*
*  LED class is the base class.
*  It stores the channel, group, color mode, IPs and port numbers of the object.
*/

// Constructor of LED object
LED::LED(const char* name, const int& Ext_ch, const int& Int_ch, const color_mode& c, const class_t& ct)
{
    channelName = new char[strlen(name) + 1];

    if (channelName == nullptr)
    {
        throw MyException("Space allocation failed");
    }
    if (Ext_ch < 0 || Int_ch < 0)
    {
        throw MyException("Channel number is lower than 0");
    }
    strcpy(channelName, name);
    setExternalChannel(Ext_ch);
    setInternalChannel(Int_ch);
    setColorMode(c);
    setClassType(ct);

    DriverIP = "0.0.0.0";
    DriverPort = 0;
    SenderIP = "0.0.0.0";
    SenderPort = 0;
}

// Destructor of LED object
LED::~LED()
{
    delete[] channelName;
    channelName = nullptr;
}

// Sets the name of LED object.
void LED::setName(const char* n)
{
    delete channelName;
    channelName = new char[strlen(n) + 1];

    if (channelName == nullptr)
    {
        throw MyException("Space allocation failed");
    }
    strcpy(channelName, n);
}

// Sets the external channel number of LED object.
// The external channel number is the channel number visible to the sender.
// This is a mapped number to avoid number collision when multiple driver boards are connected.
void LED::setExternalChannel(const int& ch)
{
    if (ch < 0)
    {
        throw MyException("External channel number cannot be lower than 0");
```

```
    }
    external_channel = ch;
}

// Sets the internal channel number of LED object.
// This internal channel is used by the driver boards itself to identify the channel.
void LED::setInternalChannel(const int& ch)
{
    if (ch < 0)
    {
        throw MyException("Internal channel number cannot be lower than 0");
    }
    internal_channel = ch;
}

// Sets the color mode of the LED object
void LED::setColorMode(const color_mode& c)
{
    color = c;
}

// Sets the group of LED object
void LED::setGroup(const int& g)
{
    if (group < 0)
    {
        throw MyException("Invalid group number");
    }
    group = g;
}

// Sets the class type of the LED object
void LED::setClassType(const class_t& ct)
{
    classType = ct;
}

// Save contact socket of Channel Driver
void LED::setDriverInfo(const std::string& IP, const int& port)
{
    DriverIP = IP;
    DriverPort = port;
}

// Save contact socket of Channel Sender
void LED::setSenderInfo(const std::string& IP, const int& port)
{
    SenderIP = IP;
    SenderPort = port;
}

// Sets the connection type used by the LED object
void LED::setConnectionType(const connection_t& con_t)
{
    connectType = con_t;
}

// Return the name of the object as char pointer
char* LED::getName() const
{
    return channelName;
}

// Returns the external channel number of LED object.
// The external channel number is the channel number visible to the sender.
// This is a mapped number to avoid number collision when multiple driver boards are connected.
int LED::getExternalChannel() const
{
    return external_channel;
}

// Returns the internal channel number of LED object.
```

\\psf\Dropbox\Elektrotechniek 4de\Afstuderen ...\Networked LED Driver System\ChannelObject.cpp 3

```
// This internal channel is used by the driver boards itself to identify the channel.
int LED::getInternalChannel() const
{
    return internal_channel;
}

// Returns the color mode of LED object
color_mode LED::getColorMode() const
{
    return color;
}

// Returns the group of LED object
int LED::getGroup() const
{
    return group;
}

// Return the class type of the LED object
class_t LED::getClassType() const
{
    return classType;
}

// Return the IP address of the driver for the LED object
std::string LED::getDriverIP() const
{
    return DriverIP;
}

// Returns the port number of the driver for the LED object
int LED::getDriverPort() const
{
    return DriverPort;
}

// Returns the IP address of the sender for the LED object
std::string LED::getSenderIP() const
{
    return SenderIP;
}

// Returns the port number of the sender for the LED object
int LED::getSenderPort() const
{
    return SenderPort;
}

// Returns the connection type of the LED object
connection_t LED::getConnectionType() const
{
    return connectType;
}



/*
*  Strip class is a derived class from LED class.
*/

// Constructor of Strip object.
Strip::Strip(const char* name, const int& Ext_ch, const int& Int_ch, const int& len, const led_t& type, ↙
    const color_mode& c, const config_t config) : LED(name, Ext_ch, Int_ch, c, STRIP)
{
    setType(type);
    setLength(len);
    setConfig(config);
}

// Destructor of Strip object
Strip::~Strip(){}
```

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

```
// Changes the length of Strip object.
void Strip::setLength(const int& len)
{
    if (len < 0)
    {
        throw MyException("Length cannot be lower than 0");
    }
    pixelLength = len;
}

// Sets the type of led strip used
void Strip::setType(const led_t& type)
{
    ledtype = type;
}

// Sets the configuration setup of Strip object
void Strip::setConfig(const config_t& config)
{
    configuration = config;
}

// Return the length of Strip object.
int Strip::getLength() const
{
    return pixelLength;
}

// Return the type of Strip object
led_t Strip::getType() const
{
    return ledtype;
}

// Returns the configuration setup of Strip object
config_t Strip::getConfig() const
{
    return configuration;
}



/*
*  Matrix class is a derived class from LED class. This class controls and manages
*  the LED set in a matrix. A matrix has a width and a length.
*/

//Constructor of Matrix object
Matrix::Matrix(const char* name, const int& Ext_ch, const int& Int_ch, const int& wd, const int& ht,    ↙
    const color_mode& c) : LED(name, Ext_ch, Int_ch, c, MATRIX)
{
    setWidth(wd);
    setHeigth(ht);
}

// Destructor of Matrix object
Matrix::~Matrix(){}

// Changes the width of Matrix object.
void Matrix::setWidth(const int& wd)
{
    if (wd < 0)
    {
        throw MyException("Width cannot be lower than 0");
    }
    width = wd;
}

// Changes the heigth of Matrix object.
void Matrix::setHeigth(const int& ht)
{
    if (ht < 0)
```

```
    {
        throw MyException("Height cannot be lower than 0");
    }
    height = ht;
}

// Changes the width and heigth of Matrix object.
void Matrix::setDimensions(const int& wd, const int& ht)
{
    if (wd < 0)
    {
        throw MyException("Width cannot be lower than 0");
    }
    if (ht < 0)
    {
        throw MyException("Height cannot be lower than 0");
    }
    width = wd;
    height = ht;
}

// Return the width of Matrix object.
int Matrix::getWidth() const
{
    return width;
}

// Return the height of Matrix object.
int Matrix::getHeight() const
{
    return height;
}



/*
*  Raw class is a derived class from LED class. This class only
*  contains raw data.
*/

//Constructor of Raw object
Raw::Raw(const char* name, const int& Ext_ch, const int& Int_ch) : LED(name, Ext_ch, Int_ch, RGB, RAW){}

//Destructor of Raw object
Raw::~Raw(){}

// Checks if the messsage format is correct before processing data
// Checks if the received message has the proper format according to the protocol.
// Returns false when format is not correct.
bool Raw::checkSenderDTformat(const std::vector<unsigned char>& bufIn) const
{
    if (bufIn.size() > 4 && bufIn[3] == '/')
    {
        return true;
    }
    return false;
}

// Decode data sent by the sender via UDP
void Raw::decodeUDP(std::vector<unsigned char>& TXbuf, const std::vector<unsigned char>& bufIn)
{
    for (std::vector<unsigned char>::const_iterator it = bufIn.begin() + 4; it != bufIn.end(); it++)
    {
        TXbuf.push_back(*it);
    }
}

// Decode data sent by the sender via USB
void Raw::decodeUSB(std::vector<unsigned char>& TXbuf, const std::vector<unsigned char>& bufIn)
{
    // Implement in the future
}
```

\\psf\Dropbox\Elektrotechniek 4de\Afstuderen ...\Networked LED Driver System\ChannelObject.cpp 6

```
/*
*  AddressableStrip is a derived class from Strip class. Each LED in the strip
*  are individually addressable.
*/

// Constructor of AddressableStrip object.
AddressableStrip::AddressableStrip(const char* name, const int& Ext_ch, const int& Int_ch, const int&
    len, const color_mode& c, const config_t config) : Strip(name, Ext_ch, Int_ch, len, ADDRESSABLE, c,
    config){}

// Destructor of AddressableStrip object
AddressableStrip::~AddressableStrip(){}



/*
*  NonAddressableStrip is a derived class from Strip class. The LEDs in the strip
*  are NOT individually addressable.
*/

// Constructor of NonAddressableStrip object.
NonAddressableStrip::NonAddressableStrip(const char* name, const int& Ext_ch, const int& Int_ch, const
    int& len, const color_mode& c, const config_t config) : Strip(name, Ext_ch, Int_ch, len,
    NONADDRESSABLE, c, config){}

// Destructor of NonAddressableStrip object
NonAddressableStrip::~NonAddressableStrip(){}



/*
*  AddressableStripMono is a derived class from AddressableStrip class. The LEDs in the strip
*  are individually addressable and produce one color.
*/

//Default  Constructor of AddressableStripMono object
AddressableStripMono::AddressableStripMono(const char* name, const int& Ext_ch, const int& Int_ch) :
    AddressableStrip(name, Ext_ch, Int_ch, 1, MONO, OTHER){}

// Constructor of AddressableStripMono object
AddressableStripMono::AddressableStripMono(const char* name, const int& Ext_ch, const int& Int_ch, const
     int& len, const config_t config) : AddressableStrip(name, Ext_ch, Int_ch, len, MONO, config){}

// Destructor of AddressableStripMono object
AddressableStripMono::~AddressableStripMono(){}

// Checks if the messsage format is correct before processing data
// Checks if the received message has the proper format according to the protocol.
// Returns 0 when format is not correct.
bool AddressableStripMono::checkSenderDTformat(const std::vector<unsigned char>& bufIn) const
{
    if (bufIn.size() == (6 + getLength()) &&bufIn[3] == '/')
    {
        if (bufIn[4] == ALL && bufIn[5] == '/')
        {
            return true;
        }
        else if (bufIn[4] == SINGLE && bufIn[5] == '/' && bufIn[8] == '/')
        {
            return true;
        }
        else if (bufIn[4] == MULTIPLE && bufIn[5] == '/' && bufIn[7] == '/')
        {
            if (bufIn[8 + (bufIn[6] * 2)] == '/')
            {
                return true;
            }
        }
    }
```

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

```
    return false;
}

// Decode data sent by the sender via UDP
void AddressableStripMono::decodeUDP(std::vector<unsigned char>& TXbuf, const std::vector<unsigned char>&
    & bufIn)
{
    switch (bufIn[4])
    {
        case ALL:
        {
            int data = 6;

            TXbuf.push_back('/');
            TXbuf.push_back((getInternalChannel() >> 8) & 0xFF);
            TXbuf.push_back(getInternalChannel()& 0xFF);
            TXbuf.push_back('/');
            TXbuf.push_back(ALL);
            TXbuf.push_back('/');
            for (int i = 0; i < getLength(); i++)
            {
                TXbuf.push_back(bufIn[data++]);
            }
            break;
        }

        case SINGLE:
        {
            TXbuf.push_back('/');
            TXbuf.push_back((getInternalChannel() >> 8) & 0xFF);
            TXbuf.push_back(getInternalChannel() & 0xFF);
            TXbuf.push_back('/');
            TXbuf.push_back(SINGLE);
            TXbuf.push_back('/');
            TXbuf.push_back(bufIn[6]);
            TXbuf.push_back(bufIn[7]);
            TXbuf.push_back('/');
            TXbuf.push_back(bufIn[9]);
            break;
        }

        case MULTIPLE:
        {
            int i;
            int pixels = bufIn[6];
            int msgData = (8 + pixels * 2);

            TXbuf.push_back('/');
            TXbuf.push_back((getInternalChannel() >> 8) & 0xFF);
            TXbuf.push_back(getInternalChannel() & 0xFF);
            TXbuf.push_back('/');
            TXbuf.push_back(MULTIPLE);
            TXbuf.push_back('/');
            TXbuf.push_back(bufIn[6]);
            TXbuf.push_back('/');

            for (i = 8; i < msgData; i = i + 2)
            {
                TXbuf.push_back(bufIn[i]);
                TXbuf.push_back(bufIn[i + 1]);
            }

            TXbuf.push_back('/');

            for (i = msgData + 1; i<(msgData + 1 + pixels); i++)
            {
                TXbuf.push_back(bufIn[i]);
            }
            break;
        }
    }
}
```

\\psf\Dropbox\Elektrotechniek 4de\Afstuderen ...\Networked LED Driver System\ChannelObject.cpp 8

```
// Decode data sent by the sender via USB
void AddressableStripMono::decodeUSB(std::vector<unsigned char>& TXbuf, const std::vector<unsigned char>&
    & bufIn)
{
    // Implement in the future
}



/*
*  AddressableStripRGB is a derived class from AddressableStrip class. The LEDs in the strip
*  are individually addressable and produce RGB color.
*/

// Default Constructor of AddressableStripRGB object
AddressableStripRGB::AddressableStripRGB(const char* name, const int& Ext_ch, const int& Int_ch) :
    AddressableStrip(name, Ext_ch, Int_ch, 1, RGB, OTHER){}

// Constructor of AddressableStripRGB object
AddressableStripRGB::AddressableStripRGB(const char* name, const int& Ext_ch, const int& Int_ch, const
    int& len, const config_t config) : AddressableStrip(name, Ext_ch, Int_ch, len, RGB, config){}

// Destructor of AddressableStripRGB object
AddressableStripRGB::~AddressableStripRGB(){}

// Checks if the messsage format is correct before processing data
// Checks if the received message has the proper format according to the protocol.
// Returns 0 when format is not correct.
bool AddressableStripRGB::checkSenderDTformat(const std::vector<unsigned char>& bufIn) const
{
    if (bufIn.size() == (6 + (getLength()*3)) && bufIn[3] == '/')
    {
        if (bufIn[4] == ALL && bufIn[5] == '/')
        {
            return true;
        }
        else if (bufIn[4] == SINGLE && bufIn[5] == '/' && bufIn[8] == '/')
        {
            return true;
        }
        else if (bufIn[4] == MULTIPLE && bufIn[5] == '/' && bufIn[7] == '/')
        {
            if (bufIn[8 + (bufIn[6] * 2)] == '/')
            {
                return true;
            }
        }
    }
    return false;
}

// Decode data sent by the sender via UDP
void AddressableStripRGB::decodeUDP(std::vector<unsigned char>& TXbuf, const std::vector<unsigned char>&
     bufIn)
{
    switch (bufIn[4])
    {
        case ALL:
        {
            int data = 6;

            TXbuf.push_back('/');
            TXbuf.push_back((getInternalChannel() >> 8) & 0xFF);
            TXbuf.push_back(getInternalChannel() & 0xFF);
            TXbuf.push_back('/');
            TXbuf.push_back(ALL);
            TXbuf.push_back('/');
            for (int i = 0; i < (getLength() * 3); i++)
            {
                TXbuf.push_back(bufIn[data++]);
            }
```

\\psf\Dropbox\Elektrotechniek 4de\Afstuderen ...\Networked LED Driver System\ChannelObject.cpp 9

```
            break;
        }

        case SINGLE:
        {
            TXbuf.push_back('/');
            TXbuf.push_back((getInternalChannel() >> 8) & 0xFF);
            TXbuf.push_back(getInternalChannel() & 0xFF);
            TXbuf.push_back('/');
            TXbuf.push_back(SINGLE);
            TXbuf.push_back('/');
            TXbuf.push_back(bufIn[6]);
            TXbuf.push_back(bufIn[7]);
            TXbuf.push_back('/');
            TXbuf.push_back(bufIn[9]);
            TXbuf.push_back(bufIn[10]);
            TXbuf.push_back(bufIn[11]);
            break;
        }

        case MULTIPLE:
        {
            int i;
            int pixels = bufIn[6];
            int RGBdata = (8 + pixels * 2);

            TXbuf.push_back('/');
            TXbuf.push_back((getInternalChannel() >> 8) & 0xFF);
            TXbuf.push_back(getInternalChannel() & 0xFF);
            TXbuf.push_back('/');
            TXbuf.push_back(MULTIPLE);
            TXbuf.push_back('/');
            TXbuf.push_back(bufIn[6]);
            TXbuf.push_back('/');

            for (i = 8; i < RGBdata; i = i + 2)
            {
                TXbuf.push_back(bufIn[i]);
                TXbuf.push_back(bufIn[i + 1]);
            }

            TXbuf.push_back('/');

            for (i = RGBdata + 1; i<(RGBdata + 1 + pixels * 3); i++)
            {
                TXbuf.push_back(bufIn[i]);
            }
            break;
        }
    }
}

// Decode data sent by the sender via USB
void AddressableStripRGB::decodeUSB(std::vector<unsigned char>& TXbuf, const std::vector<unsigned char>&↙
     bufIn)
{
    std::vector<unsigned char> RGBdata;
    switch (bufIn[4])
    {
        case ALL:
        {
            for (std::vector<unsigned char>::const_iterator it = bufIn.begin() + 6; it != bufIn.end();  ↙
    it++)
            {
                RGBdata.push_back(*it);
            }
            TXbuf = cmd_load_data(getInternalChannel(), getLength(), RGBdata);
        }

        case SINGLE:
        {
            //Implement in the future
```

```
        }

        case MULTIPLE:
        {
            //Implement in the future
        }
    }
}



/*
*  NonAddressableStripMono is a derived class from NonAddressableStrip class. The LEDs in the strip
*  are NOT individually addressable and produce one color.
*/

// Constructor of NonAddressableStripMono object
NonAddressableStripMono::NonAddressableStripMono(const char* name, const int& Ext_ch, const int& Int_ch)↙
     : NonAddressableStrip(name, Ext_ch, Int_ch, 1, MONO, OTHER){}

// Constructor of NonAddressableStripMono object
NonAddressableStripMono::NonAddressableStripMono(const char* name, const int& Ext_ch, const int& Int_ch,↙
     const int& len, const config_t config) : NonAddressableStrip(name, Ext_ch, Int_ch, len, MONO,      ↙
    config){}

// Destructor of NonAddressableStripMono object
NonAddressableStripMono::~NonAddressableStripMono(){}

// Checks if the messsage format is correct before processing data
// Checks if the received message has the proper format according to the protocol.
// Returns 0 when format is not correct.
bool NonAddressableStripMono::checkSenderDTformat(const std::vector<unsigned char>& bufIn) const
{
    if (bufIn.size() == 5 && bufIn[3] == '/')
    {
        return true;
    }
    return false;
}

// Decode data sent by the sender via UDP
void NonAddressableStripMono::decodeUDP(std::vector<unsigned char>& TXbuf, const std::vector<unsigned   ↙
    char>& bufIn)
{
    TXbuf.push_back('/');
    TXbuf.push_back((getInternalChannel() >> 8) & 0xFF);
    TXbuf.push_back(getInternalChannel() & 0xFF);
    TXbuf.push_back('/');
    TXbuf.push_back(bufIn[4]);
}

// Decode data sent by the sender via USB
void NonAddressableStripMono::decodeUSB(std::vector<unsigned char>& TXbuf, const std::vector<unsigned   ↙
    char>& bufIn)
{
    // Implement in the future
}



/*
*  NonAddressableStripRGB is a derived class from NonAddressableStrip class. The LEDs in the strip
*  are NOT individually addressable and produce RGB color.
*/

// Default Constructor of NonAddressableStripRGB
NonAddressableStripRGB::NonAddressableStripRGB(const char* name, const int& Ext_ch, const int& Int_ch) :↙
     NonAddressableStrip(name, Ext_ch, Int_ch, 1, RGB, OTHER){}

// Constructor of NonAddressableStripRGB
NonAddressableStripRGB::NonAddressableStripRGB(const char* name, const int& Ext_ch, const int& Int_ch,  ↙
    const int& len, const config_t config) : NonAddressableStrip(name, Ext_ch, Int_ch, len, RGB, config)↙
```

\\psf\Dropbox\Elektrotechniek 4de\Afstuderen ...\Networked LED Driver System\ChannelObject.cpp 11

```
    {}

// Destructor of NonAddressableStripRGB object
NonAddressableStripRGB::~NonAddressableStripRGB(){}

// Checks if the messsage format is correct before processing data
// Checks if the received message has the proper
// format according to the protocol.
// Returns 0 when format is not correct.
bool NonAddressableStripRGB::checkSenderDTformat(const std::vector<unsigned char>& bufIn) const
{
    if (bufIn.size() == 7 && bufIn[3] == '/')
    {
        return true;
    }
    else
    {
        return false;
    }
}

// Decode data sent by the sender via UDP
void NonAddressableStripRGB::decodeUDP(std::vector<unsigned char>& TXbuf, const std::vector<unsigned
    char>& bufIn)
{
    TXbuf.push_back('/');
    TXbuf.push_back((getInternalChannel() >> 8) & 0xFF);
    TXbuf.push_back(getInternalChannel() & 0xFF);
    TXbuf.push_back('/');
    TXbuf.push_back(bufIn[4]);
    TXbuf.push_back(bufIn[5]);
    TXbuf.push_back(bufIn[6]);
}

// Decode data sent by the sender via USB
void NonAddressableStripRGB::decodeUSB(std::vector<unsigned char>& TXbuf, const std::vector<unsigned
    char>& bufIn)
{
    // Implemnt in the future
}



/*
*  MatrixMono is a derived class from Matrix class. The LEDs in the Matrix
*  are individually addressable and produce one color.
*/

// Default Constructor of MatrixMono object
MatrixMono::MatrixMono(const char* name, const int& Ext_ch, const int& Int_ch) : Matrix(name, Ext_ch,
    Int_ch, 1, 1, MONO){}

// Constructor of MatrixMono object
MatrixMono::MatrixMono(const char* name, const int& Ext_ch, const int& Int_ch, const int& wd, const int&
     ht) : Matrix(name, Ext_ch, Int_ch, wd, ht, MONO){}

// Destructor of MatrixMono object
MatrixMono::~MatrixMono(){}

// Checks if the messsage format is correct before processing data
// Checks if the received message has the proper
// format according to the protocol.
// Returns 0 when format is not correct.
bool MatrixMono::checkSenderDTformat(const std::vector<unsigned char>& bufIn) const
{
    if (bufIn.size() == (6 + (getWidth()*getHeight())) && bufIn[3] == '/')
    {
        if (bufIn[4] == ALL && bufIn[5] == '/')
        {
            return true;
        }
        else if (bufIn[4] == SINGLE && bufIn[5] == '/' && bufIn[8] == '/' && bufIn[11] == '/')
```

```
        {
            return true;
        }
        else if (bufIn[4] == MULTIPLE && bufIn[5] == '/' && bufIn[7] == '/')
        {
            if (bufIn[8 + (bufIn[6] * 4)] == '/')
            {
                return true;
            }
        }
    }
    return false;
}

// Decode data sent by the sender via UDP
void MatrixMono::decodeUDP(std::vector<unsigned char>& TXbuf, const std::vector<unsigned char>& bufIn)
{
    switch (bufIn[4])
    {
    case ALL:
    {
        int data = 6;

        TXbuf.push_back('/');
        TXbuf.push_back((getInternalChannel() >> 8) & 0xFF);
        TXbuf.push_back(getInternalChannel() & 0xFF);
        TXbuf.push_back('/');
        TXbuf.push_back(ALL);
        TXbuf.push_back('/');
        for (int i = 0; i < (getWidth()*getHeight()); i++)
        {
            TXbuf.push_back(bufIn[data++]);
        }
        break;
    }

    case SINGLE:
    {
        TXbuf.push_back('/');
        TXbuf.push_back((getInternalChannel() >> 8) & 0xFF);
        TXbuf.push_back(getInternalChannel() & 0xFF);
        TXbuf.push_back('/');
        TXbuf.push_back(SINGLE);
        TXbuf.push_back('/');
        TXbuf.push_back(bufIn[6]);
        TXbuf.push_back(bufIn[7]);
        TXbuf.push_back('/');
        TXbuf.push_back(bufIn[9]);
        TXbuf.push_back(bufIn[10]);
        TXbuf.push_back('/');
        TXbuf.push_back(bufIn[12]);
        break;
    }

    case MULTIPLE:
    {
        int i;
        int pixels = bufIn[6];
        int msgData = (8 + pixels * 4);

        TXbuf.push_back('/');
        TXbuf.push_back((getInternalChannel() >> 8) & 0xFF);
        TXbuf.push_back(getInternalChannel() & 0xFF);
        TXbuf.push_back('/');
        TXbuf.push_back(MULTIPLE);
        TXbuf.push_back('/');
        TXbuf.push_back(bufIn[6]);
        TXbuf.push_back('/');

        for (i = 8; i < msgData; i = i + 2)
        {
            TXbuf.push_back(bufIn[i]);
```

```
            TXbuf.push_back(bufIn[i + 1]);
        }

        TXbuf.push_back('/');

        for (i = msgData + 1; i<(msgData + 1 + pixels); i++)
        {
            TXbuf.push_back(bufIn[i]);
        }
        break;
    }

    default:
        break;
    }
}

// Decode data sent by the sender via USB
void MatrixMono::decodeUSB(std::vector<unsigned char>& TXbuf, const std::vector<unsigned char>& bufIn)
{
    // Implement in the future
}



/*
*  MatrixRGB is a derived class from Matrix class. The LEDs in the Matrix
*  are individually addressable and produce RGB color.
*/

// Default Constructor of MatrixRGB object
MatrixRGB::MatrixRGB(const char* name, const int& Ext_ch, const int& Int_ch) : Matrix(name, Ext_ch,     ↙
    Int_ch, 1, 1, RGB){}

// Constructor of MatrixRGB object
MatrixRGB::MatrixRGB(const char* name, const int& Ext_ch, const int& Int_ch, const int& wd, const int&  ↙
    ht) : Matrix(name, Ext_ch, Int_ch, wd, ht, RGB){}

// Destructor of MatrixRGB object
MatrixRGB::~MatrixRGB(){}

// Checks if the messsage format is correct before processing data
// Checks if the received message has the proper
// format according to the protocol.
// Returns 0 when format is not correct.
bool MatrixRGB::checkSenderDTformat(const std::vector<unsigned char>& bufIn) const
{
    if (bufIn.size() == (6 + (getWidth()*getHeight()*3)) && bufIn[3] == '/')
    {
        if (bufIn[4] == ALL && bufIn[5] == '/')
        {
            return true;
        }
        else if (bufIn[4] == SINGLE && bufIn[5] == '/' && bufIn[8] == '/' && bufIn[11] == '/')
        {
            return true;
        }
        else if (bufIn[4] == MULTIPLE && bufIn[5] == '/' && bufIn[7] == '/')
        {
            if (bufIn[8 + (bufIn[6] * 4)] == '/')
            {
                return true;
            }
        }
    }
    return false;
}

// Decode data sent by the sender via UDP
void MatrixRGB::decodeUDP(std::vector<unsigned char>& TXbuf, const std::vector<unsigned char>& bufIn)
{
    switch (bufIn[4])
```

\\psf\Dropbox\Elektrotechniek 4de\Afstuderen ...\Networked LED Driver System\ChannelObject.cpp 14

```
    {
    case ALL:
    {
        int data = 6;

        TXbuf.push_back('/');
        TXbuf.push_back((getInternalChannel() >> 8) & 0xFF);
        TXbuf.push_back(getInternalChannel() & 0xFF);
        TXbuf.push_back('/');
        TXbuf.push_back(ALL);
        TXbuf.push_back('/');
        for (int i = 0; i < ((getWidth()*getHeight()) * 3); i++)
        {
            TXbuf.push_back(bufIn[data++]);
        }
        break;
    }

    case SINGLE:
    {
        TXbuf.push_back('/');
        TXbuf.push_back((getInternalChannel() >> 8) & 0xFF);
        TXbuf.push_back(getInternalChannel() & 0xFF);
        TXbuf.push_back('/');
        TXbuf.push_back(SINGLE);
        TXbuf.push_back('/');
        TXbuf.push_back(bufIn[6]);
        TXbuf.push_back(bufIn[7]);
        TXbuf.push_back('/');
        TXbuf.push_back(bufIn[9]);
        TXbuf.push_back(bufIn[10]);
        TXbuf.push_back('/');
        TXbuf.push_back(bufIn[12]);
        TXbuf.push_back(bufIn[13]);
        TXbuf.push_back(bufIn[14]);
        break;
    }

    case MULTIPLE:
    {
        int i;
        int pixels = bufIn[6];
        int RGBdata = (8 + pixels * 4);

        TXbuf.push_back('/');
        TXbuf.push_back((getInternalChannel() >> 8) & 0xFF);
        TXbuf.push_back(getInternalChannel() & 0xFF);
        TXbuf.push_back('/');
        TXbuf.push_back(MULTIPLE);
        TXbuf.push_back('/');
        TXbuf.push_back(bufIn[6]);
        TXbuf.push_back('/');

        for (i = 8; i < RGBdata; i = i + 2)
        {
            TXbuf.push_back(bufIn[i]);
            TXbuf.push_back(bufIn[i + 1]);
        }

        TXbuf.push_back('/');

        for (i = RGBdata + 1; i<(RGBdata + 1 + pixels * 3); i++)
        {
            TXbuf.push_back(bufIn[i]);
        }
        break;
    }

    default:
        break;
    }
}
```

\\psf\Dropbox\Elektrotechniek 4de\Afstuderen ...\Networked LED Driver System\ChannelObject.cpp 15

```
// Decode data sent by the sender via USB
void MatrixRGB::decodeUSB(std::vector<unsigned char>& TXbuf, const std::vector<unsigned char>& bufIn)
{
    std::vector<unsigned char> RGBdata;
    switch (bufIn[4])
    {
    case ALL:
    {
        for (std::vector<unsigned char>::const_iterator it = bufIn.begin() + 6; it != bufIn.end(); it++)
        {
            RGBdata.push_back(*it);
        }
        TXbuf = cmd_load_data(getInternalChannel(), (getWidth()*getHeight()), RGBdata);
    }

    case SINGLE:
    {
        //Implement in the future
    }

    case MULTIPLE:
    {
        //Implement in the future
    }
    }
}
```

Appendix 4: Database.h and Database.cpp source code

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

```
//
//  Database.h
//  Networked LED Driver System server
//
//  Created by Winer Bao on 20/02/15.
//  for Tekt Industries Pty. Ltd.
//  Copyright (c) 2015 Winer Bao. All rights reserved.
//

#ifndef __Networked_LED_Driver_System_Server_Database__
#define __Networked_LED_Driver_System_Server_Database__

#include <stdio.h>
#include <map>
#include "Definitions.h"
#include "ChannelObject.h"

struct AdminSocket
{
    std::string adminIPaddr = "0.0.0.0";
    int port;
};

class Mapping
{
public:
    void registerDriver(const int& Ext_channel, const int& Int_channel, const std::string& IP, const int↙
    & port, connection_t con_t);
    void deregisterDriver(const int& channel);
    void registerSender(const int& channel, const std::string& IP, const int& port);
    void deregisterSender(const int& channel);
    void registerAdmin(const std::string& IP, const int& port);
    void deregisterAdmin();
    void changeStrip(const int& Ext_channel, const int& length, const led_t& msgType, const config_t&   ↙
    msgConfig, const color_mode& color, const int& group);
    void changeMatrix(const int& Ext_channel, const int& width, const int& height, const color_mode&    ↙
    color, const int& group);
    void changeRaw(const int& Ext_channel, const int& group);
    std::string getDatabase_DriverInfo();
    std::string getDatabase_SenderInfo();
    bool doesChannelExist(const int& channel) const;
    bool doesChannelBelongToDriver(int channel, std::string IP, int port);
    bool doesChannelBelongToSender(const int& channel, const std::string& IP, const int& port);
    bool isChannelOccupied(const int& channel);
    bool isDriverOccupied(const int& channel) const;
    bool doesAdminExist() const;
    bool doesBelongToAdmin(const std::string& IP, const int& port) const;
    bool checkSenderDTformat(const int& channel, const std::vector<unsigned char>& bufIn);
    void decodeUDP(const int& channel, std::vector<unsigned char>& TXbuf, const std::vector<unsigned    ↙
    char>& bufIn);
    void decodeUSB(const int& channel, std::vector<unsigned char>& TXbuf, const std::vector<unsigned    ↙
    char>& bufIn);
    std::string getDriverIPaddr(const int& channel);
    int getDriverPort(const int& channel);
    class_t getClass(const int& channel);
    color_mode getColorMode(const int& channel);
    connection_t getConnectionType(const int& channel);
    int getLength(const int& channel);
    led_t getType(const int& channel);
    config_t getConfig(const int& channel);
    int getWidth(const int& channel);
    int getHeight(const int& channel);

private:
    AdminSocket databaseAdmin;
    std::map<int, LED*> database;
    LED* createDerivedStrip(const int& Ext_channel, const int& Int_channel, const std::string& IP, const↙
     int& port, const int& length, const led_t& msgType, const config_t& msgConfig, const color_mode&   ↙
    colorMode);
    LED* createDerivedMatrix(const int& Ext_channel, const int& Int_channel, const std::string& IP,     ↙
    const int& port, const color_mode& colorMode, const int& width, const int& height);
    LED* createDerivedRaw(const int& Ext_channel, const int& Int_channel, const std::string& IP, const  ↙
```

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

```
    int& port);
    template<typename T>
    T* createObject(int Ext_channel, int Int_channel) const;
};

// Creates a T type object and returns a pointer to this object
template<typename T>
T* Mapping::createObject(int Ext_channel, int Int_channel) const
{
    std::string name = std::to_string(Ext_channel);

    T* LED_ptr = new T(name.c_str(), Ext_channel, Int_channel);
    return LED_ptr;
}

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

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

```
//
//  Database.cpp
//  Networked LED Driver System server
//
//  Created by Winer Bao on 20/02/15.
//  for Tekt Industries Pty. Ltd.
//  Copyright (c) 2015 Winer Bao. All rights reserved.
//

#include "Database.h"
#include <assert.h>

// Creates an default object (LED and Derived Classes)
// The newly created object/channel is added to the database.
// Admin must change driver setting via the Admin program before sending data.
void Mapping::registerDriver(const int& Ext_channel, const int& Int_channel, const std::string& IP,
    const int& port, connection_t con_t)
{
    LED* obj_ptr = createDerivedRaw(Ext_channel, Int_channel, IP, port);   // Create default object
    obj_ptr->setGroup(1);                                   // Set default group
    obj_ptr->setConnectionType(con_t);                      // Set default connection type
}

// Remove driver/channel from database
void Mapping::deregisterDriver(const int& channel)
{
    delete database[channel];
    database.erase(channel);
}

// Stores Sender's information in database.
// The channel becomes unavailable to others.
void Mapping::registerSender(const int& channel, const std::string& IP, const int& port)
{
    LED* LEDptr = database[channel];

    assert(LEDptr != nullptr);
    LEDptr->setSenderInfo(IP, port);
}

// Removes Sender's information from database.
// The channel becomes available to others.
void Mapping::deregisterSender(const int& channel)
{
    LED* LEDptr = database[channel];

    assert(LEDptr != nullptr);
    LEDptr->setSenderInfo("0.0.0.0", 0);
}

// Stores Admin's information in database.
// The admin program becomes unavailable to others.
void Mapping::registerAdmin(const std::string& IP, const int& port)
{
    databaseAdmin.adminIPaddr = IP;
    databaseAdmin.port = port;
}

// Removes Admin's information from database.
// The admin program becomes available to others.
void Mapping::deregisterAdmin()
{
    databaseAdmin.adminIPaddr = "0.0.0.0";
    databaseAdmin.port = 0;
}

// Changes a class object to a derived strip class object.
void Mapping::changeStrip(const int& Ext_channel, const int& length, const led_t& msgType, const
    config_t& msgConfig, const color_mode& color, const int& group)
{
    LED* ptr = database[Ext_channel];
    const std::string driverIP = ptr->getDriverIP();
```

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

```
    const int driverPort = ptr->getDriverPort();
    const std::string senderIP = ptr->getSenderIP();
    const int senderPort = ptr->getSenderPort();
    const connection_t connectType = ptr->getConnectionType();
    const int Int_channel = ptr->getInternalChannel();

    deregisterDriver(Ext_channel);
    createDerivedStrip(Ext_channel, Int_channel, driverIP, driverPort, length, msgType, msgConfig,
    color);
    ptr = database[Ext_channel];
    ptr->setSenderInfo(senderIP, senderPort);
    ptr->setGroup(group);
    ptr->setConnectionType(connectType);
    ptr->setInternalChannel(Int_channel);
    if (connectType == USB)
    {
        setUSBChannelLength(Int_channel, length);
    }
}

// Changes a class object to a derived matrix class object.
void Mapping::changeMatrix(const int& Ext_channel, const int& width, const int& height, const color_mode
    & color, const int& group)
{
    LED* ptr = database[Ext_channel];
    const std::string driverIP = ptr->getDriverIP();
    const int driverPort = ptr->getDriverPort();
    const std::string senderIP = ptr->getSenderIP();
    const int senderPort = ptr->getSenderPort();
    const connection_t connectType = ptr->getConnectionType();
    const int Int_channel = ptr->getInternalChannel();

    deregisterDriver(Ext_channel);
    createDerivedMatrix(Ext_channel, Int_channel, driverIP, driverPort, color, width, height);
    ptr = database[Ext_channel];
    ptr->setSenderInfo(senderIP, senderPort);
    ptr->setGroup(group);
    ptr->setConnectionType(connectType);
    ptr->setInternalChannel(Int_channel);
    if (connectType == USB)
    {
        setUSBChannelLength(Int_channel, width*height);
    }
}

// Changes a class object to a derived raw class object.
void Mapping::changeRaw(const int& Ext_channel, const int& group)
{
    LED* ptr = database[Ext_channel];
    const std::string driverIP = ptr->getDriverIP();
    const int driverPort = ptr->getDriverPort();
    const std::string senderIP = ptr->getSenderIP();
    const int senderPort = ptr->getSenderPort();
    const connection_t connectType = ptr->getConnectionType();
    const int Int_channel = ptr->getInternalChannel();

    deregisterDriver(Ext_channel);
    createDerivedRaw(Ext_channel, Int_channel, driverIP, driverPort);
    ptr = database[Ext_channel];
    ptr->setSenderInfo(senderIP, senderPort);
    ptr->setGroup(group);
    ptr->setConnectionType(connectType);
    ptr->setInternalChannel(Int_channel);
}

// Returns a string with all driver information.
std::string Mapping::getDatabase_DriverInfo()
{
    std::string message;
    message.append("&");
    message.append("DRIVER");
    message.append("/");
```

\\psf\Dropbox\Elektrotechniek 4de\Afstuderen ... System\Networked LED Driver System\Database.cpp 3

```
    for (std::map<int, LED* >::iterator it = database.begin(); it != database.end(); ++it)
    {
        message.append(std::to_string(it->first));
        message.append("/");
        if ((it->second)->getConnectionType() == UDP)
        {
            message.append((it->second)->getDriverIP());
            message.append("/");
            message.append(std::to_string((it->second)->getDriverPort()));
        }
        else //if ((it->second)->getConnectionType() == USB)
        {
            message.append("USB");
            message.append("/");
            message.append("0");
        }
        message.append("/");
        if ((it->second)->getClassType() == STRIP)
        {
            message.append("strip");
            message.append("/");
        }
        else if ((it->second)->getClassType() == MATRIX)
        {
            message.append("matrix");
            message.append("/");
        }
        else if ((it->second)->getClassType() == RAW)
        {
            message.append("raw");
            message.append("/");
        }

        if ((it->second)->getColorMode() == MONO)
        {
            message.append("mono");
            message.append("/");
        }
        else if ((it->second)->getColorMode() == RGB)
        {
            message.append("RGB");
            message.append("/");
        }

        Strip* ptrStrip = dynamic_cast<Strip*>((it->second));
        if (ptrStrip != nullptr)
        {
            if (ptrStrip->getType() == ADDRESSABLE)
            {
                message.append("addressable");
                message.append("/");
            }
            else if (ptrStrip->getType() == NONADDRESSABLE)
            {
                message.append("non-addressable");
                message.append("/");
            }
            if (ptrStrip->getConfig() == LINE)
            {
                message.append("line");
                message.append("/");
            }
            else if (ptrStrip->getConfig() == OTHER)
            {
                message.append("other");
                message.append("/");
            }
            message.append(std::to_string(ptrStrip->getLength()));
            message.append("/");
        }
```

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

```
        Matrix* ptrMatrix = dynamic_cast<Matrix*>((it->second));
        if (ptrMatrix != nullptr)
        {
            message.append(std::to_string(ptrMatrix->getWidth()));
            message.append("/");
            message.append(std::to_string(ptrMatrix->getHeight()));
            message.append("/");
        }
    }
    return message;
}

// Returns a string with all sender information.
std::string Mapping::getDatabase_SenderInfo()
{
    std::string message;
    message.append("&");
    message.append("SENDER");
    message.append("/");

    for (std::map<int, LED* >::iterator it = database.begin(); it != database.end(); ++it)
    {
        if ((it->second)->getSenderIP() != "0.0.0.0")
        {
            message.append(std::to_string(it->first));
            message.append("/");
            message.append((it->second)->getSenderIP());
            message.append("/");
            message.append(std::to_string((it->second)->getSenderPort()));
            message.append("/");
            if ((it->second)->getClassType() == STRIP)
            {
                message.append("strip");
                message.append("/");
            }
            else if ((it->second)->getClassType() == MATRIX)
            {
                message.append("matrix");
                message.append("/");
            }
            else if ((it->second)->getClassType() == RAW)
            {
                message.append("raw");
                message.append("/");
            }
        }
    }
    return message;
}

// Checks if there is a driver board connected to this channel.
// Returns true when there is one connected.
bool Mapping::doesChannelExist(const int& channel) const
{
    if (database.find(channel) == database.end())
    {
        return false;
    }
    return true;
}

// Checks if the Sender owns this channel.
// If the Sender is not the owner of the channel, false is returned.
bool Mapping::doesChannelBelongToSender(const int& channel, const std::string& IP, const int& port)
{
    LED* LEDptr = database[channel];

    assert(LEDptr != nullptr);
    if ((LEDptr->getSenderIP() == IP) && (LEDptr->getSenderPort() == port))
    {
        return true;
    }
```

```
    return false;
}

// Checks if the Driver owns this channel.
// If the Driver is not the owner of the channel, false is returned.
bool Mapping::doesChannelBelongToDriver(int channel, std::string IP, int port)
{
    LED* LEDptr = database[channel];

    assert(LEDptr != nullptr);
    if ((LEDptr->getDriverIP() == IP) && (LEDptr->getDriverPort() == port))
    {
        return true;
    }
    return false;
}

// Checks if there is a owner of this channel.
// If no owner is found, a false is returned.
bool Mapping::isChannelOccupied(const int& channel)
{
    LED* LEDptr = database[channel];

    assert(LEDptr != nullptr);
    if ((LEDptr->getSenderIP() == "0.0.0.0") && (LEDptr->getSenderPort() == 0))
    {
        return false;
    }
    return true;
}

// Check if there is a owner of this channel/driver
// If no owner is found, a false is returned.
bool Mapping::isDriverOccupied(const int& channel) const
{
    if (database.find(channel) == database.end())
    {
        return false;
    }
    return true;
}

// Checks if there is a admin program connected to the server.
// Returns true when there is one connected.
bool Mapping::doesAdminExist() const
{
    if (databaseAdmin.adminIPaddr == "0.0.0.0")
    {
        return false;
    }
    return true;
}

// Checks if the Admin owns this channel.
// If the Admin is not the owner of the channel, false is returned.
bool Mapping::doesBelongToAdmin(const std::string& IP, const int& port) const
{
    if (databaseAdmin.adminIPaddr == IP && databaseAdmin.port == port)
    {
        return true;
    }
    return false;
}

//Change object to an Strip object
LED* Mapping::createDerivedStrip(const int& Ext_channel, const int& Int_channel, const std::string& IP, ↙
    const int& port, const int& length, const led_t& msgType, const config_t& msgConfig, const          ↙
    color_mode& colorMode)
{
    LED* obj_ptr;
    if (msgType == ADDRESSABLE && colorMode == RGB)
    {
```

```
        AddressableStripRGB* ptr = createObject<AddressableStripRGB>(Ext_channel, Int_channel);
        ptr->setLength(length);
        ptr->setConfig(msgConfig);
        obj_ptr = ptr;
    }
    else if (msgType == ADDRESSABLE && colorMode == MONO)
    {
        AddressableStripMono* ptr = createObject<AddressableStripMono>(Ext_channel, Int_channel);
        ptr->setLength(length);
        ptr->setConfig(msgConfig);
        obj_ptr = ptr;
    }
    else if (msgType == NONADDRESSABLE && colorMode == RGB)
    {
        NonAddressableStripRGB* ptr = createObject<NonAddressableStripRGB>(Ext_channel, Int_channel);
        ptr->setLength(length);
        ptr->setConfig(msgConfig);
        obj_ptr = ptr;
    }
    else //if (msgType == NONADDRESSABLE && colorMode == MONO)
    {
        NonAddressableStripMono* ptr = createObject<NonAddressableStripMono>(Ext_channel, Int_channel);
        ptr->setLength(length);
        ptr->setConfig(msgConfig);
        obj_ptr = ptr;
    }
    database[Ext_channel] = obj_ptr;
    obj_ptr->setDriverInfo(IP, port);
    return obj_ptr;
}

//Change the object into a Matrix object
LED* Mapping::createDerivedMatrix(const int& Ext_channel, const int& Int_channel, const std::string& IP,↙
     const int& port, const color_mode& colorMode, const int& width, const int& height)
{
    LED* obj_ptr;
    if (colorMode == RGB)
    {
        MatrixRGB* ptr = createObject<MatrixRGB>(Ext_channel, Int_channel);
        ptr->setWidth(width);
        ptr->setHeigth(height);
        obj_ptr = ptr;
    }
    else //if (colorMode == MONO)
    {
        MatrixMono* ptr = createObject<MatrixMono>(Ext_channel, Int_channel);
        ptr->setWidth(width);
        ptr->setHeigth(height);
        obj_ptr = ptr;
    }
    database[Ext_channel] = obj_ptr;
    obj_ptr->setDriverInfo(IP, port);
    return obj_ptr;
}

// Change the object into a Raw object
LED* Mapping::createDerivedRaw(const int& Ext_channel, const int& Int_channel, const std::string& IP,   ↙
    const int& port)
{
    LED* obj_ptr = createObject<Raw>(Ext_channel, Int_channel);
    database[Ext_channel] = obj_ptr;
    obj_ptr->setDriverInfo(IP, port);
    return obj_ptr;
}

// Dynamic function. Checks message format.
bool Mapping::checkSenderDTformat(const int& channel, const std::vector<unsigned char>& bufIn)
{
    return database[channel]->checkSenderDTformat(bufIn);
}

// Dynamic function. Decode UDP messagae function of an object
```

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

```
void Mapping::decodeUDP(const int& channel, std::vector<unsigned char>& TXbuf, const std::vector
    <unsigned char>& bufIn)
{
    database[channel]->decodeUDP(TXbuf, bufIn);
}

// Dynamic function. Decode USB messagae function of an object
void Mapping::decodeUSB(const int& channel, std::vector<unsigned char>& TXbuf, const std::vector
    <unsigned char>& bufIn)
{
    database[channel]->decodeUSB(TXbuf, bufIn);
}

// Dynamic function. Returns IP adress of a driver board/channel.
std::string Mapping::getDriverIPaddr(const int& channel)
{
    return database[channel]->getDriverIP();
}

// Dynamic function. Returns port number of a driverboard/channel.
int Mapping::getDriverPort(const int& channel)
{
    return database[channel]->getDriverPort();
}

// Dynamic function. Returns the class type of a channel
class_t Mapping::getClass(const int& channel)
{
    return database[channel]->getClassType();
}

// Dynamic function. Returns the color mode of a channel
color_mode Mapping::getColorMode(const int& channel)
{
    return database[channel]->getColorMode();
}

// Dynamic function. Returns the connection type used
connection_t Mapping::getConnectionType(const int& channel)
{
    return database[channel]->getConnectionType();
}

// Dynamic function. Returns the strip length of a Strip channel.
int Mapping::getLength(const int& channel)
{
    const Strip* stripPtr = dynamic_cast<const Strip* >(database[channel]);

    assert(stripPtr != nullptr);
    return stripPtr->getLength();
}

// Dynamic function. Returns the led type of a Strip channel.
led_t Mapping::getType(const int& channel)
{
    const Strip*  stripPtr = dynamic_cast<const Strip* >(database[channel]);

    assert(stripPtr != nullptr);
    return stripPtr->getType();
}

// Dynamic function. Returns the led configuration of a Strip channel.
config_t Mapping::getConfig(const int& channel)
{
    const Strip* stripPtr = dynamic_cast<const Strip* >(database[channel]);

    assert(stripPtr != nullptr);
    return stripPtr->getConfig();
}

// Dynamic function. Returns the width of a Matrix channel
int Mapping::getWidth(const int& channel)
```

\\psf\Dropbox\Elektrotechniek 4de\Afstuderen ... System\Networked LED Driver System\Database.cpp 8

```
{
    const Matrix* matrixPtr = dynamic_cast<const Matrix*>(database[channel]);

    assert(matrixPtr != nullptr);
    return matrixPtr->getWidth();
}

// Dynamic function. Returns the height of a Matrix channel
int Mapping::getHeight(const int& channel)
{
    const Matrix* matrixPtr = dynamic_cast<const Matrix*>(database[channel]);

    assert(matrixPtr != nullptr);
    return matrixPtr->getHeight();
}
```