FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Ethernet Stage Lighting Protocol: An Alternative System for Professional Lighting Equipment with Legacy DMX Compatibility

Pedro Emanuel Fernandes Magalhães

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Prof. Dr. Paulo Portugal

March 1, 2015

U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

MIEEC - MESTRADO INTEGRADO EM ENGENHARIA ELETROTÉCNICA E DE COMPUTADORES 2014/2015

A Dissertação intitulada

"Ethernet Stage Lightning Protocol: An Alternative System for Professional Lightning Equipment with Legacy DMX Compatibility"

foi aprovada em provas realizadas em 20-02-2015

o júri

Presidente Professor Doutor Armando Jorge Miranda de Sousa
Professor Auxiliar do Departamento de Engenharia Eletrotécnica e de Computadores da Faculdade de Engenharia da Universidade do Porto

Professor Doutor Jaime Francisco Cruz Fonseca
Professor Associado Departamento Eletrónica Industrial da Universidade do Minho

Professor Doutor Paulo José Lopes Machado Portugal
Professor Auxiliar do Departamento de Engenharia Eletrotécnica e de Computadores da Faculdade de Engenharia da Universidade do Porto

O autor declara que a presente dissertação (ou relatório de projeto) é da sua exclusiva autoria e foi escrita sem qualquer apoio externo não explicitamente autorizado. Os resultados, ideias, parágrafos, ou outros extratos tomados de ou inspirados em trabalhos de outros autores, e demais referências bibliográficas usadas, são corretamente citados.

Autor - Pedro Emanuel Fernandes Magalhães

Faculdade de Engenharia da Universidade do Porto

Abstract

This document introduces an alternative to the current stage lighting protocols, presenting a flexible and reliable solution that, by studying the previous technologies and taking in consideration their advantages, explores new and intuitive features that are part of an architecture developed for this Dissertation.

The architecture of the system is not only based on current technologies that allow its expandability, but it also follows the principles that defined stage lighting, therefore increasing the complexity of the network without affecting the overall performance and limiting the rigging process. It also allows the inclusion of devices compatible with the previous protocols, embedding them seamlessly into the new system.

A working prototype using standard development tools has been developed implementing the basic features described by the system architecture, using standard development hardware and software platforms, therefore easily portable.

Acknowledgements

I would like to thank Professor Paulo Portugal at Faculdade de Engenharia da Universidade do Porto for accepting me as a Dissertation student, and for all the help and support throughout its planning and realization.

I would also like to thank Sirilanka Espetáculos, Lda for the chance to work as a lighting technician for them for the past five years. It was this opportunity that led me to work with lighting equipment in the first place and eventually led to the proposition of this Dissertation.

Lastly, but not less importantly, I would like to thank my family for their continued support during the years spent in this course, and my friends, inside and outside the academic and musical environment, for all the help and encouragement.

To all the mentioned entities, thank you.

Pedro Magalhães

"What is the purpose of knowledge if it lays abandoned in dead shelves?"

Cidália Fernandes

Contents

List of Figures

List of Tables

Abbreviations

DMX	*Digital MultipleX*
AMX	*Analog MultipleX*
LED	*Light-Emitting Diode*
PAR	*Parabolic Aluminized Reflector*
RGB	*Red Green Blue*
MAB	*Mark After Break*
RDM	*Remote Device Monitoring*
Art-Net	*Artistic Network*
TCP	*Transmission Control Protocol*
UDP	*User Datagram Protocol*
IP	*Internet Protocol*
DHCP	*Dynamic Host Configuration Protocol*
MAC	*Media Access Control*

Chapter 1

Introduction

Intelligent Lighting is the usage of controllable light fixtures to illuminate a stage or theatre, differing from the traditional stationary lighting which relied on simple static lighting positioning. Today, the concept of lighting has evolved with the technology in a way that it is used in almost every outdoor or indoor concert, theatres and even in home lighting.

Figure 1.1: Avolites Sapphire Touch controlling the lighting equipment. [1]

The various types of lighting equipment, such as moving heads, LED spots, washes, etc, have the ability of changing colour, shape and movement of their output to create different environments, or focus the audience into a specific spot, actor or performer. The programming of a show is crucial to the application of these concepts, so all of the effects and cues have to be created by the programmer, lighting engineer or technician responsible for the show. Overall, their performance defines the visual experience, which is a vital part of the show.

Lighting systems require a high level of control, so different technologies were created over the years to accommodate the increasing complexity of lighting equipment, cabling and rigging. However, the manufacturers were creating their own proprietary control solutions, not compatible with products from other companies, so it was necessary to have multiple controllers to handle the different types of machines.

In 1986 an universal protocol was developed to prevent the manufacturers to diverge, and to ensure that all the devices could be controlled by one single endpoint, using the same communication standard. This protocol is the DMX512, or simply DMX, and it is, even today, the standard for lighting equipment. [4]

Figure 1.2: Pink Floyd, The Division Bell tour, 1994, the greatest show in the history of mankind.

The complexity of these systems has been increasing since the development of moving heads, which is leading the the DMX technology to a limit. The limitations of the standard DMX512 protocol are becoming more obvious, which imposes more restrictions in its usage.

1.1 Framework and Motivation

During the Summer of 2009 and until 2013, I have worked with a small band as a lighting technician for their shows, where various DMX devices were used. The experience and problem-solving skills gained with working with the fixtures and the controllers directly led to the realization of the limitations within this system, so the search for a flexible alternative combining the robustness of the previous solutions and Ethernet technology began at least two years before the realization of this Dissertation.

1.2 Objectives

The first objective of this Dissertation is the development of an Ethernet based alternative to replace the DMX standard to control lighting equipment. That system should be able to handle the technological evolution and offer new advantages and solutions for common problems present in the current protocol. However, most of the existing lighting equipment has the DMX interfaces as standard, so this new system also has to allow to communicate with those legacy devices, integrating them seamlessly into the new system.

The second objective is the creation of a working prototype implementing part of the developed system that concerns the compatibility with legacy devices.

1.3 Document Structure

This document is structured as following:

- Second chapter: State-of-the-art and study of the technologies to be used in the developed system;
- Third chapter: Complete description of the system requirements and architecture;
- Fourth chapter: Development of a working prototype for proof of concept;
- Fifth chapter: Achieved results and conclusions.

Chapter 2

Technologies

This section will introduce the intelligent lighting systems and the current technologies used, including their advantages and issues.

2.1 Introduction

A lighting system is constituted by two basic elements types, controllers and devices. The controllers, or consoles, are the interface with the user, or lighting technician. This controller is widely programmable, and it is connected to the devices on stage through one or more cables. The devices have a variety of control parameters which the console controls. The devices receive the signals from the controller and change their parameters accordingly. In a basic set-up, one controller can control a multitude of devices with different control parameters. In more advanced set-ups, several controllers can be used, but they are never connected between themselves, they only affect the devices each one is connected to.

The lighting crew, technicians and engineers are responsible for assembling all the devices and connecting them to a controller, programming and execution of the show.

The controller and the devices are part of a master-slave system, where the controller is the master, sending commands to the devices, which are the slaves.

2.1.1 Fixtures

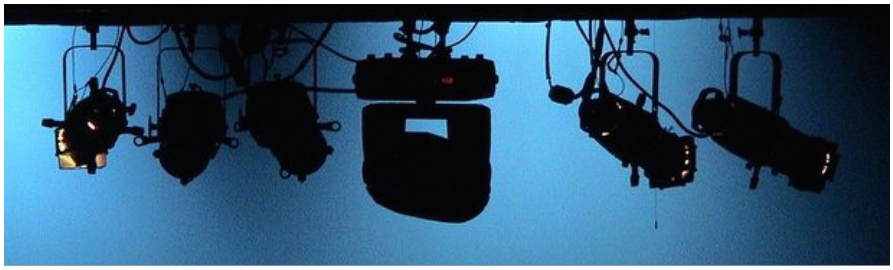

Figure 2.1: Different types of stage lighting devices. [2]

Figure 2.2: Examples of a standard PAR and LED PAR lights.

The fixtures, or lighting instruments, are the part of the system that creates light. They vary in size, shape, weight, light power output and technology, which determines their purpose on stage. These are the slaves of the system, receiving the signals from the controller and changing their outputs as requested by the programmer.

Most of these fixtures can be divided in two major categories, although some of them belong to both: Spot lights and Wash lights. The spot lights are focused lights that allow the controlled projection of shapes across a surface, with the aid of filters positioned in front of the light source. The wash lights are less controllable, and can be used, for example, as colour fillers, although they are not limited to this application. [5]

Every device has a number of control parameters. For example, a moving head usually has parameters to control the movement, the colours and shapes, which usually amounts to 8 to 16 control parameters or variables.

2.1.1.1 Parabolic aluminized reflector

Th classic fixtures used in stage lighting are the PAR lamps. These are easy and straightforward to use as they usually only have one control variable, the Dimmer, which is the basic intensity control, varying from completely off to fully on. Several filters, such as colour filters and gobos, or projection filters, can be placed in front of the PAR in order to change the light output, which is usually white by default. [6]

LED PAR lights are most commonly used to replace classic lamps as they can dynamically change the colour output, a feature that was only achievable with the aid of filters in regular PAR lights. They also allow a wide variety of colours in one single device, which previously required the use of a large number of classic PAR lamps, each one with different filters.

2.1.1.2 Scanners/Moving mirrors and Moving heads

Moving mirrors or scanners are devices that rely on a motorized mirror to project the light, which is generated and filtered in a stationary casing. The moving heads use larger motors to move

the entire fixture around, allowing a wider projection angle, whereas the scanners have limited movement range.

These devices have their own advantages and disadvantages, which limits the application and usage. For example, the scanners have a limited range of movement, but they are extremely fast since the motors only need to move a small mirror. The moving heads are heavier and therefore slower, but the movement range is expanded.

LED technology and more energy and size efficient discharge lamps allowed the moving heads to become smaller and faster, so these are the preferred types of devices used in intelligent lighting for their flexibility and weight.

Figure 2.3: Sharpy and A-Leda Wash K20, moving heads manufactured by Clay Paky. [3]

The concept of LED PAR lights has been applied to moving heads, which results in a moving wash set-up, with RGB capabilities, as seen in figure 2.3.

2.1.1.3 Other devices

Other devices used with this system are, for example, the smoke or fog machines, that can use regular smoke, or ice for a low height haze effect. These are usually controlled the same way a PAR lamp is, with only one parameter that controls the intensity of the output.

Figure 2.4: A smoke machine and a laser.

2.1.2 Controller

Figure 2.5: Avolites Sapphire Touch console.

The controller or console is the device that, as the name implies, controls the whole system, it is where the configuration, programming and computation occurs. The information is sent to the fixtures through one or more cables. It is, therefore, the master of this master-slave system.

The most basic controllers are programmable dimmer stations, which include a number of faders to control different channels, and have some memorizing capabilities. These, however do not support advanced computations, since they rely on fixed memories. The most complex controllers, which are widely used and more capable of performing advanced functions with a larger number of devices, are usually regular computers that connect to an interface to provide the necessary signal to the fixtures. Most of the consoles with faders and built-in screens have, in fact, a computer inside to handle the heavy computational requirements.

The controllers, therefore, have the computational power to calculate the positions of every device connected to it, and also interface with the lighting technician or engineer. The software used depends on the manufacturer, and the processes of

Classic consoles have a fixed number of faders that represent one individual parameter. These are fairly simple to use with dimmer lights and basic LED panels, but are very strict when it comes to more complex devices with movement motors, for example. These usually provide a set of memories modifiable by the user, and that can be accessed quickly.

Modern controllers allow more abstraction than the previous ones because the user does not interact directly with the channel numbers in a fader; instead, it is possible to create common memories, or palettes, to deal with the variables of all the devices. For example, a colour wheel in a classic controller is set manually by moving the fader of that channel to the position that represents that colour. In a modern controller, it is possible to create palettes to record all these parameters and recall them on specific devices. This is different from recalling a memory in a classic controller, because they memorize the entire output, instead of the specific outputs of one or more devices.

Modern controllers also allow complex computation processes, or programmable shapes. For example, it is possible to set a shape such as a device moves in circles, up and down, sideways, etc.

More recently, it is even possible to draw custom traces on a screen for the device to follow the shape. These operations are executed in the controller, sent through the output, read and executed by the devices. All the computation is in the console rather than the devices, which act as the slaves of the system only.

2.2 State of the Art

This chapter describes some of the technologies that are being currently used to control lighting equipment: The DMX512, standard for lighting communication, and the technologies developed around this protocol to solve some of its expandability issues.

2.2.1 DMX512

Digital Multiplex 512, or simply DMX, is a protocol originally created in 1986 as the standard network interface between fixtures and consoles. [7] A DMX universe, or network, is one connection between the console and the devices. Each universe can hold devices with a maximum amount of 512 channels, or parameters, hence the inclusion of that number in the name.

It replaced the Analog Multiplexing, or AMX192, which only allowed 192 channels controllable with the voltage levels instead of digital signals.

The communication bus is based on a differential RS-485, so it can only theoretically hold up to 32 devices in a single bus. [8] However, that number can be increased with a proper repeater.

As the complexity of the devices increases, so has the need for more parameters of control, which means that a single device can use as many channels as the manufacturer defines - it is up to the technician or engineer to ensure the correct programming of the devices, as it will be further explained in section 2.2.1.2.

For example, if 32 devices are connected to a single DMX universe with no repeaters, it means only 16 channels can be allocated per device, which proves to not be enough for some devices. The solution found to solve this problem is in increasing the number of DMX universes in the console's output.

Figure 2.6: GrandMA Console with 4 DMX Universes.

2.2.1.1 Physical Layer

In a single DMX Universe, a controller has a DMX output and the devices have an output and an input. A cable connects the controller to the first device through the input, and another cable connects its output to the input of the next device, which represents a daisy-chain connection between them (figure 2.8).

Figure 2.7: XLR-3 female connector to XLR-3 male connector.

The standard cables to connect these devices were standard XLR-3 cables, similar to the typical differential cables used as instrument and microphone cables for audio applications. As some of the microphones need a DC power of 48 Volt, the accidental connection of one of those cables to a controller led to further damaging the light equipment, so the standard changed to only allow a five pin XLR, the XLR-5, which has the same differential pair, a ground pin and two pins that remain unused. This prevents the accidental connection of audio equipment. However, since the cables only differed in the addition of two pins, manufacturers kept producing, even today, the three pin outputs and inputs for their lighting equipment, therefore breaking the standard. It is common to have cables converting the XLR-5 output of the consoles to the XLR-3 input of the devices, as seen in figure 2.7. The usage of these cables designed for audio equipment is not ideal, but the robustness of the protocol prevents most of the problems related to cabling.

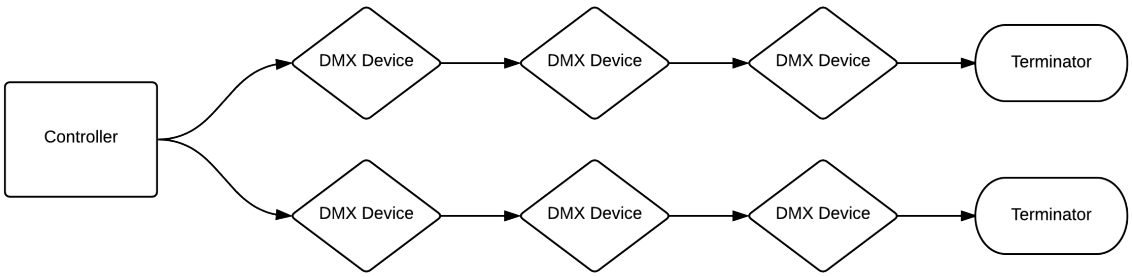

Figure 2.8: Daisy-chain connection between a controller and two DMX universes.

The last device in a DMX universe is usually fitted with a terminator, which is a 120 Ω resistor, although most of the devices usually have a dip switch with a resistor of the same value built-in. This resistor prevents the signal reflection, according to the RS-485 definition. [8]

2.2.1.2 Protocol

The DMX data is transmitted asynchronously at a speed of 250 *kBaud*. The protocol is organized in slots, which represent the DMX channels. Each channel is an eight bit value, ranging from 0 to 255, and represents one parameter of a device. The data of all the allocated channels is sent in order, so the devices only need to be configured with the starting address of their channels. [9]

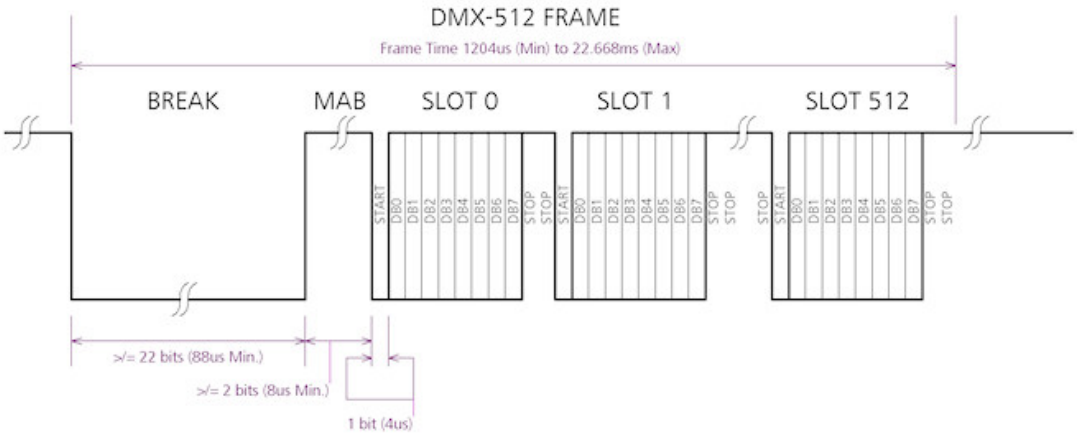

Figure 2.9: Representation of the DMX packet.

For example, a 16 channel moving head configured with the start address 20 receives the message and has to wait until the 20th slot, which corresponds to the first channel, and reads until the 36th slot. The read data corresponds to the 16 parameters of that specific moving head.

Since all the devices in the same universe receive the same message, it is possible to have multiple devices of the same type configured to the same address, which means that they will execute the exact same functions (Assuming there are no hardware differences that can alter the response of the actuators).

The message is composed by header and data. The header is formed by :

- Break: Pause between frames of at least 88 μs or 22 low bits;
- MAB, Mark-after-break: At least 2 bits or 8 μs.

At this point, the preamble of the message is formed. The first channel, or channel 0, is the start code, never used by any devices, and it must always be zero.

After the MAB, the data is formed by *n* slots:

- Slot 0:
  - 1 Start bit with the logic value 0;
  - 8 Data bits, start code 0x00;
  - 2 End bit with the logic value 1;

- Slot 1:
  - 1 Start bit with the logic value 0;
  - 8 Data bits corresponding to the data byte 1;
  - 2 End bits with the logic value 1;
- Slot *n* (up to 512):
  - 1 Start bit with the logic value 0;
  - 8 Data bits corresponding to the data byte *n*;
  - 2 End bits with the logic value 1;

A total of 513 slots can sent. After the last slot, a new break can take place and the message is retransmitted once more.

In an Universe with 512 allocated channels with the break right after the last channel, the transmission time of a single message is approximately 0.022 seconds, which represents a transmission rate of 45 Hz. Since most of the controllers have multiple DMX outputs, the number of devices can be spread through the number of existing universes, reducing the size of the DMX message, therefore reducing the transmission time. [9]

For example, 128 allocated channels (last allocated channel is 128) are transmitted in 0.006 seconds, with a rate of 166 Hz.

Since all the channels are sent until the last allocated slot, it means that controllers automatically allocate new channels in the lowest possible empty space, taking in account the number of parameters of the new device. This ensures a higher transmission rate.

Since the controller is the only one transmitting, there is no danger of collisions. There is also no need for error correction algorithms, because the packet is continuously retransmitted. This, however, makes it impossible to use with fireworks and other hazardous applications.

2.2.1.3 Use case scenario

For this scenario, the configuration process is based on an Avolites console.

"A lighting technician has at his disposal a DMX lighting console, and various DMX devices already assembled on stage, both on the top and ground rigs.

- The technician connects one cable of the console's first DMX output to the input of one device on the ground. Using more XLR cables, he proceeds to connect the output of the first device to the input of the second device, and so on, creating the daisy chain connection between them;
- Another cable is connected to the second output of the console and into the first device on the top rig, also connecting these devices in a daisy chain network;
- Two universes have been used and cannot be mixed; the two last devices are fitted with a terminator;

- The technician powers the console and the device. After initial boot, he begins configuring the console, adding each fixture available. The console allocates the DMX channels automatically, so he writes down the DMX starting address of each device;
- After setting up the console, the technician then inserts manually the starting address in each device;
- When done, the technician can now test the connections by attempting to change the parameters of each device and check their responsiveness;
- The configuration is now finished, and the technician can start programming the show."

2.2.2 RDM

Remote Device Management is an extension of the DMX512 protocol that allows bi-directional communication between controllers and devices, maintaining the compatibility with regular DMX devices that do not support this protocol without changing the physical layer. It uses the DMX phyical layer, but with a different start code to differentiate the RDM messages from the DMX messages.[10]

However, some devices are not compatible with this protocol, because if they do not read the start code, they might try to read the RDM packet as a DMX packet, which can cause unexpected results. Some splitters may also reject the RDM messages if they are configured to drop unknown or not-DMX start codes.

Figure 2.10: RDM-ready DMX splitter.

Physically, few changes are necessary in order to ensure the correct functioning of this system. The main requirement, that also belongs in the DMX512 requirements list but is often not met, although it does not cause any instability to regular systems, is the mandatory presence of the terminator mentioned in section 2.2.1.1, to avoid signal reflections.

The RDM data packets are inserted between the DMX packets when necessary. As referred in section 2.2.1.2, the start code of the DMX packet is the code 0x00. By using the start code 0x00, the RDM packet is safely transmitted and legacy devices simply ignore them.

Since it is a bi-directional protocol, there are now data collision problems, which have been resolved by defining a timed token-passing transmission, in which the controller is the only one to initiate the communication, and communicates with one device at a time, and a response is expected within a time margin.

2.2.3 Art-Net

Art-Net stands for Artistic Network, and it is an implementation of the DMX512 and RDM protocols in an Ethernet network. It is an encapsulation of those technologies and allows the flexibility of any Ethernet network. [11]

Figure 2.11: Art-Net to DMX with support for 8 universes.

It addresses the problem of expandability by allowing multiple universes through a single Art-Net system. It is possible to have up to 32768 universes in the current version.[12]

Since it transmits using UDP packets, it has the typical Ethernet and IP functions, as a DHCP server. In this protocol, each device has an IP and MAC address to which the controller uses to communicate.

2.2.4 Protocol

The ArtDMX packet consists of a header added to the DMX data packet (not considering the start and stop bits). It is formed by:

- 8 bytes with the characters to form the string “Art-Net0”;
- 4 bytes with the OpCode;
- 4 bytes with the protocol version;
- 2 bytes with the even data length up to 512;
- DMX data with the defined length.

2.3 Conclusion

After introducing the technologies referred in the section 2.2, State-of-the-art, there are a few conclusions that can be made regarding the advantages and disadvantages of their use:

- DMX512: It includes a robust differential physical layer with large resistance to electromagnetic noise, which is usually high in stage environments, and it is easy to understand. It is, however, limited to 512 channels and 32 devices in a normal network, and the configuration process, although simple, takes time since on the lighting technician to manually insert the addresses into each device.

- RDM uses a DMX line to allow a two-way communication with the controller and the devices, providing automatic configuration for those devices. But, as the predecessor, relies on a network with limits in size and parameters, and needs specific hardware in order to expand it.

- Art-Net is an encapsulation of the DMX protocol and increases the quantity of universes up to a few thousand in a single Ethernet cable, requiring specific hardware to achieve it.

Having these technologies, the purpose of re-developing a system to control these lighting devices is to include all the advantages those protocols offer, and go beyond it, offering new and more advanced functions, and have one single protocol that can travel from the controller directly into the device without any gateways, with the only exception being the legacy compatibility for DMX devices.

By using standard Ethernet as a base to develop the network, it is possible to expand the network and implement more advanced features to control the lighting equipment. This gives the users a possibility to focus more on the design and programming rather than in configuration, wiring and hardware assembly.

Chapter 3

System Architecture

3.1 Introduction

This section will define the proposed system, beginning with a full description of the architecture, a list of hardware and software requirements, and use-case scenarios. Then, a full description of the solution found will be presented, along with the explanation for the decisions made regarding the choice of technologies.

3.1.1 Use case scenarios

In order to understand the concept of this system, it was necessary to describe its behaviour with fictional scenarios exemplifying its use. By using the following scenarios, plus the general idea of the system, it was also possible to further specify the requirements of the system.

3.1.1.1 Scenario: Physical connections and configuration

"A lighting technician has a lighting console, new and old devices and the respective DMX interfaces. After the initial rigging is complete and all the devices are positioned, the technician begins the wiring process:

- A single Ethernet cable is connected to the output of the console and the input of a network switch, positioned anywhere on stage;
- Another cable is connected to one of the outputs of the switch and into the first device, that has two connectors, one for input and output; this last one is connected to the next device, daisy-chaining them;
- One or more devices are DMX, so they are connected to a small interface with the standard Ethernet in and out and a DMX output, that connects to the DMX device, which are configured for the first DMX address;
- The technician finishes connecting the devices in a star/daisy-chain topology around the central switch;

- The system is then booted up, and the technician assumes the position behind the controller, where the configuration takes place. Assuming the controller is not programmed, the automatic configuration stage takes place and the controller communicates with the devices, detecting them automatically and requesting their control parameters;
- If a configuration does not exist in a DMX interface, the technician receives a notification and a request to upload the configuration file for that device, which includes the DMX addressing;
- After the detection of all devices is completely, they automatically switch to the normal operation mode. The controller is now ready to be programmed, and the devices are now in its database;
- The technician programs the devices for the show to be presented as standard;
- After the programming is completed, the stage is ready to be used as normal. The controller and the devices can be shut down;
- After starting up for the second time, the controller will detect the configuration in the database and will wait for the requests from the devices, confirming their presence in the database, therefore with no need of reconfiguration;
- The lighting system is once again ready to be used."

3.1.1.2 Scenario: Errors and warnings

- "A warning message is generated in the device with an overheating lamp, and sent to the controller; the technician receives the warning and decides to ignore the temperature, since it does not constitute any danger;
- The device then reports one of the motors is failing to respond to the command, so the device fails to change that parameter; The technician decides to send a command to the device to disable that parameter, which means any programmed orders that affect that parameter will be ignored (assuming the console programming cannot be changed);
- The device now reports to the technician a fatal error when the temperature of the lamp is dangerously high, shutting itself off, to prevent more damage. "

The decision of implementing the self-protection mechanisms is entirely up to the manufacturers, and it does not constitute a part of this Dissertation. In this case, the only purpose of this system is to report the warnings and errors to the user.

3.1.2 Requirements

Based on the use-case scenarios in the previous section, the system has the following main requirements:

- Allow flexible network communication;
- Guarantee compatibility with existent DMX devices;
- Minimize the configuration process of all of the devices;
- Allow new functions and system operations;
- Allow the reading of various sensors and reports of exhausted hardware parts (for example, lamps and motors).

These requirements can be expanded and be further divided in hardware and software requirements.

3.1.2.1 Hardware requirements

- Allow multiple network topologies, such as star and daisy-chain;
- Ethernet to DMX interfaces to communicate with these devices;
- Compatible with standard Ethernet switches and devices;

3.1.2.2 Software requirements

- Create a protocol to handle the communications between the devices;
- Automatic device configuration with no need for used interference;
- Configure DMX devices through the implemented interface;
- Synchronized actions for multiple devices with minimal network delay;
- Minimize amount of information sent for continuous actions (controlled movements);
- Possibility to implement advanced functions and system operations in a device;

3.2 System Description

This system is similar to the concept of the DMX and other classic solutions, where a controller and a number of devices are connected hierarchically in a network. This concept has been ported to the current architecture, but has been expanded in a way that allows flexible connections.

Similarly to the DMX ideology, this system consists of a tree topology with a controller on top and the devices hierarchically distributed in the lower levels. The controller is the master of the system, as the operations are originated from there. However, the communication is bidirectional, because the devices can also send information to the controller without being asked. it is the case of two different services, such as the read/write operations, and the notifications and warnings. It is a system that respects the master-slave hierarchy, but has multiple bidirectional services.

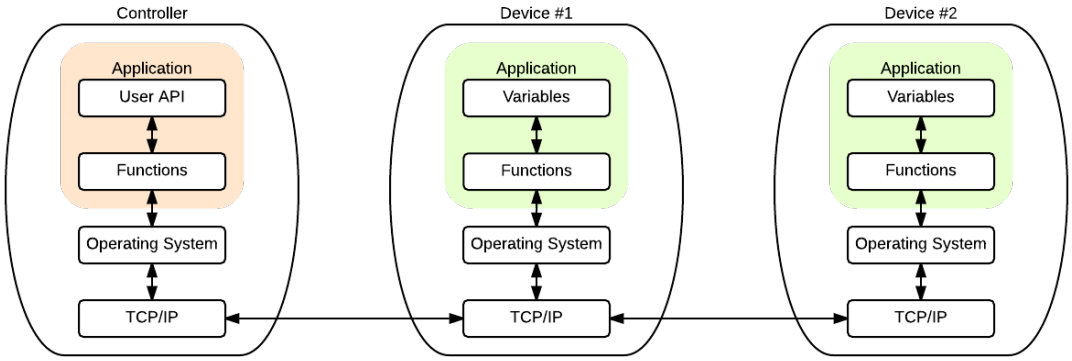

Figure 3.1: Simplified software and hardware layers

The controller and the devices are consist of a hardware layer, operating system and application, as seen on figure 3.1. This architecture has been developed as an application, on top of standard hardware and software. On the controller side, an API is provided with functions to access the variables in each device, functions as the read/write and more complex operations, such as interpolations and custom shapes.

Those complex operations are determined by a set of parameters and are executed independently by each device, which distributes the computation instead of centralizing it in the controller.

3.3 Hardware

This section introduces the hardware solutions found to meet the requirements defined in section 3.1.2.1.

3.3.1 Network Topologies

The DMX systems are connected through a shared bus, daisy-chaining the devices, as described in chapter 2.2. This set-up has, in a way, defined the way the fixtures are positioned in a stage, by keeping the cables between devices as short as possible, to reduce its length and cost.

Designing a system that replaces the DMX communication with an Ethernet based alternative and with the classic star topology would imply the use of switches with enough outputs for each device, which mean that each device would connect to that central switch. This solution would be ideal because it minimizes the number of connections between the controller and the fixtures, but the number of Ethernet cables necessary to connect the numerous devices would largely increase.

Facing this problem, the adopted solution is to follow the one that already exists in the DMX wiring system, allow a daisy-chain connection through every device by embedding a switch with two ports, one for input from the controller or the previous device, and another for the output to other device.

The daisy-chained devices can also be connected to a tree topology network using standard switches, as seen on figure 3.2. The central switch pictured is the one connecting directly to the controller, and all the other switches and devices connect to the network through it. Other switches can be further added to expand the network, always following the tree hierarchy.

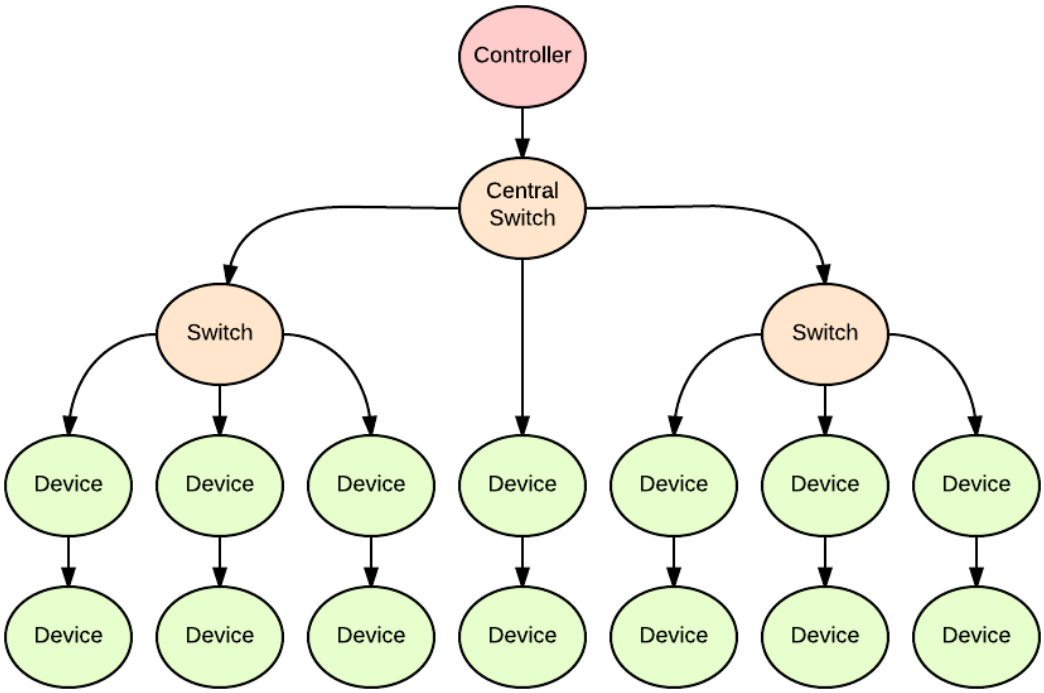

Figure 3.2: Connection diagram with one central switch and multiple devices daisy-chained.

In the previous diagram, the devices in the lowest level are not considered switches, although they are embedded in every device. These, however, only have two external ports, so they simply redirect the packets to the other port if the destination IP does not belong to the device they are embedded in.

This topology also implies that the failure of a device in the middle of a chain would isolate the devices further connected to it from the rest of the network. To find a solution for this problem, it is necessary to study the general network topologies and adapt them for this specific need, which hasn't been achieved in this Dissertation. A relatively simple solution would be to close the daisy chained connections in a ring using an additional switch, but it wouldn't completely solve the problem of islanding.

3.3.2 Physical Interfaces

This section will describe the physical interfaces that connect to the Ethernet network. One of the requirements of this system is to replace the DMX cabling with a reliable solution that will stand

the environment variables such as humidity, heat, and several connections and disconnections.

Since this Dissertation is focusing on the architecture of the system, the physical cabling will not be discussed in the next sections. It is recommended the usage of, at least, a cable protected against dust and water splashing, complying with an IP54 rating at least.

However, an analysis is required to select the best cabling and connectors, which has not been performed in this project.

3.3.2.1 Embedded Ethernet

The new devices that comply with the new system have an embedded Ethernet switch with two exterior ports, for input and output to the next device.

3.3.2.2 Ethernet - DMX

To meet the DMX compatibility requirement, it is necessary to create a device that behaves as an interface between the Ethernet and DMX protocols, and that can be installed close to the DMX fixture. However, the new protocol is different from DMX, as it will be explained in section 3.5. It requires computational power to support new functions and variables, so this interface is a fully functional microprocessor with a DMX server that behaves as any other device, completely transparent to the user, and translates the new functions and variables into DMX channels.

This device has the following requirements:

- A microprocessor compatible with the developed application;
- Support network communication by either:
  - having two Ethernet shields and implement the necessary switch, in the microprocessor, or
  - have one Ethernet shield and one separate 3-port switch to connect to it and the external input and output;

3.3.3 Controller

As mentioned in chapter 2.2, the controller is a device located at the top of the hierarchic tree, similarly to the new system. These are usually regular computers with an outside shell to create the console and a software running on top of a common operating system. Therefore, the requirements for the controllers are easily met:

- Standard computer with a Gigabit network card;
- Operating system capable of network communication;

3.4 Software

This system is almost exclusively software-driven, given that it is based in common hardware solutions. This section will cover the networking services, the protocol and the API developed.

3.4.1 Variables

Before going into the details of the network communication and the protocol, it is necessary to address the variables used within this system. Until now, each device had a specific set of parameters allocated in different channels programmed exclusively by the manufacturer. Similar devices from different manufacturers often have similar sets of variables, but they are often positioned in different channels and can also have different names according to the data sheet. Some of those devices also have different working modes to allow a different number of parameters to be controlled, allowing advanced working modes with extra channels for 16 bit pan and tilt movements, for example. This is not limited to the normal control variables, some devices include system operations in separated DMX channels to allow the controller to reset or switch off the lamps.

To solve the issues of similar variables with different designations, and since similar devices (moving heads, moving washes, LEDs, etc) share the same variable types, it was possible to organize them in three major classes:

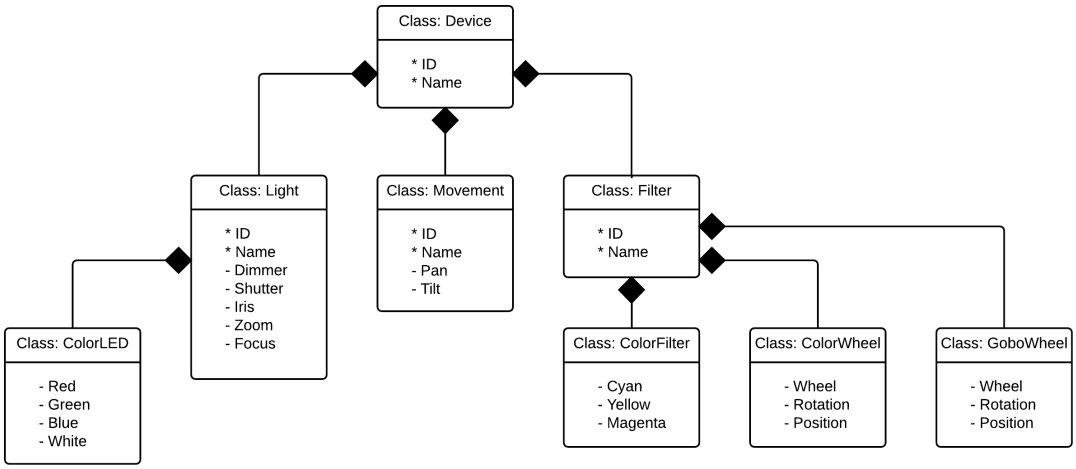

Figure 3.3: Class diagram of the device types.

- Light: Any form of light generation in the fixture;
  - ColorLED: Adds RGB colour mixing;
- Filter: Applied in front of the light to change it shape, color, or any other way;
  - ColorWheel: Rotative colour wheel that changes the colour of the projection;

  - GoboWheel: Rotative shape wheel;
  - Colour filter: Dynamic filters to add CYM colour mix [1];

- Movement: Motors and actuators associated with changing the position or direction of the light or lights.

The class diagram can be seen in figure 3.3 With these classes and their respective subclasses, it is possible to represent any lighting device, as exemplified below:

- *Name of exemplar device: Class(subclass)*
- Dimmer: Light;
- LED Panel: Light (ColourLED);
- Moving Head: Light, Filter (ColourWheel, GoboWheel), Movement;
- Moving Wash: Light, Filter (ColourFilter), Movement;
- LED Wash: Light (ColourLED), Movement.

Each variable is therefore identified by the normalized parameter as mentioned above, and an index, which allows the multiplicity. So, for example, if a LED panel has multiple ColourLED objects, each one is numbered with an index.

To solve the issue of the 8 or 16 bit resolution using a custom amount of channels, the standard size for a variable has been defined as 4 bytes. To the user, the values should appear as a percentage. 0% should equal the value 0, and 100% should equal to 4294967296. It is up to the software to increase the resolution of the percentage.

3.4.1.1 Variable codes

With the variables normalized, the next step is to make them identifiable with a code, shared between the controller and the devices. By using the same ID to identify parameters of the same type, it is easy to implement more functions that can affect a wide range of devices without increasing the network load.

The variables were given the codes in table 3.1, which will allow for the realization of the prototype in the next section.

3.4.1.2 Other devices

Other devices such as fog or smoke machines and lasers need another class for their application, because these do not constitute regular lighting fixtures, which are not being covered in this Dissertation. However, the creation of additional classes is simple and can be easily implemented, so these devices can exist as separated classes in the future.

[1] The difference between the RGB and CYM mixing is in the light source. While RGB consists of three LEDs with independent dimming control, CYM consists of three filters applied in front of the white lamp to achieve the desired colour.

Table 3.1: Variable Codes

Class	Code	Variable
Light	*0x0101*	Dimmer
	0x0102	Shutter
	0x0103	Iris
	0x0104	Zoom
	0x0105	Focus
	0x0106	Frost
ColorLED	*0x0107*	Red
	0x0108	Green
	0x0109	Blue
	0x010A	White
ColorFilter	*0x0201*	Cyan
	0x0202	Yellow
	0x0203	Magenta
ColorWheel	*0x0301*	Wheel
	0x0302	Rotate
	0x0303	Position
GoboWheel	*0x0401*	Wheel
	0x0402	Rotate
	0x0403	Position
Movement	*0x1001*	Pan
	0x1002	Tilt

3.4.2 Services and operating modes

The system supplies four basic networking services:

- TCP: Automatic configuration and device detection;
- TCP: Normal operation to accommodate control functions;
- TCP: Notifications: Warnings, errors;
- UDP: Synchronized operations.

The controller and the devices are clients and servers at the same time, but for different TCP services, so they are based on a hybrid client-server topology, as represented in figure 3.4. In every case, the devices communicate with the controller only, not with other devices. The messages can be divided in two classes:

- Controller-to-device:
  - Normal operations(read/write, interpolation, etc.);
  - Synchronized operations;
- Device-to-controller:

- Controller discovery (if accepted, might initiate configuration request);
- Warnings and errors.

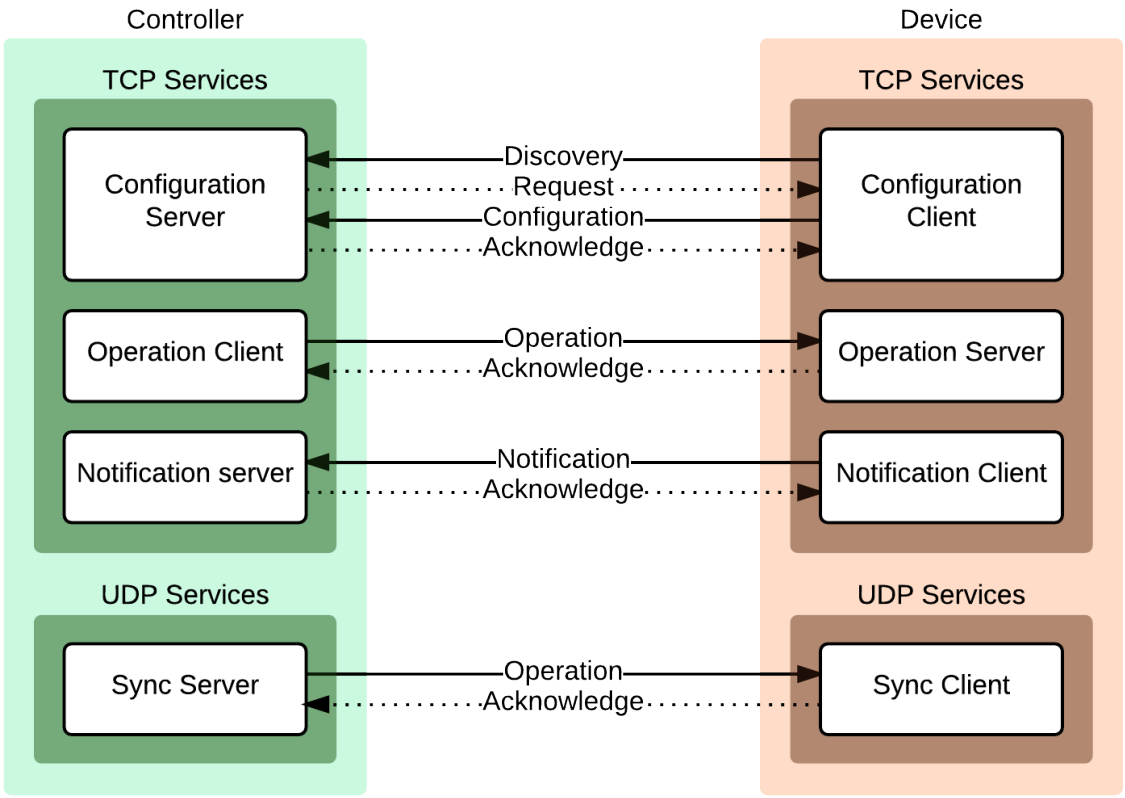

Figure 3.4: TCP and UDP System services represented as clients and servers.

These services must be always available from the controller side, but the devices are divided in two major operation modes: Configuration and normal operation, where the first is required to be completed before the second is available. This guarantees that the devices are configured and have communicated with the controller prior to any other operations.

3.5 Protocol

The protocol developed for TCP communications within this system consists of a packet containing a header and the data messages. The header is composed by an unique transaction identifier, message type which will be used to synchronize operations, a function identifier, and the data length. The result header is represented in table 3.2.

UDP communication will be specified in section 3.5.7.

In the following sections, the packets will be shown as short tables with the data only to demonstrate the construction of the messages. Each of those tables will refer to the detailed version with the header, which can be consulted in the appendix B.

Table 3.2: Header of the TCP packet

Header		
Message	Size	Example
Transaction ID	8 bytes	*0x42616420576F6C66*
Message type	1 byte	*0x00*
Function	2 bytes	*0x0001*
Data length	2 bytes	*0x0001*

3.5.1 Variables

The DMX protocol specifies that every slot has a resolution of 8 bits, or 256. Some devices use an extra channel for 16 bit resolutions, which requires the driver installed in the controller to detect this channel and use it as a field with 65536 positions.

For the new system, the predefined variable resolution is 4 bytes or 32 bits, which gives a total number of 4294967296 positions. However, it is not necessary to use such a resolution in every variable. The protocol assumes a four byte long variable, so it will scale the given variable length, shifting it to the left until the most significant bit is reached, and append zeros to the end, therefore creating the default length.

For example, a DMX interface parameter only has a one byte resolution. If that variable is passed as a character, the system will automatically convert it as seen in the next table 3.3.

Table 3.3: Byte scaling to a four byte resolution

Before shifting			
-	-	-	0xFF
Shift left until most significant byte			
-	-	0xFF	0x00
-	0xFF	0x00	0x00
0xFF	0x00	0x00	0x00

3.5.2 Functions

The function codes were divided in three classes, Configuration, Operations and Notifications, to represent the main working modes of the system. The following lists are not limited to the current elements, it will always be possible to add more functions and expand the capabilities of the system.

3.5.2.1 Configuration

During the configuration, messages are uniquely identified by the function, which represents the destination and the source.

- *0x0001* (D2C[2]): Discovery with device configuration available. Name, manufacturer and model of the device in the data message;
- *0x0002* (C2D[3]): Requesting configuration;
- *0x0003* (D2C): Configuration;
- *0x0004* (C2D): Acknowledge, enable normal operation.
- *0x0005* (D2C) Discovery from a DMX interface. If data length equals zero, no DMX file is present;
- *0x0006* (C2D): Sending DMX configuration;

3.5.2.2 Operations

The operations can enable the default control variables such as dimmer, pan, filters, or can be maintenance settings, which are identified with the first byte *0x0F*. The priority of an operation is defined by the highest transaction ID.

Below is a list of the operational functions developed for this project:

- *0x0101*: Write variable;
- *0x0102*: Read variable;
- *0x0111*: Timed linear interpolation with *n* points;
- *0x0112*: Timed cubic interpolation with *n* points;
- *0x0F03*: Disable motor or actuator of the specified variable/parameter;
- *0x0F03*: Disable motor or actuator of the specified variable/parameter;
- *0x0F01*: Enable operations (for maintenance purposes);
- *0x0F02*: Disable operations;
- *0x0F03*: Enable motor or actuator of the specified variable/parameter;
- *0x0F04*: Disable motor or actuator of the specified variable/parameter;
- *0x0FFF*: Reset device;

[2]Device to Controller
[3]Controller to Device

3.5.2.3 Notifications

The notifications provide warnings and errors developed in the device and having the controller as destination. There are four types of notifications: syntax, warnings, non-fatal errors and fatal errors.

A syntax notification happens after an invalid function was sent to the device, which will not change any state or variable.

A warning is sent to the controller after any physical fault is detected and maintenance is recommended, but not necessary.

A non-fatal error specifies that a part of the device is not functioning properly and has been disabled. Since a device can have a certain number of motors, the malfunctioning of a non-critical actuator should not affect the overall performance of the device. For example, if the motor activating a frost filter is not working, the other functions can still be used, not completely limiting the usage of that specific fixture.

A fatal error is generated when the device has activated any sort of protection to prevent more damage, either by shutting itself down or disabling full functionality.

- *0xF001*: Syntax: Unspecified;
- *0xF002*: Syntax: Device is in standby mode;
- *0xF003*: Syntax: Invalid device parameter;
- *0xF101*: Warning: Unspecified;
- *0xF102*: Warning: Device operating at high temperature;
- *0xF103*: Warning: Lamp or LED needs replacement;
- *0xF201*: Non-fatal Error: Unspecified;
- *0xF202*: Non-fatal Error: Device motor failure;
- *0xF301*: Fatal Error: Unspecified;
- *0xF302*: Fatal Error: Excessive device temperature;
- *0xF303*: Fatal Error:

3.5.3 Device detection and configuration

This system supports the automatic configuration of the devices connected to the network. This service relies on a client-server connection, in which the controller is a server and the devices the clients, and is a four step process represented in figures 3.5 and 3.6.

3.5.3.1 Embedded Ethernet devices

The process consists of:

- Search for a controller by attempting a TCP connection to the default controller IP;
- Reply from the controller which may or may not request the configuration of the device;
- Send configuration;
- Receive acknowledge.

The first message contains the device type, MAC address, name and manufacturer. After receiving it, the controller searches its database for the MAC address, which is the unique identifier. If the configuration is already present, an acknowledgement message is sent to enable the device's normal operation mode. If not, the controller will send a request to the device to send the configuration, which will then be inserted in the controller's database and an acknowledge message sent to finish the configuration phase.

The configuration requests and acknowledges have null data, so the messages are only the header with the specific function code. The discovery and configuration packets can be seen in tables 3.4 and 3.5.

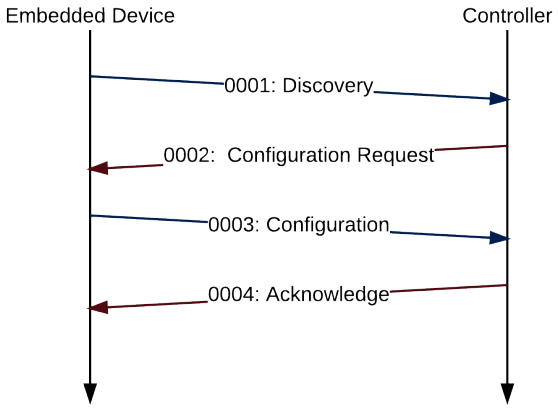

Figure 3.5: Diagram of the configuration steps for an embedded device.

Table 3.4: Discovery packet, short version of table B.7

MAC	Name Len.	Name	Manufacturer Len.	Manufacturer	Model Len.	Model

Table 3.5: Configuration packet, short version of table B.2

Number of variables	ID 1	Class 1	Index 1	ID 2	Class 2	Index 2

3.5.3.2 Ethernet - DMX interfaces for legacy devices

The Ethernet - DMX devices are configured with a host configuration file, interfacing the normalized variables with the DMX channels. The configuration steps are similar to the previous ones, but depend on the existence of such file in the device:

- If configuration file exists:
  - Send discovery message with the device type (DMX), mac address, name and manufacturer;
  - Reply from the controller which may request the configuration;
  - Configuration is send if requested;
  - Controller acknowledges.
- If configuration file does not exist:
  - Send discovery message with device type (DMX), mac address but no other details;
  - Controller sends the DMX configuration the user has selected;
  - Device resends discovery message with the MAC address, name and manufacturer;
  - Controller acknowledges.

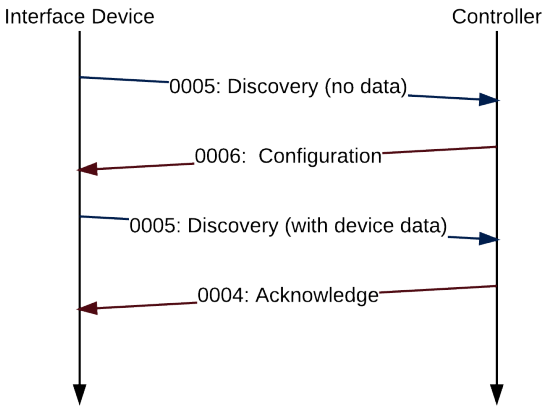

Figure 3.6: Diagram of the configuration steps for an DMX interface device.

The packets for these operations are similar to the mentioned above for Embedded Ethernet devices. The only differences are in the function code and the direction of the configuration. Before, the controller requested the configuration of the device, but in this case, the controller is the one defining that configuration. The configuration packet for these devices is in table 3.6.

The DMX channel goes up to 512, so two bytes will be used to represent it in the packet.

In case a DMX device has one or more parameters with a 16 bit resolution, the configuration will assign the two channels when converting into DMX signals. To define this in the configuration, it is necessary to send two variables with the same class and index codes. The most significant byte should be first, followed by the least significant byte.

Table 3.6: Configuration packet for DMX interfaces, short version of table B.6

Number of variables	Class 1	Index 1	Channel 1	Class 2	Index 2	Channel 2

The full packets can be consulted in the Appendix B,

3.5.4 Read/Write operations

By using the function codes 0x0101 and 0x0102, it is possible to request to a device to write and read variables, respectively. It is also possible to send multiple variables to read or write to a specific device.

To identify a variable, two parameters are necessary, the variable code and the index. For the write message, the number of variables to write is specified, followed by the first variable code, index and value.

For each variable to write, the variable code and index are sent. The write operation is similar, adding the value to write after the identifiers.

The packet construction can be seen in tables 3.7 and 3.8. More detailed versions of those tables can be seen in the Appendix B, specifically the tables B.9 and B.8.

Table 3.7: Read packet, short version of table B.9

TID	Type	Function	Length	Num. Var	Class 1	Index 1	Class 2	Index 2

Table 3.8: Write packet, short version of table B.8

TID	Type	Function	Length	Number of Variables	Class 1	Index 1	Val 1

3.5.5 System operations

The DMX devices allow the usage of specific channels to access system operations, such as resetting the device, shutting down the lamp, reversing the pan or tilt, and others defined by the manufacturers. In this system, those operations are divided from the common read/write functions and have specific codes to access those special functions.

3.5.6 Interpolation

This system allows the usage of distributed computing to perform interpolations and advanced operations that require precision and timing. Previously, all the computing power was located in the controller, and the instructions were sent to the device with the refresh rate of the DMX message. With the new system, it is possible to send a single instruction to a device to perform an interpolation or spline with timed precision.

For example, a slow tilt movement in a common DMX moving head is possible by either continuously updating the position from the controller, or setting the speed through a specific

channel. The first option, though, lacks precision and the second does not guarantee that the position will vary from one point to the other in the required time. This system, however, does.

The interpolation capabilities within this system have not been entirely studied, and it has not been implemented within the network. But, since the protocol was developed with the intuit of easily implementing new functions, it will be possible to do so in the nearby future.

It is recommended that the interpolation commands remain as short-timed as possible, since the clocks of the controller and the devices may not be synchronized, which could lead to the operations ending sooner or later than expected. However, the difference in modern microprocessors is minimal, and should not make a visible difference.

A few different interpolations can be implemented:

- Timed linear interpolation with three parameters: Start position, end position and time. The device should then calculate the speed of the motor to reach from the first to the end positions within the given time;
- Timed cubic interpolation with three or more points, to give the possibility of advanced movements and shapes.

An interpolation can be stopped by sending a write command to the same variable, or by another interpolation. Any operation that implies the change of the variable under an interpolation will stop it.

An extra parameter can be added to repeat the process after it is finished, allowing, for example, a looping "wave" effect, or a continuous shutter blink. This parameter should be used carefully, because, as mentioned above, the differences in clock synchronization may lead to different results if a number of devices is executing the same code.

By using this feature, the user does not have any automatic feedback of the state of the interpolation, only knowing the defined points and time. To solve this, the software within the controller can periodically send requests to read the variable being affected by the operation. This period does not need to be too short if it is only for the user to see the current values.

3.5.7 Synchronized operations

Another feature of this system is the possibility to declare a certain operation as synchronized. This allows the user to send an operation packet with the normal functions, but instead of the device executing them as soon as they are received, they are stored and will wait for a synchronization signal, which will deploy that operation.

This feature is useful for situations where a large number of devices is in use and they are required to execute a command at the same time. The commands are sent to each device as synchronized, and will stay stored along with the transaction ID. To deploy them, the controller simply broadcasts the transaction ID, and the devices only need to verify if there is a suspended operation with the same identifier as the received.

For this feature, a few requirements are necessary:

- The transaction ID must identify the same operation across all devices, so only one broadcast is necessary;
- Each device should be able to hold multiple synchronized operations and should deploy them in order of arrival of the respective broadcasts that trigger them. This bypasses the previous rule that uses the transaction ID to identify the operation with the highest priority;
- Every synchronized operation has a timeout, otherwise it can be stored indefinitely, which can cause memory leaks;
- Acknowledge messages are sent after the device successfully stores the synchronized operation, not after the same operation is triggered.

3.5.7.1 Network and Protocol

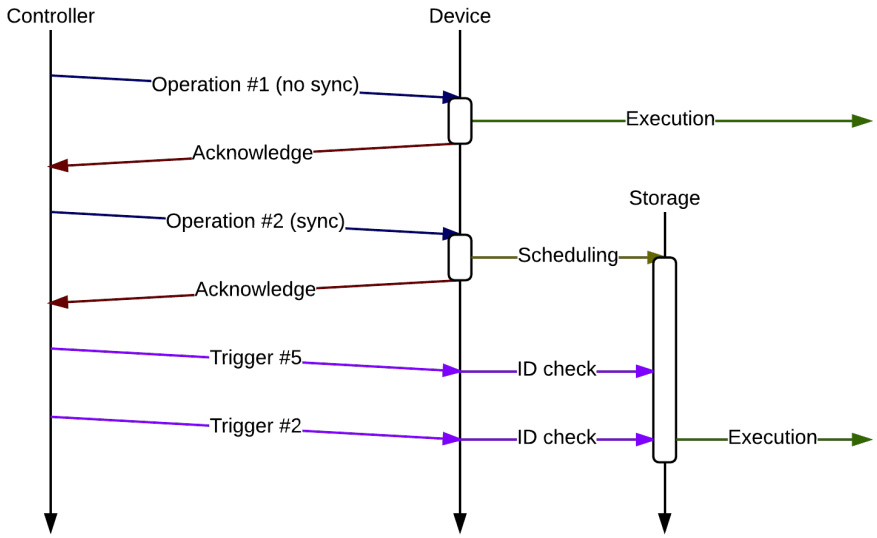

Figure 3.7: Storing and deploying a synchronizing an operation

The synchronization requires that all the devices receive a broadcast message with the transaction ID. This is achieved by sending the transaction ID to the default IP broadcast address, 255.255.255.255. This is the fourth service mentioned in section 3.4.2 and it consists of a client-server connection between the controller as server and all of the devices as clients. The connection diagram can be seen in figure 3.7.

The header of a synchronized operation is the only difference from a normal operation, by using the field *Message type* and changing it to 0x01, as shown in table 3.9. The received Acknowledge message is the header of the operation, and confirms that it has been received and stored, waiting to be triggered.

Table 3.9: Header of the synchronized TCP packet

Header		
Message	Size	Example
Transaction ID	8 bytes	*0x42616420576F6C66*
Message type	1 byte	***0x01***
Function	2 bytes	*0x0001*
Data length	2 bytes	*0x0001*

Since the purpose of this communication is to transmit the transaction ID, which is already an unique value, the packet configuration does not have a header and data configuration, it is simply the eight bytes that represent the transaction ID.

3.6 Conclusion

This concludes the section of the system architecture. Although the requirements are fully met, the system is still far from being able to be implemented in a real stage situation, which would require a deeper study of the hardware specifications and the device classes.

It is, however, a system that offers many advantages in comparison with the previous protocols. Since the base has been developed, the missing parts should be easy to develop and implement.

The network topology proposed demonstrated the flexibility of the network if used with a star or daisy-chain models. However, the best topology for such a system would be a purely star model, having the switches connecting to a single device, without any daisy chaining to avoid islanding. For some cases this solution wouldn't be possible due to the spacial requirements. The devices are, therefore, prepared to handle multiple network topologies.

Chapter 4

Implementation

4.1 Introduction

The second part of this Dissertation was the implementation of the developed system into a prototype. This section will cover the development platforms, the implementation, and the final results.

The developed prototype is an Ethernet/DMX Interface, which is the most obvious choice, given that there are no devices at the moment that comply with the new system.

4.2 Development Platforms

Initially, this project was expected to be equally hardware and software driven. But, since the requirements defined the use of standard Ethernet based solutions, the hardware became less specific and complex, which allowed the development of more advanced software functions. The prototype developed is, therefore, a proof-of-concept that has been developed on standard platforms, but it should be easily portable to others, more specific for a commercial application of this system.

4.2.1 Hardware

This section will describe the hardware used and its respective assembly to create the final prototype.

Since this prototype is a DMX interface, it needs to be able to generate the DMX signal to feed a standard DMX device. Therefore, the requirements are as follows:

- Standard development board with Ethernet;
- Serial port capable of generating a signal at 250 *kBaud*;

4.2.1.1 Selected hardware and its requirements

The hardware used for the development is a standard Raspberry Pi model B, with a Linux based operating system. This board already has a fully functional Ethernet port with network support from the operating system.

Since the DMX protocol uses the RS-485 physical layer, and the Pi has a serial port, it was necessary to use a MAX485 to create the differential signal. Therefore, the circuit in figure 4.1 was proposed.

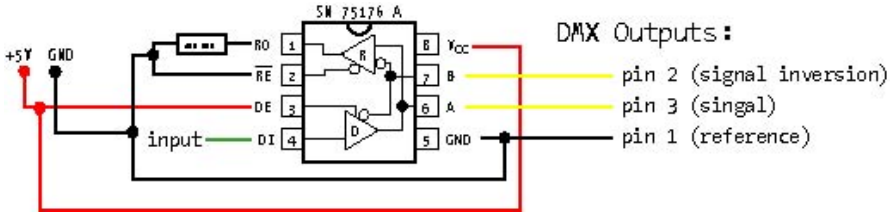

Figure 4.1: RS-232 to RS-485 circuit.

The input of the MAX485 is connected to the UART TX pin 8 of the Raspberry Pi, and the value of the resistor is 100 Ω. Since the DMX protocol is unidirectional, so only the transmission pin was used.

The final assembled prototype can be seen in figure 4.10.

4.2.2 Software

The prototype was developed using a Raspberry Pi with a Linux image for the development board. Java was the language selected for this application for its compatibility with sockets and general network services provided by the operating systems. However, the pin and serial support for this hardware was not working as expected, so an additional application was developed in C to allow the usage of the UART ports from the Raspberry Pi.

4.3 Protocol Layers

The software of the controller and the device was organized with protocol layers, which allows the presence of an API in the highest level providing the basic access functions to the user. This API is meant to be used directly with the controller's software.

For this prototype, only two of the four services were implemented: the operations and configuration. The layer organization is similar in both ends, the controller side and the device side:

- Configuration:
  - Configuration;
  - Configuration wrapper;
  - Network;
- Operations:
  - API;

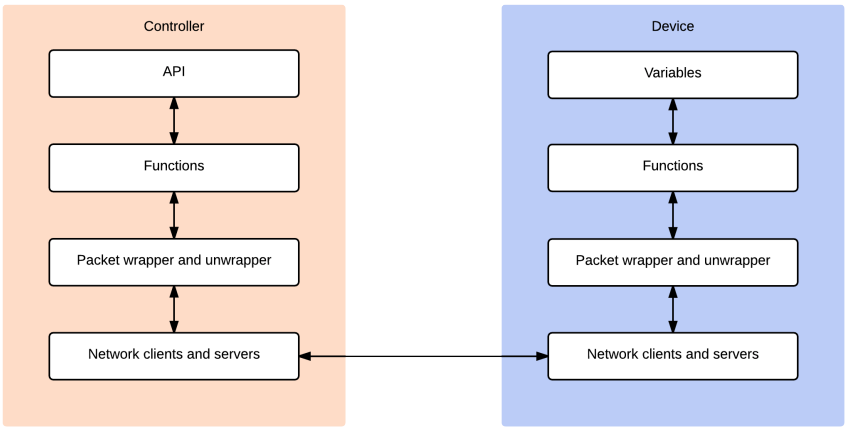

Figure 4.2: General protocol layer diagram.

- Functions;
- Packet wrapper;
- Network.

The protocol layers, in the system, represent the different classes running in separated threads. This method provides the concurrent execution of the operations.

In the next sections it will be explained how these layers communicate and what is their purpose in the system.

4.4 Threads and synchronization

Each protocol layer represents a different thread in the system, which means that for a single operation, three threads are created to accommodate the bidirectional communication. These threads, however, do not execute at the same time. Similarly to a token-passing system, only one is executing code at a time.

This was achieved by taking advantage of the Java's interrupted exceptions during a thread sleep. For example, a Runnable class that sleeps by default such as:

```
public class ExampleSleeping implements Runnable
{
  public void run()
  {
    while(true)
    {
      try
      {
        Thread.sleep(1000);
      } catch (InterruptedException e)
```

```
        {
          do_something();
        }
      }
    }
}
```

And initialized by:

```
Thread sleepyThread = new Thread(new ExampleSleeping);
sleepyThread.run();
```

Will be sleeping and can be referenced by the name “sleepyThread”. By passing this thread reference to another thread during its initialization, it is possible to execute:

```
referenceToTheSleepyThread.interrupt();
```

Which will send an interrupt to the first thread and allow the execution of the “do_something()” function. By using this concept with a state machine, it was possible to synchronize the execution of the threads, while maintaining the flexibility of the protocol layering.

4.5 Device and Variable identification

To access a variable, three parameters are necessary: the identification of the device, the variable name and the index. The names of the variables are defined according to section 3.4.1. However, it is still necessary to identify the devices.

To do so, when the controller receives the first discovery message from the devices, it also receives the information of each one of them, such as the name and manufacturer. Using the name, it generates a short version that the user can easily identify and use to access that device, and since there might have devices with the same name, these are numbered as well.

For example, a couple devices with the name “LED RGB”, as the one used in this prototype, will have a generated name of “LEDRGB”, and will be identified by an index: “LEDRGB.1”, “LEDRGB.2”, and so on. Similarly to this process, each variable is identified by its name and index: “Dimmer.0”, ColorWheel.0”, “ColorWheel.1”, etc.

Using this nomenclature, a variable is accessed by the name of the device followed by the variable: “LEDRGB.0.Red.0”, “LEDRGB.0.Green.0”, “LEDRGB.0.Blue.0”, “LEDRGB.0.Shutter.0”. These represent the variables of the current DMX prototype in the new system.

4.6 Configuration

There are two types of the available configurations for the new devices and the legacy devices, as described in the system architecture. The configuration of native devices has been successfully

implemented, and the legacy DMX has been partially implemented, for it was assumed that the DMX configuration was already present. In this case, the controller did not have the configuration, so it requested it just as a native device, so the process is very similar, only differing in the function header.

A diagram of protocol layers used in the native configuration process can be seen in figure 4.3.

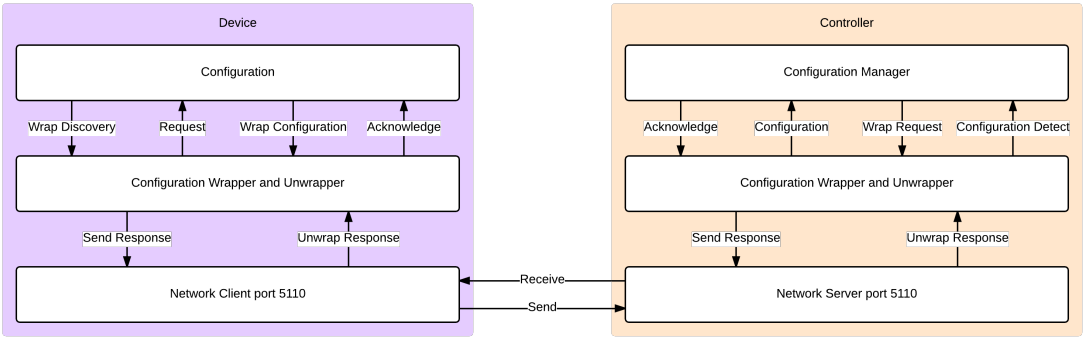

Figure 4.3: Configuration protocol layers.

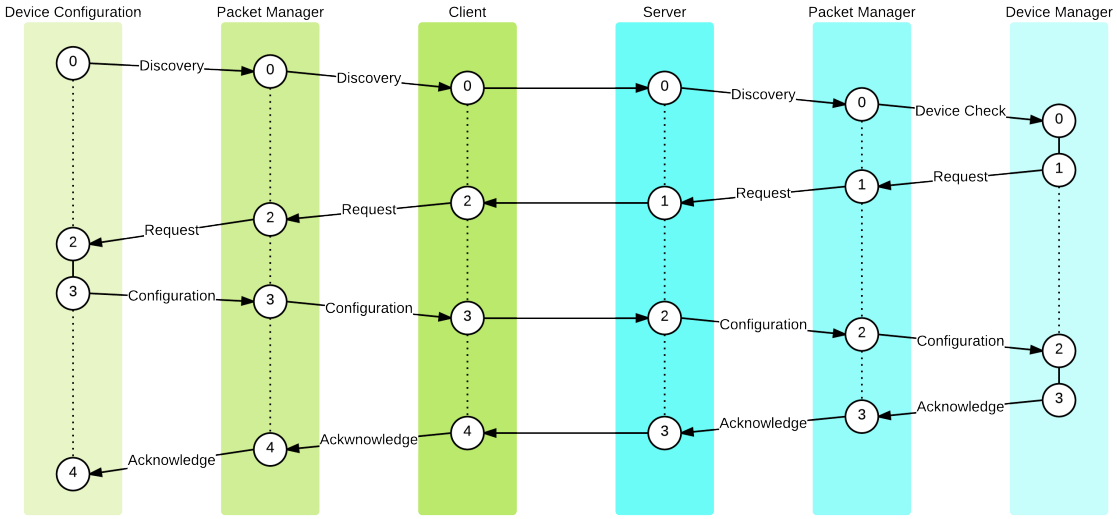

Figure 4.4: State machine diagram for the configuration process between a device and a controller

Figure 4.4 represents the state machines used in the complete configuration process. Each protocol layer is declared as a Runnable Java object with a state machine embedded. As the communication moves to the other layers, the previous ones enter in a sleep state, waiting to be interrupted by the returning message. This figure, however, only represents the states with code executing work. In reality, each time a thread is suspended, it represents an extra sleep state, which does not execute any code other than the *Thread.sleep()*, while waiting for the interruption

from other threads. The source code used in this prototype for the configuration process can be consulted in Appendix C.

4.6.1 Device

As seen on figure 4.4, the device has three protocol layers implemented for the configuration process. Each layer represents a thread, and they communicate between themselves by sharing variable references, passed during the initialization.

After the device is properly initialized, with the operating system loaded, the prototype software creates a thread to configure itself as seen in picture 4.5. This configuration goes through the following steps:

- Detect current configuration, if it is a legacy or new device, and if it has the configuration files;
- Create a configuration request based on the existence of the file;
- Attempt a TCP socket connection through the port 5110 to the controller default ID, which was 192.168.1.1 in the case of this prototype;
- Retry the connection if it fails;
- After connecting to a controller, send the configuration if requested;
- Terminate configuration when receiving the Acknowledge command.

The Acknowledge command from the controller will end the configuration at any time, and will enable the normal operations.

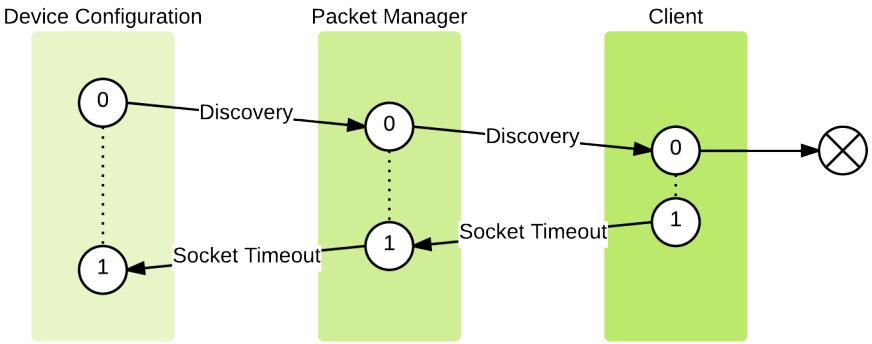

Figure 4.5: State machine diagram for the configuration process between a device and a controller

4.6.2 Controller

The controller has an embedded MySQL database to physically save the devices and their configuration. It has three basic tables:

- Device: Where all the information about each device is recorded. During the initialization of the devices, the controller will search the database for the device's MAC addresses which are their unique identifier, and request the configuration if missing;
  - Device ID;
  - MAC Address: The unique identifier of a device;
  - IP address: Initialized to null, when an IP is present the device is active;
  - Name: Name of the device;
  - Short Name: Generated automatically from the Name without any spaces for variable identification purposes;
  - Manufacturer;
- Parameters: Table to record the different available device parameters and their respective normalized variable codes;
  - Variable ID: A two bit value identifying the variable code;
  - Variable Name: String used to identify the normalized variable names;
- DeviceParameters: Assigns the variables to the device.
  - Device ID;
  - Variable ID;
  - Index;

When receiving the discovery message from a device, the controller will verify its existence in the database, based on the MAC address. If it exists and is configured, the reply will be an acknowledge message. If not, the controller will add the device to the Device table and request the configuration, which will then be inserted in the DeviceParameters table, followed by an Acknowledge message.

The case of a DMX discovery message with no data (no DMX information) has not been implemented, but in this case, it should request the user the configuration files.

4.7 Operations

4.7.1 Controller

The controller has three available functions, two for writing and one for reading values, which will be explained below. These functions do not block the main thread, because they create separated threads for the execution. Being non-blocking, they can be run multiple times for different devices, and will not interfere with other functions.

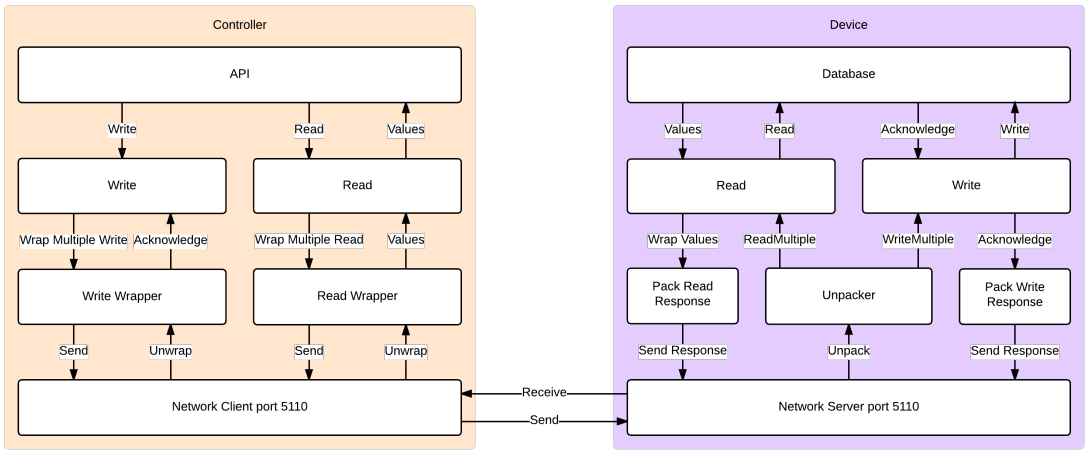

Figure 4.6: Operations protocol layers.

4.7.1.1 Function Write

Two prototypes are available through the API to access the writing functions.

```
public class Write
{
  public static void WriteMultipleInstant(String[] names, int[] values);
  public static long WriteMultipleSynced(String[] names, int[] values);
}
```

The first represents a single, instantaneous write command, and the second represents a synchronized operation that returns the transaction ID to be later deployed by accessing the following function, which takes the transaction ID as parameter to deploy the scheduled operation:

```
public class Sync
{
  public static void DeploySyncedAction(long transID);
}
```

The only difference between the two write functions is in the declaration of the header. A synchronized operation will set the message type to 0x01, while a normal operation will be identified by 0x00.

This function takes a array of strings and an array of integers as parameters of the same size, where the first one is the name of the variables to write as defined in section 4.5, and the second one is the respective values to write. It is possible to access variables of different devices, but since this prototype was the only one available, the possible variables are:

```
String[] names = new String[]{"RGBLED.0.Shutter.0", "RGBLED.0.Red.0",
    "RGBLED.0.Green.0", "RGBLED.0.Blue.0"};
```

```
int[] values = new int[]{255, 255, 0, 255};

Write.WriteMultipleInstant(names, values);
```

This would set the LED colours to pink, with the shutter to full. The values used are the DMX values, which are only one byte long. The four byte long requirement has not been completely implemented, however the space is already reserved in the packet, so all the four bytes are sent.

This function resolves the IPs for each device and the variable codes, checking their existence, throwing errors in case they don't, and organizes the variables of each device, creating separated packets in case the names belong to different devices:

```
public class PackWrite implements Runnable
{
  public PackWrite(String ip, boolean sync, long transactionID, int[]
      variables, int[] indexes, int[] values);
}
```

The PackWrite function takes the IP and values and creates the packet as seen in table B.8, which is then sent through a TCP socket on port 5112.

4.7.1.2 Function Read

This function is very similar to the function write above, except it does not take any values to be written, and the function code is obviously different. The available prototype is:

```
public class Write
{
  public static void ReadMultiple(String[] names, int[] results);
}
```

Where it takes as parameters the names of the devices as defined in section 4.5 and an array where the read values will appear. These arrays must have the same length.

4.7.2 Devices

The devices have a TCP server operating on port 5112, listening to the communications from the controller. When a packet is received, a thread of the class Unpack is created, which reads the packet header and redirects it to the correct function in the protocol layer above.

In this case, a write function in the header redirects the packet to the class UnpackWrite, which will take the data and extract the values to be written:

```
public class Unpack implements Runnable
{
  public Unpack(byte[] input, int[] result, byte[] output, Thread
      lastThread);
```

```
}

public class Write implements Runnable
{
  public Write(byte[] input, int[] ids, int[] indexes, int[] vars);
}
```

The arrays are passed by reference, so it is guaranteed that their value does not change across the threads, which requires a precise synchronization between the threads as they execute.

The variables are then inserted into the database, which the device will then use to update its outputs.

4.8 DMX Module

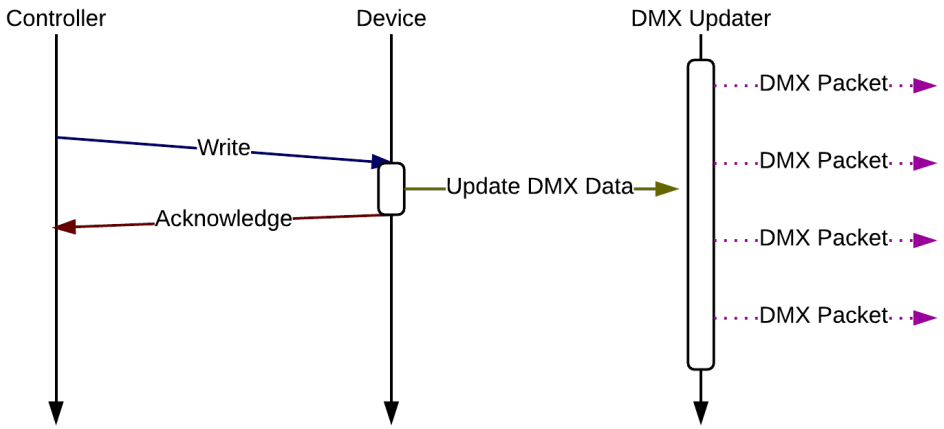

Figure 4.7: The DMX Module.

The DMX module was implemented partially in Java, partially in C, because the required libraries for Java serial communication using the Raspberry Pi did not prove to work. The DMX channel updater was implemented in Java, and the packet generation and serial communication was implemented in C, communicating through a localhost socket connection.

4.8.1 DMX packet generation

The DMX packet generation was expected to be exactly like the standard mentioned in chapter 2.2, with the minimal values as also seen in figure 4.8.

However, after the initial testing, by enabling the four channels addressed to the Lightmaxx, it has been discovered that the LEDs were blinking randomly. With further testing, the problem

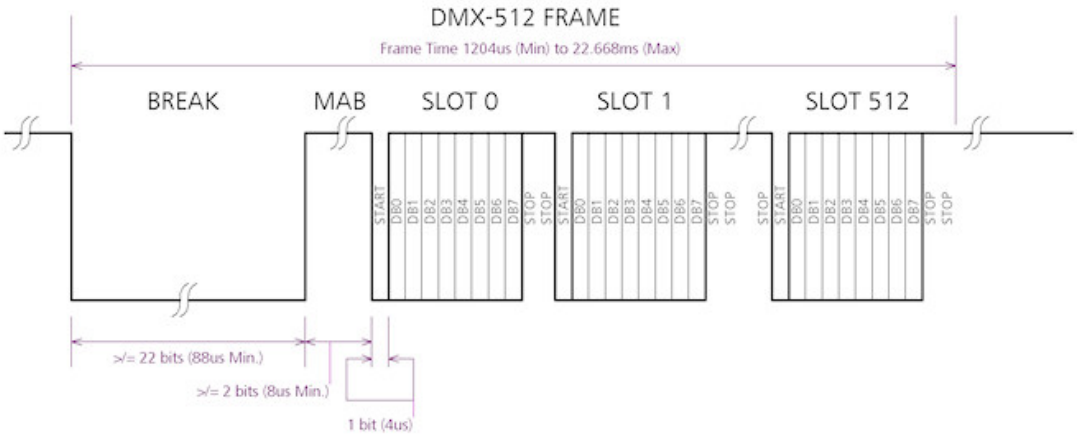

Figure 4.8: Representation of the DMX packet with the minimal values.

was addressed to the values of the Break and the MAB, Mark-After-Break, which proved to not be enough to stabilize the connection. The minimal Break time is at least 22 bits long, but has been set to 80, and the MAB time has been set to 24.

This obviously creates more delay on the packet transmission time, but has proven to stabilize the connection. Plus, since there are only four channels occupied, the packet renewal time is minimal, and the effect does not affect the final performance.

4.8.2 Serial communication in C with the Raspberry Pi

The DMX communication occurs at a baud rate of 250 kBaud. However, the Raspberry Pi only supported, with an alteration to the UART clock, the values of 230400 Baud, or 500000 Baud, both not supported by the DMX devices. Using this UART also would create another issue: it has only one static stop bit, where the DMX protocol requires two.

In order to achieve the 250000 Baud, the clock of the UART was changed to 2 MBaud, which is exactly eight times greater than the desired rate. By doing so, theoretically, the transmission time of a byte, 8 bits, at 2 MBaud is exactly the same time one single bit was read at 250 kBaud. By using this method, and reducing the number of bits to 6 to compensate for the extra start and stop bit, therefore creating the single byte, it was possible to transmit the DMX packet.

Every time a byte was sent, a byte that was composed by the start bit, six data bits and the stop bit, only one single bit was being read from the DMX fixture. With this, it was possible to create the desired start and stop bits that the DMX packet needed, but it also required the manual construction of the entire packet.

The clock of the UART was modified to achieve its speed by adding this line to the file “/boot/-config.txt”:

```
init_uart_clock=16000000
```

The UART was then initialized to 2 MBaud with the following function:

```
int openUart(int uart)
{
  uart = open("/dev/ttyAMA0", O_RDWR | O_NDELAY);
  struct termios options;
  tcgetattr(uart, &options);

  options.c_cflag = B2000000 | CS6;
  options.c_iflag = IGNPAR;
  options.c_oflag = 0;
  options.c_lflag = 0;

  tcflush(uart, TCIFLUSH);
  tcsetattr(uart, TCSANOW, &options);

  return uart;
}
```

The packet was generated byte by byte, according to the number of available channels:

```
unsigned char tx_buffer[1024];
unsigned char *p_tx_buffer;
(...)
// BREAK
for (i = 0; i < 80; i++)
  *p_tx_buffer++ = ZERO;
// MAB
for (i = 0; i < 24; i++)
  *p_tx_buffer++ = ONE;

for (int l = 0; l < allocated_channels; l++)
{
  int tempbuf = channelBuffer[l];

  // Start bit
  *p_tx_buffer++ = ZERO;

  // 8 Data bits
  for (i = 0; i < 8; i++)
    if (tempbuf % 2 == 0) { *p_tx_buffer++ = ZERO; } else *p_tx_buffer++
        = ONE; tempbuf = tempbuf >> 1;

  // Stop bits
  *p_tx_buffer++ = ONE;
  *p_tx_buffer++ = ONE;
```

```
}
```

Where “ZERO” and “ONE” are equal to 0 and 63, respectively, which is the integer value of six bits.

This method was possible because the DMX protocol does not offer bidirectional communication, so the prototype only needed to send the DMX messages without expecting a response. However, the Raspberry was used only as a development board, and its UART was used for testing purposes, so it is not meant to be implemented in a final prototype.

4.9 Partially and Not Implemented Methods

This section describes some of the functions that have been proposed on chapter 3, but only have been partially implemented or have not been implemented at all.

4.9.1 Interpolation

The interpolations are part of the advanced features that distribute the computation among the devices in the system, rather than focusing them on the controller, relying on the communication network to update the device states periodically. This feature has been partially implemented as a linear, two points interpolation running on the device prototype, with support for one variable only. The prototype to the implemented function can be seen below, and the complete version can be consulted in Appendix C:

```
public class LinearTwoValueSimpleSpline
{
   public LinearTwoValueSimpleSpline(int id, int index, int val1, int
       val2, int period);
}
```

It consists of a thread created to change the value of a specific variable, and takes as parameters the variable code and index, the initial value, final and period. Once started, it will create the given interpolation going from the initial value to the final in the specified period.

This method was implemented in the Device side only. It has, however, proven to work, although the period had a deviation of a few milliseconds. For this reason, the recommendation is to keep the period as low as possible, to reduce the deviations and to prevent devices from running out of sync if executing the same interpolations.

4.9.2 Notifications

The notifications are part of a service that has not been implemented in the prototype. It would have had two different purposes: To notify the controller of errors and warnings, and to read a variable periodically. Simply put, it would act as a publisher-subscription system, sending to the controller the state of a variable or variables.

Figure 4.9: Raspberry Pi and Fame Lightmaxx.

To enable this subscription, the same principle of the synchronized write would be used, but the device would have to interpret it as a subscription, and it would send the read packets with the variables requested to the controller periodically. This would also need a function to terminate the subscription, which the controller would send

4.9.3 Synchronized Operations

The synchronized writing function has been implemented, as it can be consulted in section 4.7.1.1. It has been implemented to work with that function only, but should support every kind of operation destined to change variables, hence the need to change only the header of the packet. This would allow the presence of synchronized interpolations, for example, which could be deployed simultaneously.

4.10 Tests and Validation

To test the concept, a Fame LightMaxx LED strip was used with the prototype. This is a DMX device with four channels: Shutter, Red, Green and Blue. The DMX input was connected to the UART output of the PI through a MAX485 circuit, as seen in figure 4.10.

To test the concept, a Fame LightMaxx LED strip was used with the prototype. This is a DMX device with four channels: Shutter, Red, Green and Blue. The DMX input was connected to the UART output of the PI through a MAX485 circuit, as seen in figure 4.10.

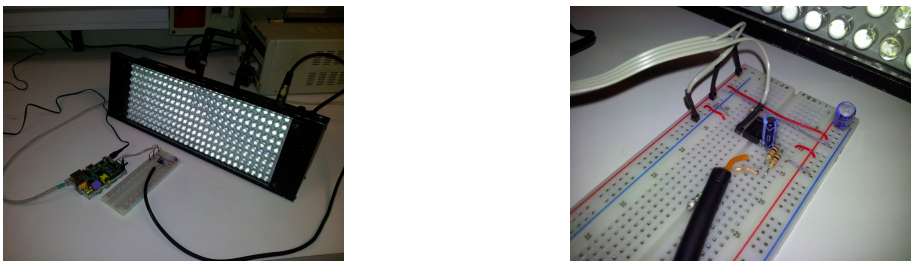

Figure 4.10: Raspberry Pi connected to the LightMaxx DMX prototype.

The system responded as expected by sending the read/write commands from the controller, such as:

```
String[] names = new String[]{"RGBLED.1.Shutter.0", "RGBLED.1.Red.0",
    "RGBLED.1.Green.0", "RGBLED.1.Blue.0"};
int[] values1 = new int[]{255, 255, 0, 0};
int[] values2 = new int[]{255, 0, 255, 255}

Write.WriteMultipleInstant(names, values1);
long delayedOperation = Write.WriteMultipleSynced(names, values2);

Thread.sleep(1000);

Sync.DeploySyncedAction(delayedOperation);
```

The code above sends two commands to the device, changing the colour output to Red instantaneously (figure 4.11), and sending the synchronization command after sleeping for one second to enable the second command, therefore changing the output colour to cyan (figure 4.12).

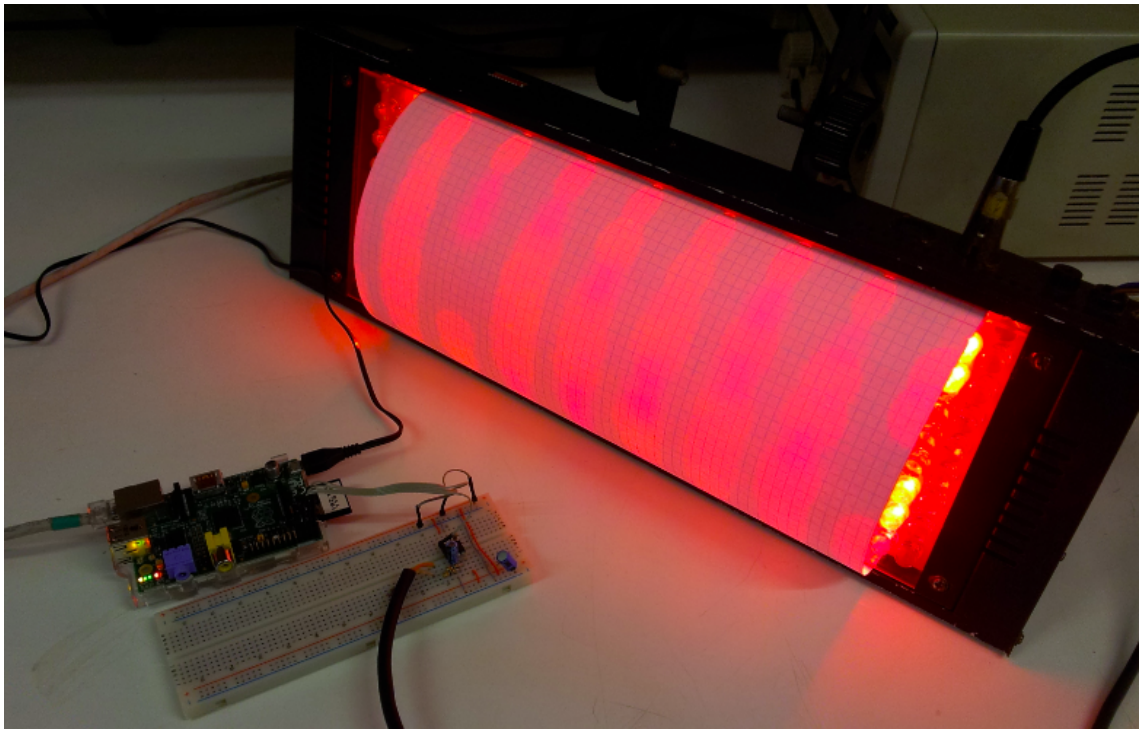

Figure 4.11: LED powered by the DMX interface (Red).

4.11 Conclusion

This concludes the chapter of the implementation of the system into a working prototype based on standard development tools. It has been proven to work with the implemented functions, which effectively shows that the system is viable.

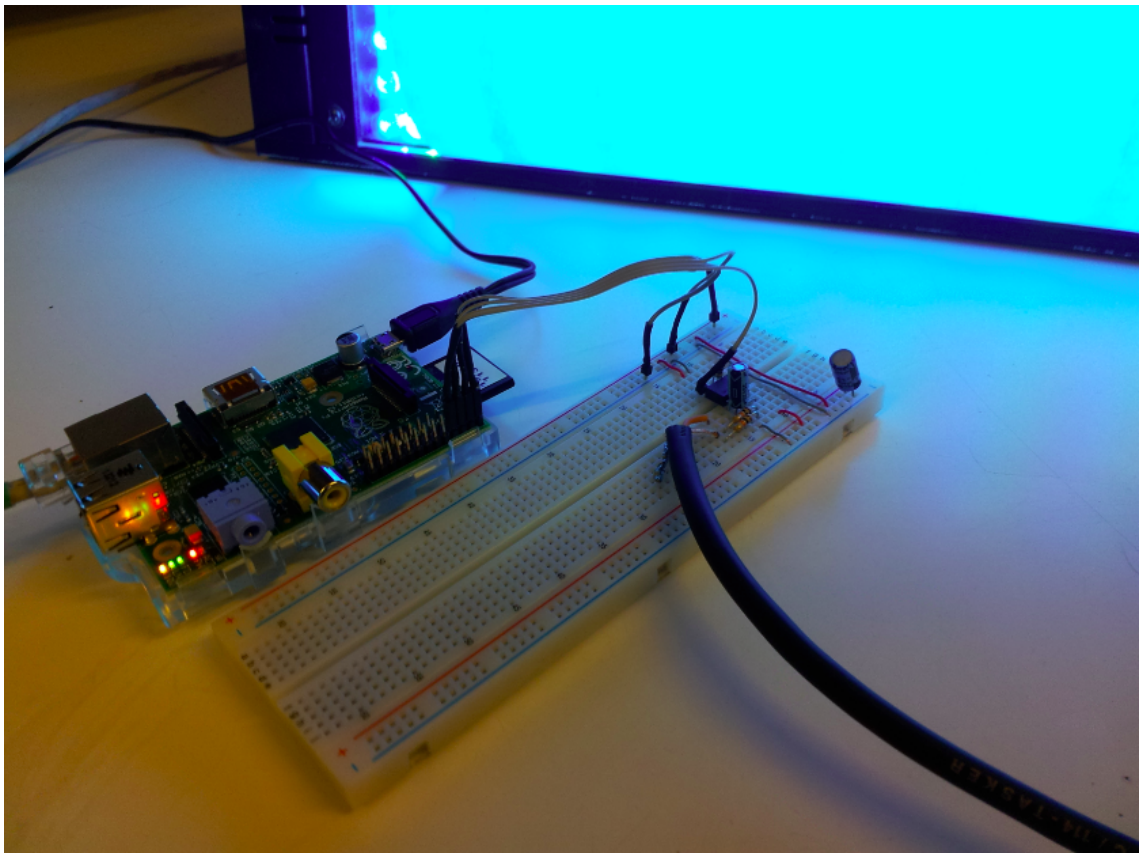

Figure 4.12: LED powered by the DMX interface (Cyan).

Chapter 5

Conclusion

This is the fifth and final chapter of this Dissertation, and concludes the work developed during this project.

5.1 Achieved Results

The purpose of this Dissertation was to develop and implement an alternative to replace the DMX protocol in stage lighting environments with support for new functions in order to simplify the configuration process and to add advanced functions that the previous protocol could not support out-of-the-box.

The study of such a system is important not only due to the increasing need of expandability and flexibility that is saturating the current technologies, which is limiting the development of new devices and controllers, but also to allow the users to focus more time on stage programming, rather than spending it on configuring it.

This system's architecture has been developed and it has been implemented in a standard development board with success, which allows it to be easily transported to other platforms. Although not all of the proposed objectives were met, the basic functions were implemented and tested, which proved that it can be further developed and used in a stage lighting environment.

5.2 Future Work

The architecture of the system has been developed to a point that would allow the continuation of the work in this area, and its possible implementation and use in a real stage environment. However, such continuation would have to be extensively studied and the requirements and specifications would have to be further developed to guarantee the viability of this project.

For example, the further study of the best network topology would be necessary to ensure the accessibility of all the devices within the network, the hardware and software requirements of the microprocessors to use as interfaces and the embedded devices to allow the increased computation needs.

Appendix A

Glossary

- Moving Head: A device used in lighting equipment that has a movable body controllable by at least two motors to move the direction of the light beam.
- Rigging: Process of mounting the light equipment and the cable in the stage structures.
- Console: Device responsible for controlling the fixtures and other lighting instruments.
- Light Engineer/Technician: Person or group responsible for the rigging, programming and controlling the lighting equipment.
- Master-Slave: Model of communication where one device has the control over the other.
- Spot light: Controllable, focused light.
- Wash light: Uncontrollable focused light.
- Gobo: Filter applied in front of a light to project a form on a surface or the air, with smoke.
- Dimmer: Intensity of a light, also referred to the dimmer lights that only have one control parameter, the intensity.
- Moving Mirror/Scanner: Device used in lighting equipment that has a fixed body and which light is projected onto a mirror with at least two motors to control the direction of the beam.
- Channel (DMX): Each parameter of a DMX device is assigned to one channel, which is an 8 bit value, ranging from 0 to 255.
- Universe (DMX): Network generated by a DMX console, to which the slave devices connect.
- Daisy-Chain: Method of connecting the devices in a network. The first has an output port which connects to the input of the second device, which also has an output to connect to the third device, and so on.
- RS-485: Differential RS-232, which is a serial communication interface.

- Baud rate: Unit for modulation rate, measured in symbols per second.
- Address (DMX): Every DMX slave device is assigned an address, which is the offset that represents the start channel reserved for that device.
- Islanding: When one or more nodes that are part of a network disconnect from the other nodes. For example, when a switch connected to a number of devices is disconnected from another switch.

Appendix B

Extended Protocol Messages

Table B.1: Embedded Ethernet Discovery (Device to Controller)

	Description	***Size***	***Example***
Header	Transaction ID	8 bytes	*0x5368696E65204F6E*
	Message type	1 byte	*0x00*
	Function	2 bytes	*0x0001*
	Data length	2 bytes	*Variable*
Data	MAC Address	6 bytes	*0x001FE103F28F*
	Name length (bytes)	2 bytes	*0x0101*
	Name of device	Variable	*"Moving Head 1"*
	Manufacturer length	2 bytes	*0x00FF*
	Manufacturer	Variable	*"Stage Lighting, Ltd"*
	Model length	2 bytes	*0x0000*
	Model	Variable	*"Spot v1"*

Table B.2: Embedded Ethernet Request (Controller to Device

	Description	***Size***	***Example***
Header	Transaction ID	8 bytes	*0x4461726B53696465*
	Message type	1 byte	*0x00*
	Function	2 bytes	*0x0002*
	Data length	2 bytes	*0x0000*

Table B.3: Embedded Ethernet Configuration (Device to Controller

	Description	***Size***	***Example***
Header	Transaction ID	8 bytes	*0x5468652057616C6C*
	Message type	1 byte	*0x00*
	Function	2 bytes	*0x0003*
	Data length	2 bytes	*Variable*
Data	Number of variables	2 bytes	*0x0003*
	ID 1	2 bytes	*0x0001*
	Class 1	2 bytes	*0x0101*
	Index 1	2 bytes	*0x0000*
	ID 2	2 bytes	*0x0002*
	Class 2	2 bytes	*0x0101*
	Index 2	2 bytes	*0x0001*
	ID 3	2 bytes	*0x0003*
	Class 3	2 bytes	*0x0102*
	Index 3	2 bytes	*0x0000*

Table B.4: Embedded Ethernet and DMX/Ethernet Interface Acknowledges (Controller to Device)

	Description	***Size***	***Example***
Header	Transaction ID	8 bytes	*0x46696E616C437574*
	Message type	1 byte	*0x00*
	Function	2 bytes	*0x0004*
	Data length	2 bytes	*0x0000*

Table B.5: DMX/Ethernet Interface Discovery (Device to Controller, no configuration)

	Description	***Size***	***Example***
Header	Transaction ID	8 bytes	*0x54686520456E646C*
	Message type	1 byte	*0x00*
	Function	2 bytes	*0x0005*
	Data length	2 bytes	*0x0006*
Data	MAC Address	6 bytes	*0x001FE103F28F*

Table B.6: Ethernet/DMX Configuration (Controller to Device)

	Description	***Size***	***Example***
Header	Transaction ID	8 bytes	*0x6573735269766572*
	Message type	1 byte	*0x00*
	Function	2 bytes	*0x0006*
	Data length	2 bytes	*Variable*
Data	Number of variables	2 bytes	*Variable*
	Class 1	2 bytes	*0x0101*
	Index 1	2 bytes	*0x0000*
	DMX Channel 1	2 bytes	*0x0001*
	Class 2	2 bytes	*0x0101*
	Index 2	2 bytes	*0x0000*
	DMX Channel 2	2 bytes	*0x0002*
	Class 3	2 bytes	*0x0102*
	Index 3	2 bytes	*0x0000*
	DMX Channel 3	2 bytes	*0x0003*

Table B.7: DMX/Ethernet Interface Discovery (Device to Controller, with configuration)

	Description	***Size***	***Example***
Header	Transaction ID	8 bytes	*0x4F63746176617269*
	Message type	1 byte	*0x00*
	Function	2 bytes	*0x0005*
	Data length	2 bytes	*Variable*
Data	MAC Address	6 bytes	*0x001FE103F28F*
	Name length (bytes)	2 bytes	*0x0101*
	Name of device	Variable	*"Dimmer 1"*
	Manufacturer length	2 bytes	*0x00FF*
	Manufacturer	Variable	*"Dimmingmark Ltd."*
	Model length	2 bytes	*0x0000*
	Model	Variable	*"Simple Dimmer"*

Table B.8: Function Write

	Description	***Size***	***Example***
Header	Transaction ID	8 bytes	*0x756D20546F756368*
	Message type	1 byte	*0x00*
	Function	2 bytes	*0x0101*
	Data length	2 bytes	*0x0001*
Data	Number of values to write	2 bytes	*0x0002*
	Variable code	2 bytes	*0x0101*
	Variable index	2 bytes	*0x0000*
	Value	4 bytes	*0x000000FF*
	Variable code	2 bytes	*0x0102*
	Variable index	2 bytes	*0x0000*
	Value	4 bytes	*0xFFFF0000*

Table B.9: Function Read

	Description	***Size***	***Example***
Header	Transaction ID	8 bytes	*0x4F66202020476F64*
	Message type	1 byte	*0x00*
	Function	2 bytes	*0x0101*
	Data length	2 bytes	*0x0001*
Data	Number of values to write	2 bytes	*0x0002*
	Variable code	2 bytes	*0x0101*
	Variable index	2 bytes	*0x0000*
	Value	4 bytes	*0x00FF0000*
	Variable code	2 bytes	*0x0102*
	Variable index	2 bytes	*0x0000*
	Value	4 bytes	*0xFFFFFFFF*

Appendix C

Additional Source Code

This appendix contains part of the relevant code developed for the prototype. The Controller and the Device were developed in Java, with the exception of the DMX Updater, which is in C.

C.1 Controller prototype

C.1.1 Configuration

C.1.1.1 Server.java

```
public class Server implements Runnable
{
   ServerSocket server = null;
   Socket client = null;
   int port;
   boolean running = false;
   public Server(int port)
   {
      this.port = port;
      running = true;
   }
   public void run()
   {
      try
      {
         server = new ServerSocket(port);
         while(running)
         {
            client = new Socket();
            client = server.accept();
            new Thread(new ClientHandler(client)).start();
         }
```

```
        } catch (IOException e)
        {
            running = false;
        }
    }
}
```

C.1.1.2 ClientHandler.java

```
public class ClientHandler implements Runnable
{
    int state = 0; // State machine status
    static int maxBufferSize = 1024;

    byte[] rawOutput = new byte[maxBufferSize];
    byte[] input = new byte[maxBufferSize];
    // Socket initialization
    int[] result = new int[2];
    Socket client = null;
    DataInputStream in = null;
    DataOutputStream out = null;
    boolean initialized = false;

    Thread thisThread;
    Thread nextThread;

    public ClientHandler(Socket client)
    {
        this.client = client;
        initialized = true;
    }
    public void run()
    {
        while (!initialized);
        thisThread = Thread.currentThread();
        byte[] raw = new byte[maxBufferSize];
        while (state < 5)
        {
            if (state == 0) //
            {
                try
                {
                    in = new DataInputStream(client.getInputStream());
                    int i = 0;
```

```
            do
               raw[i++] = in.readByte();
            while (in.available() != 0);

            for (int l = 0; l < i; l++)
               input[l] = raw[l];

            result[1] = i; // To store the length of the input
            String ip =
                client.getInetAddress().toString().substring(1);
            // Create thread to detect configuration in the next layer
            nextThread = new Thread(new PackConfigDetect(input,
                result, rawOutput, ip, thisThread));
            nextThread.start();
            state = 1;
         }
         catch (IOException e)
         {
            e.printStackTrace();
         }
      }
      else if (state == 1)
      {
         try
         {
            Thread.sleep(1000);
         } catch (InterruptedException e)
         {
            if (result[0] == Codes.sys_slip_request)
               state = 2;
            else
               state = 4;
         }
      }
      else if (state == 2)
      {
         try
         {
            in = new DataInputStream(client.getInputStream());
            out = new DataOutputStream((client.getOutputStream()));
            byte[] output = new byte[result[1]];

            for (int i = 0; i < result[1]; i++)
               output[i] = rawOutput[i];
```

```
            // Write the output to th socket
            out.write(output);

            int i = 0; // iterador
            raw = new byte[maxBufferSize];
            while(in.available() == 0) // Wait until response arrives
               Thread.sleep(1);
            do
               raw[i++] = in.readByte(); // Read response
            while (in.available() != 0);

            for (int l = 0; l < i; l++)
               input[l] = raw[l];

            result[1] = i;
         } catch (IOException e)
         {
            e.printStackTrace();
         } catch (InterruptedException e1)
         {
            e1.printStackTrace();
         }
         nextThread.interrupt();
         state = 3;
      }
      else if (state == 3)
      {
         try
         {
            Thread.sleep(1000);
         } catch (InterruptedException e)
         {
            state = 4;
         }
      }

      else if (state == 4)
      {
         try
         {
            byte[] output = new byte[result[1]];

            for (int i = 0; i < result[1]; i++)
               output[i] = rawOutput[i];
            // Send Acknowledge
```

```
                out = new DataOutputStream((client.getOutputStream()));
                out.write(output);
                out.close();
            } catch (IOException e)
            {
                e.printStackTrace();
            }
            state = 5;
        }
    }
    try
    {
        in.close();
        out.close();
        client.close();
        client = null;
    } catch (IOException e)
    {
        e.printStackTrace();
    }
    nextThread.interrupt();
  }
}
```

C.1.1.3 PackConfigDetect.java

```
public class PackConfigDetect implements Runnable
{
   int headerLen = 13;
   int maxBufferSize = 1024;
   byte[] rawInput;
   int[] result;
   byte[] output;
   Thread thisThread;
   Thread lastThread;
   boolean initialized;
   int state = 0;
   String ip, mac;
   long transID = 0;

   public PackConfigDetect(byte[] input, int[] result, byte[] output,
       String ip, Thread lastThread)
   {
      this.rawInput = input;
```

```
        this.result = result;
        this.output = output;
        this.ip = ip;
        this.lastThread = lastThread;
        this.initialized = true;
    }
    public void run()
    {
        while(!initialized);
        thisThread = Thread.currentThread();

        while(state < 12)
        {
            if (state == 0)
            {
                byte[] input = new byte[result[1]];

                for (int l = 0; l < result[1]; l++)
                    input[l] = rawInput[l];

                int i = 0;
                int dataLength = input.length;

                for (int c = 0; c < 8; c++)
                    transID = (transID << 8) + (input[i++] & 0xff);

                int protVersion = input[i++];
                result[0] = input[i++] << 8 | input[i++];
                int dataSize = input[i++] << 8 | input[i++];

                StringBuilder sb = new StringBuilder();
                for (int t = 0; t < 6; t++)
                    sb.append(String.format("%02X", input[i++]));
                mac = sb.toString();

                int nameSize = input[i++] << 8 | input[i++];
                byte[] nameBytes = new byte[nameSize];

                for (int l = 0; l < nameSize; l++)
                    nameBytes[l] = input[i++];

                int brandSize = input[i++] << 8 | input[i++];

                byte[] brandBytes = new byte[brandSize];
                for (int l = 0; l < brandSize; l++)
```

```
            brandBytes[l] = input[i++];

        int modelSize = input[i++] << 8 | input[i++];
        byte[] modelBytes = new byte[modelSize];
        for (int l = 0; l < modelSize; l++)
            modelBytes[l] = input[i++];

        String name = new String(nameBytes);
        String brand = new String(brandBytes);
        String model = new String(modelBytes);

        if (dataSize == dataLength - headerLen) // First check if
            data is complete
        {
            new Thread(new DeviceCheck(ip, mac, name, brand, model,
                result, thisThread)).start();
            state = 1;
        }
        else
        {
            result[0] = Codes.e_unspecified;
            state = 12;
        }
    }
    else if (state == 1)
    {
        try
        {
            Thread.sleep(1000);
        } catch (InterruptedException e)
        {
            if (result[0] == Codes.sys_slip_request)
                state = 2;
            else if (result[0] == Codes.sys_slip_ack || result[0] ==
                Codes.sys_dmx_valid || result[0] ==
                Codes.sys_dmx_invalid)
                state = 6;
            else
                state = 20;
        }
    }
    else if (state == 2)
    {
        int i = 0;
        // Create header
```

```
            byte[] header = new byte[headerLen];
            // Transaction ID
            header[i++] = (byte) ((transID >> 56) & 0xFF);
            header[i++] = (byte) ((transID >> 48) & 0xFF);
            header[i++] = (byte) ((transID >> 40) & 0xFF);
            header[i++] = (byte) ((transID >> 32) & 0xFF);
            header[i++] = (byte) ((transID >> 24) & 0xFF);
            header[i++] = (byte) ((transID >> 16) & 0xFF);
            header[i++] = (byte) ((transID >> 8) & 0xFF);
            header[i++] = (byte) ((transID) & 0xFF);
            // Mode
            header[i++] = (byte) 1;
            // Function
            header[i++] = (byte) ((result[0] >> 8) & 0xFF);
            header[i++] = (byte) ((result[0]) & 0xFF);
            // Data size
            header[i++] = (byte) ((0 >> 8) & 0xFF);
            header[i++] = (byte) ((0) & 0xFF);
            result[1] = i;
            for (i = 0; i < headerLen; i++)
            {
               output[i] = header[i];
            }
            state = 3;
            lastThread.interrupt();
         }
         else if (state == 3)
         {
            try
            {
               Thread.sleep(1000);
            } catch (InterruptedException e)
            {
               state = 4;
            }
         }
         else if (state == 4)
         {
            byte[] input = new byte[result[1]];
            for (int l = 0; l < result[1]; l++)
               input[l] = rawInput[l];

            int i = 0;
            int dataLength = result[1];
```

```
            for (int c = 0; c < 8; c++)
               transID = (transID << 8) + (input[i++] & 0xff);

            int protVersion = input[i++];
            result[0] = input[i++] << 8 | input[i++];

            int dataSize = input[i++] << 8 | input[i++];
            if (dataSize == dataLength - headerLen)
            {
               if ( result[0] == Codes.sys_slip_config)
               {
                  int numberOfValues = input[i++] << 8 | input[i++];
                  int[] idsToConfig = new int[numberOfValues];
                  int[] codesToConfig = new int[numberOfValues];
                  int[] indexesToConfig = new int[numberOfValues];
                  for (int l = 0; l < numberOfValues; l++)
                  {
                     idsToConfig[l] = input[i++] << 24 | input[i++] << 16
                         | input[i++] << 8 | input[i++];
                     codesToConfig[l] = input[i++] << 24 | input[i++] <<
                         16 | input[i++] << 8 | input[i++];
                     indexesToConfig[l] = input[i++] << 24 | input[i++]
                         << 16 | input[i++] << 8 | input[i++];
                  }
                  new Thread(new ConfigSLIP(mac, idsToConfig,
                      codesToConfig, indexesToConfig, result,
                      thisThread)).start();
                  state = 5;
               }
            }
            else
            {
               result[0] = Codes.e_unspecified;
               state = 12;
            }
         }
         else if (state == 5)
         {
            try
            {
               Thread.sleep(1000);
            } catch (InterruptedException e)
            {
               state = 6;
            }
```

```
            }
            else if (state == 6)
            {
               int i = 0;
               byte[] header = new byte[headerLen];
               header[i++] = (byte) ((transID >> 56) & 0xFF);
               header[i++] = (byte) ((transID >> 48) & 0xFF);
               header[i++] = (byte) ((transID >> 40) & 0xFF);
               header[i++] = (byte) ((transID >> 32) & 0xFF);
               header[i++] = (byte) ((transID >> 24) & 0xFF);
               header[i++] = (byte) ((transID >> 16) & 0xFF);
               header[i++] = (byte) ((transID >> 8) & 0xFF);
               header[i++] = (byte) ((transID) & 0xFF);
               header[i++] = (byte) 1;
               header[i++] = (byte) ((result[0] >> 8) & 0xFF);
               header[i++] = (byte) ((result[0]) & 0xFF);
               header[i++] = (byte) ((0 >> 8) & 0xFF);
               header[i++] = (byte) ((0) & 0xFF);
               result[1] = i; // Tamanho
               for (i = 0; i < headerLen; i++)
               {
                  output[i] = header[i];
               }
               state = 12;
               lastThread.interrupt();
            }
         }
      }
   }
```

C.1.2 Operations

C.1.2.1 Write.java

```
public class Write
{
   public static void WriteMultipleInstant(String[] names, int[] vals)
   {
      handleWriteThreads(names, vals, false);
   }
   public static long WriteMultipleSynced(String[] names, int[] vals)
   {
      return handleWriteThreads(names, vals, true);
   }
```

```
private static long handleWriteThreads(String[] names, int[] vals,
    boolean sync)
{
   long transactionID = Database.getNewTransactionID();
   int length = names.length;
   ArrayList<Integer> id = new ArrayList<Integer>();
   ArrayList<Integer> id_var = new ArrayList<Integer>();
   ArrayList<Integer> index = new ArrayList<Integer>();
   ArrayList<Integer> values = new ArrayList<Integer>();
   int devidtemp = 0, paridtemp = 0;
   for (int i = 0; i < length; i++)
   {
      String[] nameParts = names[i].split("\\.");
      int partLen = nameParts.length;
      devidtemp = Database.getIdDeviceFromDesig(nameParts[0]);
      if (devidtemp > 0)
         id.add(devidtemp);
      else
      {
         ErrorHandler.throwError(Codes.op_write, Codes.e_var);
         return 0;
      }
      paridtemp = Database.getParamID(nameParts[1]);
      if (paridtemp >= 0 && Database.deviceHasParameter(devidtemp,
          paridtemp))
         id_var.add(paridtemp);
      else
      {
         System.out.println("Dev " + devidtemp + " par " + paridtemp +
             " " + Database.deviceHasParameter(devidtemp, paridtemp));
         ErrorHandler.throwError(Codes.op_write, Codes.e_par);
         return 0;
      }
      index.add(Integer.parseInt(nameParts[2]));
      values.add(vals[i]);
   }
   ArrayList<Integer> p_id = new ArrayList<Integer>();
   ArrayList<Integer> p_id_var = new ArrayList<Integer>();
   ArrayList<Integer> p_index = new ArrayList<Integer>();
   ArrayList<Integer> p_values = new ArrayList<Integer>();
   int c = 0, ind = 0;
   while (!id.isEmpty())
   {
      p_id.clear();
      p_id_var.clear();
```

```
            p_values.clear();
            p_index.clear();
            p_id.add(id.remove(0));
            p_id_var.add(id_var.remove(0));
            p_index.add(index.remove(0));
            p_values.add(values.remove(0));
            boolean next = false;
            while (!next && id.size() > 0)
            {
               if (id.get(0) == p_id.get(0))
               {
                  p_id.add(id.remove(0));
                  p_id_var.add(id_var.remove(0));
                  p_index.add(index.remove(0));
                  p_values.add(values.remove(0));
               }
               else
                  next = true;
            }
            int len = p_id.size();
            int[] final_id_var = new int[len];
            int[] final_index = new int[len];
            int[] final_values = new int[len];
            for (int i = 0; i < len; i++) {
               final_id_var[i] = p_id_var.get(i);
               final_index[i] = p_index.get(i);
               final_values[i] = p_values.get(i);
            }
            String ip = Database.getIP(p_id.get(0));
            if (!ip.equals(""))
               new Thread(new PackWrite(ip, sync, transactionID,
                   final_id_var, final_index, final_values)).start();
            else
            {
               ErrorHandler.throwError(Codes.op_write, Codes.e_dev);
               return 0;
            }
         }
         return transactionID;
      }
   }
```

C.1.2.2 PackWrite.java

```
public class PackWrite implements Runnable
{
   static int maxBufferSize = 1024;
   static int headerLen = 13;
   int[] result = new int[]{Codes.op_write, 0};
   int state = 0;
   boolean initialized = false;
   String ip;
   boolean sync;
   long transactionID;
   int[] var;
   int[] index;
   int[] val;
   Thread thisThread;
   byte[] rawInput;
   int[] inputIterator = new int[]{0};

   public PackWrite(String ip, boolean sync, long transactionID, int[]
       var, int[] index, int[] val)
   {
      this.ip = ip;
      this.sync = sync;
      this.transactionID = transactionID;
      this.var = var;
      this.index = index;
      this.val = val;
      initialized = true;
   }

   public void run()
   {
      while (!initialized);
      thisThread = Thread.currentThread();
      rawInput = new byte[maxBufferSize];

      while(state < 2 && result[0] > 0)
      {
         if (state == 0 && result[0] > 0)
         {
            int i = 0;
            byte[] data = new byte[maxBufferSize];
            int numberOfValues = var.length;
            // Number of values
            data[i++] = (byte) ((numberOfValues >> 24) & 0xFF);
            data[i++] = (byte) ((numberOfValues >> 16) & 0xFF);
```

```
data[i++] = (byte) ((numberOfValues >> 8) & 0xFF);
data[i++] = (byte) ((numberOfValues) & 0xFF);

for (int c = 0; c < numberOfValues; c++)
{
   // Variable Code
   data[i++] = (byte) ((var[c] >> 24) & 0xFF); // Id
   data[i++] = (byte) ((var[c] >> 16) & 0xFF);
   data[i++] = (byte) ((var[c] >> 8) & 0xFF);
   data[i++] = (byte) ((var[c]) & 0xFF);
   // Index
   data[i++] = (byte) ((index[c] >> 24) & 0xFF); // Index
   data[i++] = (byte) ((index[c] >> 16) & 0xFF);
   data[i++] = (byte) ((index[c] >> 8) & 0xFF);
   data[i++] = (byte) ((index[c]) & 0xFF);
   // Value
   data[i++] = (byte) ((val[c] >> 24) & 0xFF); // Value
   data[i++] = (byte) ((val[c] >> 16) & 0xFF);
   data[i++] = (byte) ((val[c] >> 8) & 0xFF);
   data[i++] = (byte) ((val[c]) & 0xFF);
}

byte[] header = new byte[headerLen];
int dataLen = i;
header[i++] = (byte) ((transactionID >> 56) & 0xFF);
header[i++] = (byte) ((transactionID >> 48) & 0xFF);
header[i++] = (byte) ((transactionID >> 40) & 0xFF);
header[i++] = (byte) ((transactionID >> 32) & 0xFF);
header[i++] = (byte) ((transactionID >> 24) & 0xFF);
header[i++] = (byte) ((transactionID >> 16) & 0xFF);
header[i++] = (byte) ((transactionID >> 8) & 0xFF);
header[i++] = (byte) ((transactionID) & 0xFF);
header[i++] = (byte) ((sync) ? 1 : 0);
header[i++] = (byte) ((result[0] >> 8) & 0xFF);
header[i++] = (byte) ((result[0]) & 0xFF);
header[i++] = (byte) ((dataLen >> 8) & 0xFF);
header[i++] = (byte) ((dataLen) & 0xFF);
// Build final packet
byte[] toSend = new byte[headerLen + dataLen];
int l = 0;
for (i = 0; i < headerLen; i++) {
   toSend[l++] = header[i];
}
for (i = 0; i < dataLen; i++) {
   toSend[l++] = data[i];
```

```
            }
            new Thread(new Connection(sync, ip, 5112, toSend, rawInput,
                inputIterator, result, thisThread)).start();
            if (!sync)
                state = 1;
            else
                state = 20;
        }
        else if (state == 1 && result[0] > 0)
        {
            try
            {
                Thread.sleep(1000);
            } catch (InterruptedException e)
            {
                state = 2;
            }
        }

        else if (state == 2 && result[0] > 0)
        {
            byte[] input = new byte[inputIterator[0]];
            for (int l = 0; l < inputIterator[0]; l++)
                input[l] = rawInput[l];

            int i = 0;
            int dataLength = input.length;

            long transID = 0;
            for (int c = 0; c < 8; c++)
                transID = (transID << 8) + (input[i++] & 0xff);

            int protVersion = input[i++];
            result[0] = input[i++] << 8 | input[i++];

            int dataSize = input[i++] << 8 | input[i++];
            int numberOfWrittenVariables = input[i++] << 24 | input[i++]
                << 16 | input[i++] << 8 | input[i++];
            if (dataSize == dataLength - headerLen)
            {
                if (result[0] < 0 || numberOfWrittenVariables !=
                    var.length) // N error
                    result[0] = Codes.e_var;
            }
            else
```

```
                result[0] = Codes.e_unspecified;
             state = 2;
          }
       }
       if (result[0] < 0)
       {
          ErrorHandler.throwError(Codes.op_write, result[0]);
       }
    }
}
```

C.1.3

C.2 Device prototype

C.2.1 DMX Updater

```
#define ONE 63;
#define ZERO 0;

pthread_t tid[1];

int available_channels = 10;
int allocated_channels = 5;
int channelBuffer[20] = { 0 };

bool redo = true; // Flag to recreate the DMX data
bool running = true;
bool reading = false; // Flag to indicate that the data is being read
bool writing = false; // Flag to indicate that the data is being written

void* serverHandler(void *arg);

int openUart(int uart)
{
  uart = open("/dev/ttyAMA0", O_RDWR | O_NDELAY);

  if (uart == -1)
    printf("Unable to open UART\n");

  struct termios options;
  tcgetattr(uart, &options);
  options.c_cflag = B2000000 | CS6;  //Magic: Baud rate of 2 MBaud, 6 bits
  options.c_iflag = IGNPAR;
```

```
  options.c_oflag = 0;
  options.c_lflag = 0;
  tcflush(uart, TCIFLUSH);
  tcsetattr(uart, TCSANOW, &options);

  return uart;
}

int main(void)
{
  // UART INIT
  int uart = -1;

  uart = openUart(uart);

  unsigned char tx_buffer[1024];
  unsigned char *p_tx_buffer;
  int i = 0;
  // Create serverHandler thread
  pthread_create(&(tid[i]), NULL, &serverHandler, NULL);
      struct timespec tstart={0, 0}, tend={0, 0};

  while(running)
  {
    clock_gettime(CLOCK_MONOTONIC, &tstart);
    if (redo & !writing)
    {
      reading = true;
      redo = false;
      p_tx_buffer = &tx_buffer[0];
      // Create DMX packet
      // BREAK
      for (i = 0; i < 80; i++)
        *p_tx_buffer++ = ZERO;
      // MAB
      for (i = 0; i < 24; i++)
        *p_tx_buffer++ = ONE;

      int l, r, tempbuf;
      for (l = 0; l < allocated_channels; l++)
      {
        // Populate channels
        tempbuf = channelBuffer[l];
        // Start bit
        *p_tx_buffer++ = ZERO;
```

```
    // 8 Data bits
    if (tempbuf % 2 == 0) { *p_tx_buffer++ = ZERO; } else
        *p_tx_buffer++ = ONE; tempbuf = tempbuf >> 1;
    if (tempbuf % 2 == 0) { *p_tx_buffer++ = ZERO; } else
        *p_tx_buffer++ = ONE; tempbuf = tempbuf >> 1;
    if (tempbuf % 2 == 0) { *p_tx_buffer++ = ZERO; } else
        *p_tx_buffer++ = ONE; tempbuf = tempbuf >> 1;
    if (tempbuf % 2 == 0) { *p_tx_buffer++ = ZERO; } else
        *p_tx_buffer++ = ONE; tempbuf = tempbuf >> 1;
    if (tempbuf % 2 == 0) { *p_tx_buffer++ = ZERO; } else
        *p_tx_buffer++ = ONE; tempbuf = tempbuf >> 1;
    if (tempbuf % 2 == 0) { *p_tx_buffer++ = ZERO; } else
        *p_tx_buffer++ = ONE; tempbuf = tempbuf >> 1;
    if (tempbuf % 2 == 0) { *p_tx_buffer++ = ZERO; } else
        *p_tx_buffer++ = ONE; tempbuf = tempbuf >> 1;
    if (tempbuf % 2 == 0) { *p_tx_buffer++ = ZERO; } else
        *p_tx_buffer++ = ONE; tempbuf = tempbuf >> 1;

    // Stop bits
    *p_tx_buffer++ = ONE;
    *p_tx_buffer++ = ONE;
    *p_tx_buffer++ = ONE;
  }

  for (r = 1; r < available_channels; r++)
  {
    // Populate remaining empty channels
    *p_tx_buffer++ = ZERO;

    *p_tx_buffer++ = ZERO;
    *p_tx_buffer++ = ZERO;
    *p_tx_buffer++ = ZERO;
    *p_tx_buffer++ = ZERO;
    *p_tx_buffer++ = ZERO;
    *p_tx_buffer++ = ZERO;
    *p_tx_buffer++ = ZERO;
    *p_tx_buffer++ = ZERO;

    *p_tx_buffer++ = ONE;
    *p_tx_buffer++ = ONE;
    *p_tx_buffer++ = ONE;
  }
  reading = false;
}
```

```
    if (uart0_filestream != -1)
    {
      int count = write(uart0_filestream, &tx_buffer[0], 1024);
      if (count < 0)
        printf("UART TX error\n");
    }
    clock_gettime(CLOCK_MONOTONIC, &tend);
    nanosleep((struct timespec[]){{0, 600000 - (tend.tv_nsec -
        tstart.tv_nsec)}}, NULL) // Sleep enough time to allow the
        transmission of the packet
  }
  return 0;
}
```

References

[1] Avolites Ltd. Avolites portal. http://www.avolites.com/portals/0/Images/Content/The-Script-021636_Jamie_FOH_best.jpg. Last checked on January 5, 2015.

[2] Abigail Raney. Types of stage lighting. http://www.ehow.com/about_5347272_types-stage-lighting.html. Last checked on January 5, 2015.

[3] Clay Paky. Clay Paky. http://www.claypaky.it/. Last checked on January 5, 2015.

[4] USITT. DMX512 FAQ. http://www.usitt.org/content.asp?contentid=373, 2008. Last checked on April 2, 2014.

[5] Wikipedia. Stage Lighting. http://en.wikipedia.org/wiki/Stage_lighting. Last checked on January 5, 2015.

[6] Wikipedia. Parabolic aluminized reflector light. http://en.wikipedia.org/wiki/Parabolic_aluminized_reflector_light. Last checked on January 5, 2015.

[7] Ujjal Kar. What is DMX512? http://www.dmx512-online.com/whats.html. Last checked on January 6, 2015.

[8] Wikipedia. RS-485. http://en.wikipedia.org/wiki/RS-485. Last checked on January 16, 2015.

[9] Ujjal Kar. The DMX512 packet. http://www.dmx512-online.com/packt.html. Last checked on January 7, 2015.

[10] Wikipedia. RDM (lightning). http://en.wikipedia.org/wiki/RDM_(lighting), 2014. Last checked on January 9, 2015.

[11] Wikipedia. Art-Net. http://en.wikipedia.org/wiki/Art-Net. Last checked on January 7, 2015.

[12] Artistic Licence Holdings. Specification for the Art-Net 3 Ethernet Communication Protocol.