INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelor's Thesis

Using a Raspberry Pi as a PC-DMX interface

Florian Edelmann

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelor's Thesis

Using a Raspberry Pi as a PC-DMX interface

Florian Edelmann

Supervisor:	Prof. Dr. Dieter Kranzlmüller
Advisors:	Dr. Nils gentschen Felde Tobias Guggemos
Date of submission:	27th November 2017

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 27. November 2017

.....
(Unterschrift des Kandidaten)

Abstract

DMX is a common standard in the entertainment technology industry that allows controlling lighting fixtures (like spotlights or strobe lights on a stage) connected over a bus. The DMX source driving the bus is usually a mixing desk console; alternatively, lighting control software on a computer together with a physical PC-DMX interface is often used. The high costs of professional desk consoles and DMX interfaces limits small associations like youth groups to less expensive interfaces that often lack useful features.

This Bachelor's Thesis defines requirements for a PC-DMX interface in this use case, including a price limit and advanced features like the availability of a DMX input port to enable haptic control of software functions. A market study is conducted, which reveals that no existing products match all requirements, so a system design is established, building upon the affordable single-board computer Raspberry Pi. The software basis is provided by *Open Lighting Architecture* (*OLA*), an open-source software that is able to convert different protocols to and from DMX and process the data internally.

OLA is then extended to support DMX output through a USB-DMX adapter by reverse engineering and implementing its protocol. DMX input directly on Raspberry Pi's hardware is possible for the first time through the implementation of SPI bus sampling; other tested approaches were not successful. The validation of the finished PC-DMX interface shows satisfactory fulfillment of the requirements. Future work to improve the system design by making it even less expensive, more easy to use or by adding extra features is possible.

Contents

1 Introduction

In theaters, at parties, concerts or for meditative purposes, e.g. in church, there is a demand for professional lighting to convey or support the desired atmosphere. Often, one wants to create custom light shows for a specific event or application.

This thesis describes the design and build of a feature-rich, inexpensive and open interface between lighting control software on a computer and light fixtures like spotlights, strobe lights, moving heads and scanners. It is built on top of the open-source project *Open Lighting Architecture* running on a Raspberry Pi.

High quality lighting equipment is available from a wide variety of manufacturers and nearly every light fixture is controllable via the DMX protocol. Traditionally, one uses DMX mixing desks to generate the DMX signal. However, those desk consoles are usually very expensive and therefore not accessible for small associations.

In particular, in our local parish youth, there is a technics team I am part of that arranges sound and lighting engineering for hosted parties and other events in church. Since purchasing full-featured DMX desk would have exceeded the budget, the free version of the PC lighting control software *e:cue*[1] was used instead. The computer running the software is connected to light fixtures via a translator box (*PC-DMX interface*). Unfortunately, this box is still rather pricey and does only accept e:cue's proprietary exchange protocol which is output by their own software.

Eventually, in this use case, the software – being the free version after all – became too limited: More ideas developed on how to design light shows than the program allowed. There are many different completely free lighting programs that can be tried and compared, but for all the same problem persists: The DMX data must be transmitted to the DMX line bus from the computer with an auxiliary interface.

Additionally, the wish to use a small DMX desk as an input for the software formed, making haptic (instead of mouse-driven) controlling possible. This would allow controlling the speed of a chaser[2] by changing its corresponding hardware fader or triggering a specific function in the software whenever a DMX input channel exceeds a defined level. Another possible use case could be directly forwarding a few DMX channels from the input to the output and letting the software handle all other channels.

After a market study, it was clear that such advanced features are not available in any professional yet inexpensive DMX interface. The decision was made to build a PC-DMX interface on my own. It should communicate with the computer over open protocols to allow usage with several different lighting control programs. The core of that box is a Raspberry Pi running an open-source software called *Open Lighting Architecture* (*OLA*) that is able to

[1] `http://ecue.de`

[2] A chaser is a sequence of scenes defined in a lighting software. A scene in turn consists of fixtures set to a certain state; e.g. spotlight 1 is red, spotlights 2 and 3 are off, spotlight 4 is yellow. The program then cycles through those scenes whenever an event occurs (e.g. a button is clicked) or an adjustable amount of time passes.

convert different protocols to and from DMX and process the data internally. This software needs to be extended to fulfill all needs.

This thesis documents the development of the interface from the gathering of requirements and planning the approach, to setting up and extending OLA up until the finished product. All the code improvements and additions I made to the existing project were also contributed back to allow other users to benefit from my work.

Structure of this thesis

First, some technical background about the required protocols and technologies is given, most notably the DMX and Art-Net protocols, which are needed to understand the goals of the project. Then, the requirements of the PC-DMX interface are analyzed and incorporated into a system design.

Afterwards, the implementation is presented by outlining the structure and concepts of OLA, which much of this thesis is based on, documenting the electrical and chassis build, and finally describing the code and proceedings for both the DMX output and input plugins.

Subsequently, the implemented features are validated against their specification and an outlook to future improvements and use is given.

2 Technical background

For this thesis, a basic understanding of some concepts, protocols and specifications is needed. Particularly, it must be clear to the reader how to use DMX hardware and how the low level works technically, so that my programming work can be apprehended. In this chapter, I try to give a sufficient overview about the most important topics.

It is assumed that the reader knows about Raspberry Pi programming and the standard ISO/OSI layer model. It is helpful to already have worked with *git* sometime.

2.1 DMX

DMX is used in this paper to refer to the standard *DMX512-A* [EST13], which is short for "Digital Multiplex with 512 individual pieces of information" [USI]. The original standard was defined by the *United States Institute for Theatre Technology* (*USITT*) in 1986 and revised and extended several times. Details of the protocol itself are given in the next section. First, the general usage of DMX devices is outlined.

2.1.1 Typical DMX setup

A typical (very simple) DMX lighting setup according to [Ben12] which is shown schematically in figure 2.1 is as follows: A DMX desk console outputs a DMX signal that is sent via cable to the first light fixture (a simple dimmer in this case). The signal is then looped through to the next fixture, which is a LED head lamp here. The final fixture in this example to be "daisy-chained" is a moving head. Its output DMX signal is not used in another fixture and therefore a terminating resistor (here a black and yellow "stick") finishes the DMX line.

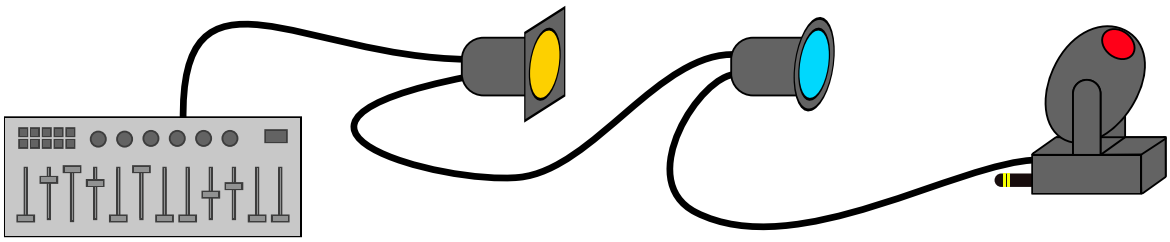

Figure 2.1: Schematical DMX lighting setup example.

The DMX signal basically consists of 512 integer values between 0 and 255, each representing the value of one channel. Every fixture f in the DMX line listens to a specific number n_f of these channels (fixed by the manufacturer) to control its features. The user now has to give each fixture an address a_f to mark the first channel it has to listen to, e.g. via a DIP switch or a small display on each of the fixtures. The order of the addresses given to the fixtures does not have to match the order of the fixtures in the line.

To show how this works in practice, the example is continued (see also table 2.1): At first, the LED head lamp is given address $a_{LED} = 1$. It needs $n_{LED} = 3$ channels to control its red, green and blue (RGB) components separately, thus listens to channels 1, 2 and 3. To make the LED light yellow, we would need to set channels 1 and 2 (red and green) to 255 and channel 3 (blue) to 0. The next address we can assign without conflict is $a_{LED} + n_{LED} = 4$. We give this address to the dimmer ($a_{dim} = 4$) which needs only this one channel to control its brightness. Now we reserve address 5 for future use of another dimmer, so that the two dimmer channels will be next to each other on the desk console. Finally the moving head is assigned address $a_{mov} = 6$. We assume that its RGB values, pan / tilt movement and gobo wheel[3] can be controlled and therefore 6 channels are needed.

Table 2.1: Example DMX addresses and channel numbers.

fixture	address	number of channels	controlled channels
LED head	1	3	ch. 1: red
			ch. 2: green
			ch. 3: blue
dimmer	4	1	ch. 4: brightness
unused			ch. 5: –
moving head	6	6	ch. 6: red
			ch. 7: green
			ch. 8: blue
			ch. 9: pan movement
			ch. 10: tilt movement
			ch. 11: gobo wheel

The 512 channels being signalled through one line are called a *DMX universe*. A desk console can output multiple universes, which allows to address more fixtures in total.

Instead of using a physical DMX desk, its task can also be fulfilled by a software. The computer running it is then connected to the DMX line via a physical interface.

2.1.1.1 DMX Splitters and Mergers

There are two notable hardware components that can be used in a DMX line to make the wiring between fixtures and DMX sources more flexible:

[3] A gobo is a stencil in front of the lamp that shapes the emitted light beam. Typically, multiple gobos are mounted in a wheel that rotates according to the DMX value in the corresponding channel to allow the selection of one gobo at a time.

- A DMX splitter is used as a T piece to forward one input signal into two (or more) output lines.
- A DMX merger combines the signals from its two input universes A and B (seldomly also more) into one output signal. Mergers typically have different modes of operation, such as the following. [Sho15]
  - *Backup*: As long as A is a valid signal, loop it through; else use B.
  - *Merge*: Use A's first x channels, then fill the remaining 512 minus x channels with B's first channels.
  - *LTP* ("latest takes precedence"): Loop the universe through that has changed later. Sometimes this algorithm is also applied per channel instead of per universe.
  - *HTP* ("highest takes precedence"): For each of the 512 channels, use the highest value of the both corresponding channels in A and B.

2.1.2 DMX protocol

This section gives a short summary of the official *DMX512-A* standard [EST13] by the *Entertainment Services and Technology Association* (*ESTA*).

The DMX protocol defines a serial signal (shown in figure 2.2) at a baud rate of 250000 bits per second. It consists of individual packets, each of which is initiated by the *reset sequence* (an arbitrarily long `low` *BREAK* signal followed by the `high` *mark after BREAK* (*MAB*) and slot 0). A slot contains 8-bit data (least significant bit first), prepended by a `low` start bit and followed by two `high` stop bits. The data in slot 0 is called the *start code*, it is always `0x00` for DMX packets. Other start codes can be used to indicate manufacturer-specific information or special functions; those packets shall be ignored by standard DMX receivers, so they are not relevant in this project.

After the reset sequence, the channel values of the universe are transmitted in one slot each. Since channels are transmitted serially, it is possible for fixtures in the DMX line to count received channels and start listening as soon as their address matches the current channel number. The channel number therefore does not have to be transmitted separately. Slots can be separated by a `high` *mark between slots* (*MBS*[4]).

Following the last channel, a `high` *mark before BREAK* (*MBB*) or the next BREAK indicates the end of the packet. At least one packet per second should be transmitted, though faster updates are desirable to ensure fast respondence of the fixtures and, in particular, smooth light fading. To increase the refresh rate, not all 512 channels have to be transmitted in a packet.

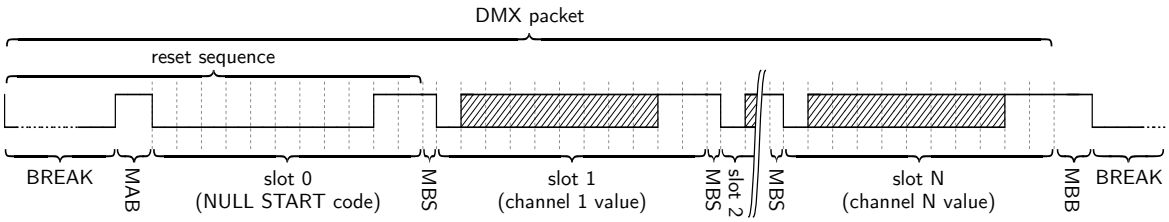

Figure 2.2: DMX timing diagram.

[4]This abbreviation is not used in the official standard. I introduced it for simpler referencing.

Table 2.2: DMX timing. Values in parenthesis only apply to receivers. From [EST13]

Entity	Min	Typical	Max
Bit rate	245kbit/s	250kbit/s	255kbit/s
Bit time	3.92µs	4µs	4.08µs
BREAK	92µs (88µs)	176µs	–
MAB	12µs (8µs)	–	< 1.00s
MBS	0	–	< 1.00s
MBB	0	–	< 1.00s
DMX packet	1204µs (1196µs)	–	1.00s (1.25s)
Refresh rate	830Hz (836Hz)	–	1Hz (0.8Hz)

If one considers packets with all 512 channels, one can work out from table 2.2 a minimum packet time of 22.7ms, or a maximum refresh rate of 44Hz.

An extension to DMX that will not be important for this thesis but should still be mentioned is *Remote Device Management* (*RDM*). It allows setting the fixtures' DMX address and other options from the RDM controller, which is an extended DMX desk or software. It works by interleaving the unidirectional DMX signal with bidirectional RDM packets.

2.1.2.1 Electrical specification

DMX uses the electrical specifications defined in the industry standard EIA-485 (also known as RS-485) which describes balanced transmission-line signaling [TII08]. "Balanced" means data bits are encoded via the potential difference between the twisted-pair cables *Data +* and *Data –*. This decreases interference liability, since noise adds to both data lines equally – effectively cancelling itself out in the difference –, and thus makes line lengths of up to 1.2km possible. For DMX lines though, the recommendation is to stay below 300m [Ben12].

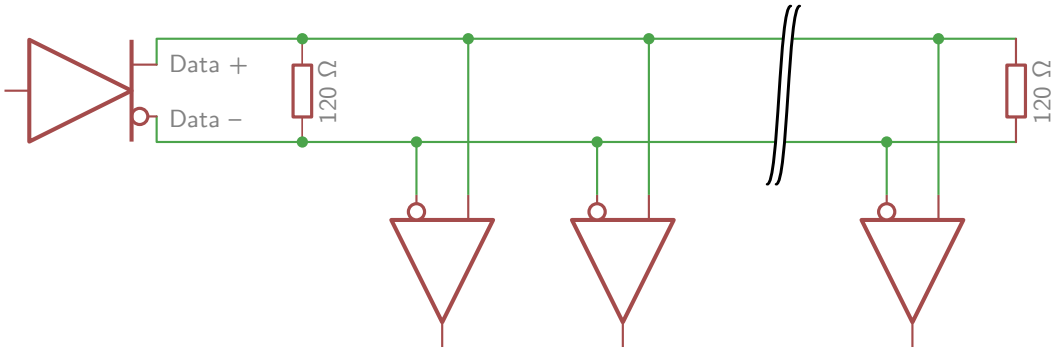

Figure 2.3: EIA-485 bus with one transmitter and up to 32 receivers.

In figure 2.3, the electrical schematic of such a bus is shown. On the left, the transmitter converts the raw signal into the differential signal. All the receivers on the bottom (up to 32

are allowed) do the same in the other direction.[5] At both the near end (transmitter side) and the far end (after last receiver), a termination resistor of 120 ohms is installed to minimize reflections that could interfere with the signal.

The DMX standard requires 5-pin XLR connectors, except where they are "physically impossible to mount" [EST13]. Even so, most DMX hardware is equipped with a 3-pin XLR connector instead or additionally. This is due to the fact that only 3 pins are needed and 3-pin XLR cables are common in event technology because they are also used for microphones.

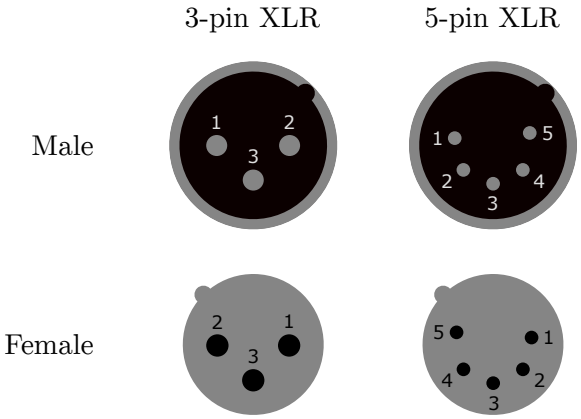

Figure 2.4: XLR connectors used for DMX.

Table 2.3: XLR pin assignment for DMX.

Pin Number	3-pin XLR	5-pin XLR
1	Ground	Ground
2	*Data –*	*Data –*
3	*Data +*	*Data +*
4	–	*Data 2 –* (optional)
5	–	*Data 2 +* (optional)

2.1.3 Art-Net and sACN protocols

DMX only allows one or two universes per line, which may make cabling impractical in some use cases. To overcome this issue, the English lighting equipment company *Artistic Licence* created a free-to-use DMX over UDP[6] protocol called *Art-Net* [Art17] in 1998. It allows sending multiple DMX universes over a standard IP network and thus highly extends the

[5] Actually, all devices connected to the bus are allowed to both transmit and receive (e.g. used in RDM). Thus, usually so-called "bus transceiver chips" that can convert in both directions are deployed. Since communication over DMX will always be uni-directional in this thesis, I simplified the figure.

[6] User Datagram Protocol

flexibility of lighting systems since a single Ethernet link or a wireless network can be used for large parts of the transport way.

There are lighting controllers and fixtures that work directly with Art-Net (exclusively or alongside traditional DMX), all others can be connected via an *Art-Net Node* that converts to and from DMX. Often, the protocol is used for communication between a lighting software on a computer and one such Node acting as the DMX source for fixtures, like in figure 2.5.

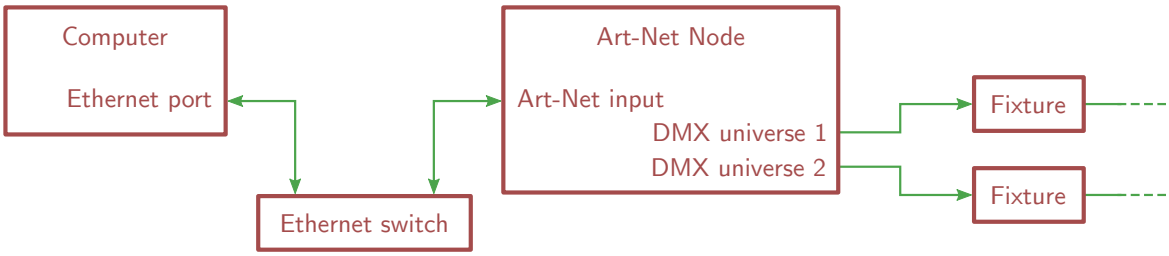

Figure 2.5: Schematical connection of an Art-Net Node.

Streaming ACN (*sACN*), which was standardized as ANSI E1.31 in late 2016 [EST16], is ESTA's open protocol with the same goals and shares most of its high level properties with Art-Net. In this thesis, the details and differences of both protocols will not be covered.

2.2 UART

The *Universal Asynchronous Receiver Transmitter* (*UART*) is an interface present on many microcontrollers that allows communication over serial bus lines. There is no extra clock signal, the receiver synchronizes itself through the fixed data format: Data bits are transmitted sequentially, framed in slots with `low` start and `high` stop bits. If the signal is `low` for longer than one slot time, the *break condition* is fulfilled.

Various parameters have to be fixed at both receiver and transmitter to avoid misunderstanding: Baud rate, the number of data bits per slot (usually 5 to 9), bit numbering (most or least significant bit first), the number of stop bits used (one or two) and if each slot should additionally contain a parity bit.

As the DMX timing protocol is a specialization of this specification, sending and receiving DMX data via a UART is possible. One caveat though is the non-standard baud rate of 250kbit/s.

2.3 SPI

The *Serial Peripheral Interface* (*SPI*) is a synchronous data transmission interface between a master and multiple slave devices designed by Motorola [Dem15]. Only the independent slave configuration will be discussed here, which can be seen in figure 2.6.

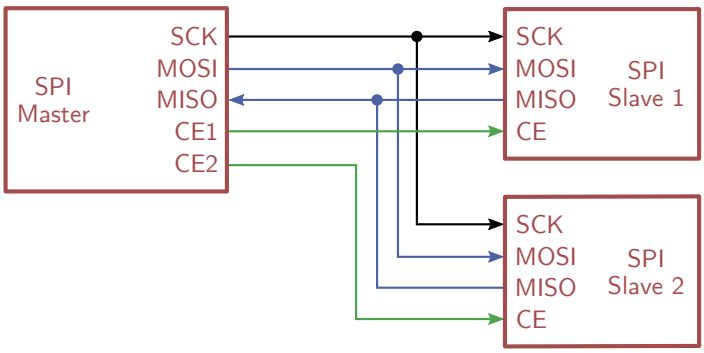

Figure 2.6: Schematic of SPI master and slaves.

By applying a `low` signal at one of the *Slave select* / *Chip enable* (*CE*) pins, the master can notify the corresponding slave that it wants to communicate with. After that, the master generates a clock signal at the *SCK* (*Serial clock*) pin and simultaneously reads at the *MISO* (*Master In, Slave Out*) pin and transmits at the *MOSI* (*Master Out, Slave In*) pin one bit per clock cycle. After the data transmission is completed (e.g. one byte is sent to the slave who then may answer with one byte, but that depends on the protocol fixed between the devices), the master resets all pins to their idle levels.

There are multiple SPI modes that define when the bit read / write operation should happen in relation to the clock signal. For simplicity, only mode 0 is shown here, in which *SCK*'s idle status is `low` and data transfer starts with the first rising edge in the clock signal (see figure 2.7).

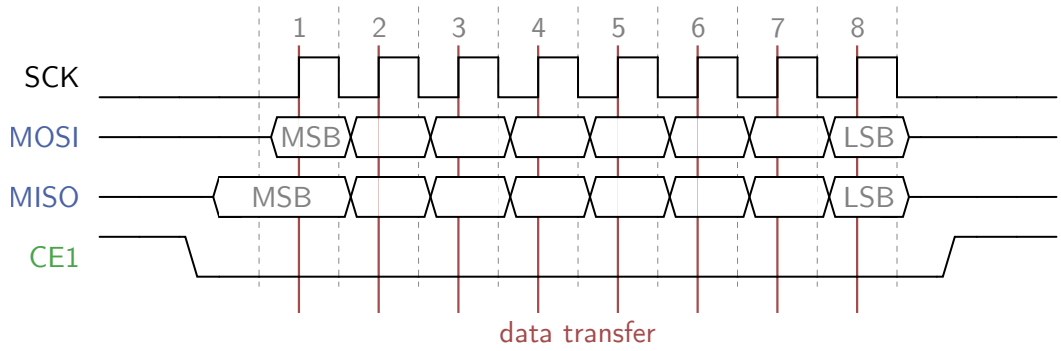

Figure 2.7: SPI timing diagram. MSB and LSB are short for *most significant bit* and *least significant bit*, respectively. Adapted from [Dem15].

3 Requirement analysis and system design

In this chapter, the plan for the PC-DMX interface is outlined. First, I will define my requirements and examine several existing products on the market on how they match or fail these requirements. This will then lead to the design of the interface described in this thesis.

3.1 Requirement analysis

The requirements defined in this section are designed for the specific use case of small associations like technics teams in youth groups (i.e. not professional event management companies or the like). Their budget is usually very limited but their expertise does not have to be – i.e. the PC-DMX interface must offer features for advanced users while still being affordable.

A youth technics team may not have found its optimum workflow yet and may want to improve it by trying out different free lighting control programs, e.g. QLC+[7], DMX Control[8] or FreeStyler DMX[9]. The interface should support that by being compatible with as many of them as possible.

Connection to the computer shall be possible via Ethernet to allow extending the cable via standard network equipment like switches, routers and Wi-Fi access points. USB connection is not sufficient because USB devices need special drivers and configuration, which would make the interface less portable, e.g. if a quick replacement computer in an emergency situation is needed. Among the protocols used for data transmission, at least one should be open, i.e. either sACN or Art-Net[10] should be supported.

The interface should support output of at least two DMX universes to be able to address a sufficiently large number of fixtures and input of at least one to allow haptic control of DMX channels using a mixing desk. The method how DMX input signals are handled should be configurable:

- Either the DMX channel values are sent via Art-Net to the control software (default *Art-Net Node*-like mode), e.g. to control software functions with hardware mixing desk faders,
- or it acts like a DMX splitter, forwarding its DMX input signal on both DMX outputs,
- or the DMX input channels are merged with the channel values provided over the network into one of the output universes using one of the merge modes described in section 2.1.1.1.

[7] `http://www.qlcplus.org/`
[8] `https://www.dmxcontrol.org/`
[9] `http://www.freestylerdmx.be/`
[10] Although Art-Net is not strictly open, it is free-to-use and supported by a wide variety of software and hardware.

All input and output DMX signals should be processed with a high refresh frequency, so that delays remain low and smooth light fading is possible.

Using the interface should be as simple as possible for end-users. That means that neither in-depth knowledge about the DMX protocol or computer networks nor special know-how about this specific interface should be required to use it. However, end-users are assumed to know how to work with DMX software and hardware in general. More complicated functions like the flexible input mapping mentioned above should be trivial enough to be understandable in a short period of time.

The whole setup should cost less than 100€ and be extensible, i.e. widened future requirements like the need for more DMX universes should be easy to implement without a redesign and rebuild of the whole hardware and software.

Additionally, it would be appreciated if both hardware and software were open-source to allow others to extend and improve the interface.

3.2 Market study

A market study was conducted to see how existing products do fulfill the requirements defined in the previous section. The price limit on 100€ was fixed to narrow down the products in the first step. An overview of the products described here is given in table 3.1.

Since the price limit of 100€ narrows down the range of professional hardware to various USB-to-DMX adapters and one Art-Net Node by *Eurolite*[11] – all of them with support for only one output universe, none with DMX input –, only "Do It Yourself" projects are left.

A popular one, *DMXControl Projects*' *Nodle U1*[12], can also only be connected via USB. Thus, it has to be explicitly supported by lighting programs – which several do. Nevertheless, it fails the network connection requirement.

Some members of the *DMXControl* forum and wiki created an Art-Net Node based on a commercially available AVR construction kit[13]. It was refined and eventually ported to its own hardware to support two DMX universes[14]. Unfortunately, this is still not sufficient.

The same applies to GitHub user *mtongnz*'s Art-Net Node based on the Wi-Fi-enabled *ESP8266* microcontroller[15].

The project that matches most of my requirements is the *Quad Art Net Box* by Ulrich Radig[16]: It supports four DMX output universes, one of them can be toggled to an input. Sources and schematics are available online, and an assembly kit can be ordered. However, there is no information given about whether the DMX input can be flexibly merged into one DMX output universe or if it is always forwarded to the Art-Net output.

Rather than doing all the work in one microcontroller like in the previous projects, *raspberrypi-dmx.org*[17] uses a much more powerful Raspberry Pi with an additional co-processor on an extension board (sometimes called *shield*) that is plugged into the GPIO

[11] `https://www.steinigke.de/en/mpn70064842-eurolite-art-net-dmx-node-1.html`
[12] `https://www.dmxcontrol-projects.org/hardware/nodle-u1.html`
[13] `https://wiki.dmxcontrol.de/wiki/Art-Net-Node_f%C3%BCr_25_Euro`
[14] `https://wiki.dmxcontrol.de/wiki/ArtNetNode_auf_einer_Platine`
[15] `https://github.com/mtongnz/ESP8266_ArtNetNode_v2`
[16] `https://www.ulrichradig.de/home/index.php/dmx/quad-art-net`
[17] `http://www.raspberrypi-dmx.org/raspberry-pi-art-net-dmx-out`

(General Purpose Input / Output) pins. Thereby, only the software needs to be replaced (by re-flashing the SD card) to match the use case: USB, Art-Net, sACN, Open Sound Control and MIDI can all be converted to DMX with the correct SD card image. Unfortunately, the extension board hardware does only support one input and one output universe.

Table 3.1: Available products overview. Values in parenthesis specify alternate modes.

Product	Simultaneous output / input universes	Flexible input mapping	Open / Extensible	Connection to computer
Professional USB-to-DMX adapters	1 ✗ / 0 ✗	*N/A* ✗	✗ / ✗	USB ✗
Eurolite Art-Net/DMX Node 1	1 ✗ / 0 ✗	*N/A* ✗	✗ / ✗	Art-Net ✓
DMXControl Projects Nodle U1	1 ✗ / 1 ✓	✗	✓ / ✗ [18]	USB ✗
DMXControl Wiki AvrArtNodeV2.0	2 ✓ / 0 ✗ (0 ✗ / 2 ✓)	*N/A* ✗	✓ / ✗	Art-Net ✓
mtongnz ESP8266_ArtNetNode_v2	2 ✓ / 0 ✗ (1 ✗ / 1 ✓)	✗	✓ / ✗	Art-Net ✓
Ulrich Radig Quad Art Net Box	3 ✓ / 1 ✓ (4 ✓ / 0 ✗)	✗	✓ / ✗	Art-Net ✓
Raspberry Pi Art-Net 3 -> DMX Out	1 ✗ / 1 ✓	✗	✓ / ✓	Art-Net + others ✓

Another interesting project I have found during my research is the *Open Lighting Architecture* (*OLA*) software, which will be further explained in section 4.1. It aims to be a universal protocol translator for DMX signals, supporting different devices through plugins. It can be installed on the Raspberry Pi and a plugin providing native DMX output through its UART port is already available.

In conclusion, none of the existing products fulfills all requirements, but there are a few different approaches and projects that provide a good starting point for building an PC-DMX interface myself.

[18] *DMXControl Projects* do state in their manual that "future extensions should be possible" [DMX13]. However it seems to me that such extensions would still require completely redesigned hardware.

3.3 System design

The single-board computer Raspberry Pi will form the basis of the adapter. It has Ethernet and USB ports and a CPU powerful enough to run OLA. An extension board interfaced to its GPIO pins must be developed. It has to be equipped with two EIA-485 bus transceivers to provide one DMX input and, through the aforementioned UART plugin, one output. The second output will be supplied by an USB-to-DMX adapter that was available to me, the *DMXCreator 512 Basic*[19]. Its protocol has to be reverse-engineered and incorporated into OLA's USB plugin (see section 4.4). Initial observation of the protocol suggested the feasibility of this approach.

An OLA plugin that allows direct DMX input on the Raspberry Pi is yet to be developed. Initially, I planned to extend the UART plugin. However, prototyping the protocol recognition using the UART port did not succeed, so the SPI bus will be used to sample the DMX signal instead. Further details of this technique are explained in section 4.5.

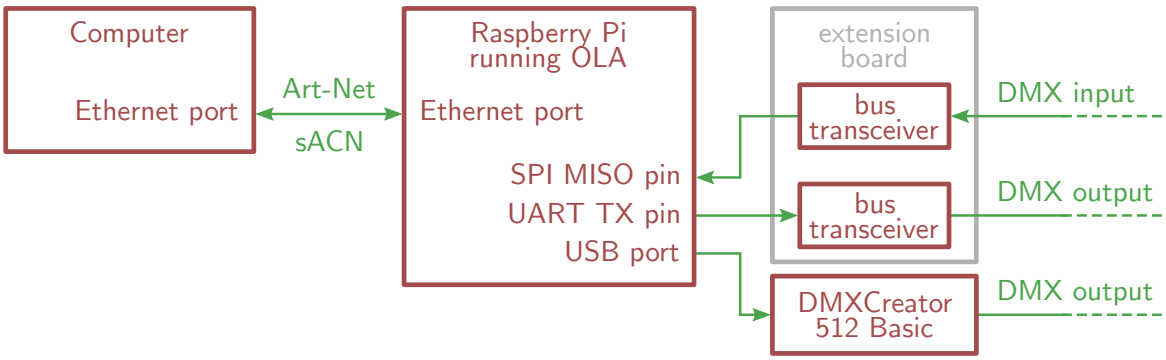

Figure 3.1: Schematic of the planned PC-DMX interface.

[19] `http://www.dmx512.ch/512.html`

4 Implementation

In chapter 3, requirements were specified which the desired PC-DMX interface has to fulfill. A market study revealed that no existing products match these requirements, so a system design was developed. This chapter describes the steps I took to implement this design.

First, relevant parts of the *Open Lighting Architecture* (*OLA*) project, which will be used as the software basis, are outlined, before it is initially installed on a Raspberry Pi. Afterwards, the physical extension board and implementation of both required extensions to the OLA software are explained. The build of a chassis completes this chapter.

4.1 The Open Lighting Architecture Project

The *Open Lighting Architecture* (*OLA*) project I discovered during my market study is described in [Hes15] as

> *[...] free, open source software originally created by Simon Newton and now developed by a team of contributors around the world. It runs on Linux or Mac and is capable of interfacing with USB DMX512 hardware, DMX512 over IP protocols, and the Raspberry Pi's GPIO pins. The application includes a web interface for easily creating, monitoring, and configuring DMX universes. OLA is one part of the larger Open Lighting Project, which aims to build high-quality, free software for the entertainment lighting industry.*

As it provides the basis of my implementation, I briefly explain some concepts in the project that are needed later.

4.1.1 Terminology

In OLA, some keywords are used extensively [OLP]:

- A *port* is a point where at most 512 DMX channel values are passed to (*output port*) or read in (*input port*). It can either be physical or virtual (like in Art-Net).
- A *device* groups ports together, it consists of at least one port.
- A *plugin* provides support for recognizing, connecting to and communicating with one or more devices. It has to be compiled along with OLA (i.e. cannot be downloaded and connected afterwards) and thus has to be part of the project. At runtime, plugins can be enabled and disabled independently.
- An OLA *universe* is an internal set of 512 DMX channel values. It can be *patched* by the user to input ports to receive new data and / or to output ports to transmit its current channel values.

4.1.2 Relevant existing plugins

At the time of writing, there are already more than 20 plugins available in OLA's source code. Amongst them, the following are of special interest for this thesis.

4.1.2.1 Art-Net Plugin

The *ArtNet Plugin*[20] implements the Art-Net protocol version 3, which supports at most four input universes and four output universes per IP address. This plugin creates input ports and output ports accordingly, which can be patched to OLA universes and thereby relay DMX data to / from external lighting programs. Since the Art-Net protocol is designed in a backwards-compatible manner, both newer and older client software are able to communicate with OLA.

This plugin does not need additional hardware, it uses the network ports that are already available.

4.1.2.2 UART Plugin

The *Native UART DMX Plugin*[21] instantiates one output port that directly generates the DMX signal via the UART port of the host device, usually a Raspberry Pi.

This signal at a GPIO pin must then only be run through a bus transceiver chip to transform it into a balanced EIA-485 signal with a valid potential difference.

Richard Ash, the initial author of this plugin, outlines the difficulties he had to face for his implementation in a blog post[22]:

- "DMX-512 runs a a [sic] non-standard (for PC) baud rate of 250kbaud."
  Fortunately, this issue could be solved on Linux by using the `termios2` interface for UART setup.
- "DMX-512 uses serial break signals [...]. These cannot be sent by just writing characters out of the serial port."
  Again, the `termios2` interface provides methods to start and end the BREAK. In between, a standard `usleep` call interrupts the sending thread for the specified time; imprecisions do not matter in this case.
- "DMX-512 has relative tight timing requirements for various elements of the signal – if your computer suddenly stops sending data for a while, then the lights you are controlling may go out or flicker randomly."
  This is true indeed, however both his own project experience and my testing have proven the output to be reliable enough for smooth light fading.

[20] `https://github.com/OpenLightingProject/ola/tree/master/plugins/artnet`. It is actually not named *Art-Net Plugin* at the time of writing. I opened issue #1328 on GitHub to fix this.
[21] `https://github.com/OpenLightingProject/ola/tree/master/plugins/uartdmx`
[22] `http://eastertrail.blogspot.de/2014/04/command-and-control-ii.html`

4.1.2.3 USB Plugin

The *USB DMX Plugin*[23] provides support for a variety of USB-to-DMX adapters. Each of them is controlled by a "sub-plugin" that extends the common basis implementation. This simplifies access to the `libusb` library and reduces code duplication.

Each sub-plugin gets notified about a newly plugged in USB device and can claim it if vendor ID, device ID and possibly other information match predefined values. Then, it is responsible for creating ports and communicating with the device.

4.1.2.4 SPI Plugin

The *SPI Plugin*[24] allows to directly operate LED pixel strips with SPI-controllable LED drivers like WS2801 or LPD8806[25]. Advanced functions like using hardware SPI multiplexers and multiple pixel strips are available but beyond the scope of this explanation.

4.1.3 Project organization with GitHub

The project repository is hosted at GitHub[26]. Its `master` branch always contains the newest development version, released versions are tagged commits in the git history (like `0.10.5`). For every bigger version change (like from `0.9.x` to `0.10.0`), a version branch is created (`0.10`) that allows future bug fix commits to be targeted against the released version without having to include newer features from the `master` branch.

Pull requests from contributors' forks are automatically run against the project's tests and linters and have to be approved by both main developers, Simon Newton and Peter Newman. This allows spotting bugs and inconsistencies early and ensures good code quality.

4.2 Initial setup of OLA on the Raspberry Pi

In this section, I explain the steps which were needed to install OLA on a Raspberry Pi 1 model B+ from scratch. Newer versions of the single-board computer should work as well, but may need some slight adjustments.

First, a recent Raspbian Lite image from the Raspberry Pi homepage[27] has to be downloaded and flashed[28] onto the microSD card that the Raspberry Pi will boot from. The microSD card should be at least 4GB in size[29]. Secure shell (SSH) access is disabled by default in Raspbian. Since SSH is required for connecting remotely to the Raspberry Pi, it must be enabled by putting a new (empty) file named `ssh` in the card's root directory.

[23] `https://github.com/OpenLightingProject/ola/tree/master/plugins/usbdmx`
[24] `https://github.com/OpenLightingProject/ola/tree/master/plugins/spi`
[25] see `https://opendmx.net/index.php/OLA_LED_Pixels`
[26] `https://github.com/OpenLightingProject/ola`
[27] `https://raspberrypi.org/downloads/raspbian/`. There are also pre-configured OLA images available from `http://dl.openlighting.org/`, but since I need the latest *git* version to apply my own changes and those images were not updated for several years, the manual procedure is the better way.
[28] Instructions: `https://raspberrypi.org/documentation/installation/installing-images/README.md`
[29] I managed to install OLA on a 2GB card, but that required removing various packages and constantly scratching at the space limit. I later switched to a 4GB card.

After booting up the Raspberry Pi with the newly flashed microSD card and connecting it to the network with an Ethernet cable, the IP address has to be found out[30] so that a secure shell can be opened. In this shell, all following commands are executed.

Before continuing, all packages, firmware and the kernel should be updated to their latest versions:

```
sudo apt update
sudo apt upgrade
sudo rpi-update
```

4.2.1 Building and installing OLA

Building OLA from source for the first time takes several hours. Thus, it may be helpful to overclock Raspberry Pi's processor via `raspi-config`; the *Medium* setting worked reliably for me. A reboot is needed for the change to take effect.

Some prerequisite packages are required for building OLA and need to be installed with `apt`. Thereafter, the latest source code from GitHub is downloaded, built and the resulting binaries get copied to the correct paths.

```
sudo apt install git libcppunit-dev libcppunit-1.13-0 uuid-dev pkg-config ↩
    libncurses5-dev libtool autoconf automake g++ libmicrohttpd-dev ↩
    libmicrohttpd10 protobuf-compiler libprotobuf-lite9 python-protobuf ↩
    libprotobuf-dev libprotoc-dev zlib1g-dev bison flex make libftdi-dev ↩
    libftdi1 libusb-1.0-0-dev liblo-dev libavahi-client-dev
git clone https://github.com/OpenLightingProject/ola.git
cd ola
autoreconf -i
./configure
make
sudo make install
sudo ldconfig
```

Note: It may be possible to cross-compile OLA on a more powerful machine. However, I could not find any advice on how to do this for such a big project depending on the *autotools* build toolchain and therefore instead decided to try as much new code as possible on my work computer and build only those versions on the Raspberry Pi that have already been built successfully there.

After the install is complete, the OLA daemon can be started with `olad` and its web interface accessed at port 9090.

OLA should be started automatically as soon as the Raspberry Pi has booted, which can be achieved by an *init script*. I used OLA's official one[31] as a basis, but simplified it a bit, changed it for user *pi* and included GPIO pin initialization (see init-olad.sh[32] and listing 4.1). The script needs to be made executable and registered with the following commands.

[30] Instructions: `https://raspberrypi.org/documentation/remote-access/ip-address.md`

[31] `https://github.com/OpenLightingProject/ola/blob/master/debian/ola.olad.init`

[32] I henceforth use this font for references to files that are part of this thesis. A list of all files and further information is provided at the end of the document.

```
sudo mv init-olad.sh /etc/init.d/olad
sudo chmod a+x /etc/init.d/olad
sudo update-rc.d olad defaults
```

Listing 4.1: Excerpt from init-olad.sh.

```
31 /sbin/start-stop-daemon --start --background --make-pidfile --pidfile $PIDFILE ↩
       --umask 0002 --chuid $USER --exec $DAEMON -- $DAEMON_ARGS
32
33 # set GPIO24 high (drive enable of IC1) and GPIO16 low (drive enable of IC2)
34 echo "24" > /sys/class/gpio/export
35 echo "16" > /sys/class/gpio/export
36 sleep 1
37 echo "out" > /sys/class/gpio/gpio24/direction
38 echo "out" > /sys/class/gpio/gpio16/direction
39 sleep 1
40 echo "1" > /sys/class/gpio/gpio24/value
41 echo "0" > /sys/class/gpio/gpio16/value
```

4.2.2 Enabling UART

In `/boot/config.txt`, `enable_uart=0` needs to be changed to `enable_uart=1` to make the port usable. The maximum baud rate is 115200bit/s (less than the required 250kbit/s), so another line `init_uart_clock=16000000` has to be added to the same file to increase the limit.

By default, shell and kernel messages are output on the serial connection. This behavior must be disabled via `raspi-config`. Finally, to allow access to the UART port, the default user *pi* has to be added to the *dialout* group:

```
sudo usermod -a -G dialout pi
```

OLA's UART plugin needs to be enabled and configured so that it uses the correct UART port. This can be done by changing the contents of file `/home/pi/.ola/ola-uartdmx.conf` to the following.

```
1 enabled = true
2 device = /dev/ttyAMA0
3 /dev/ttyAMA0-break = 100
4 /dev/ttyAMA0-malf = 100
```

4.2.3 USB configuration

Recognized USB devices are accessible for members of the *plugdev* group, so *pi* should be added there like above. To make all of OLA's supported USB devices recognized, OLA's `udev` rules are imported with the following commands.

```
sudo wget -O /etc/udev/rules.d/10-ola.rules https://raw.githubusercontent.com/↩
    OpenLightingProject/raspberrypi/master/etc/udev/rules.d/10-local.rules
sudo udevadm control --reload-rules
```

4.2.4 Network settings

To make it easier to directly connect the PC-DMX interface to computers that do not have a DHCP server running (which possibly applies to most end user systems), it is assigned a static IP address. The computer's IP address then only has to be in the same subnet to be able to communicate. The following lines need to be added to `/etc/dhcpcd.conf`.

```
# static ip
interface eth0

static ip_address=192.168.0.10/24
static routers=192.168.0.1
static domain_name_servers=192.168.0.1
```

OLA's web interface is accessible at port 9090 by default, which can be changed with a command line parameter. However, since ports below 1024 can not be opened without root privileges[33] and `olad` refuses to run as root, well-known port 80 for web servers can not be used. These commands install forwarding rules from port 80 to 9090 as a workaround.

```
sudo sysctl -w net.ipv4.ip_forward=1
sudo sysctl -w net.ipv4.conf.all.route_localnet=1
sudo iptables -A PREROUTING -t nat -i eth0 -p tcp --dport 80 -j DNAT --to ↩
    127.0.0.1:9090
sudo mkdir /etc/iptables
sudo sh -c "iptables-save > /etc/iptables/rules.v4"
```

To make these rules persist after a reboot, the following lines are added to `/etc/rc.local`.

```
sysctl -w net.ipv4.conf.all.route_localnet=1
iptables-restore < /etc/iptables/rules.v4
```

4.3 Electrical installation

The next goal is to build the extension board that hosts both bus transceiver chips for UART output and SPI input and is connected via Raspberry Pi's GPIO pins, as designed in section 3.3. I developed the schematic in figure 4.1 based on Richard Ash's blog post (see section 4.1.2.2) and examples in the *SN75176B* transceiver chip data sheet [TI15] (though similar transceiver chips like the *MAX485* could also be used instead) with the EAGLE software[34].

[33] See RFC 1700 [RP94]. It was obsoleted by RFC 3232, but the information about privileged ports still remains valid.

[34] `https://www.autodesk.com/products/eagle/overview`

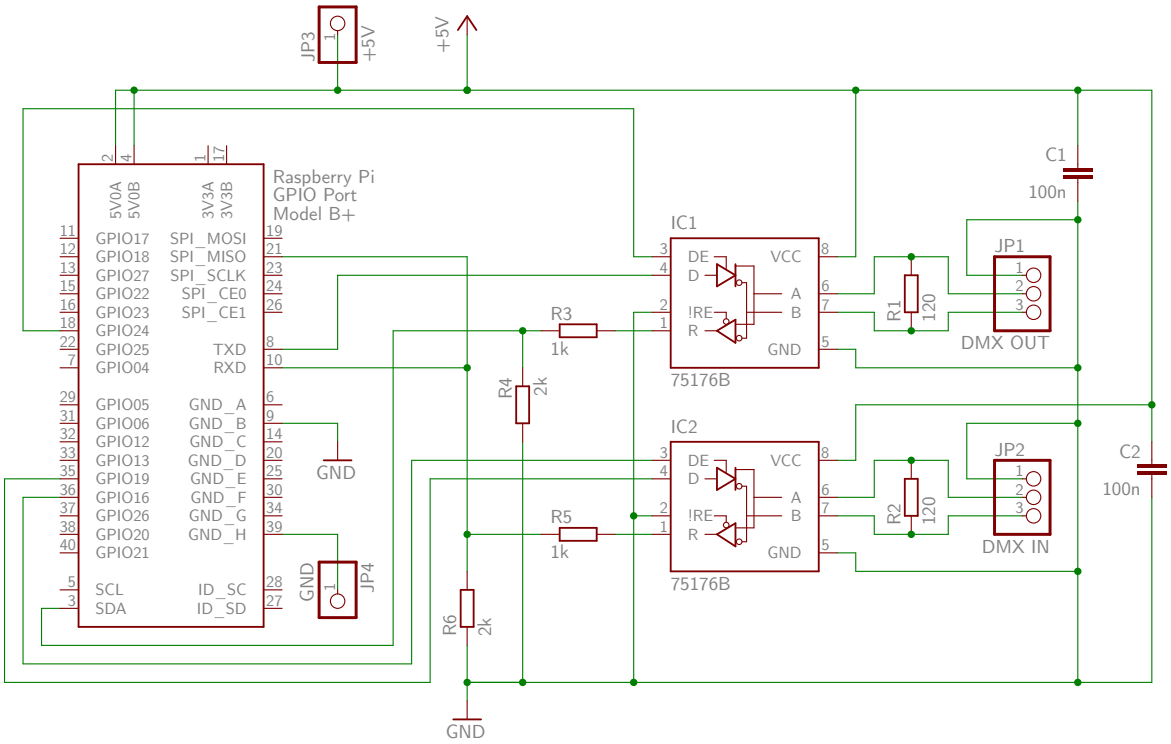

Figure 4.1: Extension board schematic. See extension-board.sch.

The schematic is designed such that the *receive* (*R*), *drive* (*D*) and *drive enable* (*DE*) pins of both transceiver chips *IC1* and *IC2* – not just one direction for each chip – are connected to the Raspberry Pi. Currently, only one direction for each chip is supported, but this design allows bidirectional use of the chips in the future (see chapter 6). The negated *receive enable* (*!RE*) pin is hard-wired to ground (*GND*) to enable receiving, as this cannot cause harm to data transmission.

Received data at the *R* pins are forwarded to the GPIO pins via voltage dividers (resistors *R3*/*R4* and *R5*/*R6*) to reduce the 5V output signal to the allowed 3.3V for Raspberry Pi's inputs. *IC2*'s received data (of the DMX input line) additionally to SPI's *MISO* (*Master In, Slave Out*) pin go also into UART's *RXD* (receive) pin as that was my first try to make receiving DMX data on the Raspberry Pi work. *IC1*'s received data (of the DMX output line) go into the *SDA* pin to keep the option open to use the I^2C bus to parse data there; otherwise it can be used as a simple GPIO pin.

Resistors *R1* and *R2* are DMX termination resistors as defined in the DMX standard. Both chips' supply voltage (*VCC*) pins are connected to ground via small capacitors (*C1* and *C2*) to mitigate voltage peaks of the power supply.

All other parts in the schematic are plug connectors; *JP1* and *JP2* go to the DMX XLR connectors, *JP3* and *JP4* allow an external voltage supply to power the Raspberry Pi through the extension board instead of the onboard micro USB port.

This schematic was then transformed into a two-sided layout that can either be printed to create a *PCB* (printed circuit board) or soldered manually on a drilled board, which I did. The layout is shown in figure 4.2, the finished board in figure 4.3.

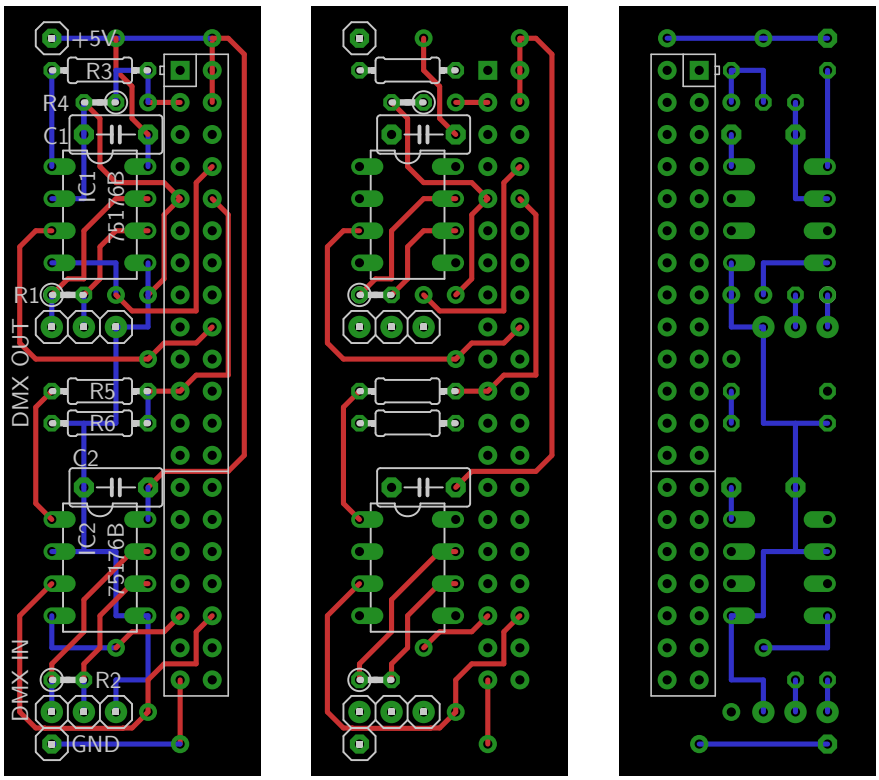

Figure 4.2: Extension board layout. See extension-board.brd. Complete view, top view without labels, bottom view without labels (mirrored).

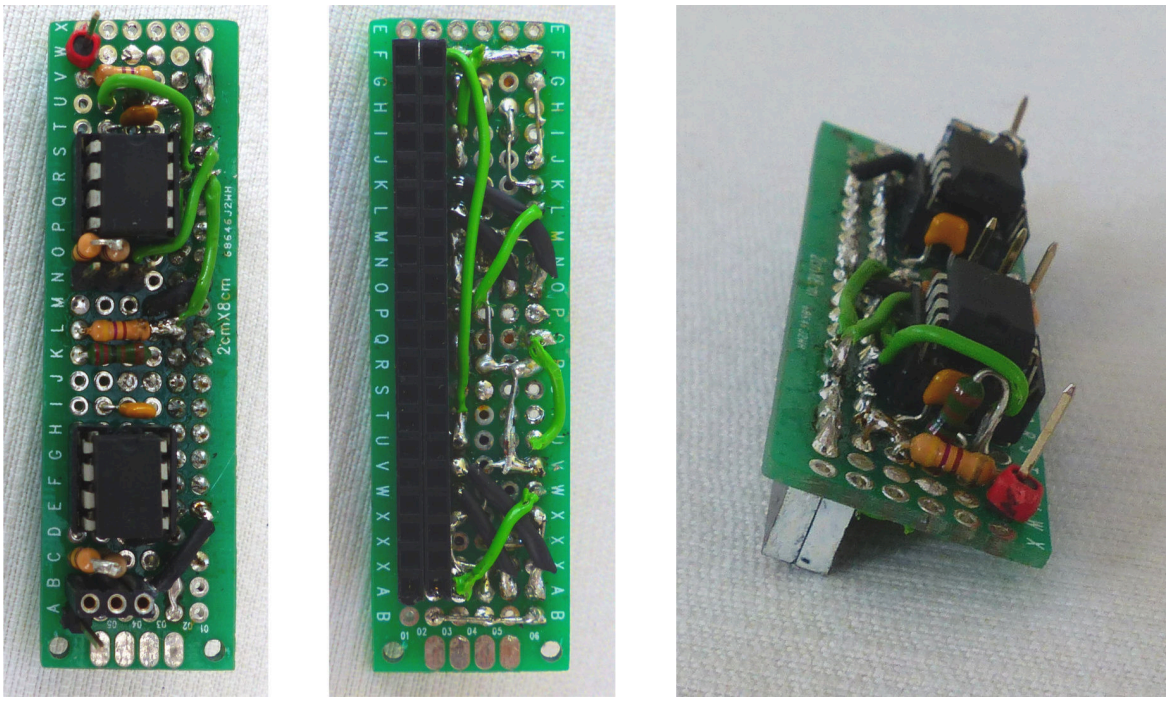

Figure 4.3: The finished extension board. Top view, bottom view, side view.

After installing the extension board on Raspberry Pi's GPIO pins, fixtures connected via *IC1*'s DMX line can be controlled by changing an OLA universe's channel values in the web interface. The UART output port just has to be patched to that universe.

In an earlier version, termination resistors were missing, which resulted in an increased number of transmission errors. Also, both transceiver chips' *Drive enable* pin were hard-wired to *VCC*. Thus, whenever the receiver chip (*IC2*) was connected to a DMX line with another DMX source, it always tried to pull the bus to `low` and eventually broke down.

4.4 Implementation of the DMXCreator 512 Basic protocol as OLA USB sub-plugin

The second DMX output port shall be provided by the *DMXCreator 512 Basic* USB-to-DMX adapter[35]. It is only officially supported by VXCO's Windows lighting software *DMXCreator* [VXC11]. To make it usable in OLA, its protocol must be reverse engineered and then re-implemented as an extension for the USB DMX Plugin.

4.4.1 Reverse engineering the protocol with Wireshark

To capture the traffic between the DMXCreator software and USB adapter, I installed the network analyzer software *Wireshark* with USB support on Windows[36] and went through the steps in table 4.1. By investigating the resulting capture file dmxcreator.pcap, I found out the following:

1. Only when the source DMX data changes (e.g. while fading a slider), USB traffic can be observed. The adapter generates a DMX signal with a valid refresh rate itself. This was verified by unplugging power of a connected fixture, which instantly reset to the correct color when it was plugged in again.
2. For every DMX data change, one packet with a constant byte string is sent to USB endpoint `0x01` of the device and then either one or two data packets (with 256 bytes of payload each) are sent to endpoint `0x02`:
   a) If the change occurs in the first half of the universe (channels 1 to 256), the byte string is `0x80 0x01 0x00 0x00 0x00 0x01` and only one data packet is sent.
   b) If the change occurs in the second half (channels 257 to 512), the byte string is `0x80 0x01 0x00 0x00 0x00 0x02` and two data packets are sent.
3. The data packets' payload consists of half a universe's DMX channel values as one byte string. Hence, two data packets are needed if the changed channel is located in the second half.
4. The USB device's vendor ID is `0x0a30`, its product ID is `0x0002` (included in packet 88; can also be retrieved by running `lsusb` on Linux).

[35] `http://www.dmx512.ch/512.html`
[36] Instructions: `https://wiki.wireshark.org/CaptureSetup/USB`

Table 4.1: Procedure for capturing DMXCreator's USB communication protocol. The *pcap packets* column corresponds to the packets in dmxcreator.pcap that were captured during each step.

#	Step	*pcap* packets
1.	Start to capture USB traffic with Wireshark.	0 – 26
2.	Plug in the DMXCreator 512 Basic USB adapter.	27 – 97
3.	Start the DMXCreator software.	98 – 103
4.	Patch *eurolite LED PAR-64 RGB Spot* fixture (5 channels: Red, Green, Blue, Dimmer, Flash) to DMX address 1.	
5.	Patch another *eurolite LED PAR-64 RGB Spot* fixture to DMX address 367.	
6.	Go to main window.	104 – 106
7.	Slowly fade the first fixture's *Red* channel (DMX channel 1) down from 255 to 0.	107 – 260
8.	Slowly fade the second fixture's *Blue* channel (DMX channel 369) down from 255 to 86.	261 – 386
9.	Slowly fade the second fixture's *Green* channel (DMX channel 368) down from 255 to 0.	387 – 224
10.	End capturing.	

4.4.2 Extending OLA's USB DMX Plugin

The reverse-engineered protocol now had to be incorporated into OLA's USB DMX Plugin. Since this plugin does not depend on Raspberry Pi's embedded hardware, development could happen completely on my more powerful work computer to speed up build times.

Note: All code blocks in this section can be found unshortened in OLA's GitHub pull request 1136[37].

First, the user starting `olad` has to be given permissions to communicate with the USB adapter. This can be achieved by adding the following rule to the udev rules from section 4.2.3 in `/etc/udev/rules.d/10-ola.rules` and reloading the rules. It allows all members of the *plugdev* group access to USB devices identified by the given vendor and product IDs.

```
19 # udev rules for the DMXCreator 512 Basic device
20 SUBSYSTEM=="usb|usb_device", ACTION=="add", ATTRS{idVendor}=="0a30", ATTRS{↩
       idProduct}=="0002", GROUP="plugdev"
```

I also added the rule to OLA's debian/ola.udev file to have it included in released OLA `.deb` packages. When installing these, the rules gets extracted to the correct location.

[37] https://github.com/OpenLightingProject/ola/pull/1136/files

DMXCreator512BasicFactory (factory class)

For creating a new sub-plugin of the USB DMX Plugin, I followed its developer information document[38]. Every sub-plugin has a *factory* class, whose `DeviceAdded` method is called whenever a new USB device is found. This method is expected to return `false` if the device should not or cannot be claimed by this sub-plugin. In *DMXCreator*'s case, the only identifying attributes of the USB adapter are its vendor and product IDs.

Listing 4.2: Excerpt from DMXCreator512BasicFactory.cpp.

```
36 const uint16_t DMXCreator512BasicFactory::VENDOR_ID = 0x0a30;
37 const uint16_t DMXCreator512BasicFactory::PRODUCT_ID = 0x0002;
38
39 bool DMXCreator512BasicFactory::DeviceAdded(
40     WidgetObserver *observer,
41     libusb_device *usb_device,
42     const struct libusb_device_descriptor &descriptor) {
43   if (descriptor.idVendor != VENDOR_ID || descriptor.idProduct != PRODUCT_ID) {
44     return false;
45   }
46
47   LibUsbAdaptor::DeviceInformation info;
48   if (!m_adaptor->GetDeviceInfo(usb_device, descriptor, &info)) {
49     return false;
50   }
51
52   OLA_INFO << "Found a new DMXCreator 512 Basic device";
```

The USB adapter does not return a serial number, so because it is not possible to distinguish between different devices, only one at a time is supported.

```
58   if (info.serial.empty()) {
59     if (m_missing_serial_number) {
60       OLA_WARN << "We can only support one device without a serial number.";
61       return false;
62     } else {
63       m_missing_serial_number = true;
64     }
65   }
```

In the case that the device *should* be claimed, the method creates a new widget instance, passes it to `BaseWidgetFactory`'s `AddWidget` method (located in *WidgetFactory.h*), which tries to initialize it, and returns the result. OLA supports detecting *hot-plugged* devices (i.e. devices plugged in after OLA was started) in the default asynchronous mode of the underlying `libusb` library, but a fallback version using `libusb`'s synchronous methods shall be implemented as well. Thus, two classes `AsynchronousDMXCreator512Basic` and `SynchronousDMXCreator512Basic` are implemented as child classes of the widget class `DMXCreator512Basic`.

[38] https://github.com/OpenLightingProject/ola/blob/master/plugins/usbdmx/README.developer.md

DMXCreator512Basic (widget class)

The protocol allows sending a whole universe (instead of only the first half), thus I decided to always use this method. In DMXCreator512Basic.cpp, I declared a constant array with the bytes which should be sent to USB endpoint `0x01`.

Listing 4.3: Excerpt from DMXCreator512Basic.cpp.

```
55 // if we only wanted to send the first half of the universe, the last byte would
56 // be 0x01
57 static const uint8_t status_buffer[6] = {
58   0x80, 0x01, 0x00, 0x00, 0x00, 0x02
59 };
```

Both the synchronous and asynchronous implementations of `DMXCreator512Basic` use the *Facade* software pattern, i.e. method calls to the widget class are passed through to a child class of `ThreadedUsbSender` or `AsyncUsbSender`, respectively. I explain my approach to implementing the synchronous version here because it is a bit easier to understand. The asynchronous version provides the same functionality while using asynchronous `libusb` methods and callback functions.

For initialization, the USB device must be opened and claimed. With the resulting device handle, the `ThreadedUsbSender` child class can be instantiated and a pointer of it is saved in the instance variable `m_sender`.

```
160 bool SynchronousDMXCreator512Basic::Init() {
161   libusb_device_handle *usb_handle;
162
163   bool ok = m_adaptor->OpenDeviceAndClaimInterface(
164       m_usb_device, 0, &usb_handle);
165   if (!ok) {
166     return false;
167   }
168
169   std::auto_ptr<DMXCreator512BasicThreadedSender> sender(
170       new DMXCreator512BasicThreadedSender(m_adaptor, m_usb_device,
171                                            usb_handle));
172   if (!sender->Start()) {
173     return false;
174   }
175   m_sender.reset(sender.release());
176   return true;
177 }
```

Whenever new DMX data shall be sent via this USB adapter, the widget's `SendDMX` method is called, which just forwards the `DmxBuffer` object to `m_sender` if it is already available (i.e. if the `Init` function was called correctly before), whose infinite loop then in turn calls `TransmitBuffer` in the next iteration.

```
179 bool SynchronousDMXCreator512Basic::SendDMX(const DmxBuffer &buffer) {
180   return m_sender.get() ? m_sender->SendDMX(buffer) : false;
181 }
```

In `TransmitBuffer`, the provided `DmxBuffer` first is compared with the last transmitted one. If they are the same, nothing needs to be transmitted at all.

```
96  bool DMXCreator512BasicThreadedSender::TransmitBuffer(
97      libusb_device_handle *handle, const DmxBuffer &buffer) {
98
99    if (m_dmx_buffer == buffer) {
100     // no need to update -> sleep 50µs to avoid timeout errors
101     usleep(50);
102     return true;
103   }
104
105   m_dmx_buffer = buffer;
```

Else, both halves of the universe are copied into `m_universe_lower` and `m_universe_upper`. If the provided `DmxBuffer` does not contain all 512 channels (the `length` variable gets set to the number of copied channels), the rest is filled up with zeros.

```
107   unsigned int length = CHANNELS_PER_PACKET;
108   m_dmx_buffer.Get(m_universe_data_lower, &length);
109   memset(m_universe_data_lower + length, 0, CHANNELS_PER_PACKET - length);
110
111   length = CHANNELS_PER_PACKET;
112   m_dmx_buffer.GetRange(CHANNELS_PER_PACKET, m_universe_data_upper, &length);
113   memset(m_universe_data_upper + length, 0, CHANNELS_PER_PACKET - length);
```

Afterwards, `status_buffer` is sent to USB endpoint `0x01` and both halves are consecutively sent to endpoint `0x02`. If any operation fails, `false` is returned, so that the thread stops and the device handle gets closed.

```
115   bool r = BulkTransferPart(handle, ENDPOINT_1, status_buffer,
116                             sizeof(status_buffer), "status");
117   if (!r) {
118     return false;
119   }
120
121   r = BulkTransferPart(handle, ENDPOINT_2, m_universe_data_lower,
122                        CHANNELS_PER_PACKET, "lower data");
123   if (!r) {
124     return false;
125   }
126
127   r = BulkTransferPart(handle, ENDPOINT_2, m_universe_data_upper,
128                        CHANNELS_PER_PACKET, "upper data");
129   return r;
130 }
```

Integration

At several locations, the new classes had to be integrated into the USB DMX Plugin. However, the required snippets are very similar to all existing sub-plugins, hence, I will not include them here, but only provide a list of changed files and functions for reference:

- AsyncPluginImpl.cpp:
  instantiated factory class in `Start` method; overloaded `NewWidget` method
- AsyncPluginImpl.h:
  overloaded `NewWidget` method
- SyncPluginImpl.cpp:
  instantiated factory class in constructor; overloaded `NewWidget` method
- SyncPluginImpl.h:
  overloaded `NewWidget` method
- SyncronizedWidgetObserver.h [sic][39]:
  overloaded `NewWidget` method
- WidgetFactory.h:
  overloaded virtual `NewWidget` method

Additionally, the USB adapter was included in UsbDmxPlugin.cpp's plugin description[40] and the new files had to be added to Makefile.mk to allow recompiling with `make` and `make install`.

Now, patching the new *DMXCreator 512 Basic USB Device* to a universe in OLA's web interface and sending DMX data through it is working as expected.

The new code was merged back into OLA's project repository on GitHub[41], after improving my initial version together with the project's main developers Peter Newman and Simon Newton.

4.5 Implementation of the OLA Native SPI DMX Plugin

Two output ports are provided by the UART and USB DMX plugins, the input port is yet to be implemented. In this section I briefly describe my first two approaches and why they failed. Afterwards, the working solution with SPI is explained in detail.

4.5.1 Insufficiency of Raspberry Pi's UART input

My first idea was to extend the UART plugin to also support DMX input. This seemed perfect since the UART and DMX protocols are so similar and very little software overhead would be needed.

However, there is a big catch: Receiving and recognizing the BREAK signal is very difficult because it is just forwarded to the application as a null byte and thus indistinguishable from

[39] I raised issue #1331 on GitHub to correct the typing error.
[40] Today, after changes in OLA's plugin structure, the plugin description is located in *README.md* and read in from there.
[41] `https://github.com/OpenLightingProject/ola/pull/1136`

data channels that are just set to zero. There are options in the `termios` C interface's `c_iflag` input flags to change this:

Listing 4.4: Excerpt from the *termios* man page. Note that octal \377 is 255 in decimal.

```
BRKINT If IGNBRK is set, a BREAK is ignored.  If it is not set but
       BRKINT is set, then a BREAK causes the input and output queues
       to be flushed, and if the terminal is the controlling terminal
       of a foreground process group, it will cause a SIGINT to be
       sent to this foreground process group.  When neither IGNBRK
       nor BRKINT are set, a BREAK reads as a null byte ('\0'),
       except when PARMRK is set, in which case it reads as the
       sequence \377 \0 \0.

IGNPAR Ignore framing errors and parity errors.

PARMRK If this bit is set, input bytes with parity or framing errors
       are marked when passed to the program.  This bit is meaningful
       only when INPCK is set and IGNPAR is not set.  The way erro-
       neous bytes are marked is with two preceding bytes, \377 and
       \0.  Thus, the program actually reads three bytes for one
       erroneous byte received from the terminal.  If a valid byte
       has the value \377, and ISTRIP (see below) is not set, the
       program might confuse it with the prefix that marks a parity
       error.  Therefore, a valid byte \377 is passed to the program
       as two bytes, \377 \377, in this case.

       If neither IGNPAR nor PARMRK is set, read a character with a
       parity error or framing error as \0.

INPCK  Enable input parity checking.

ISTRIP Strip off eighth bit.
```

However, regardless of which settings I tried, after some time only random data bytes were decoded. The UART seemed to be confused by the frequent BREAKs.

I was unable to figure out where this issue arose from because I could not verify if the signal was captured by the UART correctly. So I decided to look for a way to receive data that permits access to the "raw" DMX signal, as this would potentially be a more stable approach. It would require implementing parsing myself but thereby also give me full control over it.

4.5.2 Bit bang reading with pigpio library

The next idea was to constantly poll one GPIO pin's value and getting the raw signal that way. This technique is known as "bit bang reading" in the *pigpio* library[42]. I had concerns about the speed and precision of the read process, since Linux is not a real time operating system and scheduling could delay read operations so that they already sample the pin when the next bit is fed in. At 250kbit/s, these delays could already be significant.

[42] `http://abyz.me.uk/rpi/pigpio/cif.html#gpioSerialReadOpen`

Fortunately, *pigpio* provides a diagnose tool called *piscope* that displays the bit banged signal. I generated DMX data with an external DMX interface and inspected the signal captured by *piscope*. An example capture image together with the expected signal can be seen in figure 4.4.

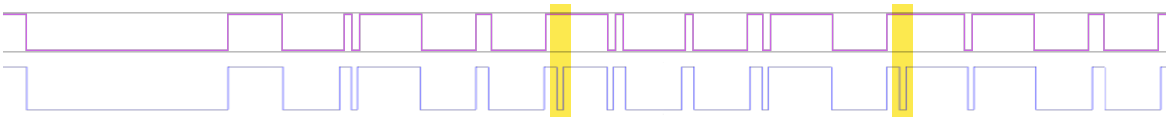

Figure 4.4: *pigpio* "bit bang read" signal shown in *piscope* (top) versus sent data. The sent channel values are 255, 0, 0, 127, 0, 0, 255, 255, 0, 0, ...

This revealed that often a short pulse, i.e. a quick change from `low` to `high` to `low` or the other way around, was not visible at all, e.g. in slot 4 and 7 in figure 4.4. Additionally, sampling seemed to happen once every 5µs, since slots' stop bits (2 `high` bits) lasted sometimes 5µs and sometimes 10µs, but never 8µs as expected.

In conclusion, "bit bang reading" a GPIO pin is not sufficiently accurate for the 250kbit/s DMX signal.

4.5.3 Using SPI to sample DMX

An idea I came across in a *Raspberry Pi StackExchange* answer[43] while researching GPIO bit banging speeds was sampling the native SPI port (see section 2.3) for arbitrary DMX data. For this, the DMX line is connected (via the bus transceiver) to Raspberry Pi's *MISO* (*Master In, Slave Out*) pin, the other SPI pins are left unconnected. The intention is that SPI is designed for much higher speeds than UART, so a stable clock frequency is important. Unfortunately, not many details were provided in that post, and it seems to be a very uncommon technique, so I had to figure out most steps myself.

Raspberry Pi's SPI controller (acting as SPI master) has a core frequency of 250MHz that can be divided by any even number[44]. The goal is to sample the connected DMX signal 8 times per bit to have enough tolerance if sometimes the sample time falls exactly on an edge, so the required sample frequency is 250kbit/s · 8/bit = 2MHz. The required clock divider 250MHz / 2MHz = 125 is odd, so 124 will be used instead (odd divisors are rounded down) and parsing of the sampled bits must be flexible enough to account for this inaccuracy. However, since a valid DMX receiver has to accept any signal with a bit rate of 245kbit/s to 255kbit/s (see section 2.1.2), flexibility must be ensured anyway.

[43] `https://raspberrypi.stackexchange.com/a/2044`

[44] According to the BCM2835 manual [Bro12], only powers of two can be used as clock divider, but this is incorrect according to `https://raspberrypi.stackexchange.com/a/3444` and testing by myself.

$$f_{eff} = \frac{250\text{MHz}}{124} \approx 2.02\text{MHz}$$

$$\frac{f_{eff}}{245\text{kbit/s}} \approx \frac{8.23 \text{ sampled bits}}{\text{DMX bit}}$$

$$\frac{f_{eff}}{250\text{kbit/s}} \approx \frac{8.06 \text{ sampled bits}}{\text{DMX bit}}$$

$$\frac{f_{eff}}{255\text{kbit/s}} \approx \frac{7.91 \text{ sampled bits}}{\text{DMX bit}}$$

4.5.3.1 Enabling SPI

Enabling SPI can be done with `raspi-config`. To increase the buffer size (i.e. the number of bytes that can be received/transmitted in one operation), `spidev.bufsiz=65536` should be added to the kernel options in `/boot/cmdline.txt`. After a reboot, the value returned by `cat /sys/module/spidev/parameters/bufsiz` should be the requested 65536.

Note: Since there is much contradicting information available online, I want to clarify: In newer firmware versions, no additional steps (like manually enabling a kernel module or blacklisting another) are needed.

To test the configuration, the *MISO* and *MOSI* pins can be wired together for a *loopback* test:

```
wget https://raw.githubusercontent.com/raspberrypi/linux/rpi-3.10.y/↩
    Documentation/spi/spidev_test.c
gcc -o spidev_test spidev_test.c
./spidev_test --device /dev/spidev0.0 --speed 2000000
```

Output should look like the following.

```
spi mode: 0
bits per word: 8
max speed: 2000000 Hz (2000 KHz)

FF FF FF FF FF FF
40 00 00 00 00 95
FF FF FF FF FF FF
FF FF FF FF FF FF
FF FF FF FF FF FF
DE AD BE EF BA AD
F0 0D
```

Additionally, I ran the test while connecting the *MISO* pin to either +3.3V, ground, or leaving it unconnected. As expected, the output was `FF`s only, `00`s only and again `00`s only, respectively.

4.5.3.2 Receiving DMX data

Based on the loopback test program above, I wrote spi-receive.c, which reads in 8192 bytes[45] four times from the SPI MISO bus and outputs them in binary format to the console. As the loopback test code and the existing SPI Plugin (see section 4.1.2.4) do too, it uses the `spidev` interface, which enables SPI communication from user space (not as part of the kernel).

A single SPI transfer with `spidev` is configured by an `spi_ioc_transfer` struct like shown below. The transfer itself is then executed with `ioctl(fd, SPI_IOC_MESSAGE(1), &tr)`.

Listing 4.5: Excerpt from spi-receive.c.

```
43   struct spi_ioc_transfer tr = {
44     // don't transmit anything
45     .tx_buf = 0,
46
47     // save received bytes into `rx` buffer (at appropriate offset)
48     .rx_buf = (unsigned long)(rx + BYTES_PER_TRANSFER*offset),
49
50     // bytes to send/receive in this transfer operation
51     .len = BYTES_PER_TRANSFER,
52
53     // don't delay after data bytes are sent
54     .delay_usecs = delay,
55
56     // overwrite speed temporarily to 2MHz
57     .speed_hz = speed,
58
59     // overwrite bits per word temporarily to 8
60     .bits_per_word = bits_per_word,
61   };
```

The `transfer` function, where the struct above is created and used to initiate the SPI data transfer, is called four times. Afterwards, the `printBinary` function prints the `rx` buffer as binary numbers. It uses the `BYTE_TO_BINARY` macro to deconstruct bytes into 8 ones and zeros.

```
71 #define BYTE_TO_BINARY_PATTERN "%c %c %c %c %c %c %c %c "
72 #define BYTE_TO_BINARY(byte)  \
73   (byte & 0x80 ? '1' : '0'), \
74   (byte & 0x40 ? '1' : '0'), \
75   (byte & 0x20 ? '1' : '0'), \
76   (byte & 0x10 ? '1' : '0'), \
77   (byte & 0x08 ? '1' : '0'), \
78   (byte & 0x04 ? '1' : '0'), \
79   (byte & 0x02 ? '1' : '0'), \
80   (byte & 0x01 ? '1' : '0')
```

[45] On Raspberry Pi, *spidev*'s buffer size is 4096 bytes by default. Because this limit is too low (see end of section 4.5.3.4), its maximum size was increased to 65536 in the previous section.

I tried the program with an example DMX signal and saved the resulting binary data to dmx-spi-data.txt. Then these data were loaded and plotted in *GNU Octave*[46] with the following commands. A screenshot of the resulting plot is shown in figure 4.5.

```
load "dmx-spi-data.txt"
data2 = dmx_spi_data(2, :)  # copy 2nd chunk
stairs(data2 * 0.9 + 0.05)  # show square signal
axis([17000 19500 0 1])     # show x values (bits) 17000...19500, y values 0...1
```

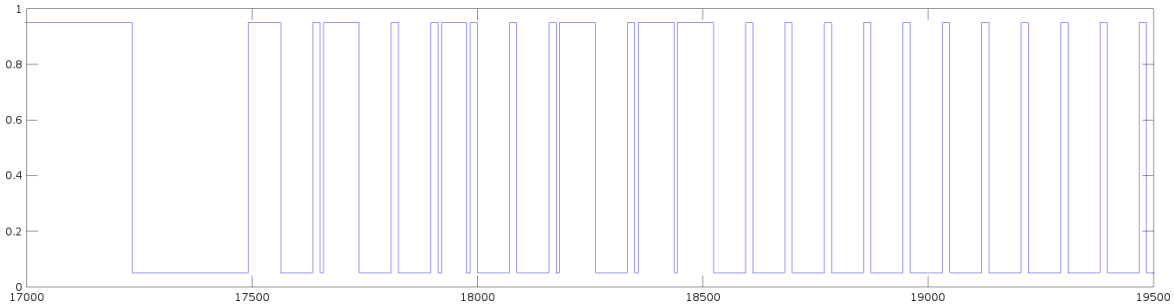

Figure 4.5: Received SPI data (excerpt) plotted with GNU Octave.

I repeated the procedure multiple times and the plots kept looking very promising. No short pulses were missed and due to the oversampling, the timing was also very accurate[47]. The only problem I noticed was that two consecutive chunks (i.e. received in multiple transfer operations) are separated by an uncaptured gap, even if the `transfer` calls are right after each other in the code.

That means that the resulting bytes cannot be parsed as a (possibly infinitely) long stream of bits, but rather each chunk must be parsed on its own. As a result, there are chunks that do only contain the start of a DMX packet, some do only contain the end. These are useless though, since it is not clear how many channels have been transmitted before. Thus, the refresh rate is lower for higher channels. The problem is visualized in figure 4.6.

[46] `https://www.gnu.org/software/octave/`

[47] Actually, the "real" DMX signal that the *piscope* bit banged signal in figure 4.4 was compared against, was a plot of the same data received with SPI.

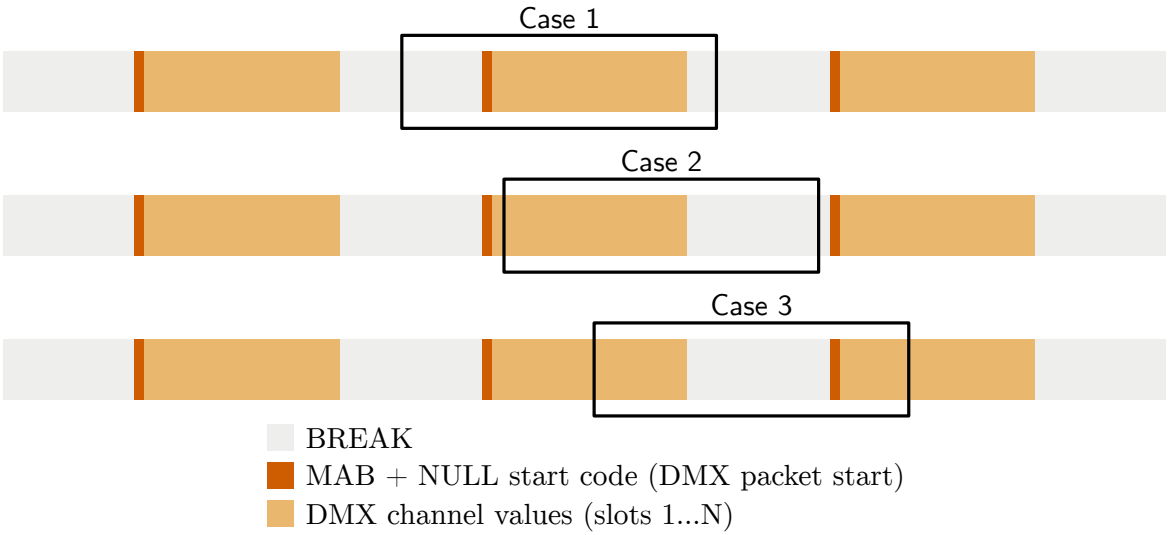

Figure 4.6: Received SPI chunks versus DMX signal stream. *(Note: Sizes and proportions are not to scale.)* DMX packets can be either fully enclosed in one SPI chunk (case 1; this is the optimum case) or only partially. If only the end of a DMX packet is contained (case 2), the chunk is useless. If a DMX packet's start is included (case 1 and case 3), all channel values until the chunk end can be correlated to their respective channel numbers. Since it is less likely that a DMX packet starts right at the chunk's beginning than somewhere in the middle, higher channels are updated less often.

4.5.3.3 Parsing received SPI chunks

As mentioned earlier, receiving the raw signal requires implementing parsing the DMX channel values from the sampled data myself. This parsing has to obey timing constraints of the DMX protocol.

My approach to implementing this is a state machine that processes the sampled data bit by bit, proceeds to the next state if the received data follow the DMX protocol and goes back to the initial state otherwise. A flow chart of this state machine is pictured in figure 4.7.

After a DMX slot's start bit is detected, always the middle of the following 8 DMX bits (= 8 "SPI bytes") is sampled to construct the channel value. The subsequent stop bits decide how to proceed:

a) Either the two stop bits are `high` and arbitrarily many `high` bits as *mark between slots / mark before BREAK* follow. Then, at the next falling edge, the parser saves the constructed DMX channel value to the correct position and continues in the *in data start bit* state for the next slot. An exception is the last channel: If the just saved DMX value was written to channel 512, then it is clear that no more slots can follow, so the DMX packet is completed and the state gets changed to *in BREAK* instead.

b) Or there are `low` bits where the stop bits should be. If the constructed channel value is also zero, that was actually not a data slot, but the beginning of the BREAK. So all channel values from here on are set to zero, the DMX packet is completed and the state machine proceeds to *in BREAK*.

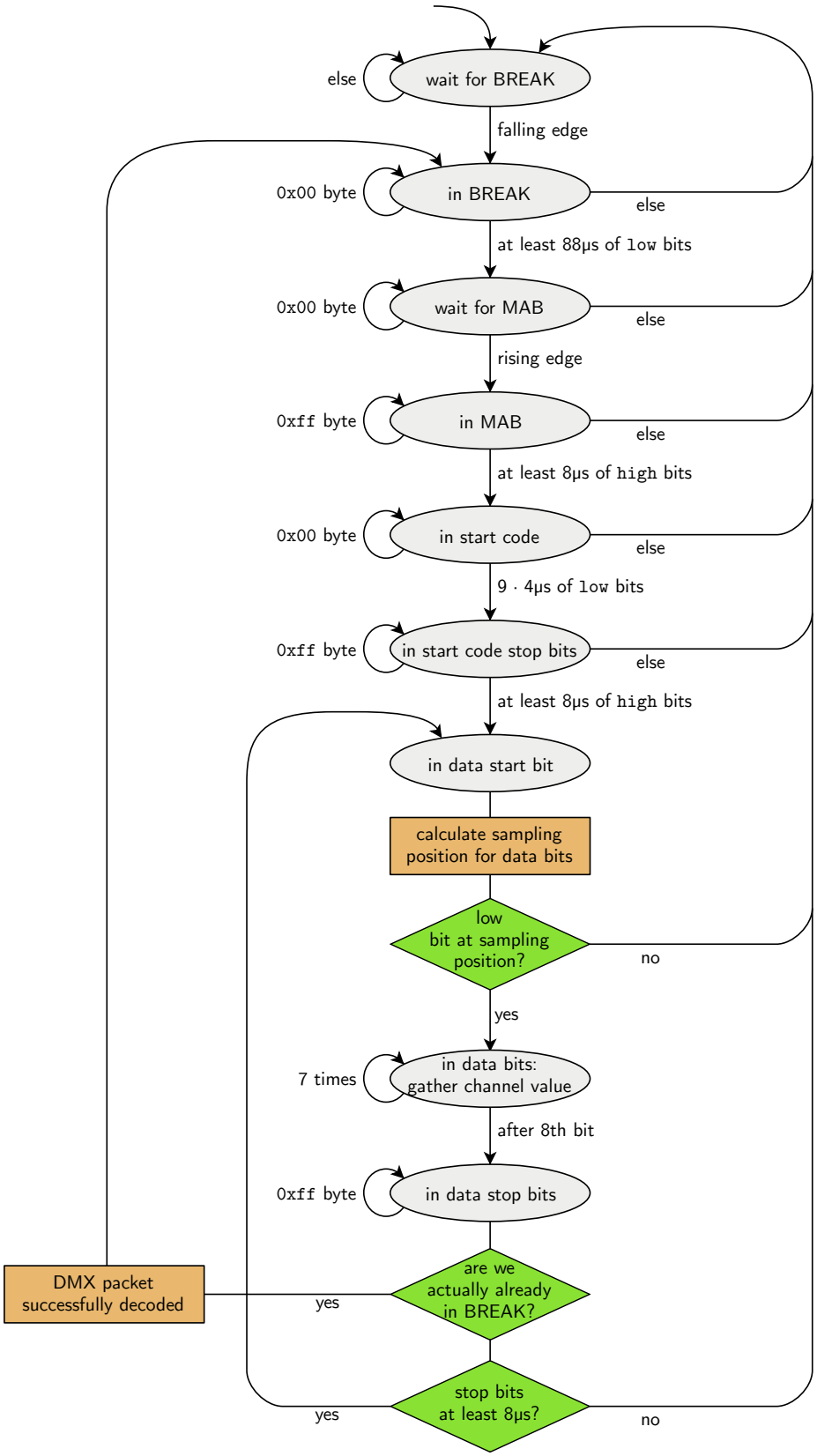

Figure 4.7: Parsing DMX from sampled SPI data with a state machine.

To recognize falling and rising edges, I wrote two helper functions. `DetectFallingEdge` returns the number of zeros if the passed byte has the form 1^n0^{8-n} ($n \in \{0, \ldots, 7\}$), and -1 otherwise. Note that this case can occur if the byte contains either only ones or random spikes. `DetectRisingEdge` works equivalently.

I show the `WaitForMab` function as a simple state handler example. It looks for the first rising edge after the BREAK to change to the `IN_MAB` state.

Listing 4.6: Excerpt from SPIDMXParser.cpp.

```
259 void SPIDMXParser::WaitForMab() {
260   uint8_t byte = chunk[chunk_bitcount];
261   if (byte != 0) {
262     int8_t ones = DetectRisingEdge(byte);
263     if (ones > 0) {
264       ChangeState(IN_MAB);
265       state_bitcount = ones;
266     } else {
267       ChangeState(WAIT_FOR_BREAK);
268     }
269   }
270   chunk_bitcount++;
271 }
```

Another notable state handler is `InDataStartbit` because it calculates the sampling position of the DMX data bits. This sampling position should always be in the middle of a byte, and thus depends on `state_bitcount`, i.e. the number of "SPI bits" that belong to the current state, set by the previous state handler (i.e. `InStartcodeStopbits` or `InDataStopbits`). Possibly, the middle bit was already contained in the last handled byte, so the current byte is reset to the previous one in this case. Table 4.2 lists all possible cases.

```
385 void SPIDMXParser::InDataStartbit() {
386   uint8_t byte = chunk[chunk_bitcount];
387
388   if (state_bitcount >= 4) {
389     // look at the last byte again and don't increase chunk_bitcount
390     byte = chunk[chunk_bitcount - 1];
391     sampling_position = state_bitcount - 4;
392   } else {
393     // next byte will be handled in next step as usual
394     chunk_bitcount++;
395     sampling_position = state_bitcount + 8 - 4;
396   }
397
398   // start bit must be zero
399   if ((byte & (1 << sampling_position))) {
400     ChangeState(WAIT_FOR_BREAK);
401   } else {
402     current_dmx_value = 0x00;
403     ChangeState(IN_DATA_BITS);
404   }
405 }
```

Table 4.2: Calculating the sampling position in `InDataStartbit` state handler. `d` represents a data bit. The desired sampling position is indicated with an arrow.

state bit count	previous byte, current byte	backtrack?	new current byte	sampling bit number
8	`00000000 dddddddd` ↑	yes	`00000000` ↑	4
7	`10000000 0ddddddd` ↑	yes	`10000000` ↑	3
6	`11000000 00dddddd` ↑	yes	`11000000` ↑	2
5	`11100000 000ddddd` ↑	yes	`11100000` ↑	1
4	`11110000 0000dddd` ↑	yes	`11110000` ↑	0
3	`11111000 00000ddd` ↑	no	`00000ddd` ↑	7
2	`11111100 000000dd` ↑	no	`000000dd` ↑	6
1	`11111110 0000000d` ↑	no	`0000000d` ↑	5

All other state handler functions can be inspected in SPIDMXParser.cpp.

The `ParseDmx` method, which is given an SPI chunk to parse, iterates through the chunk's bytes and calls the respective state handlers. Whenever a DMX packet end is detected, `PacketComplete` is called, which in turn invokes a callback function if it has been set before via SPIDMXParser.h's `SetCallback` method.

Note: The code listings above are part of the OLA plugin I wrote. Details are outlined in the next section. Initially, I tested the code in a standalone version that could process the output of spi-receive.c's `printCArray` function.

4.5.3.4 Wrapping the code into an OLA plugin

The final step was to create a new OLA plugin that provides an input port for every SPI port found on the system. This port can then be patched to an OLA universe which is forwarded via Art-Net to the lighting software or directly to a DMX output port.

OLA's *OSC (Open Sound Control) Plugin* includes information about its development process[48] which is a good reference for writing a new plugin. As a name, I chose *Native SPI DMX Plugin*, following *Native UART DMX Plugin*'s convention. All classes are prefixed with `SPIDMX`; their namespace, as well as directory name, is `spidmx`. I based the general plugin structure on UART plugin's:

[48] `https://github.com/OpenLightingProject/ola/blob/master/plugins/osc/README.developer.md`

- The *Plugin* class (SPIDMXPlugin.h) hooks into OLA's plugin infrastructure and searches for SPI devices.
- For every found SPI device, a new OLA *Device* (SPIDMXDevice.h) is instantiated, which in turn creates one instance of each of the following classes:
  - A *Widget* class (SPIDMXWidget.h) that abstracts away the required *spidev* calls.
  - A *Thread* class (SPIDMXThread.h) that repeatedly calls the Widget's `ReadWrite` method and saves the resulting data. The Thread itself instantiates a new *Parser* (SPIDMXParser.h) to decode the raw SPI data.
  - An *InputPort* class (SPIDMXPort.h) connects the Thread to OLA's plugin mechanism by forwarding DMX data and notifying the Thread whenever it is patched or unpatched to / from a universe.
- A README.md document describes the plugin.

The UML class diagram in figure 4.8 clarifies those relations.

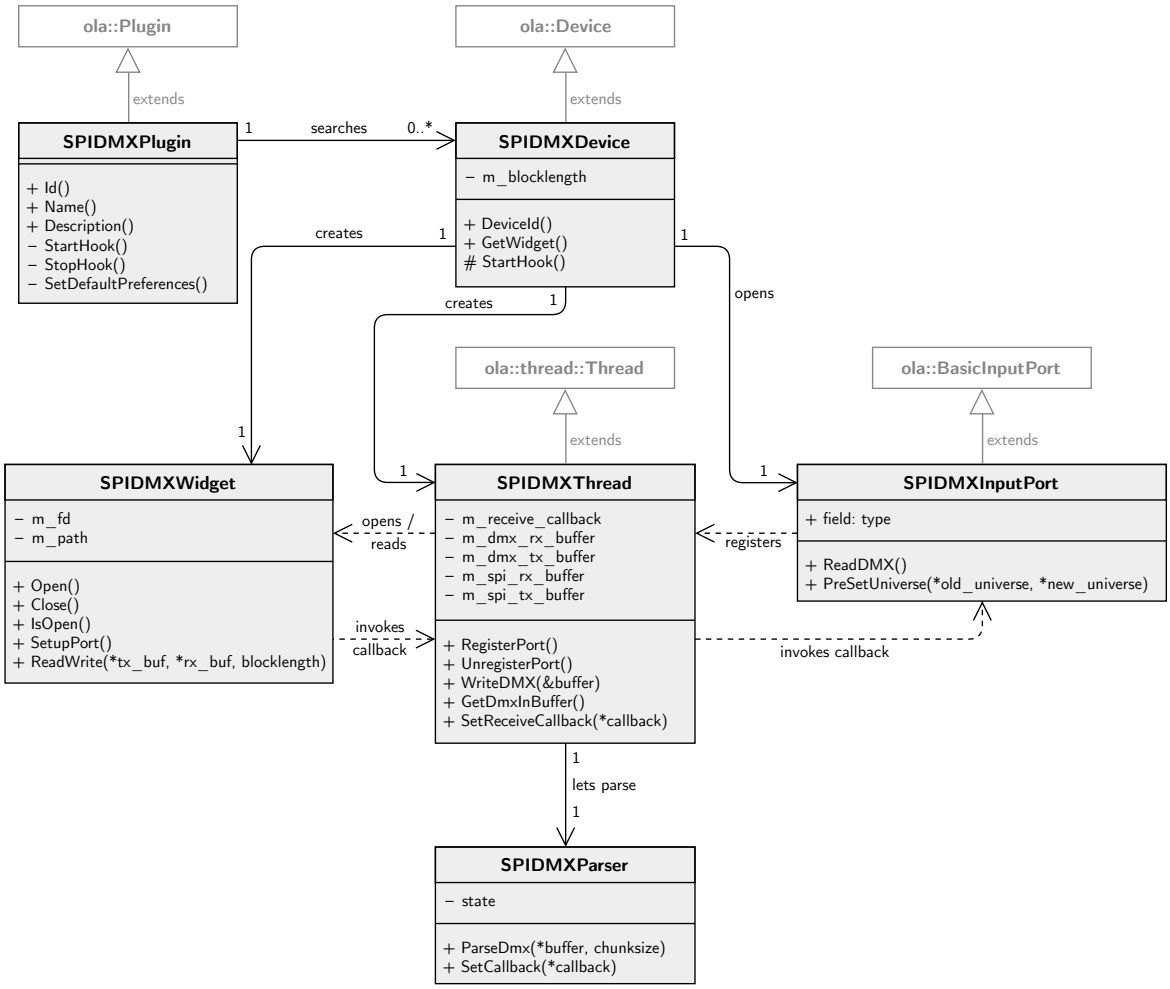

Figure 4.8: *Native SPI DMX Plugin* UML class diagram. Note that only important fields and methods are included.

Whenever the Parser detects a DMX packet end, its `PacketComplete` function invokes the callback function that was set in the constructor, called from SPIDMXThread.cpp. Its callback in turn is set in SPIDMXPort.h's `PreSetUniverse` method (listing 4.7) which is always called when the universe that the port is patched to changes.

This callback chain needs special attention because one has to remember which pointers could still reference a callback variable, especially as it could be one in another thread. Mistakes here easily lead to segmentation faults.

Listing 4.7: Excerpt from SPIDMXPort.h.

```
54   bool PreSetUniverse(Universe *old_universe, Universe *new_universe) {
55     if (!old_universe && new_universe) {
56       return m_thread->SetReceiveCallback(NewCallback(
57         static_cast<BasicInputPort*>(this),
58         &BasicInputPort::DmxChanged));
59     }
60     if (old_universe && !new_universe) {
61       return m_thread->SetReceiveCallback(NULL);
62     }
63     return true;
64   }
```

Since plugins can be enabled / disabled at build time, the *autotools* toolchain must be informed about the new plugin and its operating system dependencies (namely *spidev*), which was done by adding a single line to configure.ac:

```
836 PLUGIN_SUPPORT(spidmx, USE_SPIDMX, [$have_spi])
```

Additionally, the plugin's Makefile.mk, which is similar to other plugins' Makefiles, had to be included from plugins/Makefile.mk.

At runtime, the plugin is loaded in olad/DynamicPluginLoader.cpp, where the build constant `USE_SPIDMX` determines if the plugin was enabled at build time.

```
215 #ifdef USE_SPIDMX
216   m_plugins.push_back(
217       new ola::plugin::spidmx::SPIDMXPlugin(m_plugin_adaptor));
218 #endif  // USE_SPIDMX
```

Each plugin is assigned a plugin ID constant in common/protocol/Ola.proto, in this case `OLA_PLUGIN_SPIDMX = 23`. It was temporarily 10000 during development and changed to its final value shortly before merging into the `master` branch.

After integrating the new plugin into the existing infrastructure, a complete rebuild of the project was required:

```
autoreconf
./configure
make
sudo make install
sudo ldconfig
```

The finished SPI DMX Plugin was tested with multiple DMX sources and works fine. As explained earlier, higher channels are updated less often, resulting in higher latency. Hence, they should not be used for light fading, which should be smooth by definition.

The chunk size of one transmission – which has the largest impact on this – is configurable through OLA's plugin settings file (`/home/pi/.ola/ola-spidmx.conf`). However, the value of 8192 bytes seems to be a good compromise between a high probability of including higher channels in a chunk on one hand, and not having too large uncaptured gaps between chunks on the other.

Note: All code changes and additions from this section can be found in OLA's GitHub pull request 1289[49].

4.6 Chassis build

To make the DMX interface robust and portable, I housed the Raspberry Pi together with the extension board in a plastic chassis.

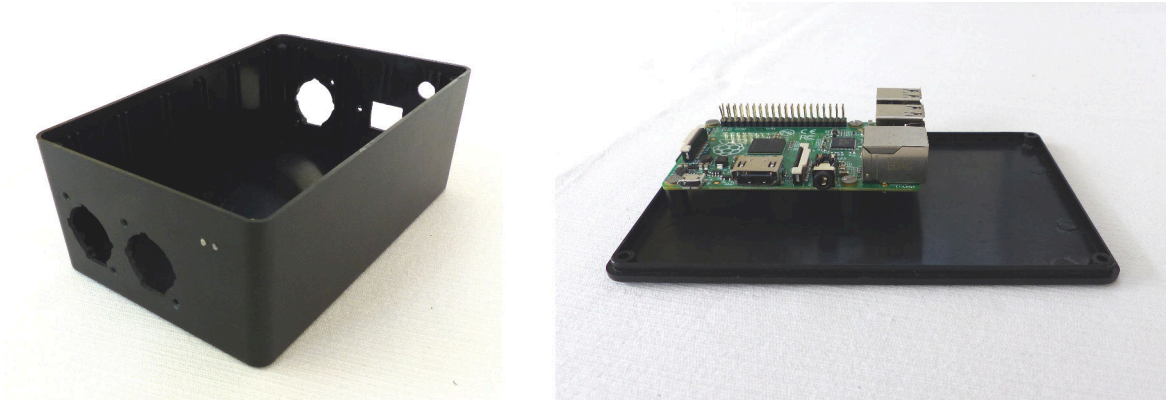

Figure 4.9: Plastic chassis. Top half with holes for mounting connectors, bottom half with mounted Raspberry Pi.

All external plugs (2 female XLR connectors, 1 male XLR connector, 1 Ethernet jack and 1 power supply plug) are connected via pluggable extension wires. This allows for a modular installation and easy replacement of faulty parts. To fit the *DMXCreator 512 Basic* USB adapter into the chassis, I had to remove its casing, replace its cable with a more flexible one and enclose it in a heat shrink tube.

[49] `https://github.com/OpenLightingProject/ola/pull/1289/files`

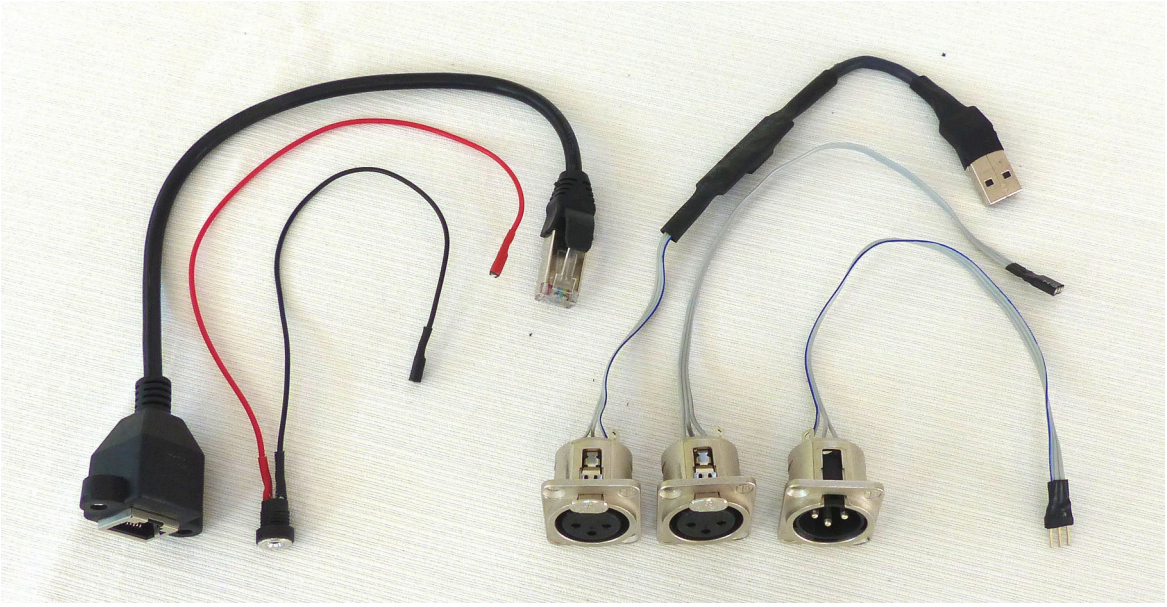

Figure 4.10: Connectors with extension cables.

Because Raspberry Pi's indicator LEDs are useful to know if it has powered up and booted correctly, I mounted transparent plastic cords that act as optical waveguides.

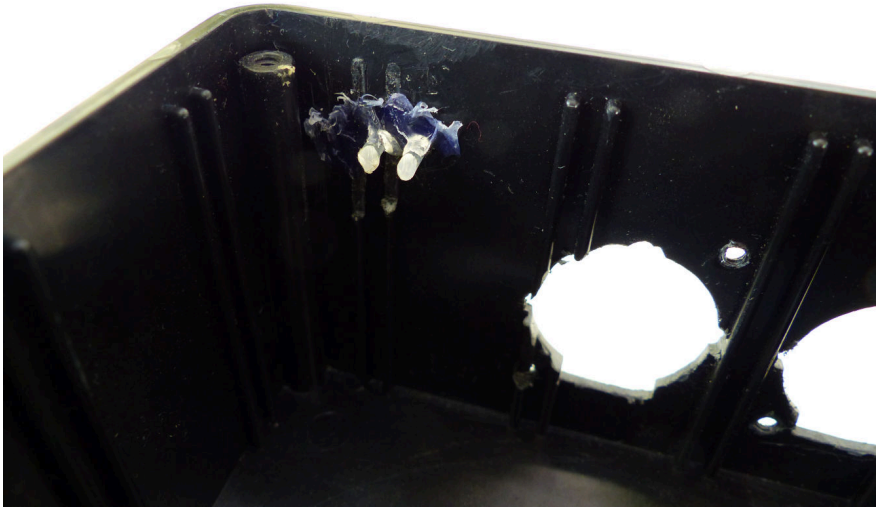

Figure 4.11: Optical waveguides for indicator LEDs.

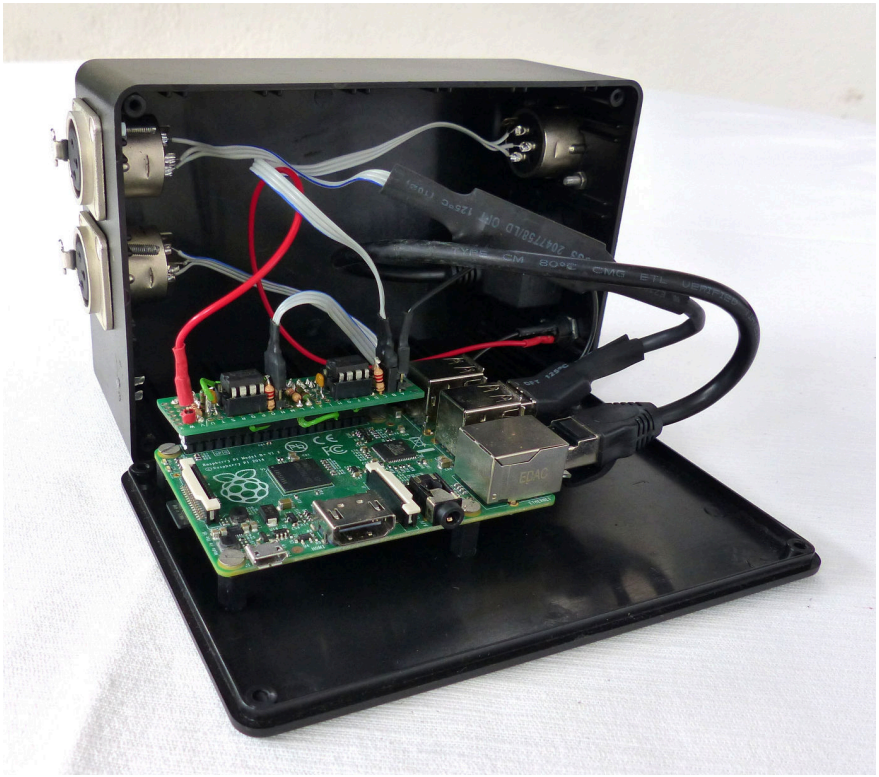

Figure 4.12: Open chassis with all connectors mounted and connected.

Figure 4.13: Finished PC-DMX interface.

5 Validation

The PC-DMX interface implementation – both hardware and software – needs to work reliably. This means in particular that it should fulfill all requirements I defined in section 3.1 at all times, except when outer circumstances, e.g. power loss, prevent it from doing so.

Code quality checks and unit tests can help as automatic tools to ensure this. However, as I will outline in the first subsection, I cannot fully rely on them.

5.1 Unit tests

The Open Lighting Architecture project provides a test infrastructure and various existing tests. My code changes and additions did not cause these existing tests to fail, which suggests that no regressions were introduced.

Testing the new code with actual hardware support is very difficult. A DMX signal would have had to be sent through an output port back into an input port to check its delay and data integrity. That procedure would depend on two plugins and thereby violate the principle of isolation for unit tests: If the test fails, it is not clear which of the components caused it to. Also, hardware testing is not automatable in continuous integration tools.

Instead, I manually tested the PC-DMX interface's functionality by connecting light fixtures and comparing expected and observed light output. Indeed, the interface was already successfully deployed at several events that lasted for around 10 hours. It ran stable for the whole period of time, indicating that no mistakes were made in the implementation that would noticeably affect the behavior.

5.2 Code quality

To ensure a high code quality standard, the OLA project has defined a code style that has to be adhered to. The automatic *lint checks* that are run after every commit in GitHub pull requests to prevent violation are fulfilled by my code changes.

The consistent code style improves readability, which makes careful reviews by the project maintainers in the GitHub pull requests easier. Discussion there also helped find bugs and inconsistencies before merging the new features into the `master` branch.

5.3 Fulfillment of requirements

Many of the requirements are already fulfilled by OLA as the software basis. That applies to supporting different lighting control programs using both proprietary and open network protocols, ease of use through the web interface and being open-source. Also, OLA directly

allows flexibly patching DMX inputs and outputs to universes, which permits all wanted use cases and possibly even more.

Two DMX outputs and one DMX input are supported right now, more can be added by simply connecting appropriate hardware via USB. DMX output frequency is satisfactory, which was verified manually by connecting light fixtures at different DMX addresses and fading them smoothly using a lighting control software. DMX input is also picked up often enough to allow smooth fading for approximately the lower 400 channels, fading higher channels feels rough.

The costs of Raspberry Pi 1 Model B+, 4GB microSD card, circuit board, electronic components, plug connectors chassis and power supply add up to about 60€, staying far below the limit of 100€. The *DMXCreator 512 Basic* USB adapter though is not included. It was apparently sold for over 400€[50] before it was discontinued, but it was available to me for free anyway, so it would not be fair to count it. As substitute, other very cheap USB-DMX adapters (around 20€) could possibly be integrated into OLA with some effort. Other options are mentioned in the next chapter.

Table 5.1: Fulfillment of requirements. See section 3.1.

Requirement	Fulfillment
Works with multiple lighting control programs	✓
Connection to computer possible via Ethernet	✓
Open protocol between computer and interface supported	✓
2 DMX output ports	✓
1 DMX input port	✓
Input universe handling configurable	✓
High refresh rate for DMX input and output	mostly ✓ (higher DMX input channels are refreshed less often, see above)
Easy usability for end-users	mostly ✓ (computer's network settings have to be changed)
Costs below 100€	partially ✓ (*DMXCreator 512 Basic* USB-DMX adapter exceeds limit, see above)
Extensible	✓
Open-source	✓

[50] `http://vxco.ch/wp-content/uploads/2012/10/DMXCREATOR-_PREISLISTE_CH_D_2.15.pdf`

6 Conclusion and future work

The goal of this thesis was to create an inexpensive yet feature-rich interface between lighting control software on a computer and DMX fixtures. The detailed requirements were defined and a market study was conducted to identify strengths and weaknesses in existing products. No product fulfilled all requirements, so a system design was developed and implemented, which included reverse engineering the protocol of a USB-DMX adapter and sampling the SPI bus.

As a result, the new PC-DMX interface is a big improvement over the proprietary *e:cue* interface used before in my parish youth: Haptic input is now possible using any DMX desk console (though not entirely for a full universe) and it can be used with multiple lighting control programs to compare their features and concepts.

There are several subjects that can be improved for future versions and reproductions to make the interface less expensive, more easy to use or to add extra features.

More Raspberry Pi-native DMX output ports

To stay below the limit of 100€ for future replications, the output port that is currently provided by the *DMXCreator 512 Basic* USB adapter needs to be replaced. If this second output could be generated directly with Raspberry Pi's hardware, the costs for a USB adapter would diminish completely.

An approach that looks promising is using the SPI port not only for DMX input, but also for output, i.e. making use of the *MOSI* pin as well. Each DMX channel value would have to be encoded in 11 bytes; one `low` byte as the DMX start bit, then one byte for each DMX data bit and two `high` bytes as stop bits. In total, this adds up to $11 \cdot 512 = 5632$ bytes, plus some more for the reset sequence, which is suitable to go together with the current input implementation.

Another idea is bit banging UART output on a GPIO pin. Timing was too unreliable for parsing DMX input data but things could indeed look different for output. There is a benchmark measuring GPIO bit banging speed[51], which suggests maximum possible speed is not an issue when using a native C implementation, like *pigpio*'s `gpioWaveAddSerial` function[52]. However, inaccurate timing could of course still be a problem, so further investigation is needed to draw a conclusion.

Remote device management (RDM)

As explained briefly at the end of section 2.1.2, *Remote Device Management* (*RDM*) is a bidirectional extension to DMX that allows setting fixture's options remotely. RDM controllers are usually even more expensive than regular DMX desk consoles and PC interfaces.

[51] `http://codeandlife.com/2012/07/03/benchmarking-raspberry-pi-gpio-speed/`
[52] `http://abyz.me.uk/rpi/pigpio/cif.html#gpioWaveAddSerial`

OLA has support for RDM internally, but there is currently no way to receive / transmit RDM packets natively on the Raspberry Pi.

Note that both directions (input / output) need to be supported for RDM. Thus, the *SPI DMX Plugin* could be modified to support DMX output (see previous section) and make the input use configurable: Either one wants to use both input and output ports individually, or only one output port with RDM support is desired.

Improve network management

Another expensive addition to regular DMX systems are so-called *wireless DMX* solutions, which use radio signals to transmit DMX data over the air to their respective receivers. Those are useful when a long distance has to be bridged between the DMX source and the first lighting fixture.

Since network protocol like Art-Net and sACN can be carried over Wi-Fi without problems, it would be possible to use it for communication between the PC-DMX interface and the controlling computer. One would only need to add a wireless network adapter to both the Raspberry Pi and the computer (if it is not a laptop with builtin Wi-Fi support anyway) and connect both to an external access point.

Currently, as one static IP address is set for Raspberry Pi's network adapter, it is not possible to use multiple PC-DMX interfaces in the same setup without manually changing the IP addresses to differ from each other. Additionally, an IP address in the same subnet must be chosen for the computer, which is tedious extra setup work. Especially in a wireless system, it would be a desired feature to have them managed automatically.

An idea would be to have one PC-DMX interface set into *master* mode, e.g. with a hardware switch. It then runs a DHCP server and acts as the access point for all other DMX interfaces in *slave* mode and also the controlling computer.

Postface

Since all parts of this PC-DMX interface are open-source, anybody reading this is welcome to build upon my hardware engineering and to contribute to the *Open Lighting Architecture* project like I did. I hope that my work will inspire others to implement their own PC-DMX interfaces and share their experiences.

For questions and remarks, the author can be contacted via email[53] or GitHub[54].

[53] `florian-edelmann@online.de`
[54] `https://github.com/FloEdelmann`

List of Figures

List of Tables

List of Listings

List of Files

Files listed here are referenced from this document, the full files are included in the *zip* archive handed in together with this Bachelor's Thesis. In some cases, an excerpt is included in the text (see the list of listings). For readers who can only access the printed or PDF version of this document, online versions of the files are provided where possible.

dmx-spi-data.txt
: Example DMX data received with spi-receive.c
referenced on page 33

dmxcreator.pcap
: An example *pcap* file that was captured with Wireshark during executing the procedure described in table 4.1.
referenced on pages 23–24

spi-receive.c
: A simple program that reads in 8192 bytes four times from SPI's *MISO* bus and outputs them in binary format.
referenced on pages 32–33, 37

init-olad.sh
: Init script that is executed automatically after Raspberry Pi has booted. It starts the OLA daemon and initializes GPIO pins.
referenced on pages 18–19

eagle/extension-board.brd
: EAGLE electrical circuit layout of the extension board.
referenced on pages 21–22

eagle/extension-board.sch
: EAGLE electrical circuit schematic of the extension board.
referenced on pages 20–21

ola/configure.ac
: Source file for OLA's build process where I added a line to support the *SPI DMX Plugin*.
`https://github.com/OpenLightingProject/ola/blob/f9e7a7668768eea30ed015e7080a656887a0373f/configure.ac`
referenced on page 39

ola/common/protocol/Ola.proto
: Specifies all plugin ID constants.
`https://github.com/OpenLightingProject/ola/blob/f9e7a7668768eea30ed015e7080a656887a0373f/common/protocol/Ola.proto`
referenced on page 39

ola/debian/ola.udev

udev rules that allow users in the *plugdev* group to communicate with USB devices recognized by OLA.

`https://github.com/OpenLightingProject/ola/blob/8c0aea98458b807ea9d94b241b7e729ef03158c3/debian/ola.udev`

referenced on page 24

ola/olad/DynamicPluginLoader.cpp

Loads and initializes OLA's plugins if they were not disabled at build time.

`https://github.com/OpenLightingProject/ola/blob/f9e7a7668768eea30ed015e7080a656887a0373f/olad/DynamicPluginLoader.cpp`

referenced on page 39

ola/plugins/Makefile.mk

Includes all individual plugin Makefiles.

`https://github.com/OpenLightingProject/ola/blob/f9e7a7668768eea30ed015e7080a656887a0373f/plugins/Makefile.mk`

referenced on page 39

ola/plugins/spidmx/Makefile.mk

Makefile for *SPI DMX Plugin*'s files.

`https://github.com/OpenLightingProject/ola/blob/f9e7a7668768eea30ed015e7080a656887a0373f/plugins/spidmx/Makefile.mk`

referenced on page 39

ola/plugins/spidmx/README.md

Plugin description of the *SPI DMX Plugin*.

`https://github.com/OpenLightingProject/ola/blob/f9e7a7668768eea30ed015e7080a656887a0373f/plugins/spidmx/README.md`

referenced on page 38

ola/plugins/spidmx/SPIDMXDevice.cpp

Represents an SPI device and manages thread, widget and input / output ports.

`https://github.com/OpenLightingProject/ola/blob/f9e7a7668768eea30ed015e7080a656887a0373f/plugins/spidmx/SPIDMXDevice.cpp`

ola/plugins/spidmx/SPIDMXDevice.h

Header file for SPIDMXDevice.cpp.

`https://github.com/OpenLightingProject/ola/blob/f9e7a7668768eea30ed015e7080a656887a0373f/plugins/spidmx/SPIDMXDevice.h`

referenced on page 38

ola/plugins/spidmx/SPIDMXParser.cpp

Parses an SPI buffer into a *DmxBuffer* and notifies a callback when a DMX packet is received completely.

`https://github.com/OpenLightingProject/ola/blob/f9e7a7668768eea30ed015e7080a656887a0373f/plugins/spidmx/SPIDMXParser.cpp`

referenced on pages 36–37

ola/plugins/spidmx/SPIDMXParser.h

Header file for SPIDMXParser.cpp.

https://github.com/OpenLightingProject/ola/blob/f9e7a7668768eea30ed015e7080a656887a0373f/plugins/spidmx/SPIDMXParser.h

referenced on pages 37, 38

ola/plugins/spidmx/SPIDMXPlugin.cpp

Looks for possible SPI devices to instantiate and is managed by the OLA daemon.

https://github.com/OpenLightingProject/ola/blob/f9e7a7668768eea30ed015e7080a656887a0373f/plugins/spidmx/SPIDMXPlugin.cpp

ola/plugins/spidmx/SPIDMXPlugin.h

Header file for SPIDMXPlugin.cpp.

https://github.com/OpenLightingProject/ola/blob/f9e7a7668768eea30ed015e7080a656887a0373f/plugins/spidmx/SPIDMXPlugin.h

referenced on page 38

ola/plugins/spidmx/SPIDMXPort.h

Represents the input / output ports that hook into the thread.

https://github.com/OpenLightingProject/ola/blob/f9e7a7668768eea30ed015e7080a656887a0373f/plugins/spidmx/SPIDMXPort.h

referenced on pages 38–39

ola/plugins/spidmx/SPIDMXThread.cpp

This thread runs while one or more ports are registered. It simultaneously reads / writes SPI data and then calls the parser. This is repeated in an infinite loop.

https://github.com/OpenLightingProject/ola/blob/f9e7a7668768eea30ed015e7080a656887a0373f/plugins/spidmx/SPIDMXThread.cpp

referenced on page 39

ola/plugins/spidmx/SPIDMXThread.h

Header file for SPIDMXThread.cpp.

https://github.com/OpenLightingProject/ola/blob/f9e7a7668768eea30ed015e7080a656887a0373f/plugins/spidmx/SPIDMXThread.h

referenced on page 38

ola/plugins/spidmx/SPIDMXWidget.cpp

A wrapper around the required *spidev* calls.

https://github.com/OpenLightingProject/ola/blob/f9e7a7668768eea30ed015e7080a656887a0373f/plugins/spidmx/SPIDMXWidget.cpp

ola/plugins/spidmx/SPIDMXWidget.h

Header file for SPIDMXWidget.cpp.

https://github.com/OpenLightingProject/ola/blob/f9e7a7668768eea30ed015e7080a656887a0373f/plugins/spidmx/SPIDMXWidget.h

referenced on page 38

ola/plugins/usbdmx/AsyncPluginImpl.cpp

Asynchronous implementation of the *USB DMX Plugin*.

`https://github.com/OpenLightingProject/ola/blob/8c0aea98458b807ea9d94b241b7e729ef03158c3/plugins/usbdmx/AsyncPluginImpl.cpp`

referenced on page 28

ola/plugins/usbdmx/AsyncPluginImpl.h

Header file for AsyncPluginImpl.cpp.

`https://github.com/OpenLightingProject/ola/blob/8c0aea98458b807ea9d94b241b7e729ef03158c3/plugins/usbdmx/AsyncPluginImpl.h`

referenced on page 28

ola/plugins/usbdmx/DMXCreator512Basic.cpp

Implementation of the *DMXCreator 512 Basic* USB protocol.

`https://github.com/OpenLightingProject/ola/blob/8c0aea98458b807ea9d94b241b7e729ef03158c3/plugins/usbdmx/DMXCreator512Basic.cpp`

referenced on pages 26–27

ola/plugins/usbdmx/DMXCreator512Basic.h

Header file for DMXCreator512Basic.cpp.

`https://github.com/OpenLightingProject/ola/blob/8c0aea98458b807ea9d94b241b7e729ef03158c3/plugins/usbdmx/DMXCreator512Basic.h`

ola/plugins/usbdmx/DMXCreator512BasicFactory.cpp

Factory class for *DMXCreator512Basic* widgets.

`https://github.com/OpenLightingProject/ola/blob/8c0aea98458b807ea9d94b241b7e729ef03158c3/plugins/usbdmx/DMXCreator512BasicFactory.cpp`

referenced on page 25

ola/plugins/usbdmx/DMXCreator512BasicFactory.h

Header file for DMXCreator512BasicFactory.cpp.

`https://github.com/OpenLightingProject/ola/blob/8c0aea98458b807ea9d94b241b7e729ef03158c3/plugins/usbdmx/DMXCreator512BasicFactory.h`

ola/plugins/usbdmx/Makefile.mk

Makefile for *USB DMX Plugin*'s files.

`https://github.com/OpenLightingProject/ola/blob/8c0aea98458b807ea9d94b241b7e729ef03158c3/plugins/usbdmx/Makefile.mk`

referenced on page 28

ola/plugins/usbdmx/SyncPluginImpl.cpp

Synchronous implementation of the *USB DMX Plugin*.

`https://github.com/OpenLightingProject/ola/blob/8c0aea98458b807ea9d94b241b7e729ef03158c3/plugins/usbdmx/SyncPluginImpl.cpp`

referenced on page 28

ola/plugins/usbdmx/SyncPluginImpl.h

Header file for SyncPluginImpl.cpp.

https://github.com/OpenLightingProject/ola/blob/8c0aea98458b807ea9d94b241b7e729ef03158c3/plugins/usbdmx/SyncPluginImpl.h

referenced on page 28

ola/plugins/usbdmx/SyncronizedWidgetObserver.h

Transfers widget *add* / *remove* events to another thread.

https://github.com/OpenLightingProject/ola/blob/8c0aea98458b807ea9d94b241b7e729ef03158c3/plugins/usbdmx/SyncronizedWidgetObserver.h

referenced on page 28

ola/plugins/usbdmx/UsbDmxPlugin.cpp

USB DMX Plugin base class.

https://github.com/OpenLightingProject/ola/blob/8c0aea98458b807ea9d94b241b7e729ef03158c3/plugins/usbdmx/UsbDmxPlugin.cpp

referenced on page 28

ola/plugins/usbdmx/UsbDmxPlugin.h

Header file for UsbDmxPlugin.cpp.

https://github.com/OpenLightingProject/ola/blob/8c0aea98458b807ea9d94b241b7e729ef03158c3/plugins/usbdmx/UsbDmxPlugin.h

ola/plugins/usbdmx/WidgetFactory.h

Contains the *WidgetObserver* base class which receives notifications when Widgets are added or removed.

https://github.com/OpenLightingProject/ola/blob/8c0aea98458b807ea9d94b241b7e729ef03158c3/plugins/usbdmx/WidgetFactory.h

referenced on page 28

Bibliography

[Art17] *Art-Net 4 – Specification for the Art-Net 4 Ethernet Communication Protocol.* Protocol Release V1.4, Revision 1.4dd. Artistic Licence, 2017. – http://www.artisticlicence.com/WebSiteMaster/User%20Guides/art-net.pdf

[Ben12] Bennette, Adam: *Recommended Practice for DMX512 – A guide for users and installers.* 2nd Edition. PLASA, 2012. – ISBN 978–0–9557035–2–2

[Bro12] *BCM2835 ARM Peripherals.* Broadcom, 2012. – https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2835/BCM2835-ARM-Peripherals.pdf

[Dem15] Dembowski, Klaus: *Raspberry Pi - Das technische Handbuch: Konfiguration, Hardware, Applikationserstellung.* 2., erweiterte und überarbeitete Auflage. Springer Vieweg, 2015. – ISBN 978–3–658–08710–4

[DMX13] *Nodle U1 Bau- und Bedienungsanleitung.* Rev. 1.0. DMXControl Projects e.V., 2013. – http://www.dmxcontrol-projects.org/component/phocadownload/category/1-verein.html?download=3:nodle-u1

[EST13] *ANSI E1.11 – DMX512-A Asynchronous Serial Digital Data Transmission Standard for Controlling Lighting Equipment and Accessories.* Entertainment Services and Technology Association, 2008 (R2013). – http://tsp.esta.org/tsp/documents/docs/ANSI-ESTA_E1-11_2008R2013.pdf

[EST16] *ANSI E1.31 — Lightweight streaming protocol for transport of DMX512 using ACN.* Entertainment Services and Technology Association, 2016. – http://tsp.esta.org/tsp/documents/docs/E1-31-2016.pdf

[Hes15] Hesmond, Michael: Build your own DMX tester: With Open Lighting Architecture and Raspberry Pi. In: *Lighting&Sound America* Summer 2015 (2015). http://www.lightingandsoundamerica.com/mailing/PLASAProtocol/PSummer2015_BuildYourOwnDMXTester.pdf

[OLP] Open Lighting Project: *Using OLA.* https://wiki.openlighting.org/index.php/Using_OLA, fetched October 25th, 2017

[RP94] Reynolds, J. ; Postel, J.: Assigned Numbers / RFC Editor. Version: October 1994. https://www.ietf.org/rfc/rfc1700.txt. RFC Editor, October 1994 (1700). – RFC. – ISSN 2070–1721

[Sho15] *DMX Merge Manual.* Version 2. Showtec, 2015. – http://www.highlite.nl/silver.download/Documents%40extern%40Manuals/50359_MANUAL_GB_V2.pdf

[TI15] *SNx5176B Differential Bus Transceivers.* Rev. F. TI, 2015. – http://www.ti.com/lit/ds/symlink/sn75176b.pdf

[TII08] *Interface Circuits for TIA/EIA-485 (RS-485).* Texas Instruments Inc., 2008. – http://www.ti.com/lit/an/slla036d/slla036d.pdf

[USI] USITT: *DMX512 FAQ.* http://old.usitt.org/DMX512FAQ.aspx, fetched June 15th, 2017

[VXC11] *DMXCreator Manual.* Version 2.2. VXCO Lighting Systems, 2011. – http://www.dmx512.ch/download/dmxcreator_manual_e_22.pdf