Kim Roininen

Design and Implementation of Fibreoptics Lighting

Metropolia University of Applied Sciences

Bachelor of Engineering

Electrical and Automation Engineering

Bachelor's Thesis

25.05.2023

Abstract

Author(s):	Kim Roininen
Title:	Design and implementation of fibreoptics lighting
Number of Pages:	23 pages + 1 appendix
Date:	25 May 2023
Degree:	Bachelor of Engineering
Degree Programme:	Electrical and Automation Engineering
Specialisation option:	Electronics
Instructor(s):	Heikki Valmu, Principal Lecturer

This bachelor's thesis describes the development of a housing designed for fibre optic lighting, as well as the programming of a MCU controlled light source and familiarization with light transmission in the fibres. The personal interest in light transmission over longer distances laid the groundwork for the design and implementation work.

The aim of the project was to design and print a 3D piece, fit it with an LED light source and fibre optic ends, and program the MCU to provide lighting modes with a single pushbutton interface. The design process began with acquiring a thorough understanding of the basics of fibre optics.

Due to limited skills in designing 3D objects and undersized fibre optic spools, the final implementation was a demonstration piece with short fibres. The implementation of a single pushbutton interface and programmed lighting modes worked perfectly. The project provided a sturdy foundation for continuing to practice 3D modelling and creatively implementing optical fibre lighting in the future.

Keywords: microcontroller, programming, fibre optics, Tinkercad, lighting, Arduino

Tiivistelmä

Tekijä(t):	Kim Roininen
Otsikko:	Valokuituvalaistuksen suunnittelu ja toteutus
Sivumäärä:	23 sivua + 1 liitettä
Aika:	25.05.2023
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Sähkö- ja automaatiotekniikka
Suuntautumisvaihtoehto:	Elektroniikka
Ohjaaja(t):	Yliopettaja Heikki Valmu

Insinöörityö kuvaa valokuituvalaistukseen suunnatun kotelon sekä mikrokontrollerioh-jatun valonlähteen ohjelmoinnin kehitystä ja perehtyy valon siirtoon kuidussa. Kasvanut kiinnostus valon siirtoon pidemmillä matkoilla loi pohjan kuvatulle suunnittelu- ja toteutustyölle.

Työn tavoitteena oli suunnitella ja tulostaa 3D-kappale, sovittaa siihen LED-valonlähde sekä valokuitujen päät ja ohjelmoida mikrokontrolleriin yhdellä kytkimellä käytettävät valaistustilat. Suunnittelu aloitettiin perehtymällä valokuidun perusteisiin perusteellisesti.

Alkeellisten 3D-kappaleiden suunnittelutaitojen ja alakanttiin mitoitettujen valokuitukelojen tilaamisen vuoksi lopullisesta toteutuksesta tuli lyhyillä kuiduilla varustettu demonstraatiokappale. Yhden kytkimen toteutus sekä ohjelmoidut valaistustilat toimivat mainiosti. Työ loi vahvan pohjan jatkaa sekä 3D-mallinnuksen harjoittelua että valaistuksessa valokuidun luovaa käyttöä.

Avainsanat: mikrokontrolleri, ohjelmointi, valokuitu, Tinkercad, valaistus, Arduino

Table of Contents

Abbreviations

5050	Metric code standard way of expressing the size of a 5.0mm x 5.0mm SMD.
DC	Direct current, a unidirectional electrical current.
HCS/PCS	Hard/Plastic Clad Silica, small glass core fibre with a thin plastic cladding
IoT	Internet of Things, sensors embed in physical objects that use networks to transfer data to and from computing systems.
LED	Light emitting diode. When current flows through this semiconductor it emits light.
MCU	Microcontroller Unit.
PCB	Printed Circuit board, a sheet of fibreglass with a thin layer of copper. Widely used in electronics for signalling and component placement.
PMMA	Polymethyl methacrylate, an acrylic derivative.
POF	Plastic Optical Fibre, larger core optical fibre
RGB	A colour model in which is the red, green, and blue colours are added together to form other colours on a digital screen.
SMD	Surface-mounted device

1 Introduction

This bachelor's thesis project is inspired by the creative ways of designing and implementing visually pleasant LED lighting with optical fibre in less predictable places and materials, such as automotive interior decoration, clothing, and other accessories. Due to the refractive properties, light weight, flexibility, and fire safe characteristics of the optical fibre, it is possible to have a portable LED light source with either a small rechargeable or other conventional power source.

The main goal for this project was to find out if a LED source of light and a proper way to transfer the light into the optical fibre were possible to be designed and implemented on a rather restricted budget and tools. This main goal divides the project into three focus points which are light source control, a 3D designed object to encase the light source with the optical fibres, and the actual implementation to see if the proof-of-concept works, and then observe the results to see if there are points of improvement in the design. In the process of designing the concept it was good to find out the basics of the optical fibre and the RGB (red, green, and blue) LED (light emitting diode) technology.

This project provides a perspective for a more niche, rather creative, design experiment where the kick off for the whole idea came from looking at different lighting elements in odd places. Since the result was kept on a proof-of-concept level, the mechanical installation of the lighting ends of the optical fibre was not thought of. Originally the final implementation was supposed to be installed in a headliner of a car, but that turned out to be unrealistic due to the time constrains and the extremely limited workspace of a single bedroom housing.

2 Theory and Basics

2.1 Microcontroller Unit (MCU)

Microcontrollers are generally small and simplified low power single printed circuit board (PCB) computers that are designed to run basic timed loop functions repeatedly defined by the user, also known as an embedded application. There are mainly three distinct types of microcontrollers by multiple manufacturers currently in the market which are 8-, 16- and 32-bit variants, differentiated by their bus widths which directly affects their respective mathematical precision. An 8-bit MCU is required to use severely more resources, as in instructions and bus accesses, to perform equal calculations as its 16-bit and 32-bit counterparts. A properly configured and programmed MCU can perform in a system as an efficient layer of handling and transferring measured input signals from external sensors and trigger other inputs or outputs during set events. [1.]

The most common applications for a MCU include IoT (internet of things) devices, industrial control systems, automotive and vehicle control systems, laboratory equipment and medical systems, consumer electronics, robotics, automation systems and domestic appliances. As a part of a certain system the MCU can be configured to monitor, sense, and respond to various inputs, either digital or analog, that it detects from its connected environment. [1.]

2.2 RGB LED

There are two possible models, additive and subtractive, which can be used to modify colours visible to human vision. In subtractive model the principle is to subtract the visible wavelengths much like when mixing watercolours, the goal is to subtract the number of visible wavelengths to be reflected as seen in Figure 1. [2.]

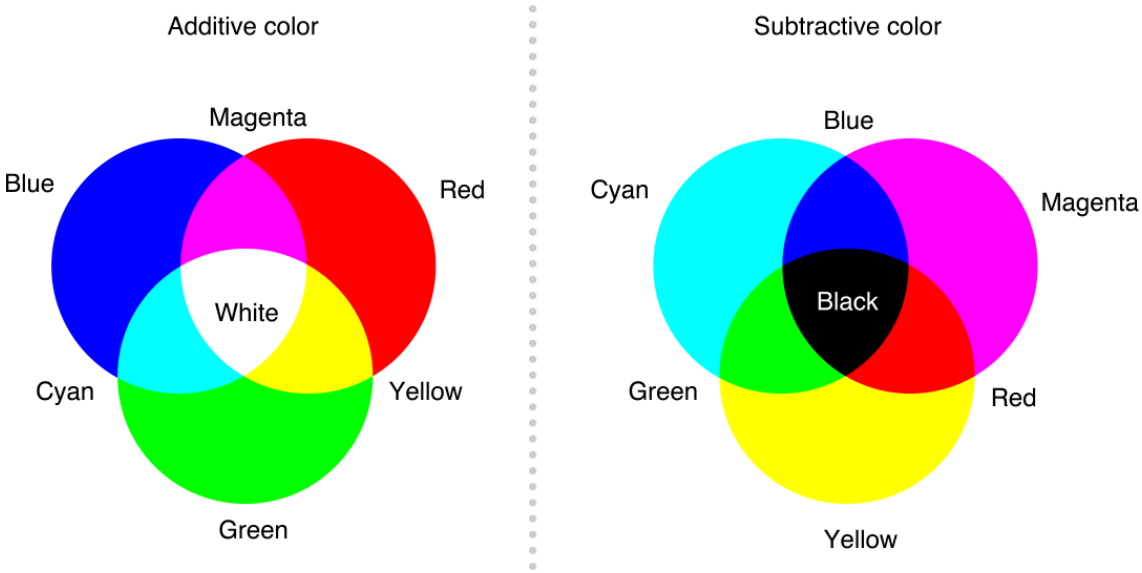

Figure 1. Additive and subtractive colour modes. On the left is visualised how white colour is constructed through 3 LEDs set to their maximum brightness. [2.]

The RGB light emitting diodes work through the additive mode where different wavelengths of visible light are layered upon each other ultimately achieving white light through maximum brightness with all three.

> Red and amber LEDs use a material system made of aluminium indium gallium phosphide (AlInGaP) whereas Blue/Green/Cyan use an indium gallium nitride (InGaN) system. This is how RGB came about, where the primary colours are red, green, and blue. [2.]

The method of deciding which visible wavelength to present is simply by adjusting the brightness of the separate LEDs. LEDs are naturally dimmable and thus they can produce 256 types of different hues of their respective colour which equals to 16 777 216 different possible hues with 3 LEDs. [2.]

2.3 Optical Fibre

Optical fibre is a material that is used to transmit and receive data in the form of light. Fibre works as a guide for the light to pass from one point to another. The fibre consists of three parts which are core, cladding and buffer. As seen in figure 2, core is the guide in which the light travels through to the receiving end. Cladding surrounds the core, and its purpose is to keep the light traveling

through the core inside the core with a phenomenon called total internal reflection. [3.]

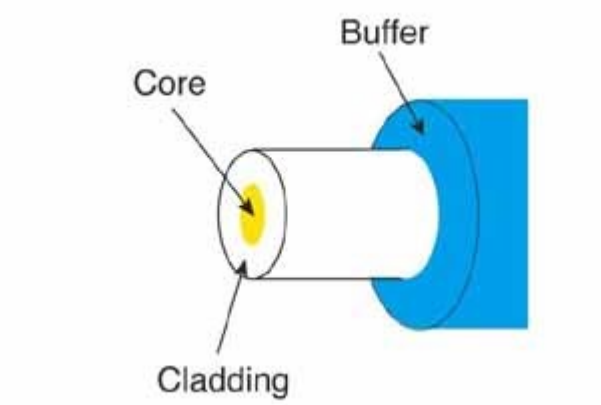

Figure 2. The main three optical fibre components visualized. [3.]

These two components in a fibre are generally made of pure glass, but some are also plastic or a mix of both materials. Buffer is the most outer layer which is there to physically protect the fibre from damage and possible moisture. Plastic optical fibre (POF) is generally manufactured with a larger core and is used for lower speed networks and shorter distances. Hard clad silica (HCS) and plastic clad silica (PCS) fibres with smaller cores are for higher speeds and longer distances. [3.]

The two main advantages of using an optical fibre to transmit light and data are the fact that it is not affected by surrounding electromagnetic fields or nearby electronic devices, and by transferring only low powered light there are no heat concerns for the optical fibre, excluding the surrounding environment of the place of installation [4, 3]. Optical waveguides are manufactured from various materials including polymers, such as PMMA (Polymethyl methacrylate), glass, and silicon. PMMA is considered an ideal material to manufacture cores for optical fibres due to its low refractive index [4,14].

Acrylic plastics, specifically PMMA, are kept in high regard for their excellent optical characteristics. They are also noted for good discolouration resistance and a high light transmitting percentage, up to 92 percent. Parts moulded from

nearly pure acrylic powders present an almost crystal-clear look and near optical perfection. [5, 8.]

2.3.1 Total Internal Reflection

The core's purpose is to transmit light, its index of refraction, a parameter in optical measurement expressing the speed of light in the material, is designed to be high. The cladding has a lower refraction where the guided light stays within if kept below a certain angle. A steeper angle for the light would be only partially reflected into the core and the rest would disperse into the cladding. [6.]

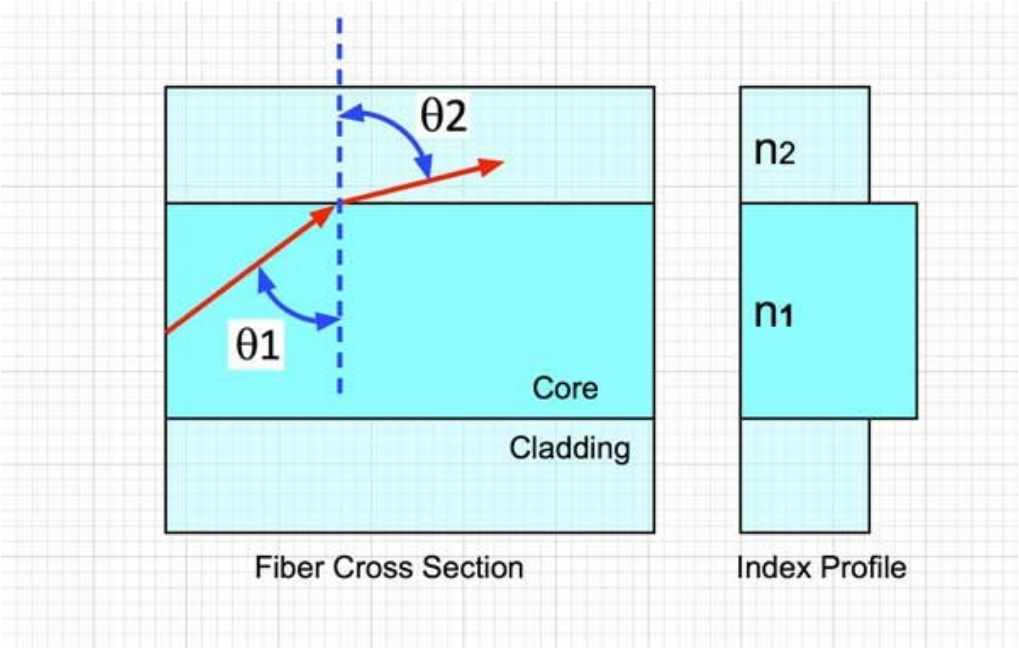

Figure 3. A cross section of a fibre as a mathematical example for the total internal reflection. On the left side the red arrow is a ray of light entering the core in angle θ_1 and dispersing into the cladding in angle θ_2. The right side visualizes the refraction properties of both the core and the cladding, where it is shown that the core has a higher refractive index. [6.]

As above in figure 3, the core has a higher refractive index than the cladding. In this example the index values are n_1 with ~1.46 and n_2 with ~1.45. To calculate the highest angle for the incoming light with the example values, the equation is as following:

$$\frac{\sin\theta_2}{\sin\theta_1} = \frac{n_1}{n_2} = \frac{v_2}{v_1} \quad (1)$$

$$n_2 \sin\theta_2 = n_1 \sin\theta_1 \quad (2)$$

$$90° - 83.2° = 6.8° \quad (3)$$

This proves that the highest incoming angle for the light to enter the core is 6.8 degrees in relation to the cladding to achieve the total internal reflection. This result is also used to define the numerical aperture of the fibre in question and the equation is a manipulation of the critical angle calculation. [6.]

$$\sin\theta_{MAX} = NA = \sqrt{n_{core}^2 - n_{clad}^2} \quad (4)$$

$$NA = \sqrt{n_{core}^2 - n_{clad}^2} = 0.17 \quad (5)$$

The total cone of acceptance for the fibre in the example is 17 degrees.

3 Materials

Most of the materials gathered for this project's implementation are deliberately chosen to be easily reusable in future projects if necessary:

- The Arduino® Nano RP2040 Connect MCU.
- Adafruit NeoPixel LED stick.
- Two 100m spools of 1mm^2 PMMA optical fibre cable without the protective buffer.
- 3D printed jig for the NeoPixel and the optical fibre strands.
- Piece of 2mm thick polycarbonate for the jig.

Tools, equipment, and other commonly used materials include wires, heat shrink tube, soldering equipment, an adjustable direct current (DC) power supply, and a multimeter.

3.1 Arduino Based Microcontroller

The microcontroller chosen for this project is the Arduino® Nano RP2040 Connect, as seen pictured from the top side in Figure 4. It is built upon the 32-bit Arm® Cortex-M architecture with a wide selection of built in features such as the U-blox® Nina W102 Bluetooth® and Wi-Fi module, an accelerometer, a gyroscope, and a microphone. [7, 1.]

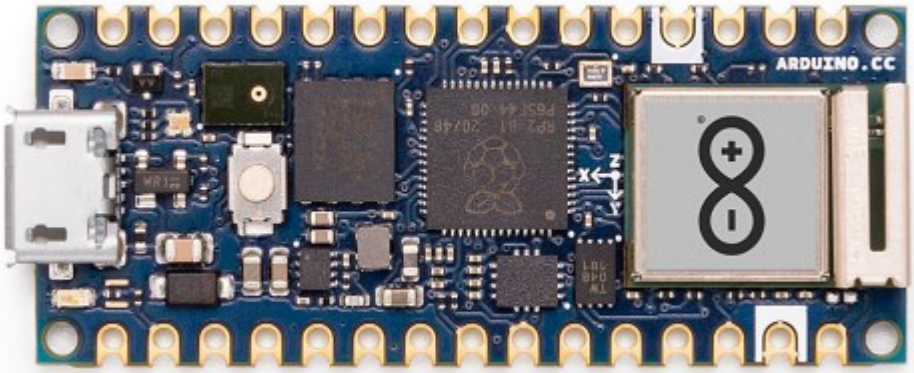

Figure 4. Arduino® Nano RP2040 Connect pictured from above. Due to the size constraints, there are no screw terminals for connectors. [7, 1.]

By default, the MCU is powered by a 3.3V DC source which is regulated from the USB 5V input with an onboard step-down buck regulator [7, 12]. To have the 5V input in use at the VUSB port, it is required to solder a short, two pads marked with a silkscreen text, on the bottom side of the PCB [8]. This streamlines the wiring since the LED stick can be connected into the MCUs 5V output instead of an external DC power supply.

3.1.1 Arduino IDE Software

The Arduino® Nano RP2040 is programmed using either the local Arduino® IDE software or the Arduino® IoT Cloud service. The advantage of using the cloud service is that it automatically saves any applications created and thus

also provides an automatic backup. The programming is simple through the previously mentioned Arduino® environments since they provide official libraries for the onboard components, verified 3rd party libraries for add-on boards, and community libraries from user projects which are published as open source. [9.]

3.2 LED Light Source

Adafruit, founded in 2005, is a New York based PCB design, manufacturing, software, and online electronics learning service company [10]. The NeoPixel product line-up is wonderful hobbyist material and a great learning aid providing nearly instant feedback through the LEDs. Adafruit provides a library to drive the NeoPixel on compatible Arduino® platforms. The NeoPixel 8 Stick is an add-on board for the Arduino environment containing eight SK6812 SMD LEDs installed on a 51.10mm*10.22mm PCB and it is possible to chain these in series to have multiple sticks controlled by a single input from the MCU. [11.]

3.2.1 SK6812

The SK6812 LED is a complete set containing a control circuit and three separate digitally controlled LEDs, titled pixels, packaged into a standard 5mm^2 (5050) surface-mounted device (SMD) casing. The integration of the control circuit simplifies the board assembly by reducing the need of external components. The SK6812 can be chained in series to control multiple units with one digital pin, as seen in figure 5 [12, 1.]

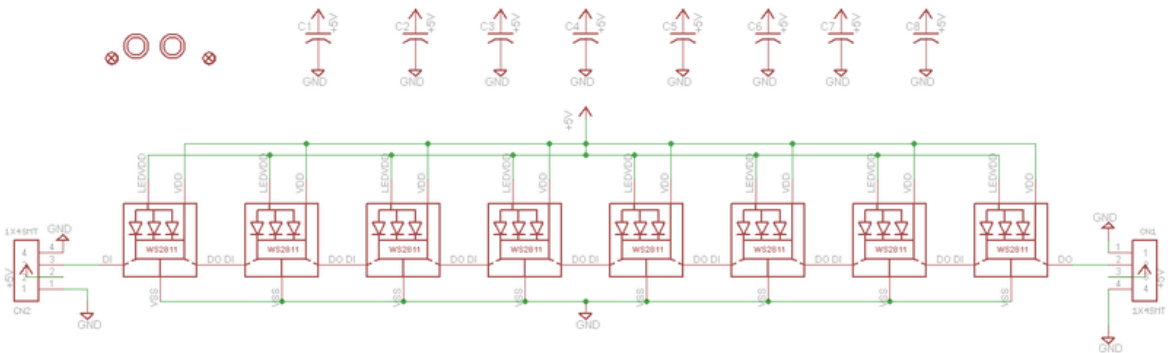

Figure 5. Schematic for the NeoPixel 8 Stick. To chain and control two or more of the LEDs, connect the DOUT to the next LEDs DIN in series and provide each component a supply voltage and a ground. [11.]

In the SK6812 datasheet [12] a distance longer than 10 meters is stated for the control signal transmission which means that more than 8 LEDs in series is possible to be controlled. The power polarity protection is also implemented as a safety precaution in case there is a mix-up with the wiring. [12, 2.]

3.3 PMMA Optical Fibre

Optical fibre purchased for this project are two unbuffered 1mm^2 thick 100-meter PMMA spools of a generic unbranded product. Since the travel distance of the light measured in this project is only meters and no actual data is transmitted to a receiver, the quality of the optical fibre is not important, only the integrity to a point where it still can transfer the light input. The optical fibres were cut to size and then attached to the 3D printed jig and pressed against a piece of polycarbonate above the NeoPixel 8 Stick.

3.4 3D Design Software and 3D Printer

3.4.1 The Software

Autodesk Tinkercad was used in designing the three-part jig for the optical fibres and the LED stick. Tinkercad is a free web-based 3D design software with an easy learning curve to start designing 3D objects. [13.]

Working on models in Tinkercad is simple. As an example, choose a starting shape and place it on to the Workplane (the blue grid seen in Figure 8) and start to modify it with various possibilities. If there is a need for a cut out or a hole in a shape: choose a shape, change it from a shape to a hole, modify it to your liking, place it to the desired position, and merge the two shapes. The files created in Tinkercad can be exported in STL form which is compatible with the Elegoo Mars 3 3D resin printer.

3.4.2 The Printer

Elegoo Mars 3 is a 3D resin printer that uses UV light to cure the printed object. The printer has a moderate 143.43mm * 89.6mm * 175mm build volume and is designed for hobby printing at home. The vapours from the resin, before curing, are possibly harmful for your health and the resin should be handled only while wearing a proper mask or in a well-ventilated space and using protective gloves. [14.]

4 Programming the MCU

4.1 Programming

This subchapter sifts through the code, Appendix 1, of this project. The libraries included in this project's code are the "Adafruit_NeoPixel" and the "Bounce2" libraries. The NeoPixel library contains all the means to control the LEDs on the stick and the bounce library is a more intricate button control environment for fine tuning the pushbutton input.

4.1.1 Definitions, Setup, and the Main Loop

The pin number 15 is defined for the MCU to control the NeoPixel and the number of LEDs on the NeoPixel is defined for the program. The strip object is initialized with different constants defined by the manufacturer, Adafruit. An enumerated variable "LedMode" is created to house four different LED modes. The default mode for the LEDs to be controlled is set to be "RAINBOW_CYCLE," this is forced in case of any power cycle. The pin for the pushbutton is defined as an integer number and a button object is created for further use. The setup function initializes and turns on all the onboard LEDs on the NeoPixel. The "buttonPin" constant is set to be "1" by default and a debounce interval of 50 milliseconds is set.

The main loop starts by updating the “Bounce” instance through the “button.update.” The “button.fell” if-loop checks whether the pushbutton is pressed for a digital “0” input to cycle through to the next mode. If the button is pressed and a “0” is identified, the “currentLedMode” switches from the current case to the next defined case which has a break after it in the predefined case structure. Each case in the case structure has a respectively named loop function defined for it.

4.1.2 The Rainbow Cycle

When the “rainbowCycle” function is enabled, the LED strip displays a smoothly transitioning rainbow effect as a result. A static variable “lastUpdate” is defined to be “0” and this will store the time for the last update for the rainbow effect. A similar static variable with identical initializing condition “firstPixelHue” is created to store the hue of the first pixel in the strip. The wait time between each update is set to 10 milliseconds. The “if” function checks the time elapsed since the last update and if it is greater than the wait time, the code inside the statement is executed.

The “lastUpdate” variable is updated to the current time using the “millis” function. Call the “strip.rainbow” function to update the LED strip colours with different variables such as the “firstPixelHue,” the hue difference between each adjacent pixel, the saturation of the colours, the brightness value of the colours, and the direction of the hue change. Next the “strip.show” function is called to update the colour data to the LED strip and make the new colours visible. Now the “firstPixelHue” variable is incremented by 256 to create a smooth transition between updates. If the previous variable becomes greater than or equal to 5 times 65536 it is reset to 0, this ensures that the hue value stays within the acceptable range and creates a continuous cycle of colours.

4.1.3 A Breathing RGB

This function creates a breathing effect which gradually changes the brightness of the LEDs while cycling through different hues. Static variables “lastUpdate,”

“hue,” and “brightness” are created and defined to initialize as “0”. A static variable “direction” is created and defined to initialize as “1”. The wait time between each update is defined to be 10 milliseconds and the same “lastUpdate” check is executed as in the first function.

The current colour is calculated based on the current hue and brightness using the “strip.ColorHSV” function by multiplying the static variable “hue” with 256 to define a new colour hue, setting the saturation to 255, and inserting the static variable “brightness”. The colour of each LED in the strip to be set in the calculated colour is done using a “for” loop and the “strip.setPixelColour” function and the “strip.show” function is called to send the updated colour data to the LED strip.

The “brightness” variable is updated based on the current direction. If the brightness reaches its maximum value, it is set to “255” and then the direction is set from “1” to “-1” which means the brightness will start to decrease. Once the brightness reaches its minimum value of “0” it is set to “0” and the direction is set back to “1”, this is where the “hue” is updated by adding 50 to the value. If the “hue” value becomes grater than or equal to “256” it is reset to “0”

4.1.4 A Breathing RGB with a Randomized Element

This function creates a breathing effect with random hues for each LED in the NeoPixel stick. The LEDs gradually change their brightness while their hues change randomly during the minimum brightness value. The same “lastUpdate” static variable is once again used and set to initialize as “0”. Static arrays, each sized with the “NUM_LEDS” value, for “hue,” “direction,” and “brightness” are created. These arrays will store the current hue, direction of brightness change, and the brightness level for each LED in the strip. The direction is set to initialize as “1” and a wait time between each update is set to 10 milliseconds.

The same “lastUpdate” check is executed. A for-loop is used to iterate through each individual LED in order. The current colour for the LED is calculated based on its hue and brightness using the “strip.ColorHSV” function. The parameter

“hue[i]” is multiplied by 256 which is the hue of the colour of the LED at index “i”. The saturation is set to “255” and the “brightness[i]” is the brightness of the colour for the LED at index “i”. The colour of the LED is set to be the calculated colour at the index “i” using the “strip.setPixelColor” function and the brightness value is updated for the LED at index “i” based on its current direction. The brightness direction calculation is identical to the previous one. Lastly the “strip.show” function is called to send the updated colour data to the LED strip.

4.1.5 The Random Colour Fade

This function creates a colour fade effect for each LED in the strip individually with a random hue. The LEDs change their brightness between minimum and maximum values, and when the brightness reaches the minimum limit, the colour of the LED changes randomly. Five different “NUM_LEDS” sized static arrays are created to store the current brightness values, the amount by which the brightness of each LED changes, the minimum and maximum brightness of each LED, and the current hue value for each LED in the strip.

A for loop is used to iterate through the LEDs. If the brightness of the LED at index “i” is equal to its maximum brightness, negate the fade amount for that LED. this will cause the brightness to decrease in the next iteration. If the brightness of the LED at index “i” is equal to its minimum brightness, make the fade amount positive and change the hue for that LED to a random value between “0” and “256”. This will cause the brightness to increase in the next iteration and change the colour. Update the brightness level for the LED at index “i” by adding its corresponding “fadeAmount.” use the “constrain()” function to ensure that the brightness value stays within the limits of “minBrightness[i]” and “maxBrightness[i].” Calculate the colour for the LED at index “i” using the “strip.ColorHSV” function with identical parameters as before and set the colour of the LED at index “i” to the calculated colour using the “strip.setPixelColor” function. The “strip.show” function is called to send the updated colour data to the LED strip and a delay of 69 milliseconds is set before the next iteration.

5 Final assembly and testing

5.1 Wiring

The wiring of this project is kept simple by chaining the LED control signal, this makes its possible to use only one digital pin on the MCU for the LEDs and another for the pushbutton.

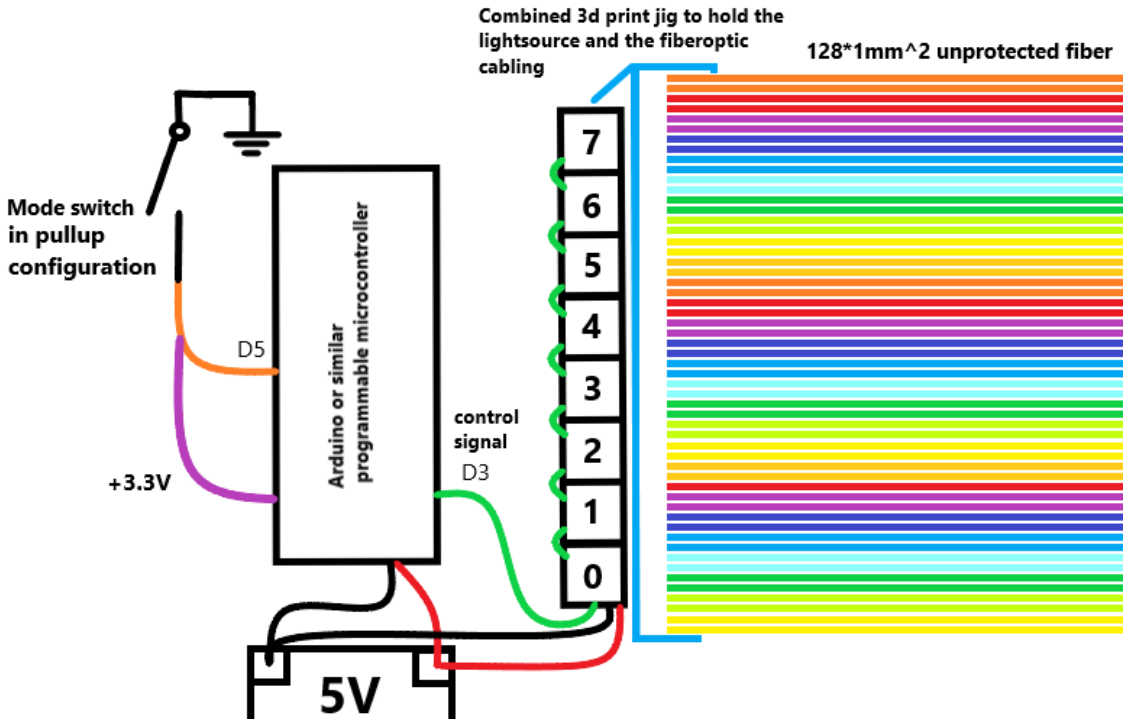

Figure 6. A block diagram of the project with the designed wiring and the 3D printed jig visualized.

On the top left in the figure 6, one pin of the pushbutton is soldered to the pin D5, and the other pin is soldered to ground with a jumper wire. A 3.3V DC signal is brought from a dedicated pin on the MCU to the pin D5 to force it to be “1” by default. Now when the button is pressed the pin D5 is grounded dropping the voltage to 0 and thus dropping the digital input signal to “0”.

5.2 Testing the Program

In the testing phase multiple pushbuttons were used and the most dependable one turned out to be a button disassembled from a keyboard (as seen in figure 7). The case structure works perfectly and almost immediately switches from one case to the next when the button is pressed and the “0” input is detected in pin D5. All four of the functions loop indefinitely as intended, which is one of the goals of this project.

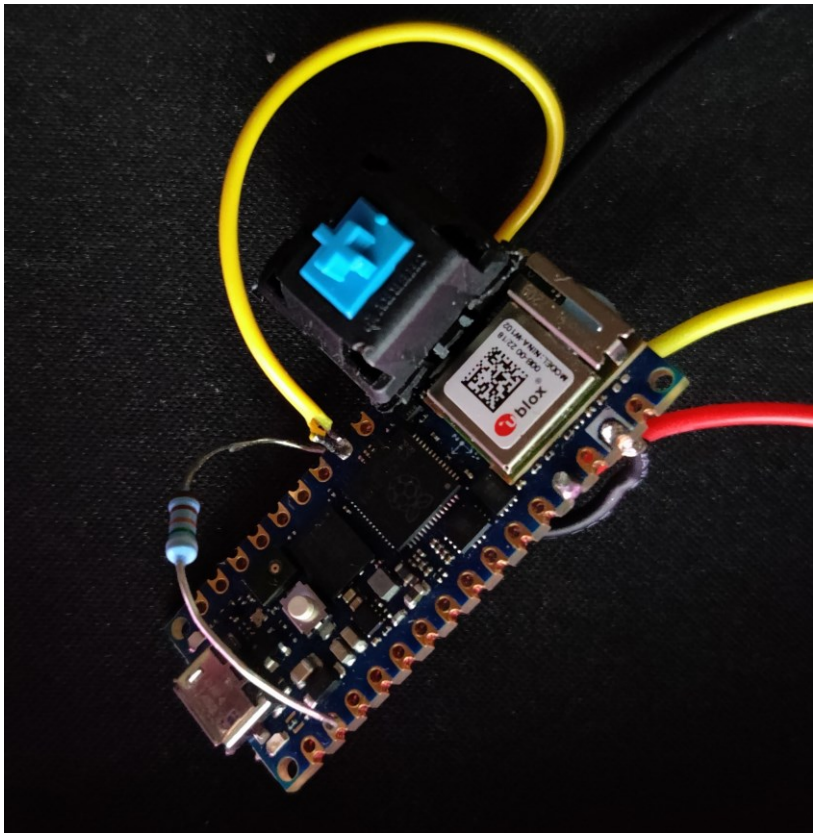

Figure 7. A pullup resistor providing the 3.3V for the digital input and the soldered keyboard switch on the MCU.

5.3 3D Prints

5.3.1 Designing the 3D Prints for the Project

The yellow and light blue objects consist of two identical halves that are meant to be joint together. This design choice was made to increase the options in the

assembly stage to see which is easier when handling the ~128 pieces of optical fibre strands. The green object on the right side in figure 8 has 4 small protrusions around the LED stick recess to target and hold the light blue object on top.

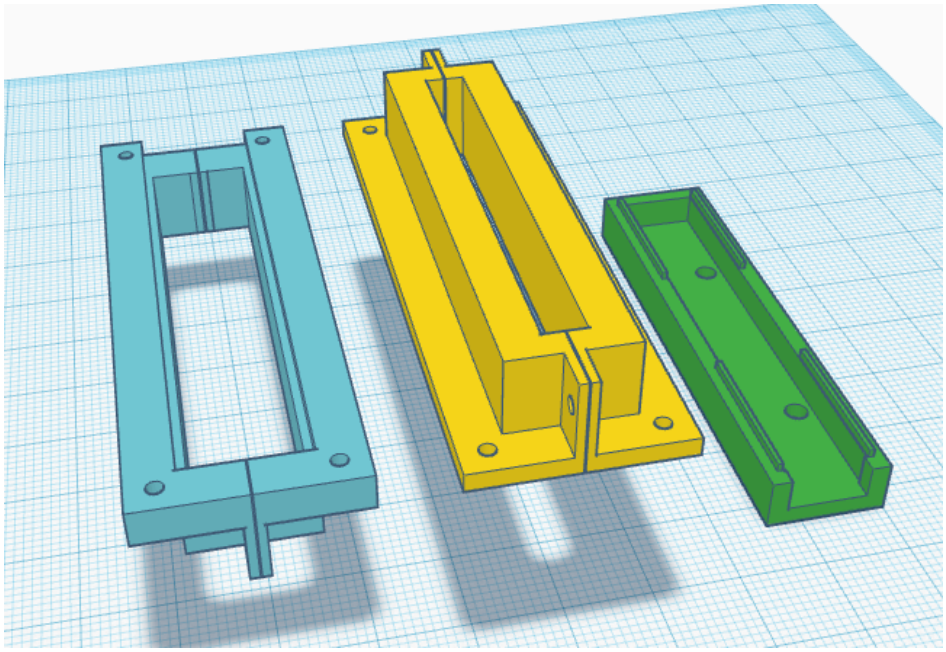

Figure 8. All three parts of the jig intended for the assembly. Yellow is designed to hold the optical fibre strands; light blue has a cut out for the polycarbonate. Both the yellow and the light blue objects consist of mirrored counterparts. Green is for attaching the NeoPixel stick through its mounting holes.

Later the blue object was converted to a unified solid object and the small notches at the edges of the gap in the green object are removed to simplify the printing process and to improve the combability.

5.3.2 Inspecting the 3D Printed Objects

Slight visible curling is present in some of the prints due to the curing process and the shrinking of the material used. This could possibly be compensated by the designer in the design phase but for now it seems that the project can still be assembled without any greater difficulty.

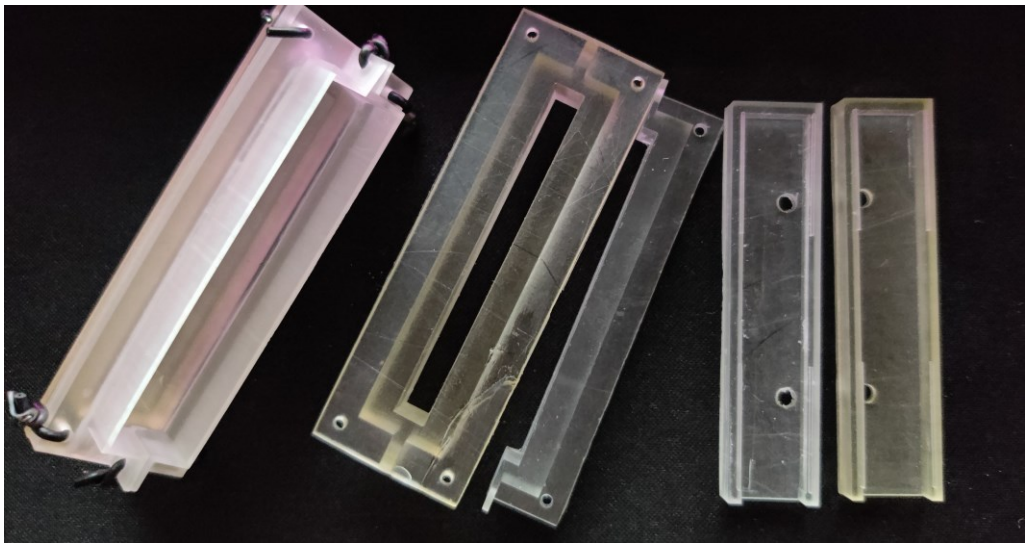

Figure 9. On the left is the assembled jig without the optical fibres installed. Middle objects are for size comparison for the length error in the first batch. Right side objects are visualizing the small improvements in the design from the first batch to the second.

When exporting the objects one by one something was either stretched or shrunken, since the first batch of the blue and yellow objects were not the same length unlike in the design, as seen above in figure 9. Two of the objects were slightly too thin and thus fragile when removing from the 3D printing surface, which caused some cracks.

5.4 Assembly

The 3D printed objects are clamped together by threading some copper single-wire through the holes and winding it tight (see figure 9) while the polycarbonate is secured with some instant glue. The NeoPixel is also fastened into its 3D printed housing by threading wire through the holes and winding it to the appropriate tightness. The holes in the NeoPixel housing had to be slightly filed to have the PCB align properly. Appropriate length multi-stranded copper wire was soldered in between the MCU and the NeoPixel to provide power, ground, and data. The MCU and the NeoPixel stick are powered by a rechargeable 5V power bank.

The insertion of the optical fibres as one turned out to be harder than thought and as a solution the strands were split into two groups. At this stage it also turned out that the fibre guide object's opening is larger than originally designed and it required double the number of fibres to fill. A total of 256 fibre threads being problematic to handle, one extra jig from the second batch was used as an extra guide to thread through the second guide on to the polycarbonate sheet glued in the counterpart. Hot glue and zip ties were used in the product to keep the fibres in place and have the jig stay intact with its additional guide.

5.5 Inspecting the Assembled Proof of Concept

The resin printed jig is transparent to a point and that makes demonstrating the proof of concept difficult since the jig itself attracts attention. Light glowing through the jig itself was not a design choice (as seen in figure 10), the main idea was to have the light shine only from the ends of the strands.

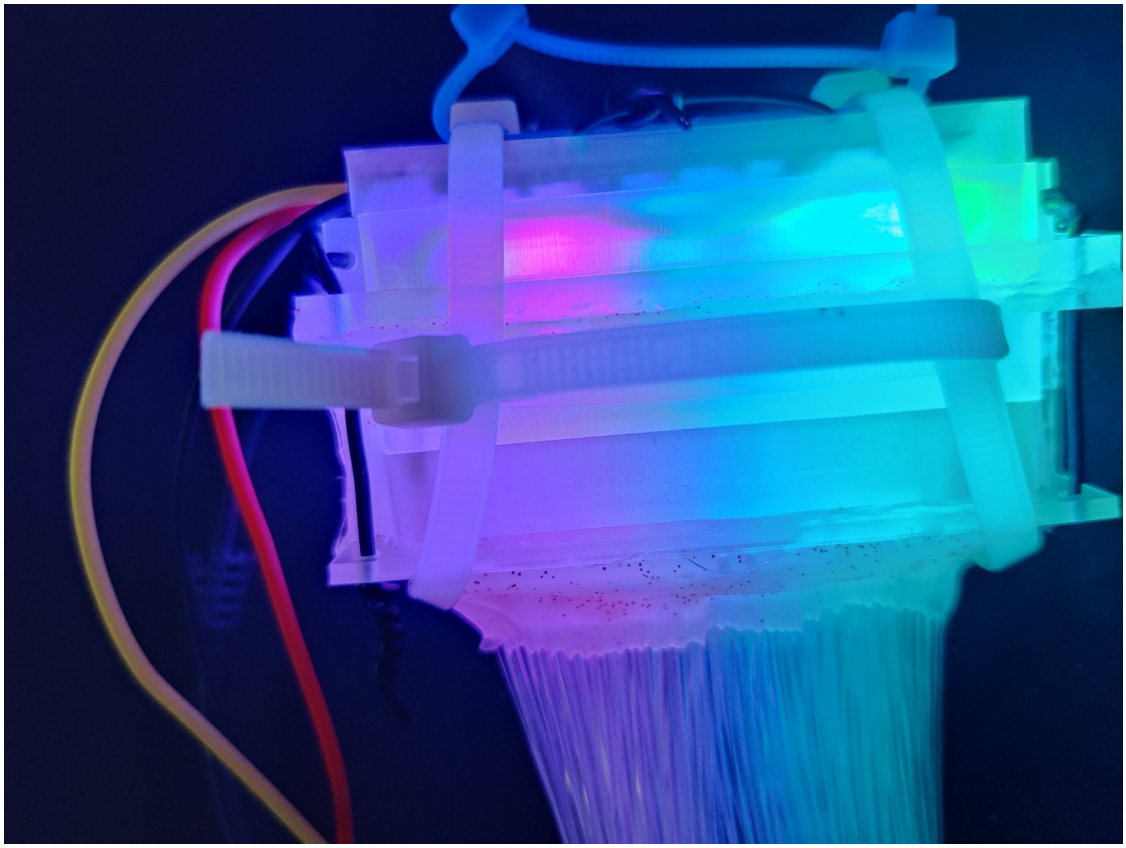

Figure 10. A picture of the transparent jig and the light shining through it.

Hot glue was not an optimal adhesive material but one that was readily available when working at home with a limited supply of materials. It takes time to cure and seems not to be very dependable in gluing larger bundles PMMA strands since the middle of the bundle is kept in place by just friction. The zip ties do add a nice do-it-yourself stamp on the overall impression. The ordered 200 metres of optical fibre may not have been measured true by the online seller and for the sake of completing the concept, intensely shorter strands of fibre had to be used. This had a negative impact on the presentation as seen below in figure 11.

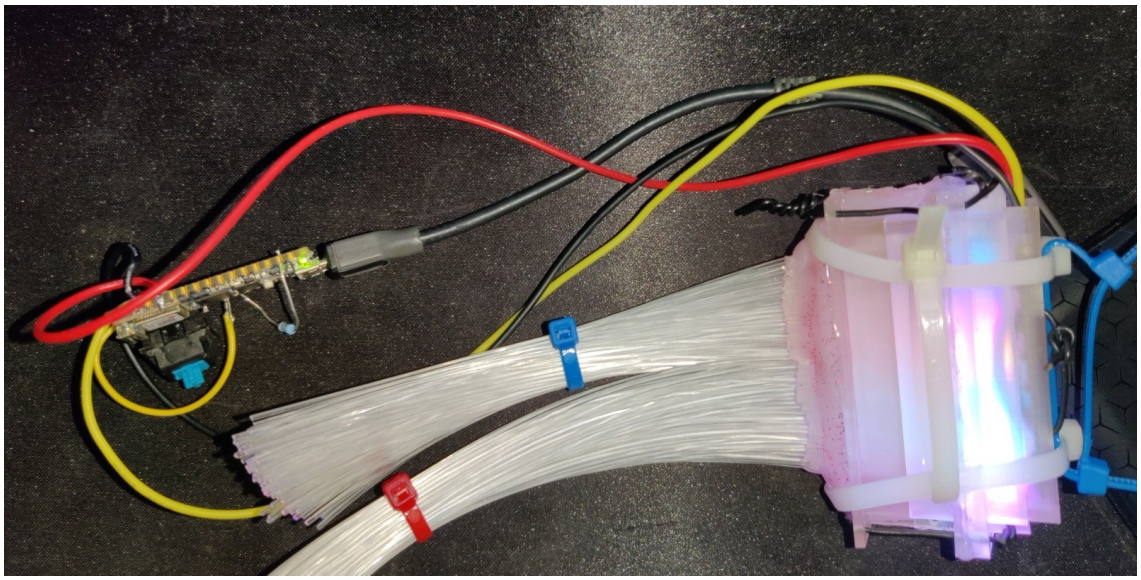

Figure 11. The assembled proof of concept with the two groups of shortened 256 optical fibres glued into the jig.

Having the fibres pressed spread equally against the sheet inside the assembled jig was not thought of thoroughly and not all the fibres receive the same amount of light, this can be observed in a dim environment by comparing the ends of the strands. The disproportion of light between fibres is also affected by the direction of the ends since the cone of light coming out is small and the uneven surface from cutting the strands from the spools.

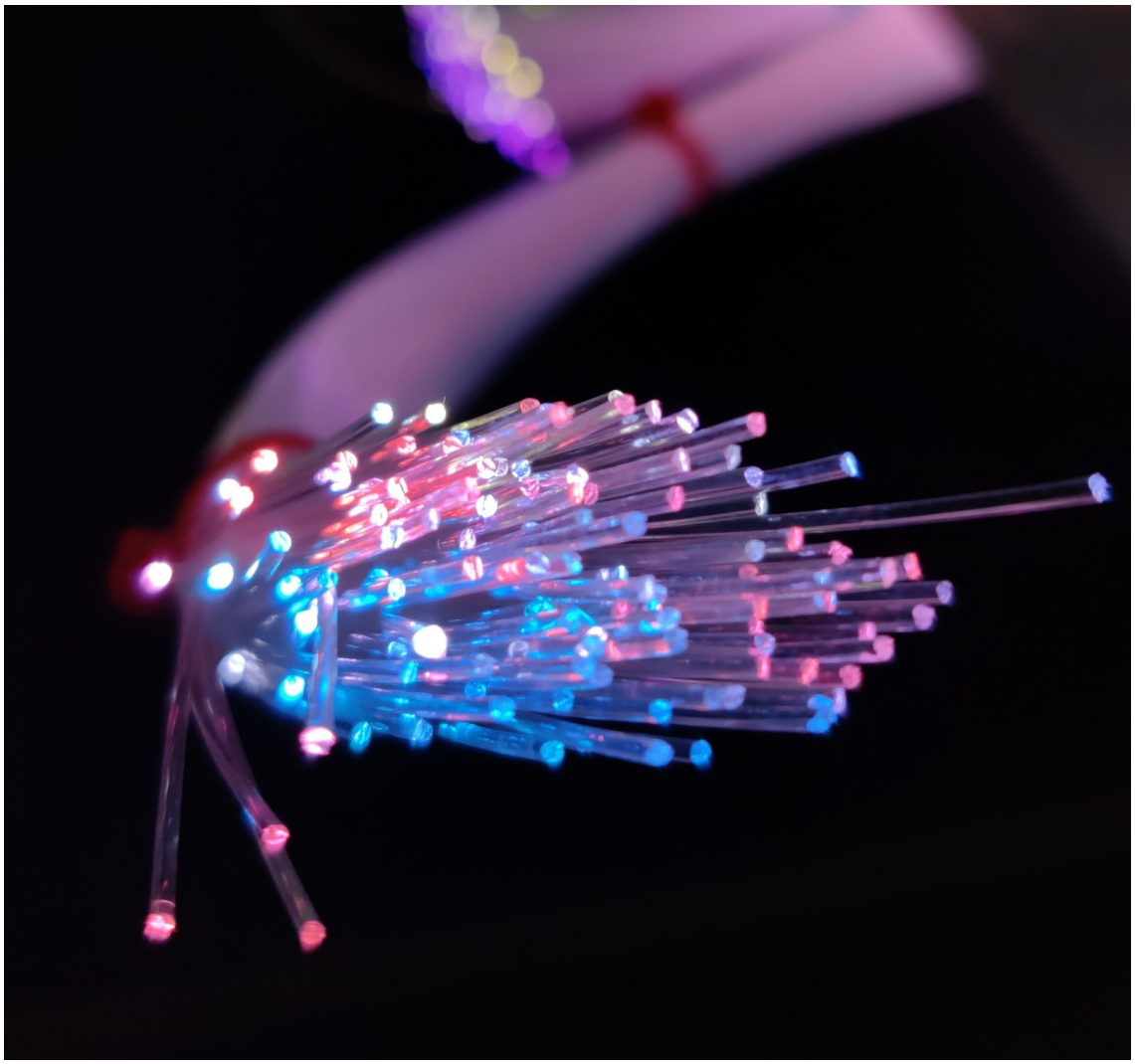

Figure 12. Ends of the fibres from the longer group of fibres.

In an actual installation the pushbutton should be wired further away from the MCU to a more user-friendly location and the jig with all the fibre shafts would be hidden away and only the ends should be visible, such as installing in a suspended ceiling. Even when ignoring most of the issues mentioned in the previous paragraphs the concept works and will most likely be developed further as time goes on. The simple one button design and the generic 5V powered MCU make this project within reach for anyone with basic programming skills and a limited budget.

6 Summary

The result of this project was to explore the use of LED lighting and optical fibre for creative and visually appealing designs. Optical fibre was chosen due to its refractive properties, flexibility, light weight, and fire safety, which allowed for portable LED lighting sources that can be incorporated into a variety of materials and places, including automotive interiors, clothing, and accessories.

The main goal of the project was to determine whether a LED light source could be designed and implemented using optical fibre while working within a restricted budget and limited tools. This goal was achieved through three focus points, which were light source control, 3D designs for encasing the light source with the optical fibres, and the actual implementation to see if the proof-of-concept worked. The project's process involved researching the basics of optical fibre and RGB LED technology.

The project provided a unique perspective on creative design experiments that could be applied to more niche markets. However, due to time constraints and limited workspace, the final installation of the lighting ends of the optical fibre was not considered. Originally, the plan was to install the lighting in the headliner of a car, but this proved to be unrealistic given the constraints. The result of the project served as a proof-of-concept that can be improved upon through further design iterations.

Sources

1 Subbaroybhat M. Everything you need to know about microcontrollers. Internet. Rs-online. 2021. <https://uk.rs-online.com/web/content/discovery/ideas-and-advice/microcontrollers-guide>.Accessed April 10, 2023.

2 Scully T. What is RGB lighting? Top 5 RGB LED strips and lights. Internet. LEDSupply. 2021. <https://www.ledsupply.com/blog/rgb-lighting-guide-to-the-top-5-rgb-led-strips-lights/>.Accessed April 10, 2023.

3 The Fiber Optic Association, Inc., Network Solutions, LLC. The FOA Reference for Fiber Optics - Optical Fiber. Internet. 2019. <https://www.thefoa.org/tech/ref/basic/fiber.html>. Accessed April 11, 2023.

4 Senior JM, Jamro MY. Optical Fiber Communications: Principles and Practice. 3rd ed. Edinburgh Gate Harlow Essex CM20 2JE England: © Pearson Education Limited; 2010.

5 Harper CA, Petrie EM. Plastics materials and processes: A concise encyclopedia. 1st ed. Hoboken, New Jersey: Wiley-Interscience; 2003.

6 The Fiber Optic Association, Inc., Network Solutions, LLC. The FOA reference for fiber optics - total internal reflection in optical fiber. Internet. 2019. <https://www.thefoa.org/tech/ref/basic/total_internal_reflection.html>. Accessed April 11, 2023.

7 Romero S. Arduino® Nano RP2040 Connect. Internet. Arduino.cc. 2023 <https://docs.arduino.cc/static/d9f882b532adf11117eea300bf46f7d6/ABX00053-datasheet.pdf>. Accessed April 10, 2023.

8 Romero S. Nano RP2040 Connect Cheat Sheet. Internet. Arduino.cc. <https://docs.arduino.cc/tutorials/nano-rp2040-connect/rp2040-01-technical-reference>. Accessed April 12, 2023.

9 Getting Started with Arduino IDE 2.0. Internet. Arduino.cc. <https://docs.arduino.cc/software/ide-v2/tutorials/getting-started-ide-v2>. Accessed April 12, 2023.

10 Adafruit Industries. Adafruit Industries, Unique & fun DIY electronics and kits. Internet. <https://www.adafruit.com/about>. Accessed April 12, 2023.

11 Adafruit Industries. NeoPixel Stick - 8 x 5050 RGB LED with Integrated Drivers. Internet. <https://www.adafruit.com/product/1426>. Accessed April 12, 2023.

12 Adafruit Industries. Technical Data Sheet. Internet. <https://cdn-shop.adafruit.com/product-files/1138/SK6812+LED+datasheet+.pdf>. Accessed April 12, 2023.

13 Autodesk Inc. Tinkercad. <https://www.tinkercad.com/3d-design>. Accessed April 12, 2023.

14 ELEGOO Mars 3 ULTRA 4K mono LCD 3D printer. Internet. ELEGOO Official. <https://www.elegoo.com/products/elegoo-mars-3-lcd-3d-printer>. Accessed April 12, 2023.

Appendix 1: The Code

kroininen_light_controller.ino

```
1   //  The neopixel library provided by Adafruit to control the LEDs and much more
2   //  Bounce2 library for more intricate button input control
3   #include <Adafruit_NeoPixel.h>
4   #include <Bounce2.h>
5   //  define NeoPixel control pin
6   //  define number of LEDs on the NeoPixel, if chained more than one stick, just add the number of LEDs
7   #define PIN 15
8   #define NUM_LEDS 8
9   //  Initializing the strip object with required constants, defined by Adafruit.
10  Adafruit_NeoPixel strip(NUM_LEDS, PIN, NEO_GRB + NEO_KHZ800);
11
12  //  Define the different LED modes
13  enum LedMode {
14    RAINBOW_CYCLE,
15    BREATHING_RGB,
16    BREATHING_RGB_RANDOM,
17    RANDOM_COLOR_FADE
18  };
19
20  //  Initialize the current LED mode to rainbow cycle,
21  //  if a power cycle is done, this will be the default mode upon power on.
22  LedMode currentLedMode = RAINBOW_CYCLE;
23
24  //  Set up the pushbutton
25  const int buttonPin = 5;    //  The const int digital input is defined as D5, without the D
26  Bounce button = Bounce();   //  Create button object for further use
27
28  void setup() {
29    strip.begin();            //  initialize the LED stick
30    strip.show();             //  turn off all the onboard LEDs
31
32    button.attach(buttonPin, INPUT_PULLUP);   //  Define the buttonPin constant to be 1 by default
33    button.interval(50);      //  Set debounce interval to 50ms
34  }
35
36  //    The main loop
37  void loop() {
38    //  Update the 'Bounce' instance
39    button.update();
40    //  Check for button press
41    if (button.fell()) {
42      switch (currentLedMode) {         //  'Case' structure, cases listed
43        case RAINBOW_CYCLE:
44          currentLedMode = BREATHING_RGB;
45          break;
46        case BREATHING_RGB:
```

```
      case BREATHING_RGB:
        currentLedMode = BREATHING_RGB_RANDOM;
        break;
      case BREATHING_RGB_RANDOM:
        currentLedMode = RANDOM_COLOR_FADE;
        break;
      case RANDOM_COLOR_FADE:
        currentLedMode = RAINBOW_CYCLE;
        break;
    }
  }

  //  Set the LED mode based on the current LED mode
  //  if '(button.fell())' -> When the button is pressed down and the loop notices,
  //  'currentLedMode' is switched to the next predefined case
  switch (currentLedMode) {
    case RAINBOW_CYCLE:
      rainbowCycle();
      break;
    case BREATHING_RGB:
      breathingRgb();
      break;
    case BREATHING_RGB_RANDOM:
      breathingRgbRandom();
      break;
    case RANDOM_COLOR_FADE:
      randomColorFade();
      break;
  //  Defined a correct loop function for respectively named case
  //  Very straight straightforward to add more cases and functions
  }
}
//  Rainbow cycle mode
//  This function is called repeatedly in the loop() function when the currentLedMode is set to
//  RAINBOW_CYCLE. The LED strip displays a smoothly transitioning rainbow effect as a result.
void rainbowCycle() {     //  Loop function with a practical yet distinctive name
//  Defining a 'static' variable and setting it to initialize into 0
//  This variable will store the time of the last update for the rainbow effect
  static unsigned long lastUpdate = 0;
//  Same type of 'static' variable with identical initializing condition
//  This variable will store the hue of the first pixel in the strip
  static long firstPixelHue = 0;
//  Setting the wait time between each update to something in milliseconds
//  Desired value is now defined as 10 milliseconds
  uint16_t wait = 10;

```

```
//  'If' function to check the time elapsed since the last update is greater than the wait time.
//  If it is, execute the code inside the statement
  if (millis() - lastUpdate > wait) {
    //  If true, update the 'lastUpdate' variable to the current time using 'millis()'
    lastUpdate = millis();
    //  Call the 'strip.rainbow()' function, which updates the colors of the strip to create
    //  the rainbow effect. The parameters passed are:
    //  'firstPixelHue': The hue of the first pixel in the strip.
    //  '3': The hue difference between each adjacent pixel.
    //  '255': The saturation of the colors.
    //  '255': The value (brightness) of the colors.
    //  'true': The direction of the hue change (true for increasing, false for decreasing).
    strip.rainbow(firstPixelHue, 3, 255, 255, true);
    //  Call the 'strip.show()' function, which sends the updated color data
    //  to the LED strip and makes the new colors visible.
    strip.show();
    //  Increment the firstPixelHue variable by 256. This value is added to create
    //  a smooth transition between updates.
    firstPixelHue += 256;
    //  If firstPixelHue becomes greater than or equal to 5 * 65536, reset it to 0. This
    //  condition ensures that the hue value stays within the acceptable range and creates
    //  a continuous cycle of colors.
    if (firstPixelHue >= 5 * 65536) {
      firstPixelHue = 0;
    }
  }
}
//  Breathing RGB mode
//  The 'breathingRgb()' function creates a breathing effect by gradually changing the brightness
//  of the LEDs while cycling through different hues.
void breathingRgb() {
  //  Define a 'static' variable lastUpdate and initialize it to '0'. This variable will store the
  //  time of the last update for the breathing effect.
  static unsigned long lastUpdate = 0;
  //  Define a 'static' variable hue and initialize it to '0'. This variable will store the current
  //  hue of the color used for the breathing effect.
  static uint16_t hue = 0;
  //  Define a 'static' variable direction and initialize it to '1'. This variable will store the
  //  direction of brightness change (1 for increasing, -1 for decreasing).
  static int direction = 1;
  //  Define a 'static' variable brightness and initialize it to '0'. This variable will store the
  //  current brightness level of the LEDs.
  static uint8_t brightness = 0;
  //  Define a variable 'wait' and set it to '10'. This variable determines the time (in milliseconds)
  //  between each update of the breathing effect.
  uint16_t wait = 10;
```

```
138
139   //  Check if the time elapsed since the last update (calculated as 'millis() - lastUpdate') is
140   //  greater than the wait time. If it is, execute the code inside the 'if' statement.
141   if (millis() - lastUpdate > wait) {
142     //  Update the lastUpdate variable to the current time using 'millis()'.
143     lastUpdate = millis();
144     //  Calculate the current color based on the current hue and brightness using the
145     //  'strip.ColorHSV()' function. The parameters passed are:
146     //  'hue * 256': The hue of the color.
147     //  '255': The saturation of the color.
148     //  'brightness': The brightness of the color.
149     uint32_t color = strip.ColorHSV(hue * 256, 255, brightness);
150     //  Set the color of each LED in the strip to the calculated color using a for loop
151     //  and the strip.setPixelColor() function.
152     for (int i = 0; i < strip.numPixels(); i++) {
153       strip.setPixelColor(i, color);
154     }
155     //  Call the strip.show() function, which sends the updated color data to the LED strip
156     //  and makes the new colors visible.
157     strip.show();
158     // Update the brightness variable based on the current direction.
159     brightness += direction;
160     //  If the brightness reaches its maximum value (255), set it to 255 and change
161     //  the direction to -1, which means the brightness will start decreasing.
162     if (brightness >= 255) {
163       brightness = 255;
164       direction = -1;
165       //  If the brightness reaches its minimum value (0), set it to 0, change the direction to 1
166       //  (which means the brightness will start increasing), and update the hue by adding 50
167       //  to it. If the hue becomes greater than or equal to 256, reset it to 0.
168     } else if (brightness <= 0) {
169       brightness = 0;
170       direction = 1;
171       hue += 50;
172       if (hue >= 256) {
173         hue = 0;
174       }
175     }
176   }
177 }
178
179 //  a breathing RGB with a random mode
180 //  The 'breathingRgbRandom()' function creates a breathing effect with random hues for each LED in
181 //  the strip. The LEDs gradually change their brightness while their hues change randomly when
182 //  the brightness reaches its minimum value.
183 void breathingRgbRandom() {
```

```
  //  Define a 'static' variable 'lastUpdate' and initialize it to '0'. This variable will store the time
  //  of the last update for the breathing effect.
  static unsigned long lastUpdate = 0;
  //  Define 'static' arrays hue, direction, and brightness, each with a size of 'NUM_LEDS'. These arrays
  //  will store the current hue, direction of brightness change, and brightness level for each LED
  //  in the strip, respectively.
  static uint16_t hue[NUM_LEDS] = {0};
  static int direction[NUM_LEDS] = {1};
  static uint8_t brightness[NUM_LEDS] = {0};
  //  Define a variable 'wait' and set it to '10'. This variable determines the time (in milliseconds)
  //  between each update of the breathing effect.
  uint16_t wait = 10;

  //  Check if the time elapsed since the last update (calculated as 'millis() - lastUpdate') is greater
  //  than the 'wait' time. If it is, execute the code inside the if statement.
  if (millis() - lastUpdate > wait) {
    //  Update the 'lastUpdate' variable to the current time using 'millis()'.
    lastUpdate = millis();
    //  Iterate through each LED in the strip using a 'for' loop:
    for (int i = 0; i < strip.numPixels(); i++) {
      //  Calculate the current color for the LED based on its hue and brightness using the 'strip.ColorHSV()'
      //  function. The parameters passed are:
      //  'hue[i] * 256': The hue of the color for the LED at index 'i'.
      //  '255': The saturation of the color.
      //  'brightness[i]': The brightness of the color for the LED at index 'i'.
      uint32_t color = strip.ColorHSV(hue[i] * 256, 255, brightness[i]);
      //  Set the color of the LED at index 'i' to the calculated color using the
      //  'strip.setPixelColor()' function.
      strip.setPixelColor(i, color);
      //  Update the brightness value for the LED at index 'i' based on its current direction.
      brightness[i] += direction[i];
      //  If the brightness of the LED at index 'i' reaches its maximum value (255), set it to '255' and change
      //  the direction for the LED at index 'i' to '-1', which means the brightness will start decreasing.
      if (brightness[i] >= 255) {
        brightness[i] = 255;
        direction[i] = -1;
        //  If the brightness of the LED at index 'i' reaches its minimum value (0), set it to '0', change
        //  the direction for the LED at index 'i' to '1' (which means the brightness will start increasing),
        //  and update the hue for the LED at index 'i' by adding a random value between '10' and '60' to it.
        //  If the hue for the LED at index 'i' becomes greater than or equal to '256', reset it to '0'.
      } else if (brightness[i] <= 0) {
        brightness[i] = 0;
        direction[i] = 1;
        hue[i] += random(10, 60);
        if (hue[i] >= 256) {
          hue[i] = 0;
        }
```

```
      }
    }
    //  Call the strip.show() function, which sends the updated color data to the LED strip and
    //  makes the new colors visible.
    strip.show();
  }
}

//  randomColorFade mode
//  The randomColorFade() function creates a color fade effect for each LED in the strip with random
//  hues. The LEDs change their brightness between minimum and maximum values, and when the brightness
//  reaches either limit, the color of the LED changes randomly.
void randomColorFade() {
  //  Define a 'static' array brightness with size 'NUM_LEDS' and initialize all elements to '0'. This
  //  array stores the current brightness level for each LED in the strip.
  static int brightness[NUM_LEDS] = {0, 0, 0, 0, 0, 0, 0, 0};
  //  Define a 'static' array 'fadeAmount' with size 'NUM_LEDS' and initialize it with different fade
  //  amounts. This array stores the amount by which the brightness of each LED changes.
  static int fadeAmount[NUM_LEDS] = {6, 5, 4, 8, 5, 6, 7, 8};
  //  Define a 'static' array 'minBrightness' with size 'NUM_LEDS' and initialize all elements to'0'.
  //  This array stores the minimum brightness level for each LED in the strip.
  static int minBrightness[NUM_LEDS] = {0, 0, 0, 0, 0, 0, 0, 0};
  //  Define a 'static' array 'maxBrightness' with size 'NUM_LEDS' and initialize it with different maximum
  //  brightness values. This array stores the maximum brightness level for each LED in the strip.
  static int maxBrightness[NUM_LEDS] = {100, 80, 90, 100, 80, 110, 100, 120};
  //  Define a 'static' array 'hue' with size 'NUM_LEDS' and initialize all elements to '0'. This array
  //  stores the current hue value for each LED in the strip.
  static uint16_t hue[NUM_LEDS] = {0};

  //  Start a 'for' loop to iterate through each LED in the strip.
  for (int i = 0; i < NUM_LEDS; i++) {
    //  If the brightness of the LED at index 'i' is equal to its maximum brightness, negate the fade amount
    //  for that LED. This will cause the brightness to decrease in the next iteration.
    if (brightness[i] == maxBrightness[i]) {
      fadeAmount[i] = -fadeAmount[i];
    //  If the brightness of the LED at index 'i' is equal to its minimum brightness, make the fade amount
    //  positive (using 'abs()') and change the hue for that LED to a random value between '0' and '256'. This
    //  will cause the brightness to increase in the next iteration and change the color.
    } else if(brightness[i] == minBrightness[i]) {
      fadeAmount[i] = abs(fadeAmount[i]);
      hue[i] = random(0, 256);
    }
    //  Update the brightness level for the LED at index 'i' by adding its corresponding 'fadeAmount'. Use the
    //  'constrain()' function to ensure that the brightness value stays within the limits of 'minBrightness[i]'
    //  and 'maxBrightness[i]'.
```

```
    brightness[i] = constrain(brightness[i] + fadeAmount[i], minBrightness[i], maxBrightness[i]);
    //  Calculate the color for the LED at index 'i' using the 'strip.ColorHSV()' function with
    //  the following parameters:
    //  'hue[i] * 256': The hue of the color for the LED at index 'i'.
    //  '255': The saturation of the color.
    //  'brightness[i]': The brightness of the color for the LED at index 'i'.
    uint32_t color = strip.ColorHSV(hue[i] * 256, 255, brightness[i]);
    //  Set the color of the LED at index 'i' to the calculated color using the 'strip.setPixelColor()' function.
    strip.setPixelColor(i, color);
  //  Close the for loop.
  }
  //  Call the strip.show() function to send the updated color data to the LED strip and make the new
  //  colors visible.
  strip.show();
  //  Add a delay of 69 milliseconds before the next iteration.
  delay(69);
}
```