EIDGENÖSSISCHE TECHNISCHE HOCHSCHULE ZÜRICH

Assignment 05

Student name: *Mélina Sladic, Lukas Probst*
Legi number: *19-953-413, 22-714-240*

Course: *Wireless Networking and Mobile Computing* – Lecturer: *Dr. Mangold*
Due date: *November 28th, 2022*

Step 2: Theory

What is the bandwidth of the optical spectrum (measured in Hz)?

The optical or visible spectrum corresponds to a band in the vicinity of 400 to 789 THz [1].

Is the visible spectrum regulated?

There is the Federal Communications Commission (FCC[1]) which is responsible for managing and licensing the electromagnetic spectrum for commercial users and for non-commercial users including: state, county and local governments. However, there is no regulation concerning frequency for the visible spectrum.

What is the difference between infrared and visible light?

Visible light has a wavelength that ranges from 380 nm to 750 nm on the electromagnetic spectrum while infrared light has longer wavelengths. Infrared light has a frequency range of 300 GHz to 400 THz and a wavelenegth ranging from 700 nm to 1 mm, the beginning of the non-visible portion of the spectrum. Infrared waves can go through dust in space with less absorption than for visible light. Thus infrared can reveal things in the environment that cannot be seen in visible light. As a result, infrared light cannot be seen by the human eye except with special technical equipment.

[1]https://www.fcc.gov/licensing

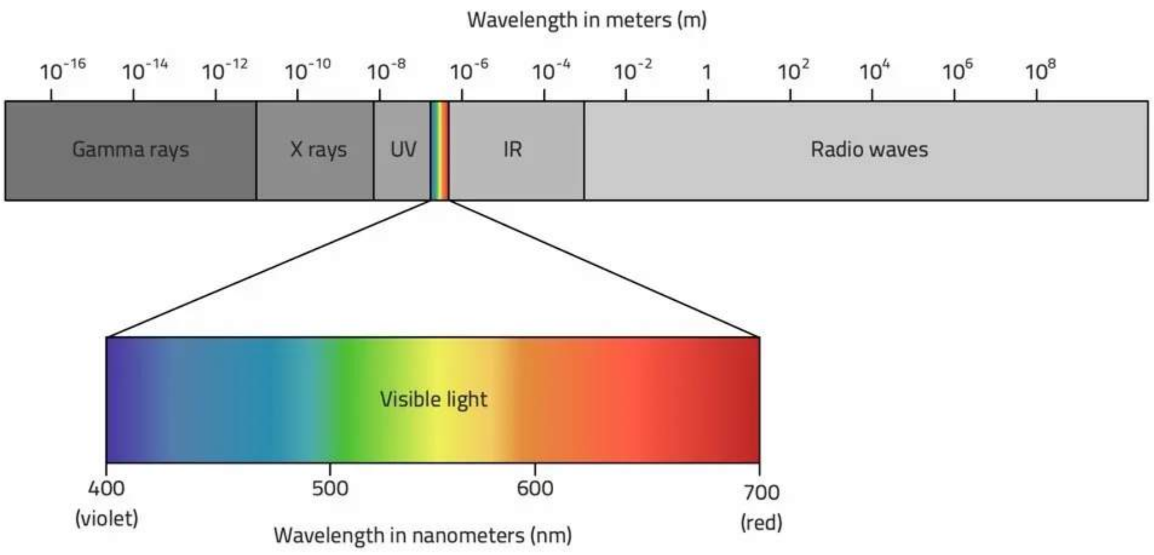


Figure 1: Components of the electromagnetic spectrum.

Can infrared light penetrate water? Can visible light?

Visible light can penetrate water: it is the best underwater communication means. However, light penetrating a water surface is scattered and absorbed as it passes downward. Indeed the water absorbs more than 50 percent of the visible light energy the deeper we go into it. Most infrared lights cannot penetrate water because their wavelengths are absorbed by water. As seen in figure 2, the ultraviolet and infrared spectrums are absorbed the most in water.

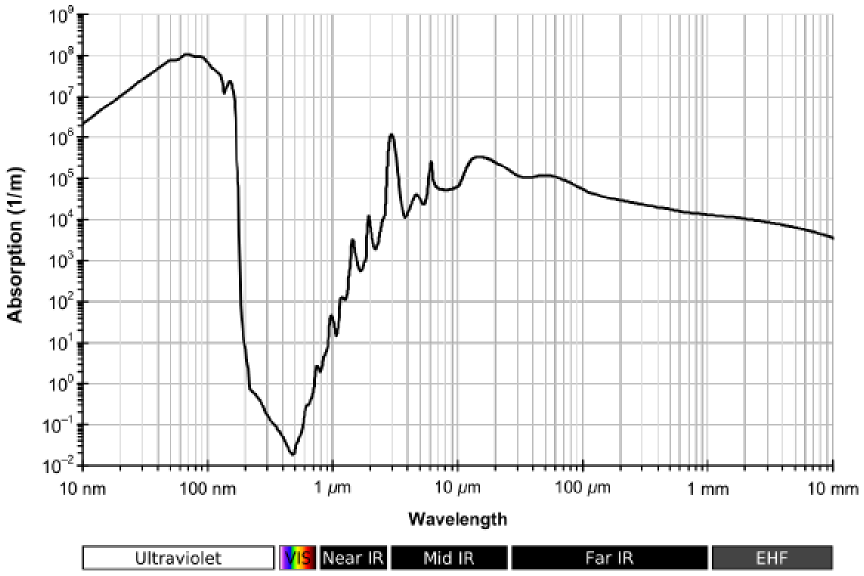


Figure 2: Liquid water absorption spectrum across a wide wavelength range (image from https://en.wikipedia.org/wiki/Electromagnetic_absorption_by_water).

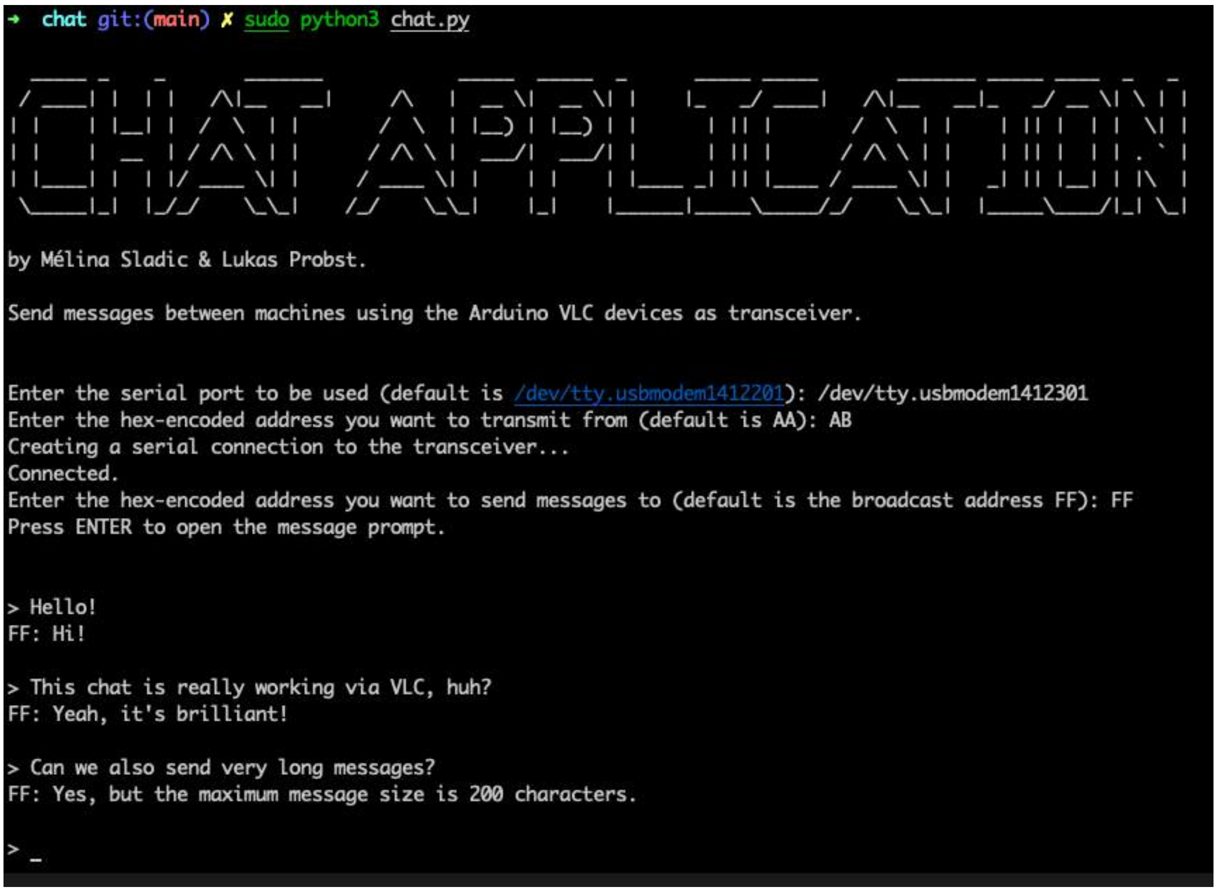
What are the benefits of using an LED instead of a photodiode as a receiver in consumer electronics?

The price of consumer gadgets is quite important. We do not need to include extra hardware in the form of a photodiode if the LED can still be used as a receiver. This significantly lowers costs, especially for gadgets where it is expensive to add a hole for a photodiode. Using a LED, we simply have to change the microchip without any additional cost. With the LED we have feedback channels all the time and the intensity of light produced by the LED is directly proportional to the current voltage. Also, with the LED, we don't have the problem of the dark current, which is present with the photodiode (dark current = the residual electric current flowing on the device when no light rays are incident on it).

Step 3: Arduino - PC Communication

We decided to use Python with the pySerial library for coding the chat & measurement application.

Step 4: Chat Application



```
→ chat git:(main) ✗ sudo python3 chat.py

by Mélina Sladic & Lukas Probst.

Send messages between machines using the Arduino VLC devices as transceiver.


Enter the serial port to be used (default is /dev/tty.usbmodem1412201): /dev/tty.usbmodem1412301
Enter the hex-encoded address you want to transmit from (default is AA): AB
Creating a serial connection to the transceiver...
Connected.
Enter the hex-encoded address you want to send messages to (default is the broadcast address FF): FF
Press ENTER to open the message prompt.


> Hello!
FF: Hi!

> This chat is really working via VLC, huh?
FF: Yeah, it's brilliant!

> Can we also send very long messages?
FF: Yes, but the maximum message size is 200 characters.


> _
```

Figure 3: Preview of our chat application.

```python
from sys import path
# Adding shared folder to the system path
```

```python
path.append('../shared')

from utility import default_port, serial_config
from serial import Serial
from signal import signal, SIGINT
from threading import Event, Thread
from broadcast import Broadcast
from cli import CLI


print("""
Chat Application

by Melina Sladic & Lukas Probst.

Send messages between machines using the Arduino VLC devices as transceiver.\n
""")

### User input
DEFAULT_PORT = default_port()
DEFAULT_SRC_ADDRESS = "AA"
DEFAULT_DEST_ADDRESS = "FF"

try:
    port = input(f"Enter the serial port to be used (default is {DEFAULT_PORT}): ") or
    DEFAULT_PORT
    src_address = input(f"Enter the hex-encoded address you want to transmit from (
    default is {DEFAULT_SRC_ADDRESS}): ") or DEFAULT_SRC_ADDRESS

    print("Creating a serial connection to the transceiver...")
    serial = serial_config(port, src_address)
    print("Connected.")

    dest_address = input(f"Enter the hex-encoded address you want to send messages to (
    default is the broadcast address {DEFAULT_DEST_ADDRESS}): ") or DEFAULT_DEST_ADDRESS
    print("Press ENTER to open the message prompt.\n")
# Writing the different exception class to catch/handle the exception
except EOFError:
    print('\nEOF exception, please enter something and try again.')
except KeyboardInterrupt:
    print('\nProgram terminated...')

exit_event = Event()
exit_event.clear()
def signal_handler(sig, frame):
    exit_event.set()
    print('\nProgram terminated...')
    exit(0)
signal(SIGINT, signal_handler)

broadcast = Broadcast(exit_event, serial, dest_address)
broadcast_thread = Thread(target=broadcast.event_loop)
broadcast_thread.start()

cli = CLI(exit_event, broadcast)
cli.event_loop()

broadcast_thread.join()
```

Listing 1: Main file chat.py.

```python
from sys import path
# Adding shared folder to the system path
path.append('../shared')
from threading import Event
from serial import Serial, SerialException
from queue import Queue, Empty
from utility import enc, dec

### Transmit messages and receive messages by listening for incoming messages from its
    queue
class Broadcast():

    # Constructor
    def __init__(self, exit_event: Event, serial: Serial, dest_address: str):
        self.exit_event = exit_event
        self.serial = serial
        self.dest_address = dest_address
        self.transmit_queue = Queue(maxsize=10)
        self.receive_queue = Queue(maxsize=10)

    # Put message into queue for transmission
    def put_transmit_queue(self, message: str):
        self.transmit_queue.put((message, self.dest_address), timeout=0.1)

    # Remove and return a message from the queue
    def get_message(self) -> str | None:
        try:
            message = self.receive_queue.get(timeout=0.1)
            self.receive_queue.task_done()
            return message
        except Empty:
            return None

    # Dispatches the oldest message from the transmit queue
    def send(self):
        try:
            (message, dest_address) = self.transmit_queue.get(timeout=0.1)
            self.transmit_queue.task_done()
        except Empty:
            return
        enc_message = enc(f"m[{message}\0,{dest_address}]\n")
        self.serial.write(enc_message)
        response = self.receive()
        # m[msg,dest_address] returns 1 on success
        assert "1" in response
        while self.receive() != "m[D]":
            pass

    def receive(self) -> str:
        dec_message = dec(self.serial.read_until()).strip()
        # R: Ready To Send (RTS), D: Data
        if dec_message.startswith("m[R,D,"):
            self.receive_queue.put((dec_message[6:-1].strip(), self.dest_address))
        return dec_message

     # Running infinite loop
    def event_loop(self):
        try:
            while not self.exit_event.is_set():
                self.receive()
                self.send()
        except SerialException:
            self.exit_event.set()
```

Listing 2: Transceiving messages is handled in broadcast.py.

```python
from threading import Event
from broadcast import Broadcast
from keyboard import add_hotkey

class CLI():

    def __init__(self, exit_event: Event, broadcast: Broadcast):
        self.exit_event = exit_event
        self.broadcast = broadcast
        self.is_accepting_input = False
        # Hotkey to accept input
        add_hotkey('enter', self.accept_input)

    def accept_input(self):
        self.is_accepting_input = True

    def event_loop(self):
        # Basically while True except program is terminated
        while not self.exit_event.is_set():
            # Try to receive a message
            received_message = self.broadcast.get_message()
            if received_message:
                (message, sender) = received_message
                print(f"{sender}: {message}")
            # Accept input and send message
            if self.is_accepting_input:
                input()
                message = input("> ")
                if len(message) <= 200:
                    self.broadcast.put_transmit_queue(message)
                else:
                    print("Maximum message size is 200 characters.")
                self.is_accepting_input = False
```

Listing 3: cli.py handles the user input.

```python
from sys import platform
from glob import glob
from time import sleep
from serial import Serial

### Settings
BAUDRATE = 115200

# Returns a default serial port based on the OS being used
def default_port():
    # Windows
    if platform.startswith("win"):
        return "COM9"
    # Linux
    elif platform.startswith("linux"):
        return next(iter(glob("/dev/tty[A-Za-z]*")), "none")
    # macOS
    elif platform.startswith("darwin"):
        return next(iter([x for x in glob("/dev/tty.*") if "usb" in x]), "none")
    else:
        return "none"

### Unicode strings are not supported, thus we need utility functions for encoding and
    decoding
def enc(string: str) -> bytes:
    return bytes(string, "ascii")

def dec(bytes: bytes) -> str:
    return bytes.decode("ascii")

# Sets defaults for the microcontroller
def serial_config(port: str, address: str) -> Serial:
    # Open the serial port and reset the device
    try:
        connection = Serial(port=port, baudrate=BAUDRATE, timeout=1)
    except:
        print(f"Port {port} could not be opened.")
        exit()
    sleep(2)
    # Set the device address (in hex)
    connection.write(enc(f"a[{address}]\n"))
    sleep(0.1) # Wait for settings to be applied
    assert dec(connection.read_until()) == f"a[{address}]\n"
    # Set the # of retransmissions to 5
    connection.write(enc("c[1,0,5]\n"))
    sleep(0.1) # Wait for settings to be applied
    assert dec(connection.read_until()) == "c[1,0,5]\n"
    # Set the FEC threshold to 30
    connection.write(enc("c[0,1,30]\n"))
    sleep(0.1) # Wait for settings to be applied
    assert dec(connection.read_until()) == "c[0,1,30]\n"
    return connection
```

Listing 4: utility.py which is located in a shared folder, contains utility functions that are mostly also used by the measurement program.

Step 5: Performance Measurement at different Distances, find out the Maximum Range

Initially, we just wanted to provide 2D graphs with distance on the x-axis and throughput or delay on the y-axis. However, since both values not only depend on the distance, but also on the payload, we decided to generate 3D graphs with Matplotlib.

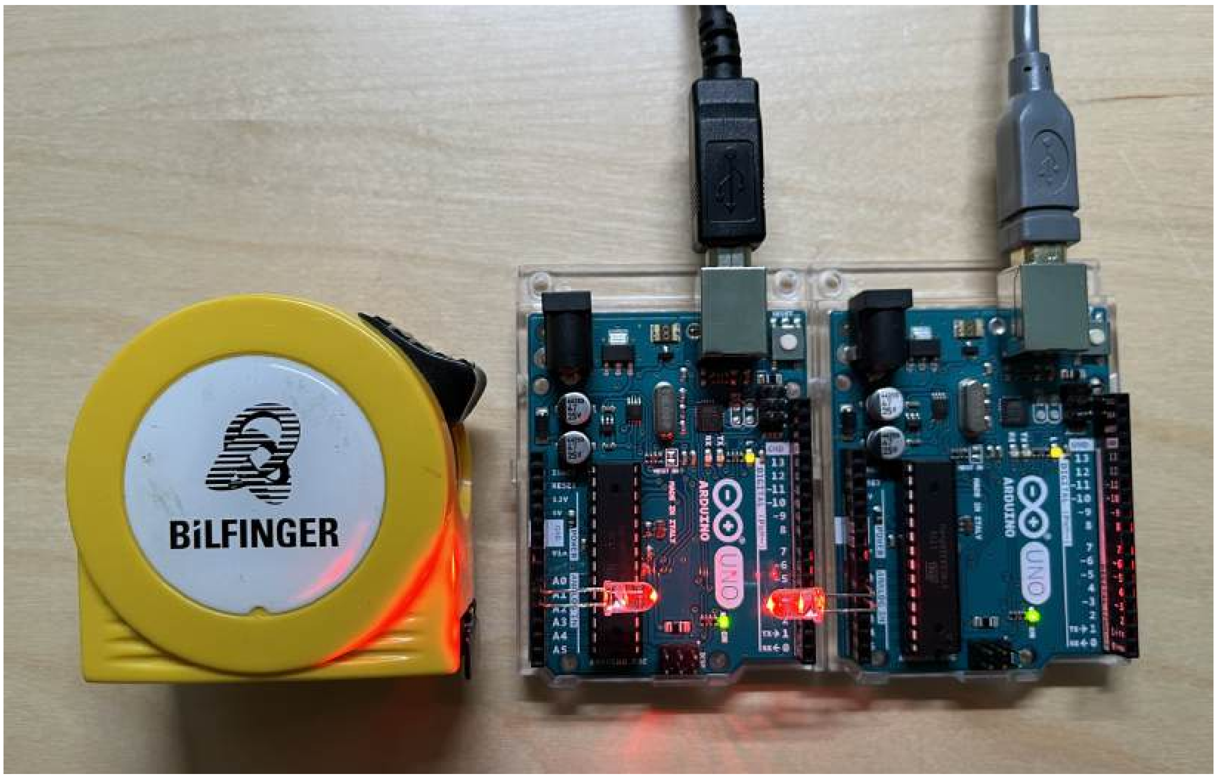


Figure 4: Experiment setup which includes two Arduinos, two transparent LEDs, two USB A-B cables and a measuring tape.

We tried sending data over a distance of 45 cm, but got a success rate of 0%. Therefore, we assume that the maximum range must lie around 40 cm where we could still receive data. However, this value should be treated with caution, as we cannot guarantee that the LEDs were precisely aligned. This proved to be difficult during the experiment, especially at longer distances, and so we sometimes got contradictory results and had to correct the alignment of the LEDs several times.

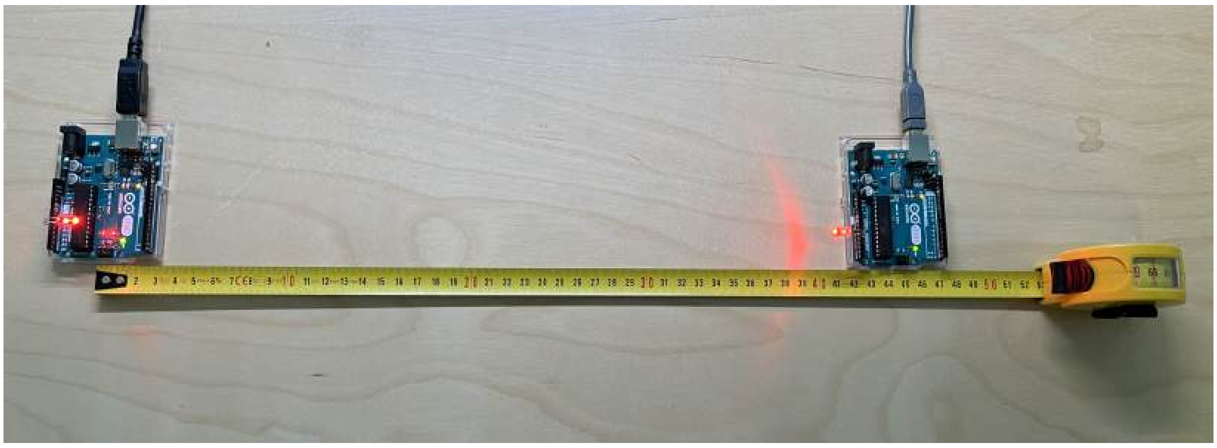


Figure 5: Maximum range in which data throughput could be measured.

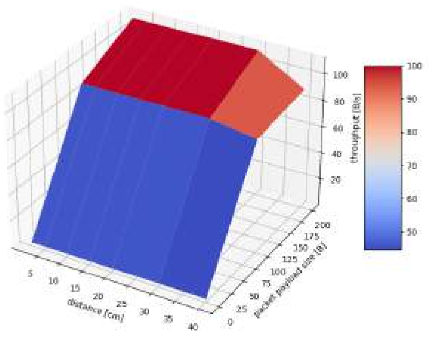


Figure 6: Data throughput measurement results.

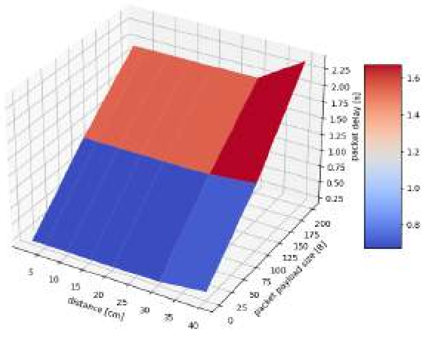


Figure 7: Packet delay measurement results.

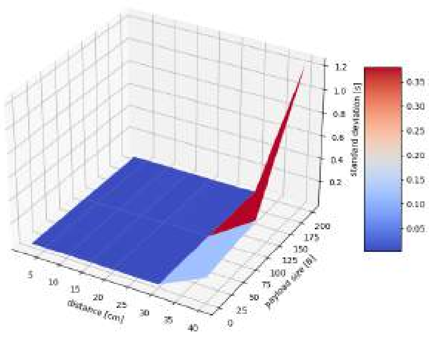


Figure 8: Standard deviation measurement results.

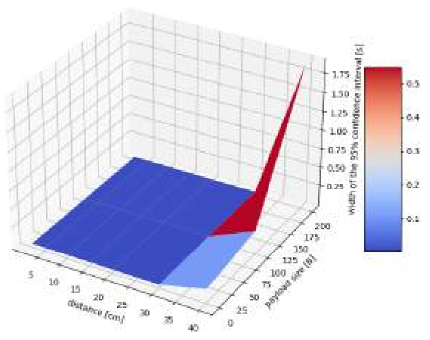


Figure 9: Confidence interval measurement results.

Description and interpretation of our findings. As a first step, we wanted to figure out the maximum possible distance between the two LEDs. We therefore tried distances of 1 m, 50 cm, 30 cm and around 25 cm we thought we found our limit. But, as said before, it is difficult to guarantee a perfect alignment, we sometimes got a 0% success rate and sometimes 100% for the same distance. Thus we did it again, tried to be even more accurate, and concluded that the maximum distance is around 40 cm. Then we chose our seven distances to be analyzed: 2 cm, 5 cm, 10 cm, 15 cm, 20 cm, 30 cm and 40 cm. Ultimately, we collected the data with all three payload sizes (1 B, 100 B, 200 B) for each distance and got 21 measurements in total. We have illustrated our results in the figures 6, 7, 8, 9.

The closer the LEDs are to each other, the better the performance. When the distance increases, the packet delay comes into play: the bigger the distance between the two LEDs, the bigger the delay (a difference of 0 to 2.4 s is observed). We observe a similar increasing pattern with the standard deviation and the confidence interval. A small standard deviation means that the data points are close to the mean, and a high standard deviation means that the data are more spread out. This means that there is more variability in the data from a 30 cm distance.

From our results, we concluded that the hit rate is 100% to around 40 cm and decreases once the distance is increased beyond that. We expect the hit rate to be between 0% and 100% in the range of 40 - 45 cm. That's what we found when the LEDs were perfectly aligned by visual estimate. This explains the fast increase of the confidence interval's width, which describes instability in the data.

In figures 6 and 7 we see an increase in throughput respectively packet delay the bigger the packet payload size. Thus even when the throughput is high, the delay is also higher. The graphs seem to show a constant performance and delay until a distance of 30 cm. From there, the performance decreases while the delay increases.

```python
from sys import path
# adding shared folder to the system path
path.append('../shared')
from utility import default_port, serial_config
from analyzer import Analyzer
from time import sleep

print("""
Performance Measurement

by Melina Sladic & Lukas Probst.

Lets you measure the performance of VLC at different distances & find out the max range.\n
""")

### User input
DEFAULT_PORT = default_port()
DEFAULT_PAYLOAD_SIZE = 100 # in byte
DEFAULT_DISTANCE = 2 # in cm
DEFAULT_MEASUREMENT_TIME = 20 # in s

try:
    port = input(f"Enter the serial port to be used (default is {DEFAULT_PORT}): ") or DEFAULT_PORT
    is_measuring = input(f"Is this device used as a transmitter and does the measuring? ").strip().lower() == "yes"
    address = "AA" if is_measuring else "AB"

    if (is_measuring):
        payload_size = int(input(f"Enter the packet payload size to be used (default is {DEFAULT_PAYLOAD_SIZE} byte, max. is 200 byte): ").strip() or DEFAULT_PAYLOAD_SIZE)
        distance = int(input(f"Enter the distance of the LEDs in cm (default is {
```

```
    DEFAULT_DISTANCE} cm): ").strip() or DEFAULT_DISTANCE)
        measurement_time = int(input(f"Enter how long the measurement should  take at most in s (default is {DEFAULT_MEASUREMENT_TIME} s): ").strip() or DEFAULT_MEASUREMENT_TIME)

    print("Creating a serial connection to the transceiver...")
    serial = serial_config(port, address)
    print("Connected.\nMeasuring...")

    if (is_measuring):
        analyzer = Analyzer(serial, payload_size, distance, measurement_time)
        analyzer.measure()
        analyzer.save_results()
    else:
        print("Receiving measurements...")
        while True:
            serial.read_until()
            # A pause of around 10 milliseconds might be needed between each command, not to overload the microcontroller's interface
            sleep(0.01)
    # Writing the different exception class to catch/handle the exception
except EOFError:
        print('\nEOF exception, please enter something and try again.')
except KeyboardInterrupt:
        print('\nProgram terminated...')
```

Listing 5: Main file measurement.py.

```
from sys import path
# adding shared folder to the system path
path.append("../shared")
from serial import Serial
from utility import dec, enc
from numpy import std, sqrt, mean
import scipy.stats as st
from random import choices
from string import ascii_lowercase
from time import sleep, time_ns
import csv

class Analyzer():

    def __init__(self, serial: Serial, payload_size: int, distance: int, measurement_time: int):
        self.serial = serial
        self.payload_size = payload_size
        self.distance = distance
        self.measurement_time = measurement_time
        self.measurement_time_ns = measurement_time * 1_000_000_000
        self.sent_payloads = 0
        self.ack_payloads = 0
        self.delays = []
        self.elapsed_time_ns = 0

    def receive(self):
        input = dec(self.serial.read_until()).strip()
        # Mesasge received and acknowledged (ACK)
        if (input == "m[R,A]"):
            self.ack_payloads += 1
        return input

    def measure(self):
        payload_str = "".join(choices(ascii_lowercase, k=self.payload_size))
        bytes_to_send = enc(f"m[{payload_str}\0,AB]\n")

        while self.sent_payloads < 30 and self.elapsed_time_ns < self.measurement_time_ns:
            sleep(0.01)
            t0 = time_ns()
            self.serial.write(bytes_to_send)
            while self.receive() != "m[D]":
                pass
            t1 = time_ns()
```

```python
            t = t1 - t0
            self.elapsed_time_ns += t
            self.sent_payloads += 1
            self.delays.append(t)
        # Wait for last ACK
        while self.receive() != "":
            pass

    def save_results(self):
        standard_deviation = std(self.delays) / 1_000_000_000
        mean_packet_delay = mean(self.delays) / 1_000_000_000
        confidence_interval = st.t.interval(alpha=0.95, df=self.sent_payloads-1, loc=mean_packet_delay, scale=standard_deviation / sqrt(self.sent_payloads))
        data_throughput = self.ack_payloads * self.payload_size / self.measurement_time
        success_rate = self.ack_payloads / self.sent_payloads

        print("\n")
        print(f"Payload size: {self.payload_size} byte")
        print(f"Distance: {self.distance} cm")
        print(f"Running time: {self.elapsed_time_ns / 1_000_000_000} s")
        print(f"Standard deviation: {standard_deviation} s")
        print(f"95% confidence interval: {confidence_interval}")
        print(f"Data throughput: {data_throughput} B/s")
        print(f"Mean packet delay: {mean_packet_delay} s")
        print(f"Acknowledged {self.ack_payloads} / {self.sent_payloads} payloads")
        print(f"Success rate: {success_rate * 100}%")

         # open the file in the write mode
        with open(f"results/{self.payload_size}B-{self.distance}cm.csv", "w+") as f:
            (cl, cr) = confidence_interval
            # write to the csv file
            f.write(", ".join([str(i) for i in [standard_deviation, data_throughput, mean_packet_delay, success_rate, cl, cr]]) + "\n")
            f.write(", ".join([str(i) for i in self.delays]))
```

Listing 6: analyzer.py measures, evaluates and saves the data.

References

[1] Stefan M. Schmid. "Software-Defined Low-Complex Visible Light Communication Networks". en. Doctoral Thesis. Zürich: ETH Zurich, 2016. DOI: 10.3929/ethz-a-010811920.