DSP Assignment 3
Wireless Data Transmission Using Lasers

Amana Ahmmad, Logan Brown
2590941A, 2641407B

November 28, 2024

1 Introduction

Data are usually transmitted wirelessly using sub-infrared frequencies: radio waves, microwaves, and so on. Transmission is usually performed using an omni-directional antenna - this is desirable as the transmitter need not know where the receiver is relative to itself.

However, there exist use cases where the transmitter may not want to send its data in all directions. If the receiver and transmitter are stationary and in known locations, but it is for some reason not practicable to lay cable between the two, a highly directional wireless system may be more ideal.

In this case, using a laser to send the data, rather than an antenna, is a possibility. This report explores the use of a laser diode to send data, in conjunction with a light-dependent resistor (LDR) to detect the signal.

2 Data framing

Since the receiver and transmitter are separated, there cannot be a clock line between the two, necessitating the use of an asynchronous scheme.

To frame the data, UART (universal asynchronous receiver-transmitter) was chosen, as it is commonly used, well documented, and well tested. Python code was written to emulate a UART transmitter (Tx) and receiver (Rx), with two classes that would output 0 or 1, or read 0 or 1 respectively.

The `UART_Tx` class allows the user to load an array of bytes, which are then "sent" automatically by the class. The `UART_Rx` class constantly listens for changes at 8 times the baud rate, and outputs a byte as soon as it determines that it has received one.

3 Set up

The laser was aimed at an LDR which was set up in a voltage-divider configuration, as outlined in Figure **??**. An Arduino connected to a computer measured the voltage across the LDR.

Whenever the output of Tx changed, the new state was reflected in the laser, to provide a binary amplitude-shift keying (ASK) scheme. There are, of course, more complex modulation schemes, offering increased data rates, that could have been used but for the sake of simplicity the scope did not go beyond ASK.

Thus, whenever the output of Tx was low (0), the laser was off, and when it was high (1), the laser was on.

4 Filtering

As LDRs react to all light, including ambient light, there was a large DC component in the voltage signal. Naturally, there was also a 50 Hz mains hum present most likely due to the visually imperceptible dimming of the ceiling lights during the mains cycle.

To remove these unwanted frequencies, a high pass and a bandstop IIR filter were chained together. The cutoff of the high pass was 5 Hz, and the cutoffs of the bandstop were 45 and 55 Hz.

5 Baud rate

Finally, the baud rate of the UARTs was set to {BAUD}. A blah blah justification

6 Appendix

6.1 uart.py

```
import numpy as np
from threading import Timer
from threading import Condition
import time

class RingBuffer:
```

```
    def __init__(self, size):
        self.buf = np.zeros(size)
        self.start = 0

    def append(self, x):
        self.buf[self.start] = x
        self.start += 1
        if self.start == len(self.buf):
            self.start = 0

    def __getitem__(self, i) -> int:
        index = self.start + i
        if index >= len(self.buf):
            index -= len(self.buf)

        return self.buf[index]

    def __str__(self) -> str:
        output = "["
        for i in range(len(self.buf)):
            output += str(self[i]) + ","
        return output[:-1] + "]"

    def __len__(self) -> int:
        return len(self.buf)


class ShiftRegister:
    def __init__(self, size):
        self.buf = RingBuffer(size)
        self.d = 0
        self.q = 0

    def __len__(self) -> int:
        return len(self.buf)

    def __str__(self) -> str:
        return str(self.buf)

    def clock(self):
        self.buf.append(self.d)
        self.q = self.buf[0]

    def clr(self):
        self.buf = RingBuffer(len(self.buf))

    def get_data(self) -> int:
        val = 0
        for i in range(len(self.buf)):
            val += int(2**(7-i) * self.buf[i])

        return val


class UART:
    def __init__(self, baud_rate):
        self._baud_rate = baud_rate
        self._buf = ShiftRegister(8)
        self._clocked_bits = 0
```

```
class UART_Rx(UART):
    def __init__(self, baud_rate, available_callback):
        UART.__init__(self, baud_rate)

        self.__clk_divisor = 8
        self.__available_callback = available_callback
        self.__clk_period = 1 / (self.__clk_divisor * self._baud_rate)
        self.__dclk_period = 1 / (self._baud_rate)
        self.__low_for = 0
        self.__high_for = 0

        # Flags
        self.__receiving = False

        # Timers
        self.__clk = Timer(self.__clk_period, self.__clk_callback)
        self.__clk.start()
        self.__dclk = Timer(self.__dclk_period, self.__dclk_callback)

        # Rx data line, accessible outside the receiver
        self.d = 1

        # Clock for debugging
        self.c = 0

    # Returns True if start bit detected
    # Should only be called ONCE per clock cycle
    def __check_start(self):
        if self.d == 1:             # If high, then we are not receiving this bit time, an
            self.__low_for = 0      # we can consider it a spurious pulse

        self.__low_for += 1
        if self.__low_for == self.__clk_divisor - 1:
            self.__low_for = 0
            self.c = 2
            self.__dclk.start()
            return True
        return False

    def __clk_restart(self):
        self.__clk = Timer(self.__clk_period, self.__clk_callback)
        self.__clk.start()

    def __dclk_restart(self):
        self.__dclk = Timer(self.__dclk_period, self.__dclk_callback)
        self.__dclk.start()

    def __dclk_callback(self):
        if self.__receiving: self.__dclk_restart()

        self._buf.d = 1 if self.__high_for > self.__low_for else 0
        self._buf.clock()

        self.c = 1

        self.__high_for = 0
        self.__low_for = 0

        self._clocked_bits += 1

        if self._clocked_bits == len(self._buf):
```

```
            self.__dclk.cancel()
            self.__dclk = Timer(self.__dclk_period, self.__dclk_callback)
            self.__receiving = False
            self.__available_callback()
            self._clocked_bits = 0

    def __clk_callback(self):
        # Can assume that callback will take less time than the clock period
        # Restart timer immediately so that clock period doesn't include processing time
        self.__clk_restart()

        if not self.__receiving:
            self.__receiving = self.__check_start()
            return

        # If reaching this point, then must be receiving
        self.__low_for  += 1 if self.d == 0 else 0
        self.__high_for += 1 if self.d == 1 else 0

    def get_buf(self):
        return self._buf.get_data()

    def stop(self):
        self.__clk.cancel()


class UART_Tx(UART):
    def __init__(self, baud_rate, q_callback):
        UART.__init__(self, baud_rate)
        self.__clk = Timer(1 / baud_rate, self.__send_data)

        # Tx data line
        self.q = 1
        self.__q_callback = q_callback
        self.__cv = None

        # Flags
        self.sending = False

    def __set_q(self, q):
        self.q = q
        self.__q_callback(q)

    def __send_data(self):
        self.__set_q(self._buf.q)
        self._buf.clock()
        clocked_bits = self._clocked_bits
        self._clocked_bits += 1

        if clocked_bits < len(self._buf):                                   # Data bit
            self.__clk = Timer(1 / self._baud_rate, self.__send_data)
            self.__clk.start()
        elif clocked_bits <= len(self._buf) + 1:                            # Stop bits
            self.__set_q(1)
            self.__clk = Timer(1 / self._baud_rate, self.__send_data)
            self.__clk.start()
        else:                                                               # Stop sending
            self.__set_q(1)
            self.sending = False
            self.__clk = Timer(1 / self._baud_rate, self.__send_data)
            self._clocked_bits = 0
```

```
            if self.__cv != None:
                self.__cv.acquire()
                self.__cv.notify_all()
                self.__cv.release()

    def load_data(self, data: int):
        val = data
        for i in range(len(self._buf)):
            self._buf.d = val // 2**(7-i)
            if val // 2**(7-i) != 0: val -= 2 ** (7-i)
            self._buf.clock()
        self._buf.d = 0

    def send_frame(self):
        self.__set_q(0)                                                    # Start bit
        self.sending = True
        self.__clk.start()

    def send_bulk(self, data, callback):
        self.__cv = Condition()
        for d in data:
            self.load_data(d)
            self.send_frame()
            while self.sending:
                self.__cv.acquire()
                self.__cv.wait()
                self.__cv.release()

        self.__cv = None
        callback()


def graph_uart():
    baud = 100
    rx_output = []
    rx_clk = []

    def update_rx(q):
        rx.d = q
    rx = UART_Rx(baud, (lambda : print(f"{rx.get_buf()}", end="")))
    tx = UART_Tx(baud, update_rx)

    idx = 0
    data = "Hello World\n"
    time.sleep(0.5)
    while idx < len(data) or tx.sending:
        rx_output.append(rx.d)
        rx_clk.append(rx.c)
        rx.c = 0
        if not tx.sending:
            tx.load_data(data[idx])
            tx.send_frame()
            idx += 1

        time.sleep(0.00001)

    while tx.sending: x = 0
    rx.stop()


    import matplotlib.pyplot as plt
```

```
    plt.plot(rx_output)
    plt.plot(rx_clk)
    plt.ylim(-0.1, 2.1)
    plt.show()

def send_file():
    baud = 100
    def finish():
        print("finished")
        rx.stop()
    def update_rx(q):
        rx.d = q
    def recv_byte():
        global out_bytes
        out_bytes.append(rx.get_buf())
        print(f"\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b{len(out_bytes)}", end="", flush=True)

    out_bytes = []
    rx = UART_Rx(baud, recv_byte)
    # rx = UART_Rx(baud, (lambda : print(f"{chr(rx.get_buf())}", end="", flush=True)))
    tx = UART_Tx(baud, update_rx)
    data = []
    with open("assets/to_send.png", "rb") as f:
        data = f.read()
    tx.send_bulk(data, finish)

    with open("assets/received.png", "wb") as f:
        f.write(bytes(out_bytes))
```