DSP Assignment 3
Wireless Data Transmission Using Lasers

Amana Ahmmad, Logan Brown
2590941A, 2641407B

December 7, 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 phototransistor 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. 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 a phototransistor which was set up in a voltage-divider configuration, as outlined in Figure ??,1. An Arduino connected to a computer measured the voltage across the resistor.

An FSK modulation scheme was used. A logic 0 corresponds to the laser being switched off, effectively 0 Hz, while a logic 1 corresponds to the laser being keyed on and off at a given frequency. This was chosen to be 100 Hz, as it would be significantly higher than any environmental optical and electronic noise, but low enough that it can be reliably sampled at 1000 Hz with minimal aliasing.

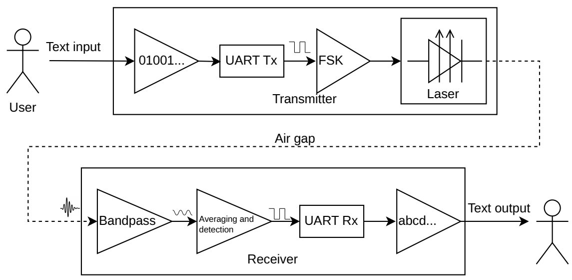

Figure 1: Dataflow diagram

4 Filtering

The phototransistor has a relatively wide wavelength response, peaking in near-infrared but extending into shorter-wavelength visible light, and longer infrared. Thus, it responds to much of the ambient light in an environment, and so there is a significant amount of noise in the signal – both optical and electrical.

The main sources of noise are the 50 Hz mains electric hum, very low-frequency drift from natural sources, and DC drift from the ambient light level.

To remove these unwanted frequencies, an 8^{th} order (or rather, a chain of 4 2^{nd} order) bandpass IIR filter was used, with the passband being 5 Hz either side of the sending frequency.

5 Baud rate

Finally, the baud rate of the UARTs was set to 10. The delay of the IIR filter and the response of the phototransistor were slow enough that sending faster than this would not allow the logic levels enough time to settle, resulting in missed bits. Naturally, this would be too slow for a real world scenario, but with more research and engineering it would be relatively trivial to make this faster.

6 Conclusion

It was possible to send large amounts of data with no loss, albeit quite slowly. The IIR filter performed well, eliminating all environmental noise and providing a clean signal to the UART receiver, though there was a significant delay in the response that limited the baud rate.

7 Appendix

7.1 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
```

```python
        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, and
            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):
```

```python
        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

        if self.__q_callback != None:
            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{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))
```

7.2 filter.py

```
import numpy as np
from scipy.signal import butter
from scipy.signal import freqz
import scipy.signal as signal
import matplotlib.pyplot as plt
import unittest

class IIRFilter():
    def __init__(self, coeffs):
       self.b = coeffs[0:3]
       self.a = coeffs[4:]

       self.tb1 = 0
       self.tb2 = 0
       self.ta1 = 0
       self.ta2 = 0
```

```
    def filter(self, sample):
        accumulator  = sample   * self.b[0]
        accumulator += self.tb1 * self.b[1]
        accumulator += self.tb2 * self.b[2]
        accumulator -= self.ta1 * self.a[0]
        accumulator -= self.ta2 * self.a[1]

        self.tb2 = self.tb1
        self.tb1 = sample
        self.ta2 = self.ta1
        self.ta1 = accumulator

        return accumulator

class Filter():
    def __init__(self, fs, f_pass):
        coeffs = butter(8, [f_pass-5, f_pass+5], btype="bandpass", fs=fs, output="sos")
        self.__iirs = [IIRFilter(c) for c in coeffs]

    def filter(self, sample):
        for iir in self.__iirs: sample = iir.filter(sample)

        return sample

def test_filter():
    # test signal
    fs = 1000
    data = np.abs(np.fft.ifft(np.ones(fs)))

    # apply the filter
    out_data = []
    filter = Filter(fs, 50)
    for d in data:
        out_data.append(filter.filter(d))

    plt.plot(np.abs(np.fft.fft(data)) / len(data), label= 'Original␣FFT')
    plt.plot(np.abs(np.fft.fft(out_data) / len(out_data)), label= 'Filtered␣FFT')
    plt.xlabel("Frequency␣(Hz)")
    plt.ylabel("Amplitude")
    plt.title("FFT␣of␣Original␣and␣Filtered␣Signals")
    plt.legend()
    plt.grid()
    plt.show()

class FilterTest(unittest.TestCase):

    def gen_test_signal(self, fs):
        return np.abs(np.fft.ifft(np.ones(fs)))

    tolerance_places = 1

    # TEST PASSBAND PERFORMANCE
    def passband_attenuation(self, f_pass):
        fs = 1000
        signal = self.gen_test_signal(fs)

        filter = Filter(fs, f_pass)
        out_signal = [filter.filter(x) for x in signal]
        self.assertAlmostEqual(np.abs(np.fft.fft(out_signal))[f_pass], np.abs(np.fft.fft(
            signal))[f_pass], places=self.tolerance_places)

    def test_passband_attenuation_20Hz(self):
        self.passband_attenuation(20)

    def test_passband_attenuation_100Hz(self):
        self.passband_attenuation(100)

    def test_passband_attenuation_200Hz(self):
        self.passband_attenuation(200)


    # TEST INDIVIDUAL IIR PASSBAND PERFORMANCE
    def iir_passband_attenuation(self, f_pass):
        fs = 1000
        signal = self.gen_test_signal(fs)
```

```
        filter = IIRFilter(butter(1, [f_pass-5, f_pass+5], btype="bandpass", fs=fs, output="
            sos")[0])
        out_signal = [filter.filter(x) for x in signal]
        self.assertAlmostEqual(np.abs(np.fft.fft(out_signal))[f_pass], np.abs(np.fft.fft(
            signal))[f_pass], places=self.tolerance_places)

    def test_iir_passband_attenuation_20Hz(self):
        self.iir_passband_attenuation(20)

    def test_iir_passband_attenuation_100Hz(self):
        self.iir_passband_attenuation(100)

    def test_iir_passband_attenuation_200Hz(self):
        self.iir_passband_attenuation(200)


    # TEST STOPBAND PERFORMANCE
    def stopband_attenuation(self, f_pass):
        fs = 1000
        signal = self.gen_test_signal(fs)

        filter = Filter(fs, f_pass)
        out_signal = [filter.filter(x) for x in signal]

        for i in range(f_pass + 10, fs // 2):
            self.assertAlmostEqual(np.abs(np.fft.fft(out_signal))[i], 0, places=self.
                tolerance_places)
        for i in range(0, f_pass - 10):
            self.assertAlmostEqual(np.abs(np.fft.fft(out_signal))[i], 0, places=self.
                tolerance_places)

    def test_stopband_attenuation_20Hz(self):
        self.stopband_attenuation(20)

    def test_stopband_attenuation_100Hz(self):
        self.stopband_attenuation(100)

    def test_stopband_attenuation_200Hz(self):
        self.passband_attenuation(200)


if __name__ == "__main__":
    unittest.main()
```

7.3 graph.py

```
import matplotlib
matplotlib.use('Qt5Agg')
from PyQt6 import QtCore, QtWidgets
from matplotlib.backends.backend_qtagg import FigureCanvasQTAgg as FigureCanvas
from matplotlib.figure import Figure
import pyqtgraph as pg # tested with pyqtgraph==0.13.7

from PyQt6.QtWidgets import QApplication, QMainWindow, QPushButton
import pyqtgraph as pg

class Graph(QMainWindow):
    def __init__(self, title, n_plots, back_samples, ylim=[0,1]):
        super(Graph, self).__init__()

        self.back_samples = int(back_samples)
        self.setWindowTitle(title)

        self.time_plot = pg.PlotWidget()
        self.time_plot.setYRange(ylim[0], ylim[1])
        # self.time_plot_curve1 = self.time_plot.plot([], symbolPen=pg.mkPen(color=(200, 200,
            255)))
        # self.time_plot_curve2 = self.time_plot.plot([])
        self.setCentralWidget(self.time_plot)

            # Setup a timer to trigger the redraw by calling update_plot.
        self.timer = QtCore.QTimer()
        self.timer.setInterval(0) # causes the timer to start immediately
        self.timer.timeout.connect(self.update_plot) # causes the timer to start itself again
            automatically
        self.timer.start()
```

```
        self.plot_curves = [self.time_plot.plot([]) for _ in range(n_plots)]
        self.samples = [[] for _ in range(n_plots)]

        self.show()

    def update_plot(self):
        for plot_curve, samples in zip(self.plot_curves, self.samples):
            plot_curve.setData(samples[-self.back_samples:])

    def add_sample(self, sample, plot):
        self.samples[plot].append(sample)
```

7.4 receiver.py

```
import uart
from filter import Filter
import config
import time
import numpy as np
import scipy.signal as signal
import matplotlib.pyplot as plt
import matplotlib
from pyfirmata2 import Arduino
from threading import Condition, Timer, Thread
from graph import Graph

matplotlib.use('Qt5Agg')
from PyQt6 import QtWidgets

samples = [[], []]

class ThreadSafeQueue():
    def __init__(self):
        self.__buf = []
        self.__cv = Condition()

    def append(self, x):
        self.__cv.acquire()
        self.__buf.append(x)
        self.__cv.release()

    def pop(self):
        self.__cv.acquire()
        x = self.__buf.pop(0) if len(self.__buf) > 0 else None
        self.__cv.release()
        return x


class Receiver():
    def __init__(self, baud, sampling_rate, board, analogue_channel, f):
        self.__uart = uart.UART_Rx(baud, self.end)
        self.__board = board
        self.__board.samplingOn(1000 / sampling_rate)
        self.__board.analog[analogue_channel].register_callback(self.__poll)
        self.__board.analog[analogue_channel].enable_reporting()
        self.__temp_buf = ThreadSafeQueue()
        self.__buf = uart.RingBuffer(int(2 * sampling_rate/f))
        self.__filter = Filter(sampling_rate, f)

        self.__filtered_graph = Graph("Filtered", 1, 5 * f, ylim=[-0.2, 0.2])
        self.__averaged_graph = Graph("Filtered␣and␣averaged", 1, 2 * sampling_rate, ylim=[0,
            0.2])
        self.__raw_graph = Graph("Raw␣data", 1, sampling_rate / 2)

        self.__update_timer = Timer(0.01, self.__update)
        self.__update_timer.start()

    def __update(self):
        self.__update_timer = Timer(0.01, self.__update)
        self.__update_timer.start()

        data = self.__temp_buf.pop()
        while (data != None):
            self.__raw_graph.add_sample(data, 0)
            x = self.__filter.filter(data)
```

```
            self.__buf.append(abs(x))
            self.__filtered_graph.add_sample(x, 0)

            x = np.max(self.__buf)

            self.__averaged_graph.add_sample(x, 0)

            thresh = 0.035
            self.__uart.d = 0 if x < thresh else 1

            data = self.__temp_buf.pop()

    def __poll(self, data):
        self.__temp_buf.append(data)

    def end(self):
        print(chr(self.__uart.get_buf()), end="", flush=True)

    def teardown(self):
        self.__averaged_graph.close()
        self.__filtered_graph.close()
        self.__raw_graph.close()
        self.__uart.stop()
        self.__board.exit()
        self.__update_timer.cancel()

app = QtWidgets.QApplication([])

PORT = Arduino.AUTODETECT
board = Arduino(PORT,debug=True)

baud, f = config.read_config()
receiver = Receiver(baud, 1000, board, 1, f)
time.sleep(1)

Thread(target = app.exec).start()

input("")
receiver.teardown()
```

7.5 transmitter.py

```
import uart
import config
import numpy as np
import matplotlib.pyplot as plt
import time

from pyfirmata2 import Arduino
from threading import Timer

class Transmitter():
    class LocalOscillator():
        def __init__(self, f):
            self.__time_offset = time.time_ns()
            self.__period_ns = 10**9 / f

        def state(self):
            t = time.time_ns() - self.__time_offset
            return (t % self.__period_ns) < (self.__period_ns / 2)


    def __init__(self, baud, board, f):
        self.__uart = uart.UART_Tx(baud, None)
        self.__board = board
        self.__output_pin = self.__board.get_pin("d:4:o")
        self.__output_pin.write(1)

        self.__lo = self.LocalOscillator(f)

        self.__lo_timer = None
        self.__lo_update_interval = 0.001
        self.update()

    def update(self):
        self.__lo_timer = Timer(self.__lo_update_interval, self.update)
```

```
        self.__lo_timer.start()

        self.__output_pin.write(self.__lo.state() if self.__uart.q else 0)

    def start(self, data):
        print("Sending...")
        self.__uart.send_bulk(data, self.end_sending)


    def end_sending(self):
        print("Done")

    def __teardown(self):
        self.__lo_timer.cancel()
        self.__board.exit()

    def prompt(self):
        message = input("Enter␣message:␣") + "\n"
        if message == "q\n":
            self.__teardown()
        else:
            self.start(map(ord, message))
            self.prompt()

PORT = Arduino.AUTODETECT
board = Arduino(PORT,debug=True)

baud, f1 = config.read_config()
transmitter = Transmitter(baud, board, f1)

transmitter.prompt()
```

7.6 config.py

```
import json

def read_config():
    with open("assets/config.json") as f:
        json_data = json.loads(f.read())
        return json_data["baud"], json_data["f"]

    raise Exception("Error␣parsing␣config")
```