BRIGHT & FURIOUS

Politechnika Warszawska

Sprawozdanie z przedmiotu PROJ2
grupa dziekańska: 4T3

22 czerwca 2023

Spis treści

1. Wstęp

1.1. Skład zespołu

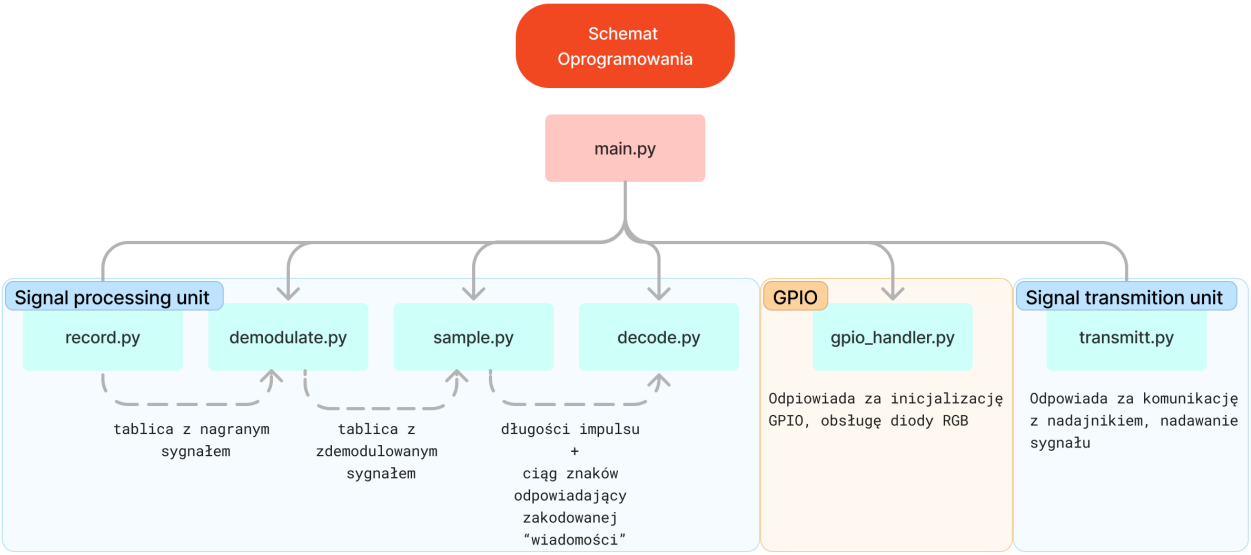
Aby wybrać temat projektu, a następnie rozpocząć nad nim pracę, najpierw musieliśmy uformować zespół i podzielić się na poszczególne role w zespole. Finalnie skład naszego zespołu prezentuje się następująco:
— **Product Owner**
— **Scrum Master**
— **Developer**
— **Developer**
— **Developer**
— **Developer**

1.2. Temat projektu i opis zadania

Tematem który nam przydzielono zostało: **Zhackowanie pilota zdalnego sterowania**. Naszym zadaniem było przechwycenie przesyłanego z pilota sygnału, następnie przetworzenie, zdekodowanie, a potem nadanie go z niezależnego urządzenia, np. z odpowiednio zaprogramowanego pilota, czy komputera. Innymi słowy, musieliśmy zrobić kopię pilota działającego na gniazdka firmy *REBEL* sterowane zdalnie.

2. Opis końcowego rozwiązania

Pracę nad wytwarzanym oprogramowaniem podzielono na mniejsze obszary co jest odzwierdziedlone przez podział na moduły.



Zdjęcie 1: Schemat - wytworzone oprogramowanie

2.1. record.py

Segment `record.py` jest odpowiedzialny za bezpośrednią komunikację przygotowywanego oprogramowania i modułu SDR. Wyzwala nagrywanie próbek sygnału przez przekształca je do prostej postaci pozwalającej na dalszą obróbkę.

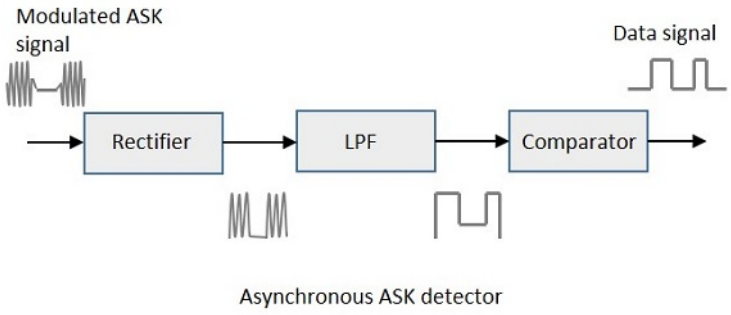
```
from scipy import signal
from numpy import max, abs, int16
from rtlsdr import RtlSdr
from scipy.io import wavfile
```

```
from datetime import datetime

N = 4096000
FREQ = 433.92e6
F_OFFSET = 0.02e6
GAIN = 15
SAMPLE_RATE = 1e6


def current_time():
    time = datetime.now()
    current_time = time.strftime("%H:%M:%S")
    return current_time


def record():
    # print("start")
    fc = FREQ - F_OFFSET
    sdr = RtlSdr()
    sdr.center_freq = fc
    sdr.sample_rate = SAMPLE_RATE
    sdr.gain = GAIN
    samples = sdr.read_samples(N)
   # decimated = signal.decimate(samples, 20) #Dla wartości 20 jest jeszcze w miarę ładnie widoczny sygnał
   # scaled = int16(decimated.real / max(abs(decimated)) * 32767)
   # wavfile.write('../../remote-control-hacking/recorded_signals_urh/1_off/out2.wav', int(sdr.sample_rate), scaled.astype("int16"))
    # print("end")
    return samples


def decimate(samples):
    # Dla wartości 20 jest jeszcze w miarę ładnie widoczny sygnał
    decimated = signal.decimate(samples, 20)
    scaled = int16(decimated.real / max(abs(decimated)) * 32767)
    return scaled
```

Plik record.py

2.2. demodulate.py

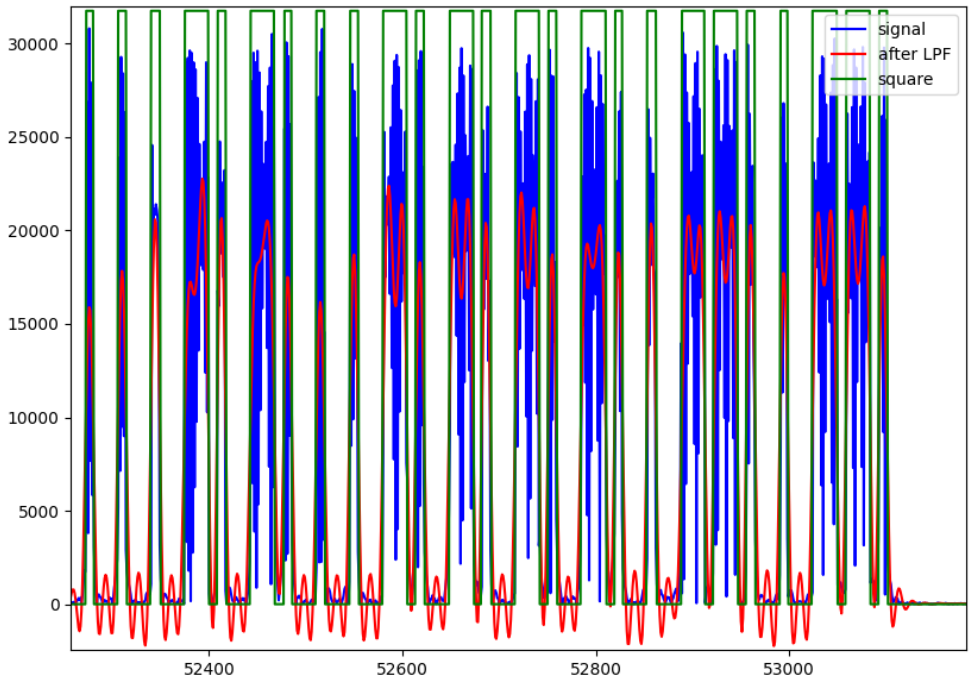
Moduł `demodulate.py` odpowiada za przeprowadzenie demodulacji ASK. Podczas tego etapu pracy nad rozwiązanie zadecydowano o wykorzystaniu demodulacji asynchronicznej. Powodami przemawiającymi za wyższością tej metody w porównaniu do demodulacji synchronicznej były prostsza implementacja oraz brak konieczności synchronizacji z sygnałem nośnym.



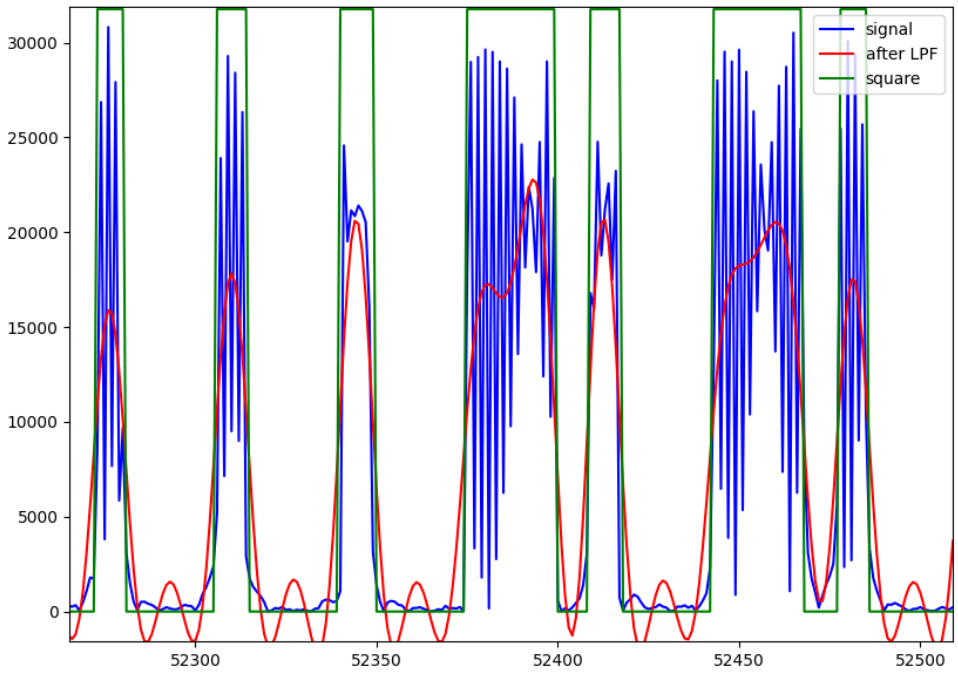
Zdjęcie 2: Schemat asynchronicznego demodulatora ASK

W naszej realizacji jako układ prostowniczy wykorzystano funkcję `abs` pakietu `numpy`. Zwraca ona tablice wartości bezwzględnych tablicy wejściowej. Następnie próbki są filtrowane przez filtr dolnoprzepustowy (w implementacji użyto filtru Butterwortha - zaczerpniętego z pakietu `scipy.signal`), jego parametry zostały dobrane eksperymentalnie. Następnie zastosowano proste przyrównanie wartości wyjściowych filtra do progu, określonego jako część amplitudy sygnału, dla wartości większej od progu przyjęto wartość nowej funkcji 1, a mniejsze otrzymały 0.

Moduł wyposażony jest w możliwość interaktywnej prezentacji postępów pracy za pomocą wykresów realizowanych za pomocą pakietu `matplotlib`. Poniżej umieszczono wykresy prezentujące poszczególne etapy demodulacji (na niebiesko - sygnał przez po wyciągnięciu z niego wartości bezwzględnej, na czerwono - sygnał po filtracji filtrem LPF, na zielono - sygnał prostokątny, wynikowy(jego amplituda została zwiększona w celu ułatwienia analizy)).



Zdjęcie 3: Proces demodulacji - zbliżenie na pojedynczy „kod nadawany przez pilota”



Zdjęcie 4: Proces demodulacji - zbliżenie na krótszy fragment sygnału

Program został wyposażony również w instrukcję `if __name__ == "__main__":` pozwalającą na uruchomienie go jako osobny projekt ładujący przygotowane wcześniej dane z pliku `csv`.

```
import numpy as np
from scipy.signal import butter, lfilter, freqz, filtfilt
import scipy.signal as sigtool

plot = True
# Filter requirements.
order = 6
fs = 10
cutoff = 0.75
T = 5.0


def rectifier(my_data):
    rect_data = (np.abs(my_data))
    return rect_data


def butter_lowpass(cutoff, fs, order=5):
    return butter(order, cutoff, fs=fs, btype='low', analog=False)


def butter_lowpass_filter(data, cutoff, fs, order=5):
    b, a = butter_lowpass(cutoff, fs, order=order)
    y = filtfilt(b, a, data)
    return y


def comparator(y, data):
    env = np.abs(data)
    treshold = np.max(env) * 0.2899
    square_sig = (y > treshold) * 1
    return square_sig


def main(data):
    n = len(data) # total number of samples
    b, a = butter_lowpass(cutoff, fs, order)
    data = rectifier(data)
    t = np.linspace(0, T, n, endpoint=False)
    y = butter_lowpass_filter(data, cutoff, fs, order)
    # print(rectifier(data))
```

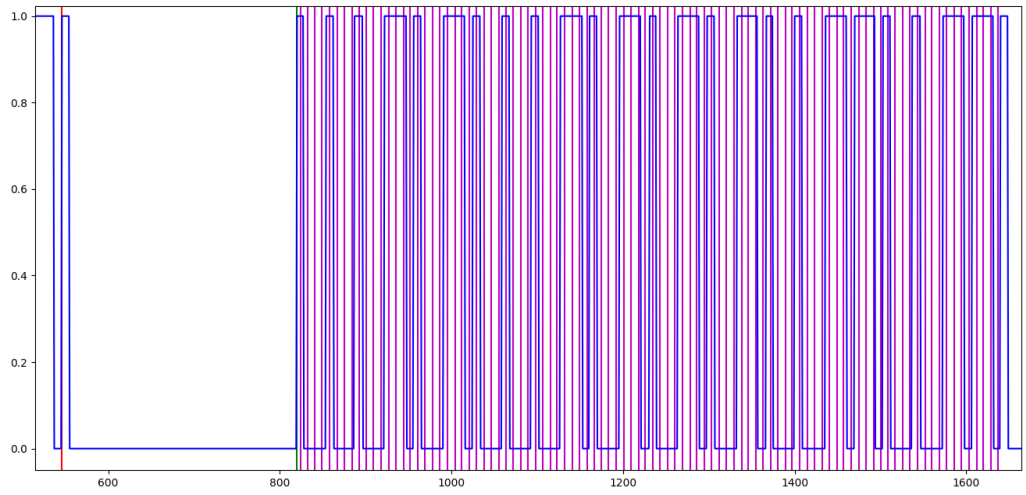
```
    square_signal = comparator(y, data)

    np.savetxt("../demodulated_signal.csv", square_signal, delimiter=",")
    return square_signal


if __name__ == '__main__':
    sample_data = np.genfromtxt('../signal.csv', delimiter=',')
    output_signal = main(sample_data)
    np.savetxt("../demodulated_signal.csv", output_signal, delimiter=",")
    print(output_signal)
```

Plik demodulate.py

2.3. sample.py

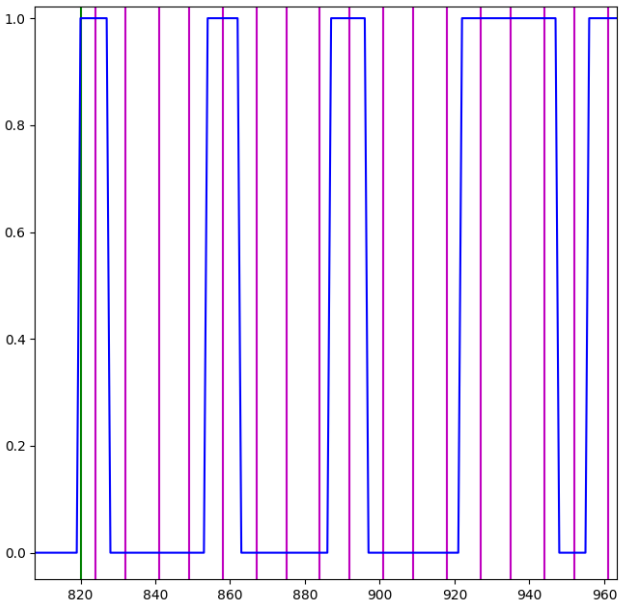
Moduł `sample.py` może pracować w dwóch trybach: w pierwszym wykorzystuje wstępnie przetworzone dane z pliku `demodulated_signal.csv` w drugim pracuje jako część całego układu przetwarzając dane otrzymane bezpośrednio z modułu `demodulate.py` i szuka w nich synchronizacji, aby próbkować dane zgodnie z zegarem nadawanym przez nadajnik.
Funkcja `find_first_impulse()` znajduje pierwszy impuls w sygnale i zwraca jego długość oraz pozycję.
Funkcja `find_sync_symbol()` sprawdza, czy kolejny impuls jest synchronizatorem i zwraca wartość True oraz długość impulsu, jeśli tak. W przeciwnym razie zwraca wartość `False`.
Funkcja `main()` szuka pierwszego impulsu i jeśli jest to synchronizator, próbkuję dane i zwraca je jako listę ciągów binarnych. Wykres jest również rysowany, aby zobaczyć, gdzie znaleziono synchronizator i próbkowane bity.

Program również został wyposażony w możliwość wizualnej weryfikacji wykonywanych przez niego zadań.



Zdjęcie 5: Proces próbkowania

Na powyższym wykresie niebieską linią oznaczono sygnał zdemodulowany, czerwoną kreską pionową oznaczono wykrycie sygnału synchronizującego, zieloną początek kodu nadawanego przez pilota, natomiast fioletowymi punkty w których sygnał jest próbkowany.



Zdjęcie 6: Proces próbkowania - zbliżenie na krótszy fragment sygnału

```
import numpy as np
from datetime import datetime
from record import current_time


def find_sync_symbol(data, start, lenght_of_pulse):
    # sync is one HIGH and 31 ZERO
    # one is already detected
    position = start + int(1.5 * lenght_of_pulse)
    isItSync = True
    for i in range(24):
        if position >= len(data):
            return False
        if data[position] != 0:
            return False

        position += lenght_of_pulse

    if isItSync:
        end = position
        while data[end] == 0:
            if (end >= len(data)):
                return False
            end += 1

        lenght_of_pulse = (end-start)/(31+1)

    return isItSync, lenght_of_pulse


def find_first_impulse(data, start=0):
    ones = np.where(data == 1)[0]
    if (start >= len(ones)):
        return False
    lenght_of_conseq = start + 1
    last = ones[start]
    while ones[lenght_of_conseq] == last+lenght_of_conseq-start:
        lenght_of_conseq += 1
        if lenght_of_conseq >= len(ones):
            break

    return lenght_of_conseq - start, ones[start]


def main(signal):
    ones = []
    sampled_data_list = []
    n = 0
```

```
    while n < len(signal):
        if not type(find_first_impulse(signal, n)) == tuple:
            break
        length_of_pulse, position = find_first_impulse(signal, n)

        if type(find_sync_symbol(signal, position, length_of_pulse)) == tuple:
            print("timestamp:", str(current_time()), "sync found")

            length_of_pulse_synced = find_sync_symbol(
                signal, position, length_of_pulse)[1]
            #print(length_of_pulse, length_of_pulse_synced)
            #length_of_pulse = (length_of_pulse_synced*31 + length_of_pulse)//32
            length_of_pulse = length_of_pulse_synced
            start = position+(32*length_of_pulse_synced)

            start += 0.5 * length_of_pulse
            sampled_data = ""
            for i in range(96):
                index = int(start + (i * length_of_pulse_synced))
                if index >= len(signal):
                    return False
                sampled_data += str(int(signal[index]))

            sampled_data_list.append(sampled_data)
            break

        ones.append(position)
        n += int(length_of_pulse)

    return sampled_data_list


if __name__ == "__main__":
    sample_data = np.genfromtxt('../demodulated_signal.csv', delimiter=',')
    sampled_signal = main(sample_data)

    print(sampled_signal)
```

Plik sample.py

2.4. decode.py

`decode.py` to moduł dekodujący sygnał. Wykorzystuje on zależność, że w nadawany kod składa się z sekwencji `1000` i `1110` następnie każdemu z nich przypisywana jest wartość odpowiednio 0 i 1. Powstała sekwencja z postaci binarnej zamieniana jest na liczbę całkowitą.

Funkcja `decode()` dzieli sygnał wejściowy na podciągi o długości n=4 i porównuje każdy z nich do słownika `decoding`, który określa, jakie wartości binarne odpowiadają poszczególnym znakom (0 i 1). Jeśli podciąg nie znajduje się w słowniku, funkcja drukuje komunikat o błędzie. W przeciwnym razie funkcja konwertuje otrzymany ciąg binarny na liczbę całkowitą i zwraca ją jako wynik.

W warunku `if __name__ == "__main__":` program testuje funkcję `decode()` na przykładowym sygnale binarnym i sprawdza, czy otrzymana liczba całkowita jest poprawna, to znaczy czy pasuje do wyniku otrzymanego za pomocą metody wykorzystanej w pierwszym semestrze do zbudowania pierwszego prototypu (Arduino + odbiornik radiowy 433 MHz). Program kończy się wydrukowaniem komunikatu `"Test passed. Signal succesfully decoded."` lub komunikatu o błędzie. Wprowadzono również możliwość testowania funkcją `decode()` listy sygnałów binarnych `signal2`.

```
verbose = False
n = 4


def decode(signal):
    signal_divided = [signal[i:i+n] for i in range(0, len(signal), n)]

    # słownik
    decoding = {
        "1000": "0",
        "1110": "1"
    }

    received_code = ""

    for sign in signal_divided:
        if not sign in decoding:
            if verbose:
                print("error", sign)
```

```
                else:
                    return False
            else:
                received_code += decoding[sign]

    if verbose:
        print(received_code)

    dec_code = int(received_code, 2)

    if verbose:
        print(dec_code)

    return received_code, dec_code


if __name__ == "__main__":
    print("atempting internal module test")
    signal = "1000100010001110100011101000100010001110100011101000111010001110100010001110111010001000111011110"
    print("passing sample data")

    decoded_signal = decode(signal)

    if decoded_signal == 1332531:
        print("Test passed. Signal succesfully decoded.")
        print("Output:", decoded_signal)

    signal2 = ['1000100010001110100011101000100010001110100011101000111010001110111011101000100010001000111011110', '0100010000100011101000111
               '1000100010001110100011101000100010001110100011101000111010001110111011101000100010001000111011110', '0100010001100011101000111
    for signal in signal2:
        decoded_signal2 = decode(signal)

        if decoded_signal2 != 1332531:
            print("Test passed. Signal succesfully decoded.")
            print("Output:", decoded_signal2)
```

Plik decode.py

2.5. gpio_handler.py

Jest to moduł odpowiadający za inicjowanie działania diody i przycisków składających się na nasz produkt. To on wywoływał podświetlenie diody na dany kolor. Biały kolor oznaczał gotowość do wykonania działania. Kolor czerwony natomiast podświetlał się, gdy sygnał był nagrywany. Żółty kolor informował o decymacji nagranego sygnału, a kolor cyjanowy oznaczał trwającą demodulację. Niebieski kolor za to sygnalizował próbkowanie sygnału. Kolor fioletowy natomiast oznaczał dekodowanie. Ostatnim kolorem był kolor zielony, ktory podświetlał się, gdy sygnał był już nagrany i użytkownik wciskał przycisk, wywołując nadawanie sygnału.

```
import RPi.GPIO as GPIO

GPIO.setwarnings(False)
GPIO.setmode(GPIO.BOARD)

TRANSMIT_PIN = 37
BUTTON_1_PIN = 3
BUTTON_2_PIN = 7

RED_LED = 15
GREEN_LED = 16
BLUE_LED = 33

GPIO.setup(BUTTON_1_PIN, GPIO.IN, GPIO.PUD_UP)
GPIO.setup(BUTTON_2_PIN, GPIO.IN, GPIO.PUD_UP)

GPIO.setup(RED_LED, GPIO.OUT)
GPIO.setup(GREEN_LED, GPIO.OUT)
GPIO.setup(BLUE_LED, GPIO.OUT)

def led(r, g, b):
    if r == 1:
        GPIO.output(RED_LED, GPIO.HIGH)
    else:
        GPIO.output(RED_LED, GPIO.LOW)
    if g == 1:
        GPIO.output(GREEN_LED, GPIO.HIGH)
    else:
        GPIO.output(GREEN_LED, GPIO.LOW)
    if b == 1:
        GPIO.output(BLUE_LED, GPIO.HIGH)
```

```
    else:
        GPIO.output(BLUE_LED, GPIO.LOW)
```

Plik gpio_handler.py

2.6. send.py

Program odpowiada za kontrolowanie modułu nadajnika radiowego zmieniając stan jednego z pinów. Zawiera funkcję przyjmującą ciąg bitów, który następnie przetwarzany jest na impulsy o odpowiedniej długości.

```
# −*− coding: utf−8 −*−
import time
import RPi.GPIO as GPIO

GPIO.setwarnings(False)
GPIO.setmode(GPIO.BOARD)

# Ustawienie pinu do kontroli nadajnika FS1000A
transmit_pin = 37
GPIO.setup(transmit_pin, GPIO.OUT)

# Sekwencja bitow do wyslania 1332675
sequence2_on = '000101000101010110000110' # gniazdko 2 wlaczenie
sequence1_on = '000101000101010100110011O' # gniazdko 1
sequence1_off = '000101000101010100111000'

# Ustawienie parametrow czasowych
zero_delay = 0.00016*0.66 # Dlugosc czasu trwania bitu 0
one_delay = 0.0005*0.66 # Dlugosc czasu trwania bitu 1
gap_zero = 0.00054*0.66 # Dlugosc czasu przerwy miedzy bitami
gap_one = 0.0002*0.66
gap_signal = 0.0053*0.66


def send_bit(bit):
    if bit == '0':
        GPIO.output(transmit_pin, GPIO.HIGH)
        time.sleep(zero_delay)
        GPIO.output(transmit_pin, GPIO.LOW)
        time.sleep(gap_zero)

    elif bit == '1':
        GPIO.output(transmit_pin, GPIO.HIGH)
        time.sleep(one_delay)
        GPIO.output(transmit_pin, GPIO.LOW)
        time.sleep(gap_one)


def sendMain(sequence):
    i = 0
    while i < 15:
        for bit in sequence:
            send_bit(bit)
        time.sleep(gap_signal)
        i += 1
    GPIO.cleanup()


if __name__ == "__main__":
    # Wysylanie sekwencji bitow
    i = 0
    while i < 15:
        for bit in sequence1_off:
            send_bit(bit)
        time.sleep(gap_signal)
        i += 1
    GPIO.cleanup()
```

Plik send.py

2.7. main.py

Plik `main.py` to głowny człon programu z jego poziomu wyzwala się większość funkcji (zatomizowany przebieg procesu nagrywania/przetwarzania i nadawania) jak i obsługuje komunikację z użytkownikiem poprzez wykrywanie zdarzeń wciśnięcia przycisków i kontrolę stanu diody.

```
import time, os
from gpio_handler import *
from sample import main as sampleMain
from decode import decode as decodeMain
from demodulate import main as demodulateMain
from send import sendMain
from record import record, initialize
from record import decimate
from time import sleep

ispulsing = False
sequence = '000101000101010101001111000'
initialize()
os.system('clear')
if __name__ == "__main__":
    print("program ready")
    while True:
        try:
            led(1, 1, 1)
            if not GPIO.input(BUTTON_2_PIN):
                led(0, 1, 0)
                print("timestamp:", str(current_time()), "sending code:", sequence)
                sendMain(sequence)

            if not GPIO.input(BUTTON_1_PIN):
                led(1, 0, 0)
                print("timestamp:", str(current_time()),
                      "signal recording - start")
                recorded_data = record()
                print("timestamp:", str(current_time()),
                      "signal recording - end")

                led(1, 1, 0)
                print("timestamp:", str(current_time()), "decimating signal")
                decimated_data = decimate(recorded_data)

                led(0, 1, 1)
                print("timestamp:", str(current_time()), "demodulating data")
                demodulated_data = demodulateMain(decimated_data)

                led(0, 0, 1)
                print("timestamp:", str(current_time()), "sampling signal")
                sampled_data = sampleMain(demodulated_data)

                if sampled_data:
                    led(1, 0, 1)
                    print("timestamp:", str(current_time()), "decoding data")
                    decoded_code = decodeMain(sampled_data[0])

                    if decoded_code != False:
                        led(1,1,1)
                        sequence = decoded_code[0] + "0"
                        print("timestamp:", str(current_time()),
                              "output:", decoded_code)
                    else:
                        print("timestamp:", str(current_time()),
                              "nie znaleziono kodu pilota")
                else:
                    print("timestamp:", str(current_time()),
                          "nie znaleziono kodu pilota")

        except KeyboardInterrupt:
            led(0,0,0)
            print("The Program is terminated manually")
            raise SystemExit
```

Plik main.py