# Propuh Pro - Tehnička dokumentacija

Za potrebe detaljnog tehničkog razumijevanja, ovdje su ponuđeni implementacijski detalji i motivacije za rješenja važnih problema u procesu razvoja ovog sistema.

U nastavku su ponuđeni detalji tehničkim licima potrebni za održavanje i nadogradnju postojećih mogućnosti sistema.

# Kontrolni uređaj

## main.py

Zadatak je napraviti korisnički interfejs gdje je moguće postaviti željenu temperaturu i brzinu ventilatora za sistem ventilacije. Izgled interfejsa prošao je više iteracija. Originalna zamisao je bila interakcija sa dva moda ekrana - operacioni i konfiguracijski mod. Međutim, pokazalo se da je bolje imati ekran za konfiguraciju svake od stavki. Osnovne stavke koje korisnik konfiguriše su:

- *Željena temperatura*
- *Kritična temperatura*
- *Brzina ventilatora* Pored pogleda na ove tri vrijednosti, postoji i pogled na *mjerenu temperaturu*.

### Display

Za display korišten je LCD sa 16 kolona i 2 reda sa I2C modulom za jednostavniju implementaciju uz pomoć biblioteke RPI-PICO-I2C-LCD. Ova biblioteka dosta pojednostavi rad sa pomenutim LCD ekranom ali se pokazala kao nedovoljno responzivna za češće osvježavanje ekrana. Konfiguracija dimenzija i izlaznih pinova displeja je data ispod.

```
from lcd_api import LcdApi
from pico_i2c_lcd import I2cLcd

# Display konfiguracija
I2C_ADDR = 0x27
I2C_NUM_ROWS = 2
I2C_NUM_COLS = 16

I2C_BUS = I2C(1, sda=Pin(26), scl=Pin(27), freq=400000)
LCD_DISPLAY = I2cLcd(I2C_BUS, I2C_ADDR, I2C_NUM_ROWS, I2C_NUM_COLS)
```

Sve interakcije sa ekranom, kao što je postavljanje ili uklanjanje sadržaja izvode se metodama *LCD_DISPLAY* objekta.

## Pogledi

Za rad sa modovima ekrana, uvedena je klasa InterfaceMode koja definiše dozvoljene modove ili poglede na informacije. Mogući modovi interfejsa su:

- Current temp - zadnja izmjerena temperatura
- Target temp - željena temperatura
- Critical temp - temperatura pri kojoj se pali alarm
- Fan mode - mod ventilatora

Svaki od navedenih modova ima prateći pogled. U prvom redu ekrana ispisan je naziv pogleda dok je u drugom redu ispisana vrijednost.

Za navigaciju između modova i izmjene vrijednosti, koriste se tasteri. Ispis obrađuje funkcija *print_configuration()* koja se poziva nakon svakog klika na taster.

## Tasteri

Tasteri dostupni korisniku su:

- NEXT_MODE_BUTTON - GPIO16
- PREVIOUS_MODE_BUTTON - GPIO19
- INCREASE_BUTTON - GPIO17
- DECREASE_BUTTON - GPIO18

Ovim tasterima korisnik može mijenjati između četri pogleda te povećati i smanjivati vrijednosti u koracima. Nad svakim od tastera definisan je prekid koji poziva funkciju koja obavlja zadatak tog tastera.

```
# Postavljanje hardverskih prekida
NEXT_MODE_BUTTON.irq(handler=next_mode, trigger=Pin.IRQ_RISING)
INCREASE_BUTTON.irq(handler=increase_value, trigger=Pin.IRQ_RISING)
DECREASE_BUTTON.irq(handler=decrease_value, trigger=Pin.IRQ_RISING)
PREVIOUS_MODE_BUTTON.irq(handler=previous_mode, trigger=Pin.IRQ_RISING)
```

### Debouncing

Kako se za konfiguraciju vrijednosti koriste fizički tasteri, pogodno je uvesti funkciju za *debouncing*.

```
debounce = 0
def debouncing():
    global debounce
    if ticks_diff(ticks_ms(), debounce) < DEBOUNCE_TIME_MS:
        return False
    else:
        debounce = ticks_ms()
        return True
```

Konstanta *DEBOUNCE_TIME_MS* određuje period na koji se blokira ponovni unos sa tastera. Vrijednost od 300ms je odabrana nakon više pokušaja upotrebe i dovoljno je kratka da ne dozvoljava bouncing bez značajno utjecaja na korisničko iskustvo.

**Korak izmjene**

Korak izmjene temperature je 0.5°C. Ova vrijednost za korak je odabrana jer predstavlja dobar kompromis između tačnosti senzora temperature i mogućnosti odabranog ventilatora. Obzirom da vrijednosti koje se dostave sa terenskog uređaja nisu nužno zaokružene na najbližu polovinu stepena, za to se ovdje koristi funkcija:

```
def round_to_nearest_half(value) -> float:
    return round(value * 2) / 2
```

LM35 - korišteni senzor, ima grešku reda 0.5°C pri 25°C.

## MQTT Komunikacija

Komunikacija sa terenskim uređajem postiže se putem MQTT brokera. Ovo se pokazalo kao veoma jednostavan pristup rješavanju problema komunikacije više uređaja.

**WiFi konektivnost**

Kako je za MQTT potrebna internet konekcija, potrebno je vršiti spajanje na WiFi mrežu. WiFI SSID i lozinka se konfigurišu u kodu na sljedeći način:

```
import network

# WiFi konfiguracija
WIFI_SSID = "naziv_mreze"
WIFI_PASSWORD = "password1234"

# Povezivanje na internet
print("Connecting to WiFi: ", WIFI_SSID)
WIFI = network.WLAN(network.STA_IF)
WIFI.active(True)
WIFI.config(pm=0xA11140)
WIFI.connect(WIFI_SSID, WIFI_PASSWORD)

LCD_DISPLAY.putstr("Connecting...")

while not WIFI.isconnected():
    pass

print("Connected to network!")
print("IP address:", WIFI.ifconfig()[0])
```

Obzirom da je sva ostala funkcionalnost uređaja ovisna o internet konekciji, ekran i sve ostale funkcionalnosti su blokirane dok se ne uspostavi WiFi konekcija sa datom mrežom. Na ekranu ostaje ispisan tekst "Connecting..." sve dok se konekcija uspostavlja. Nakon spajanja, u konzoli se ispisuje poruka o uspjehu i lokalna IP adresa.

**MQTT Broker i teme**

MQTT servis postavljen je kako slijedi:

```
import simple
# MQTT konfiguracija
MQTT_SERVER = "broker.hivemq.com"
MQTT_CLIENT_NAME = "Propuh-Pro-Control"

MQTT_TOPIC_TARGET_TEMP = b"Propuh-Pro/target_temp"
MQTT_TOPIC_CRITICAL_TEMP = b"Propuh-Pro/critical_temp"
MQTT_TOPIC_FAN_MODE = b"Propuh-Pro/fan_mode"
MQTT_TOPIC_MEASURED_TEMP = b"Propuh-Pro/measured_temp"

# Povezivanje na MQTT broker
CLIENT = simple.MQTTClient(client_id=MQTT_CLIENT_NAME, server=MQTT_SERVER,
port=1883)
CLIENT.connect()
```

Za svaku od četri stavke, postavljena je tema na MQTT brokeru *broker.himemq.com*.

*target_temp*, *critical_temp* i *measured_temp* predstavljeni su kao realni brojevi dok *fan_mode* uzima jednu od 5 dozvojenih vrijednosti definisanih u FanMode klasi.

Postavljen je klijent MQTT serivsa putem kojeg se šalju i dobavljaju vrijednosti za navedene teme.

**Slanje podataka**

Ovaj uređaj štalje vrijednosti željene i kritične temperature te mod ventilatora. Nakon bilo koje promjene vrijednosti od strane korisnika, postavlja se timer na 3 sekunde. Nakon 3 sekunde, poziva se funkcija *send_data* koja na broker postavlja navedene vrijednosti.

Ova pauza od 3 sekunde pokazala se kao pogodan pristup za slanje podataka. Obzirom da se ove vrijednosti u praksi ne mijenjaju isuviše često, nije očigledna potreba za periodičnim slanjem.

```
# Slanje podataka putem MQTT
def send_data(timer):
    publish = str(fan_mode.get_mode())
    CLIENT.publish(MQTT_TOPIC_FAN_MODE, publish)

    publish = str(target_temp)
    CLIENT.publish(MQTT_TOPIC_TARGET_TEMP, publish)

    publish = str(critical_temp)
    CLIENT.publish(MQTT_TOPIC_CRITICAL_TEMP, publish)

    print("Sent!")
```

**Dobavljanje podataka**

Podaci sa MQTT brokera se dobavljaju periodično. Funkcija *recieve_data* se poziva nakon svake sekunde i dobavlja vrijednost mjerene temperature i mod ventilatora koji je potencijalno postavljen na moblinom uređaju. Kada se ustanovi da je postavljena nova vrijednost na brokeru *check_msg* metodom MQTT klijenta, poziva se *custom_dispatcher* koji tumači dostavljenu poruku na osnovu teme.

```
# Filtriranje primljenih poruka
def custom_dispatcher(topic, msg):

    if topic == MQTT_TOPIC_MEASURED_TEMP:
        message_arrived_measured_temp(topic, msg)
    elif topic == MQTT_TOPIC_FAN_MODE:
        message_arrived_fan_mode(topic, msg)

CLIENT.set_callback(custom_dispatcher)

# Pretplata na teme
CLIENT.subscribe(MQTT_TOPIC_MEASURED_TEMP)
CLIENT.subscribe(MQTT_TOPIC_FAN_MODE)

# Provjera pristiglih podataka na MQTT
def recieve_data(timer):
    CLIENT.check_msg()
    CLIENT.check_msg()

recieve_DATA_TIMER = Timer(period=1000, mode=Timer.PERIODIC,
callback=recieve_data)
```

## Rad sa temperaturama

Za svaku od tema uvedena je varijabla koja će čuvati odgovarajuću vrijednost na uređaju kako bi se njima manipulisalo u ostatku programa.

```
interface_mode = InterfaceMode()
fan_mode = FanMode()
current_temp = 0.0
target_temp = 21.0
critical_temp = 35.0
```

Navedene početne vrijednosti za *target_temp* i *critical_temp* su odabrane iz razloga što predstavljaju uobičajen opseg temperatura za stambene prostore, mada nema razloga zašto ne mogu biti u bilo kojem drugom opsegu.

Uvedeno je ograničenje oko minimalne razlike kritične i željene temperature. Najmanja razlika je 5.0°C. Ovo je vrijednost za koju možemo pouzdano tvrditi da neće doći do alarma isključivo zbog greške u mjerenju obzirom da korišteni senzor (LM35) ima grešku reda 0.5°C pri 25°C. Ova minimalna razlika se može posebno konfigurisati putem izdvojene konstante.

```
# Dozvoljena razlika željene i kritične temperature
MINIMUM_TEMP_DIFFERENCE = 5.0
```

**Alarm**

Kada mjerena temperatura dosegne kritičnu temperaturu, dolazi do stanja alarma. Kontrolni uređaj na ekranu ispisuje posebnu treptajuću poruku s ciljem da privuće pažnju i upozori na situaciju. Za to vrijeme, neće se ispisivati standardni pogledi. Poruka o alarmu se može skloniti tasterima za promjenu pogleda. Za potrebe implementacije alarm pogleda, koristi se varibajla *alarm_now*. Varijabla *alarm* služi kao *flag* da se dogodilo prekoračenje temperature što trenutno nema drugih koristi ali otvara mogućnost za neku vrstu dijagnostike u budućnosti.

```
# Da li se u toku rada programa pojavio alarm
alarm = False
# Da li je trenutno aktivan alarm
alarm_now = False
```

# Terenski uređaj

## main.py

Zadatak je mjeriti i dostavljati temperaturu okruženja, upravljati brzinom ventilatora i signalizirati trenutno stanje. Konfiguracija ovog uređaja se dobavlja sa MQTT brokera.

### Mjerenje temperature

Temperatura se mjeri LM35 senzorom koji ima grešku reda 0.5°C pri 25°C. Ovaj senzor je odabran zbog jednostavnosti implementacije i korištenja. U opticaju je bio i DHT11 ali ne nudi dovoljnu preciznost i stabilnost u mjerenjima koja je potrebna za ovakav sistem.

Za konkretan LM35 potrebno je odrediti kalibracijsko offset kako bi mjerenje bilo reprezentativno.

Kako očekujemo da se temperatura prostora neće značajno mijenjati u kratkom periodu vremena, korišteno je periodično uzorkovanje od 10 uzoraka. Na osnovu prosječne vrijednosti uzoraka se određuje temperatura. Kalibracijski offset, broj uzoraka i ulazni pin sa A/D konverzijom se konfigurišu u kodu konstantama navedenim ispod

```
# LM35 konfiguracija
LM35_CALIBRATION_OFFSET = -1200
LM35_NUMBER_OF_SAMPLES = 10
LM35_SENSOR_PIN = ADC(Pin(28))
```

### Upravljanje ventilatorom

Ventilator se upravlja PWM signalom sa frekvencijom 500Hz. Obzirom da vrijeme za koje se mijenja brzina ventilatora nije od krucijalnog značaja, frekvencija PWM signala ne mora biti izrazito velika.

```
FAN_PWM = PWM(Pin(22))
FAN_PWM.freq(500)
```

Tačan proračun brzine rada ventilatora provodi objekt klase FanSpeedController koja sadrži svu potrebnu logiku proračuna.

```
fan_controller = FanSpeedController(22, 30, 22)
```

Detaljnije informacije o klasi FanSpeedController ponuđene su u nastavku (Pomoćne klase / FanSpeedController).

Za početne vrijednosti pri instanciranju odabrane su temperature 22°C za željenu, 30°C za kritičnu i 22°C za trenutnu. Ove vrijedonsti se ponovo postavljaju prvim dobavljanjem podataka sa MQTT servisa.

**VU Metar**

Brzinu i mod ventilatora moguće je vizuleno utvrditi na osnovu šest LED-ova koji svijetle proporcionalno brzini ventilatora

```
FAN_VU_METER_LEDS = BusOut([4, 5, 6, 7, 8, 9])

# 0 (OFF)       -> 0 upaljenih LED
# 1 (SLOW)      -> 2 upaljene LED
# 2 (NORMAL)    -> 4 upaljene LED
# 3 (FAST)      -> 6 upaljenih LED
```

## Alarm

U slučaju da FanSpeedControler odredi da je trenutna temperatura previsoka, diže se alarm. U slučaju alarma, treperi LED i pali se zujalica kako bi se vizuelno i zvukom privukla pažnja korisnika.

Zujanje alarma se može ugasiti tasterom na uređaju, što omogućuje neometano servisiranje koje je eventualno potrebno.

```
OVERHEATING_LED = Pin(11, Pin.OUT)

ALARM_PWM = PWM(Pin(27))
ALARM_PWM.freq(1000)

ALARM_OFF_BUTTON = Pin(0, Pin.IN)

# Klikom na taster se gasi alarm
ALARM_OFF_BUTTON.irq(handler=turn_alarm_off, trigger=Pin.IRQ_RISING)
```

## MQTT Konektivnost

### WiFi konektivnost

Identično kao i na kontrolnom uređaju, prvo se uspostavlja WiFi konekcija.

```
# WiFi konfiguracija
WIFI_SSID = "naziv_mreze"
WIFI_PASSWORD = "password1234"

# Povezivanje na internet
print("Connecting to WiFi: ", WIFI_SSID)
WIFI = network.WLAN(network.STA_IF)
WIFI.active(True)
WIFI.config(pm=0xA11140)
WIFI.connect(WIFI_SSID, WIFI_PASSWORD)

while not WIFI.isconnected():
    pass

print("Connected to network!")
print("IP address:", WIFI.ifconfig()[0])
```

### MQTT Broker i teme

Broker i teme koje se koriste za komunikaciju su identične kao i na kontrolnom uređaju. Jedina razlika je u nazivu klijenta

```
# MQTT konfiguracija
MQTT_SERVER = "broker.hivemq.com"
MQTT_CLIENT_NAME = "Propuh-Pro-Teren"

MQTT_TOPIC_TARGET_TEMP = b"Propuh-Pro/target_temp"
MQTT_TOPIC_CRITICAL_TEMP = b"Propuh-Pro/critical_temp"
MQTT_TOPIC_FAN_MODE = b"Propuh-Pro/fan_mode"
MQTT_TOPIC_MEASURED_TEMP = b"Propuh-Pro/measured_temp"
```

**Slanje podataka**

Terenski uređaj periodično (svakih 0.5 sekundi) šalje isključivo izmjerenu temperaturu nakon svakog uzorkovanja.

```
# Povezivanje na MQTT broker
CLIENT = simple.MQTTClient(client_id=MQTT_CLIENT_NAME, server=MQTT_SERVER,
port=1883)
CLIENT.connect()

# Uzorkovanje i slanje izmjerene temperature
def check_temperature(t):
    ...
    # Uzorkovanje
    ...
    publish = str(measured_temp)
    CLIENT.publish(MQTT_TOPIC_MEASURED_TEMP, publish)

# Tajmer za očitavanje vrijednosti LM35
CHECK_TEMP_TIMER = Timer(period=500, mode=Timer.PERIODIC,
callback=check_temperature)
```

**Dohvaćanje podataka**

Podaci sa MQTT brokera se dobavljaju periodično. Funkcija *recieve_data* se poziva nakon svake sekunde i dobavlja vrijednost željene i kritične temperature i mod ventilatora koji je potencijalno postavljen na moblinom uređaju.

Kada se ustanovi da je postavljena nova vrijednost na brokeru *check_msg* metodom MQTT klijenta, poziva se *custom_dispatcher* koji tumači dostavljenu poruku na osnovu teme.

```
# Filtriranje primljenih poruka
def custom_dispatcher(topic, msg):

    if topic == MQTT_TOPIC_FAN_MODE:
        message_arrived_fan_mode(topic, msg)
    elif topic == MQTT_TOPIC_CRITICAL_TEMP:
        message_arrived_critical_temp(topic, msg)
    elif topic == MQTT_TOPIC_TARGET_TEMP:
        message_arrived_target_temp(topic, msg)

# Pretplata na teme
CLIENT.set_callback(custom_dispatcher)
CLIENT.subscribe(MQTT_TOPIC_FAN_MODE)
CLIENT.subscribe(MQTT_TOPIC_CRITICAL_TEMP)
CLIENT.subscribe(MQTT_TOPIC_TARGET_TEMP)

# Provjera pristiglih podataka na MQTT
def recieve_data(t):
    CLIENT.check_msg()
    CLIENT.check_msg()
    CLIENT.check_msg()

# Tajmer za primanje podataka
RECIEVE_DATA_TIMER = Timer(period=1000, mode=Timer.PERIODIC,
callback=recieve_data)
```

# Pomoćne klase

## InterfaceMode

InterfaceMode je pomoćna klasa koja enkapsulira broj i naziv modova ili pogleda na ekran. Ova klasa prati *State pattern* i omogućava promjenu u naredno ili prethodno stanje. Validna stanja su predstavljena kao cijeli brojevi pa se mogu međusobno porediti. Potreba za ovom klasom nastala je jer MicroPython ne poznaje enum klase.

Konstruktor ove klase može da primi jedan parametar - početni pogled, ali je podrazumijevana vrijednost TARGET_TEMP_CONFIG. Primjer instanciranja objekta je dat ispod.

```
interface_mode = InterfaceMode(InterfaceMode.FAN_CONFIG)
```

Stanja koja ova klasa poznaje su:

```
VALID_MODES = {
    TARGET_TEMP_CONFIG,
    CRITICAL_TEMP_CONFIG,
    FAN_CONFIG,
    OPERATIONAL,
}
MODE_NAMES = {
    TARGET_TEMP_CONFIG: "TARGET_TEMP_CONFIG",
    CRITICAL_TEMP_CONFIG: "CRITICAL_TEMP_CONFIG",
    FAN_CONFIG: "FAN_CONFIG",
    OPERATIONAL: "OPERATION",
}
```

Dostupne metode su:

```
def next(self)          # Naredni mod, ciklično
def previous(self)      # Prethodni mod, ciklično
def get_mode(self)      # Dohvati trenutni mod
def get_mode_name(self) # Dohvati naziv trenutnog moda
```

# FanMode

FanMode je također *State pattern* klasa ali za definisanje modova ventilatora. Tačan intenzitet rada ventilatora nije cilj ove klase. Zapravo, cilj ove klase je da se jasno definišu mogući modovi ventilatora i omogući jasna komunikacija. Kao i za InterfaceMode, stanja se interno čuvaju kao cijeli brojevi.

Konstruktor ove klase može da primi jedan parametar - početni mod, ali je podrazumijevana vrijednost OFF. Primjer instanciranja objekta je dat ispod.

```
fan_mode = FanMode(FanMode.AUTO)
```

Stanja koja ova klasa poznaje su:

```
MODE_NAMES = {
    OFF: "OFF",
    SLOW: "SLOW",
    MEDIUM: "MEDIUM",
    FAST: "FAST",
    AUTO: "AUTO",
}

VALID_MODES = {
    OFF,
    SLOW,
    MEDIUM,
    FAST,
    AUTO,
}
```

Dostupne metode su:

```
def next(self)          # Naredni mod, ciklično
def previous(self)      # Prethodni mod, ciklično
def get_mode(self)      # Dohvati trenutni mod
def get_mode_name(self) # Dohvati naziv trenutnog moda
```

# FanSpeedController

Za enkapsulaciju logike potrebne za određivanje brzine ventilatora na osnovu relevantnih parametara, kreirana je klasa FanSpeedController.

Parametri potrebni za rad ove klase su:

- target_temp - double
- critical_temp - double
- current_temp - double
- fan_mode - FanMode objekt

Ove vrijednosti se ujedno proslijeđuju konstruktoru u navedenom redoslijedu. Za fan_mode je podrazumijevana vrijednost FanMode(FanMode.OFF).

Dostupne metode su:

```
def update_current_speed(self)
# Proračun brzine na osnovu parametara

def set_fan_mode(self, new_fan_mode)
# Postavljanje jednog od modova za ventilator

def get_fan_mode(self)
# Dobavljanje trenutnog moda za ventilator

def set_current_temp(self, new_current_temp)
# Postavljanje trenutne temperature

def set_target_temp(self, new_target_temp)
# Postavljanje željene temperature

def set_critical_temp(self, new_critical_temp)
# Postavljanje kritične temperature

def is_alarm(self)
# Provjeri je li alarmantna temperatura

def get_speed_percent(self)
# Dohvati brzinu u procentima

def get_speed_u16(self)
# Dohvati temperaturu u opsegu [0, 65535]

def get_led_binary(self)
# Dohvati stanje LED-ova

def turn_alarm_off(self)
# Ugasi alarm
```

Opsezi brzine su:

```
# 0 (OFF)       -> 0%
# 1 (SLOW)      -> 50%
# 2 (NORMAL)    -> 75%
# 3 (FAST)      -> 100%
```

Za automatski mod, proračun se vrši po opsezima proračunatog procenta kao:

```
# < -0.1        -> OFF
# [-0.1, 0.05)  -> OFF/SLOW
# [0.05, 0.2)   -> SLOW
# [0.2, 0.3)    -> SLOW/NORMAL
# [0.3, 0.5)    -> NORMAL
# [0.5, 0.6)    -> NORMAL/FAST
# >= 0.6        -> FAST
```

## BusOut

Za potrebe rada sa nizom LED-ova, uvedena je klasa BusOut. BusOut omogućujue upravljanjem sa više digitalnih izlaza istovremeno tako što se proslijedi cijeli broj koji predstavlja željeno stanje izlaza. Primjer instanciranja i postavljanja vrijednosti:

```
bus_out = BusOut(Pin(16), Pin(17), Pin(18))
bus_out.setValue(0b101)
# Pin 16 i 18 su uključeni dok je Pin 17 isključen
```

Dostupne metode su:

```
def set_value(self, val)
# Postavlja zadatu vrijednost val (integer ili binarni broj)

def size(self)
# Dohvaća broj pinova okjima se upravlja

def get_value(self)
# Dohvaća trenutno postavljenu vrijednost

def shift_right(self, shift_amount=1)
# Pomjera vrijednost izlaza udesno za dati broj bita

def shift_left(self, shift_amount=1)
# Pomjera vrijednost izlaza ulijevo za dati broj bita
```