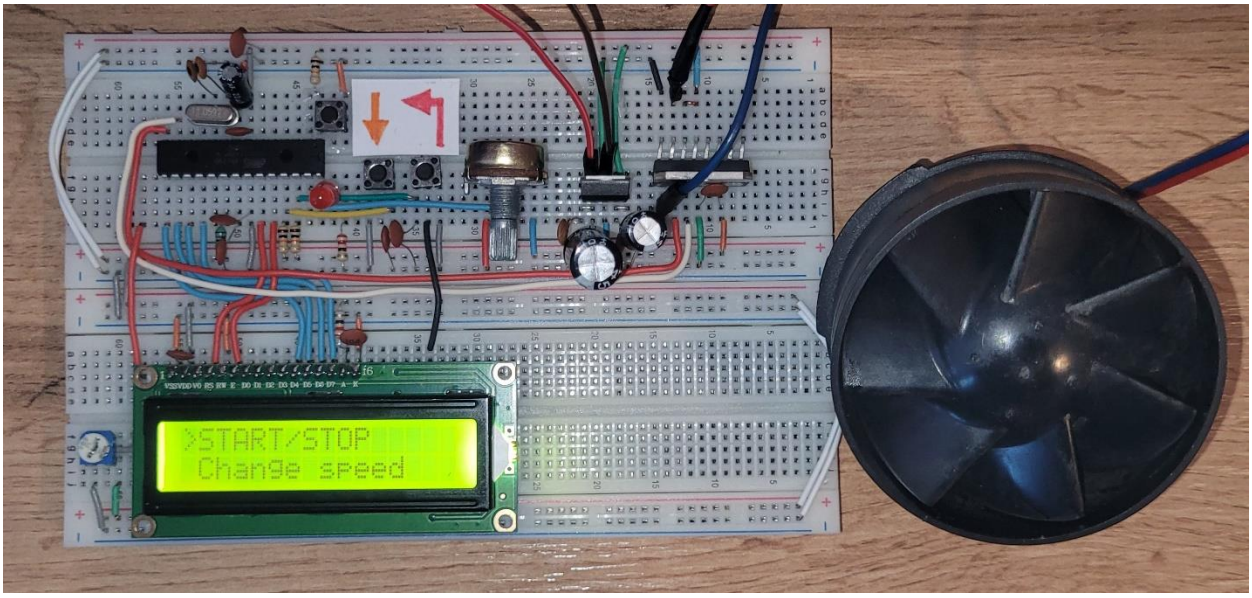
Układ sterowania wiatrakiem

Mikrokontrolery w Automatyce

2023/2024



Szymon Hrehorowicz

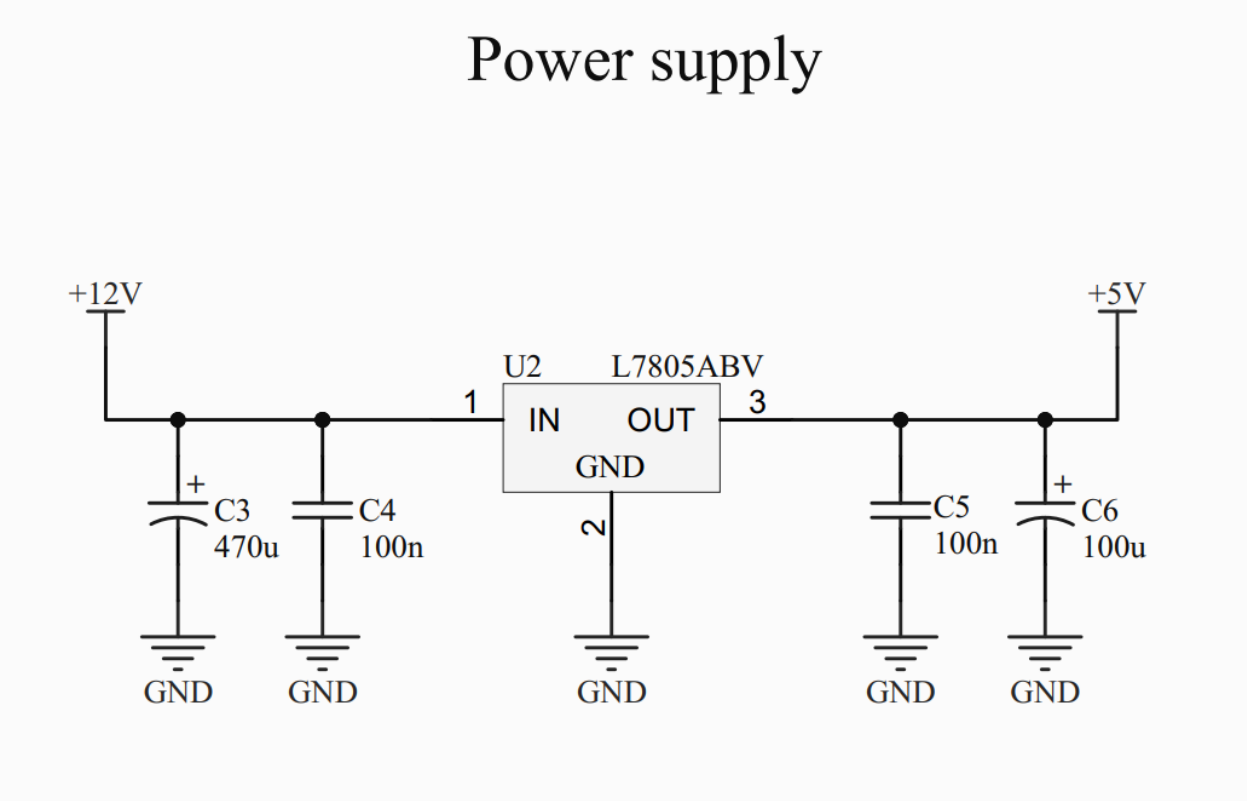
13A6 GP03

Opis projektu

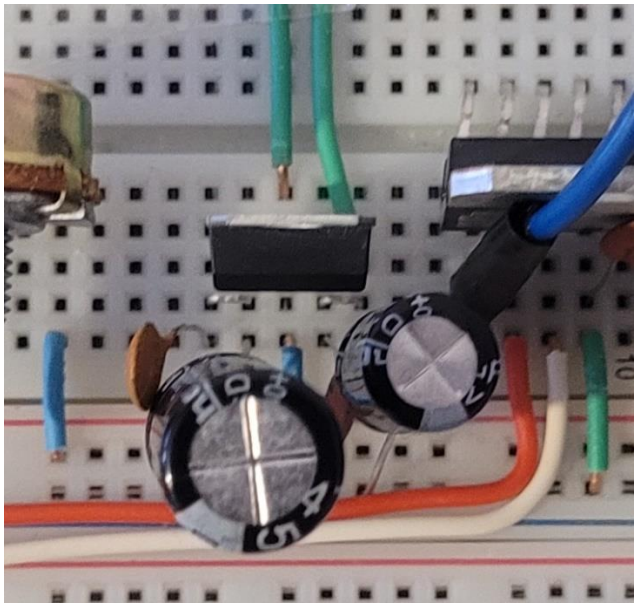
Zaprojektowany układ pozwala na sterowanie silnikiem prądu stałego. Wyposażony jest on w wyświetlacz LCD, parę przycisków oraz potencjometr będące odpowiednikiem panelu operatorskiego. Wyświetlacz pozwala na łatwą i przyjazną dla użytkownika obsługę silnika, tj. włączanie i wyłączanie maszyny, zmianę prędkości oraz kierunku jej obrotów. Układ posiada również możliwość komunikacji z komputerem za pośrednictwem protokołu USART. Komunikacja ta spełnia wszystkie funkcjonalności wcześniej opisanego panelu operatorskiego. Pozwala zatem na wdrożenie nadrzędnego systemu, który mógłby na odległość sterować silnikiem bądź grupą silników wyposażonych w zaprojektowany układ sterowania. Szczegóły implementacji poszczególnych elementów projektu zostały omówione w dalszych częściach sprawozdania.

Układ zasilania

Układ zasilany jest z 12 V zasilacza. Napięcie to jest obniżane do wartości 5 V za pomocą stabilizatora liniowego L7805ABV.



Rysunek 1 Schemat układu zasilania.



Rysunek 2 Fizyczna realizacja układu zasilania na płytce stykowej.

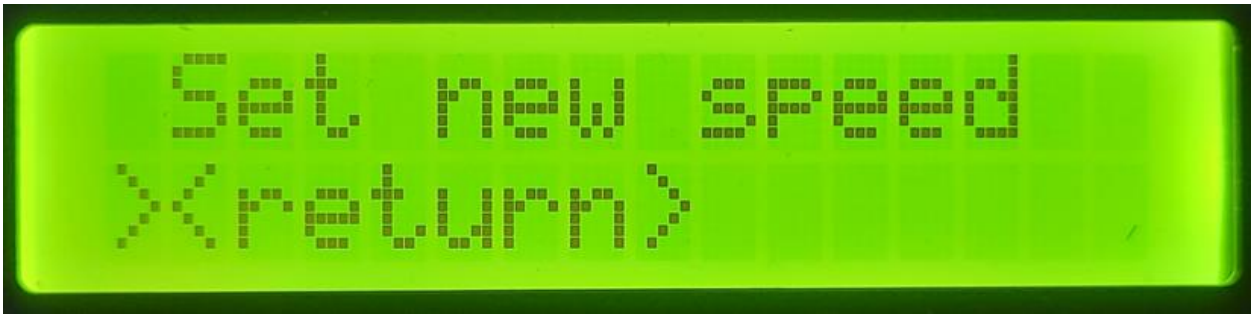
Panel operatorski

Panel operatorski składa się z dwóch części. Pierwsza z nich to wyświetlacz LCD 2x16. Na rysunkach 3-8 przedstawiono ekrany jakie układ oferuje użytkownikowi. Pozwalają one na uruchomienie i zatrzymanie silnika, sprawdzenie aktualnej nastawy prędkości obrotowej oraz jej zmianę a także na zmianę kierunku obrotów. Druga część panelu operatorskiego to dwa przyciski NO oraz potencjometr. Z ich pomocą, użytkownik porusza się po menu oraz zmienia wartość nastawy prędkości obrotowej. Realizację tej części przedstawiono na *Rys. 9*. Na tymże rysunku widać również czerwoną diodę LED. Informuje ona o ewentualnych błędach, które mogłyby wystąpić.

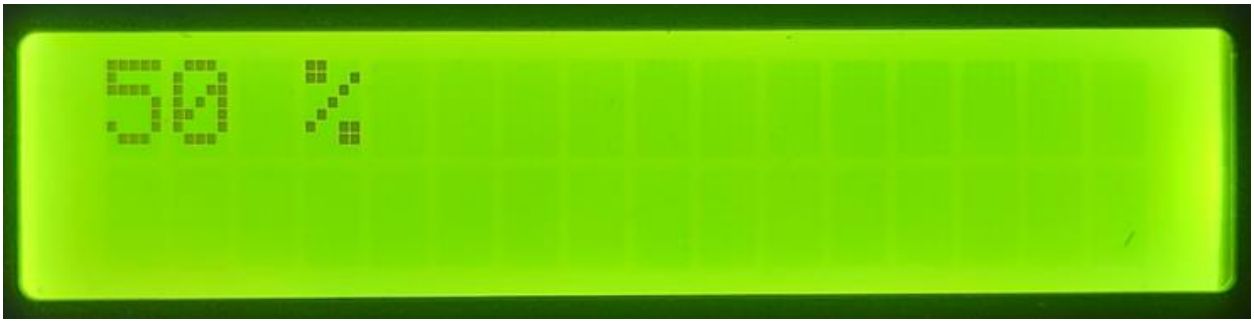
Rysunek 3 Pierwsza część menu głównego.

Rysunek 4 Druga część menu głównego.

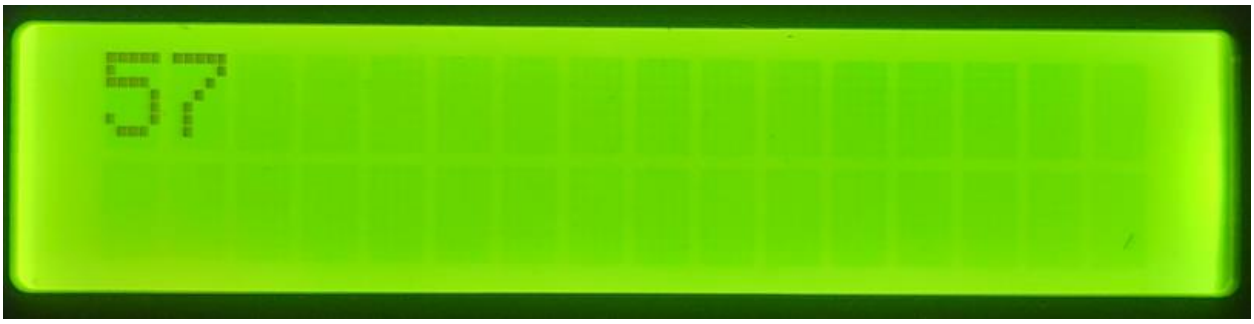
Rysunek 5 Pierwsza część podmenu sterowania prędkością obrotową.



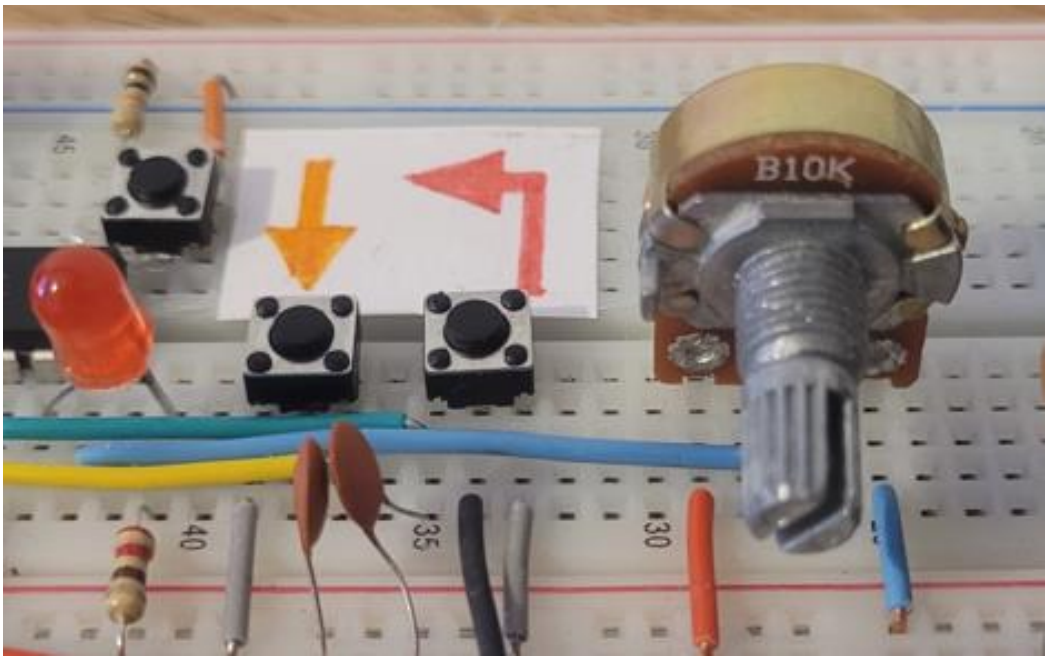
Rysunek 6 Druga część podmenu sterowania prędkością obrotową.



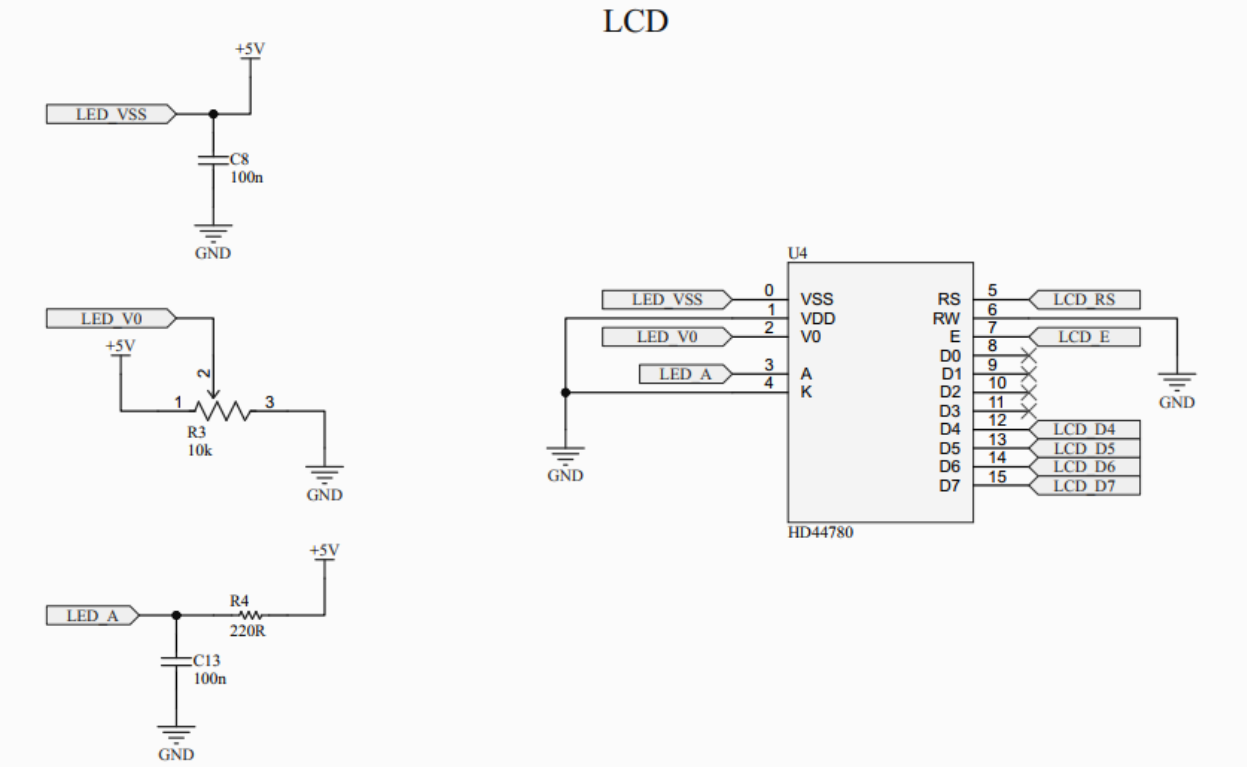
Rysunek 7 Ekran przedstawiający aktualną nastawę prędkości obrotowej (wyświetlany po wybraniu opcji „Current speed").



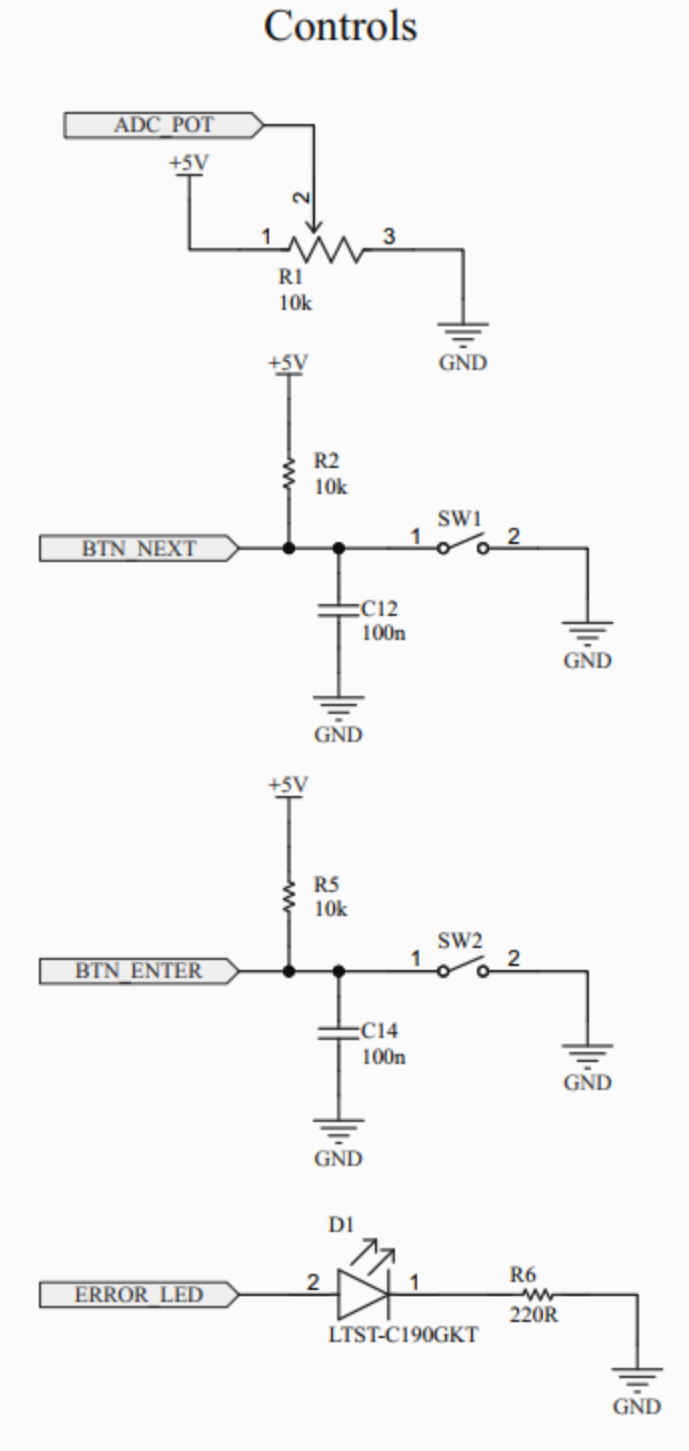
Rysunek 8 Ekran przedstawiający wybór nowej nastawy prędkości obrotowej. Na nim wyświetlana jest aktualnie wybrana, za pomocą potencjometru, wartość prędkości obrotowej (wyświetlany po wybraniu opcji „Set new speed").



Rysunek 9 Fizyczna realizacja układu obsługi panelu operatorskiego, zrealizowana na płytce stykowej. Przyciski odpowiadają kolejno (od lewej strony) na przemieszczenie się w dół menu oraz na wybór opcji bądź wejście w podmenu.



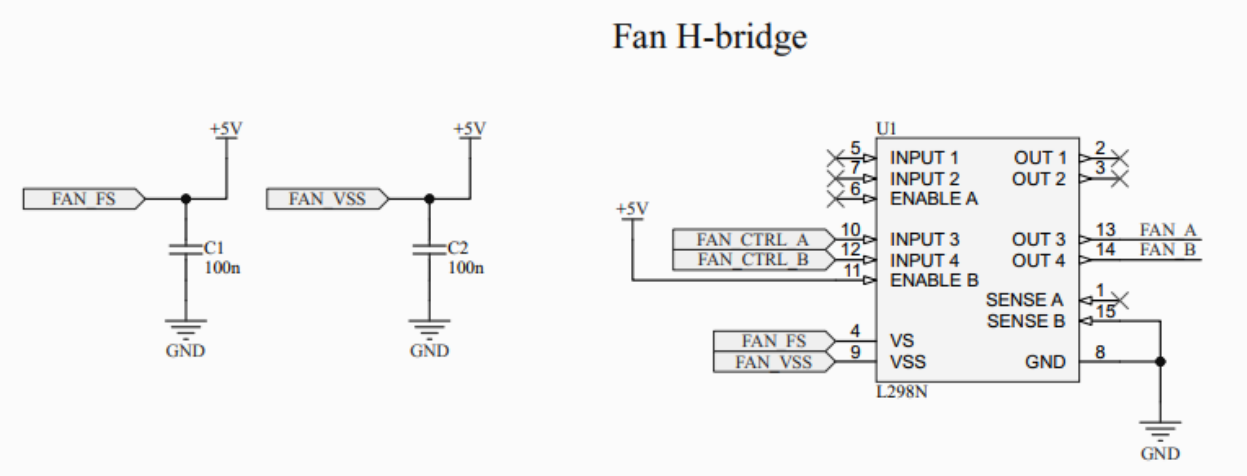
Rysunek 10 Schemat elektryczny układu wyświetlacza LCD.



Rysunek 11 Schemat elektryczny układu obsługi panelu operatorskiego oraz diody LED informującej o błędach.

Silnik

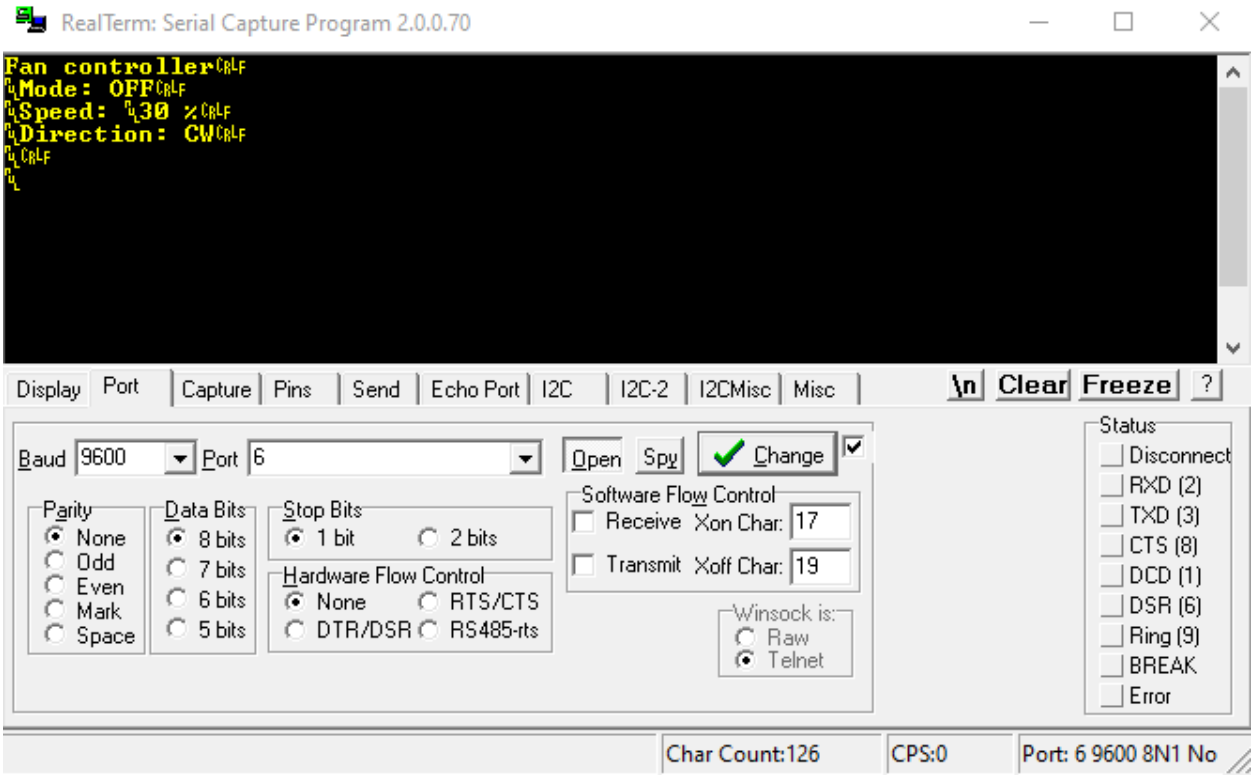
Silnik sterowany jest przy pomocy sygnałów PWM generowanych przez mikrokontroler. Sygnały te trafiają bezpośrednio do mostka H. Do realizacji mostka wykorzystano gotowy układ sclaony L298N.



Rysunek 12 Schemat elektryczny mostka H.

Komunikacja USART

Do komunikacji między urządzeniem a komputerem wykorzystano protokół USART. W *Tab. 1* przedstawiono polecenia, jakie można wysłać do urządzenia z poziomu komputera oraz oczekiwane odpowiedzi na nie.



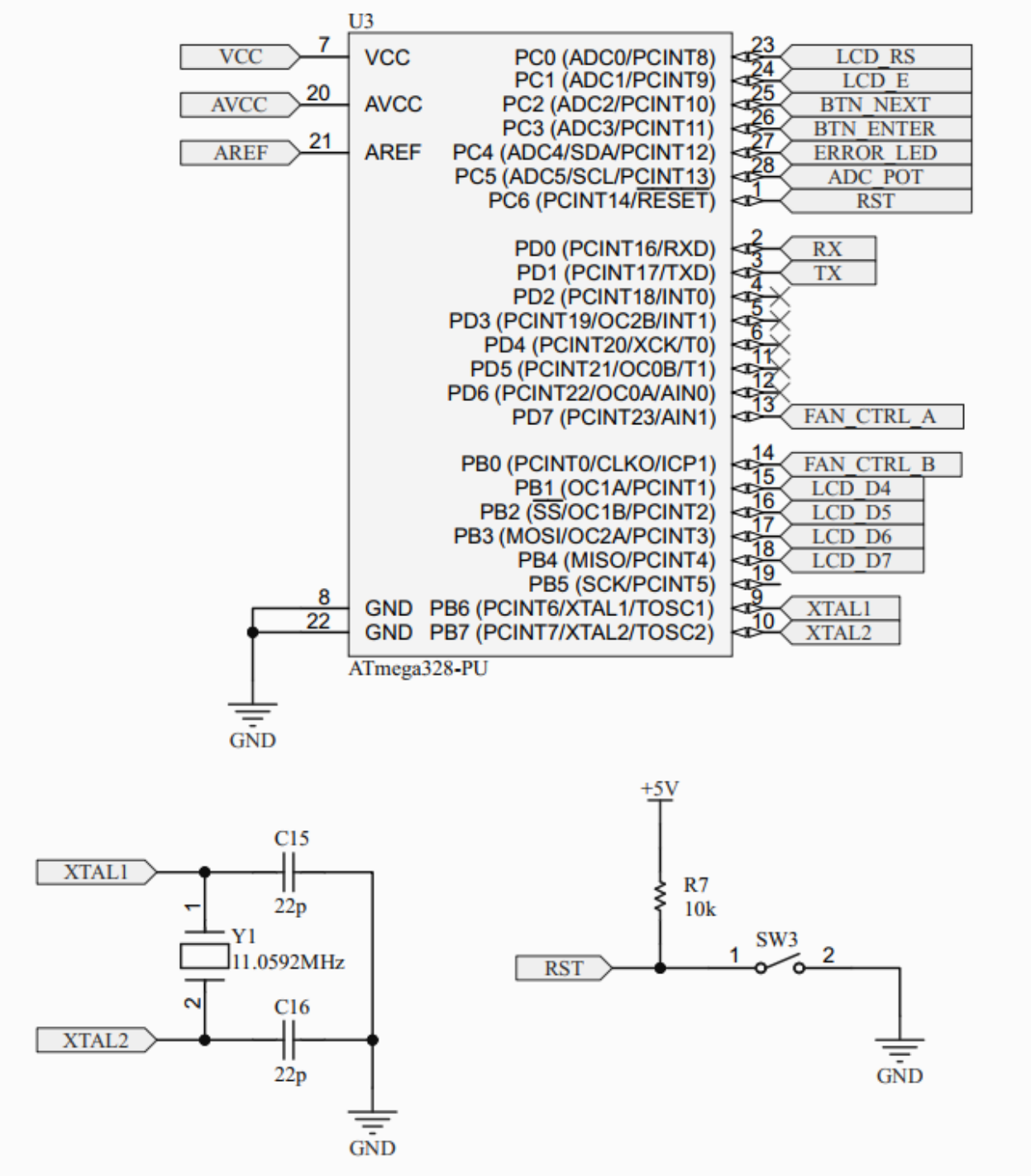
Rysunek 13 Odpowiedź układu na jedno z poleceń, w tym przypadku żądania statusu urządzenia.

Polecenie	Wartości *zmiennej*	Oczekiwana odpowiedź układu
s	-	Jak na *Rys. 13*.
v[*x*]	Wartość liczbowa z przedziału <0;100>	Zmiana wartości prędkości obrotowej na zdefiniowaną w *zmiennej*.
m[*x*]	**0** – wyłączenie wiatraka **1** – włączenie silnika	Zgodnie z wartościami *zmiennej*.
d[*x*]	**0** – kierunek zgodny z ruchem wskazówek zegara **1** – kierunek przeciwny do ruchu wskazówek zegara	Zgodnie z wartościami *zmiennej*.

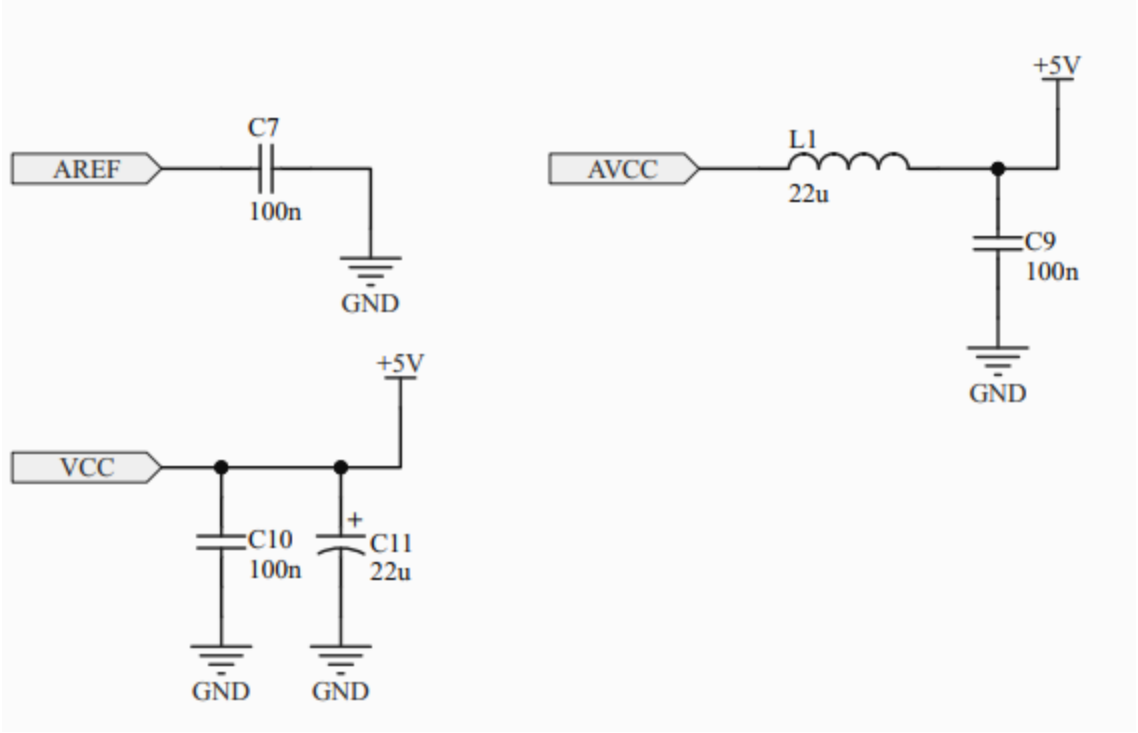
Tabela 1 Komendy, wykorzystywane w komunikacji USART.

Mikrokontroler

W projekcie wykorzystano mikrokontroler ATMega328P. Na kolejnych stronach przedstawiono kod źródłowy programu, jaki zaprogramowano urządzenie.

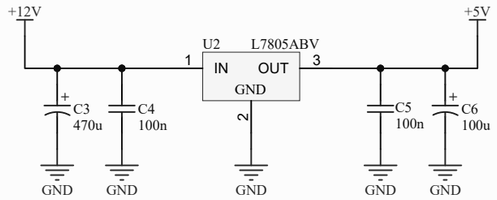


Rysunek 14 Schemat elektryczny mikrokontrolera ATMega328P wraz z jego zegarem oraz przyciskiem resetu.

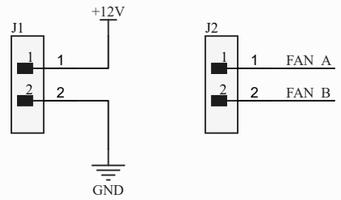


Rysunek 15 Schemat elektryczny filtracji zasilania mikrokontrolera.

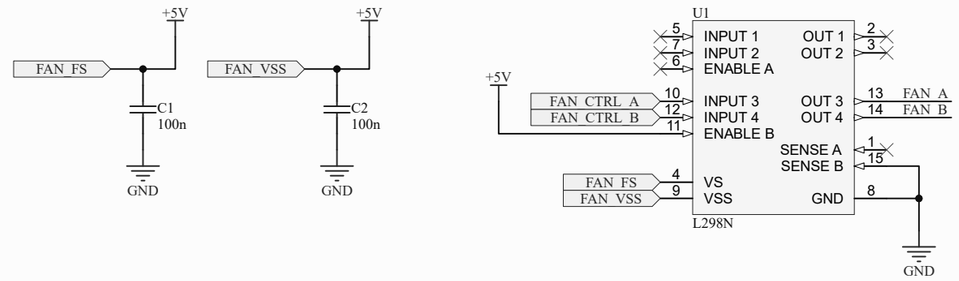
Power supply



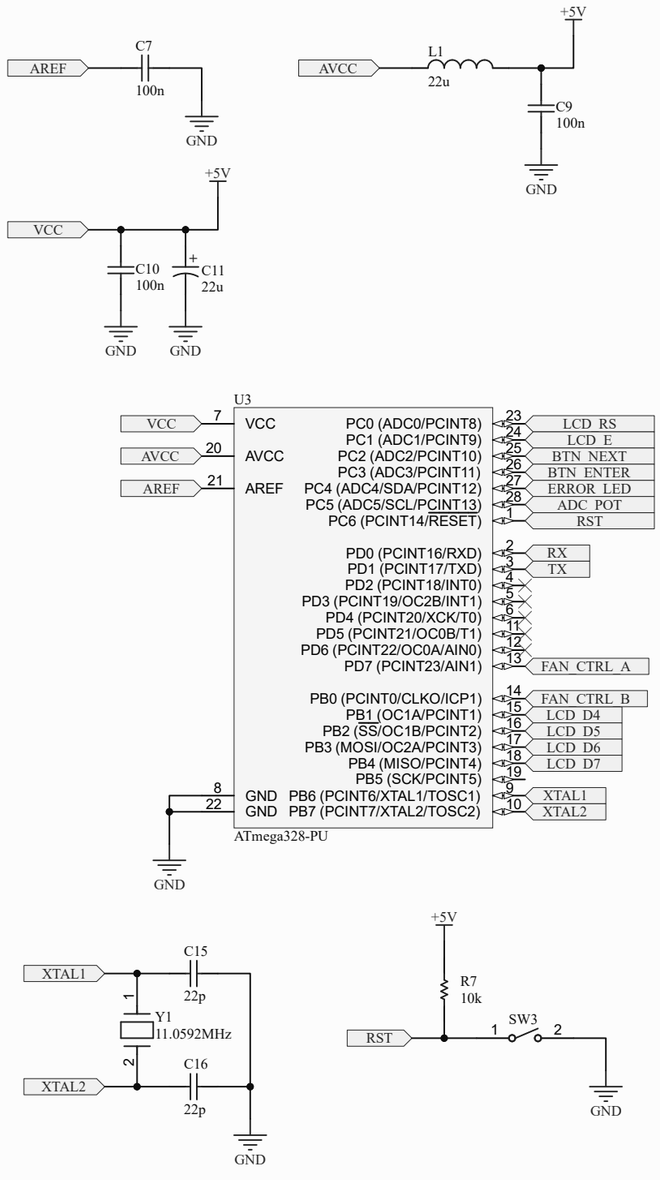
Connectors



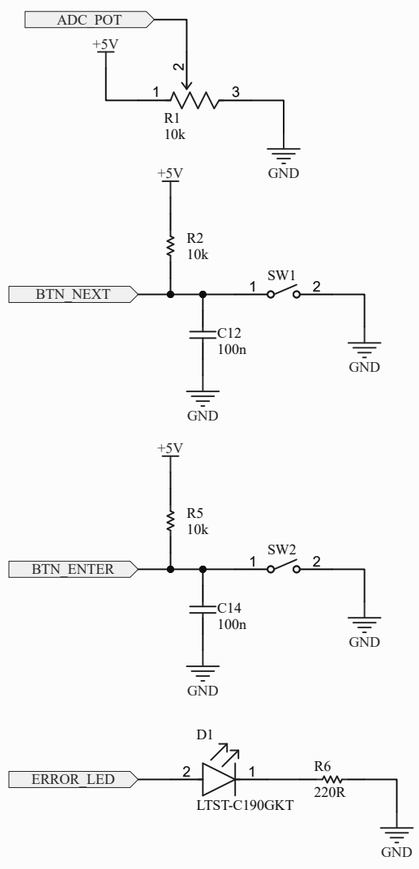
Fan H-bridge



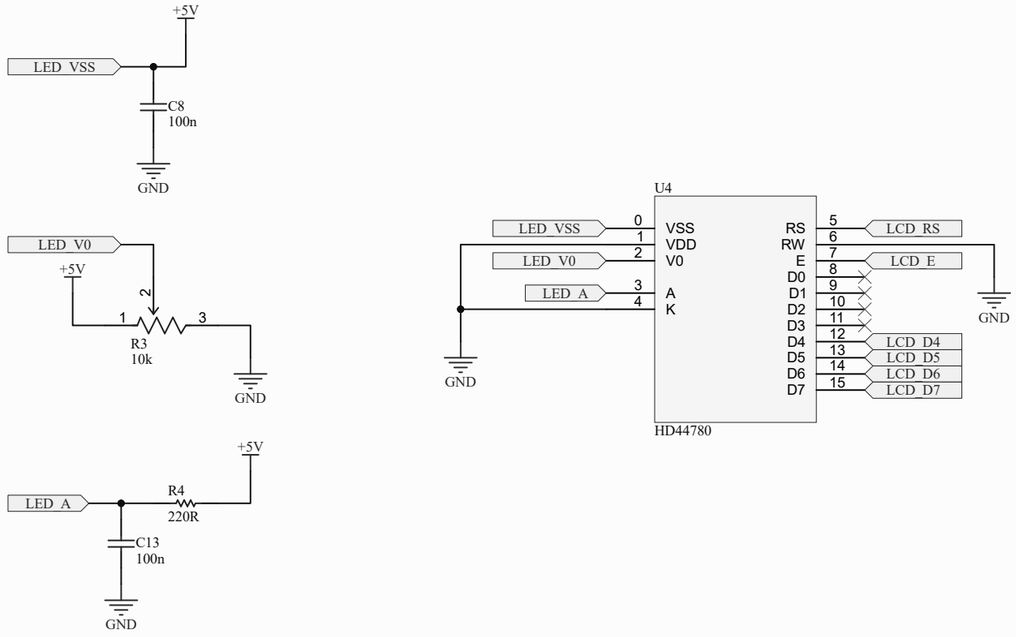
Microcontroller



Controls



LCD



Politechnika Krakowska
Mikrokontrolery w automatyce
Szymon Hrehorowicz
13A6 GP03

```c
/*
 * main.c
 *
 * Created: 1/17/2024 8:02:21 PM
 *  Author: Szymon Hrehorowicz
 */

/*
    INCLUDES
*/

#define  F_CPU 11059200UL
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>
#include <avr/pgmspace.h>
#include <util/atomic.h>
#include <stdbool.h>
#include "fan.h"
#include "lcd.h"
#include "menu.h"
#include "buttons.h"
#include "uart.h"
#include "uartControl.h"

/*
    DEFINES
*/

#define ERROR_LED      PC4

/*
    GLOBAL VARIABLES
*/

volatile fanStatus_t  fanStatus;
volatile lcdStatus_t  lcdStatus;
volatile uint8_t      buttons;
volatile uint8_t      statusRequest;
volatile uint8_t      cmdToExec;
volatile uint8_t      cmdLength;
volatile uint8_t      cmdBuff[MAX_CMD_LENGTH];
         fanHandler_t fan;
         dataQueue_t  dataQueueStart;
         uint8_t      cmdError;

/*
    FUNCTIONS
*/

void checkForErrors(void);
```

```c
int main(void)
{
    DDRC  |=  (1 << ERROR_LED);
    PORTC &= ~(1 << ERROR_LED);

    sei();

    fanStatus = fanInit(&fan);
    lcdInit();
    adcInit();
    uartInit();
    buttonsInit();
    dataQueueInit();
    menuDisplay();

    dataQueue_t *currentDataElement = &dataQueueStart;
    statusRequest = 0;

    while(1)
    {
        checkButtons();
        checkForErrors();

        if(statusRequest)
        {
            currentDataElement->action();
            currentDataElement = currentDataElement->next;
        }

        if(cmdToExec)
        {
            translateCmd();
        }
    }
}

void checkForErrors(void)
{
    if((fanStatus == FAN_ERROR) || (lcdStatus == LCD_ERROR) || cmdError)
    {
        PORTC |=  (1 << ERROR_LED);
    }else
    {
        PORTC &= ~(1 << ERROR_LED);
    }
}


/*
    INTERRUPT HANDLERS
*/
```

```
// Interrupts for fan control
ISR(TIMER0_OVF_vect)
{
    *(fan.controls.port) |= (1<<fan.controls.pin);
}

ISR(TIMER0_COMPA_vect)
{
    *(fan.controls.port) &= ~(1<<fan.controls.pin);
}

// Interrupt for buttons control
ISR(TIMER1_COMPA_vect)
{
    buttonsIntHandler();
}

// Interrupt for UART
ISR(USART_RX_vect)
{
    commandIntHandler();
}
```

```c
/*
 * adc.h
 *
 * Created: 1/18/2024 2:11:34 PM
 *  Author: Szymon
 */


#ifndef ADC_H_
#define ADC_H_

/*
    INCLUDES
*/

#include <stdint.h>

/*
    FUNCTIONS
*/

void adcInit(void);
void adcOff(void);
void adcOn(void);
uint16_t adcMeasure(void);
uint16_t adcMap(uint16_t value, uint16_t lower, uint16_t upper);

#endif /* ADC_H_ */
```

```c
/*
 * adc.c
 *
 * Created: 1/18/2024 2:11:28 PM
 *  Author: Szymon
 */

/*
    INCLUDES
*/

#include <avr/io.h>
#include "adc.h"

/*
    DEFINES
*/

#define ADC_RANGE 1024.0f

/*
    FUNCTIONS
*/

void adcInit(void)
{
    ADMUX  |= (1 << REFS0) | (1 << MUX0) | (1 << MUX2);   // Select ref voltage and channel
    ADCSRA |= (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0); // Enable ADC, 128 prescaler
}

void adcOff(void)
{
    ADCSRA &= ~(1 << ADEN);
}

void adcOn(void)
{
    ADCSRA |= (1 << ADEN);
}

uint16_t adcMeasure(void)
{
    ADCSRA |= (1 << ADSC); // Start conversion

    while(ADCSRA & (1 << ADSC));

    return (uint16_t)(ADC & 0x3FF);
}

uint16_t adcMap(uint16_t value, uint16_t lower, uint16_t upper)
{
```

```
    float fraction = (float)value / ADC_RANGE;

    uint16_t range = upper - lower;

    return lower + (fraction * range);
}
```

```c
/*
 * fan.h
 *
 * Created: 1/9/2024 12:42:42 PM
 *  Author: Szymon
 */


#ifndef FAN_H_
#define FAN_H_

#include <avr/io.h>

/*
    DEFINES
*/

// STATUS
#define FAN_OK    0u
#define FAN_ERROR 1u

// DIRECTION
#define FAN_DIR_CW  0u // clockwise
#define FAN_DIR_CCW 1u // counterclockwise

// MODES
#define FAN_OFF 0u
#define FAN_ON  1u

/*
    TYPE DEFINITIONS
*/

typedef uint8_t fanStatus_t;
typedef uint8_t fanDirection_t;
typedef uint8_t fanMode_t;

typedef struct
{
    volatile uint8_t *port;
    volatile uint8_t *ddr;
             uint8_t pin;
} controls_t;

typedef struct
{
    fanMode_t       mode;
    uint8_t           speed; // 0 - 100 %
    fanDirection_t direction;
    controls_t       controls;
} fanHandler_t;
```

```
/*
    FUNCTIONS
*/

fanStatus_t fanInit(fanHandler_t *fan);
fanStatus_t fanSetSpeed(fanHandler_t *fan, uint8_t speed);
fanStatus_t fanSetDirection(fanHandler_t *fan, fanDirection_t dir);
fanStatus_t fanStart(fanHandler_t *fan);
fanStatus_t fanStop(fanHandler_t *fan);
uint8_t     fanGetMode(fanHandler_t *fan);
uint16_t    fanGetSpeed(fanHandler_t *fan);
uint8_t     fanGetDirection(fanHandler_t *fan);
fanStatus_t fanToggleDirection(fanHandler_t *fan);


/*
    EXTERNS
*/

extern          fanHandler_t fan;
extern volatile fanStatus_t  fanStatus;

#endif /* FAN_H_ */
```

```c
/*
 * fan.c
 *
 * Created: 1/9/2024 12:42:34 PM
 *  Author: Szymon
 */

/*
    INCLUDES
*/

#include "fan.h"

/*
    DEFINES
*/

// PORTS
#define A_PORT PORTB
#define A_DDR  DDRB
#define A_PIN  PB0
#define B_PORT PORTD
#define B_DDR  DDRD
#define B_PIN  PD7

// OCR
#define FAN_OCR_LOWEST 101u
#define FAN_OCR_RANGE  255u

// SPEED
#define FAN_MIN_SPEED 0u
#define FAN_MAX_SPEED 100u

/*
    GLOBAL VARIABLES
*/

fanHandler_t fan;

/*
    FUNCTIONS
*/

fanStatus_t fanInit(fanHandler_t *fan)
{
    fanStatus_t rslt = FAN_OK;

    // Set pins as outputs
    A_DDR |= (1 << A_PIN);
    B_DDR |= (1 << B_PIN);

    // Turn off all control pins
```

```
    A_PORT &= ~(1 << A_PIN);
    B_PORT &= ~(1 << B_PIN);

    rslt = fanStop(fan);
    rslt = fanSetSpeed(fan, FAN_MAX_SPEED/2);
    rslt = fanSetDirection(fan, FAN_DIR_CW);

    // Timer initialization
    TCCR1A |= (1 << WGM01)  | (1 << WGM00); // FAST PWM
    TIMSK0 |= (1 << OCIE0A) | ( 1<< TOIE0); // Turn on interrupts

    return rslt;
}

static fanStatus_t fanSetMode(fanHandler_t *fan, fanMode_t mode)
{
    fanStatus_t rslt = FAN_OK;

    if((mode == FAN_OFF) || (mode == FAN_ON))
    {
        fan->mode = mode;
    }else
    {
        rslt = FAN_ERROR;
    }

    return rslt;
}

static fanStatus_t fanCalculateOCR(uint8_t speed, uint8_t *value_p)
{
    fanStatus_t rslt = FAN_OK;

    if((speed >= 0) && (speed <= FAN_MAX_SPEED))
    {
        *value_p = FAN_OCR_LOWEST +  ((FAN_OCR_RANGE - FAN_OCR_LOWEST) * ((float)
          speed / (float)FAN_MAX_SPEED));
    }else
    {
        rslt = FAN_ERROR;
    }

    return rslt;
}

fanStatus_t fanSetSpeed(fanHandler_t *fan, uint8_t speed)
{
    fanStatus_t rslt = FAN_OK;

    fanMode_t tmp_mode = fan->mode;

    fanStop(fan);
```

```
    if((speed >= 1) && (speed <= FAN_MAX_SPEED))
    {
        fan->speed = speed;
    }else if(speed < 1)
    {
        fan->speed = FAN_MIN_SPEED;
        fanStop(fan);
        tmp_mode = FAN_OFF;
    }else
    {
        rslt = FAN_ERROR;
    }

    if((tmp_mode == FAN_ON) && (rslt != FAN_ERROR))
    {
        fanStart(fan);
    }

    return rslt;
}

fanStatus_t fanSetDirection(fanHandler_t *fan, fanDirection_t dir)
{
    fanStatus_t rslt = FAN_OK;

    fanMode_t tmp_mode = fan->mode;

    fanStop(fan);

    if(dir == FAN_DIR_CW)
    {
        fan->direction = dir;

        fan->controls.ddr  = &A_DDR;
        fan->controls.port = &A_PORT;
        fan->controls.pin  = A_PIN;
    }else if(dir == FAN_DIR_CCW)
    {
        fan->direction = dir;

        fan->controls.ddr  = &B_DDR;
        fan->controls.port = &B_PORT;
        fan->controls.pin  = B_PIN;
    }else
    {
        rslt = FAN_ERROR;
    }

    if((tmp_mode == FAN_ON) && (rslt != FAN_ERROR))
    {
        fanStart(fan);
```

```
    }

    return rslt;
}

fanStatus_t fanStart(fanHandler_t *fan){
    fanStatus_t rslt = FAN_OK;

    rslt = fanSetMode(fan, FAN_ON);

    if(fan->speed == FAN_MAX_SPEED)
    {
        *(fan->controls.port) |= (1<<fan->controls.pin);
    }else
    {
        uint8_t newOCR = 0;

        fanCalculateOCR(fan->speed, &newOCR);


        TCNT0 = 0; // Reset Counter
        OCR0A = newOCR;
        TCCR0B |= (1 << CS02); // Start timer
    }

    return rslt;
}

fanStatus_t fanStop(fanHandler_t *fan)
{
    fanStatus_t rslt = FAN_OK;

    rslt = fanSetMode(fan, FAN_OFF);

    TCCR0B &= ~(1 << CS02);  // stop timer
    A_PORT &= ~(1 << A_PIN);
    B_PORT &= ~(1 << B_PIN);

    return rslt;
}

uint8_t fanGetMode(fanHandler_t *fan)
{
    return fan->mode;
}

uint16_t fanGetSpeed(fanHandler_t *fan)
{
    return fan->speed;
}

uint8_t fanGetDirection(fanHandler_t *fan)
```

```
{
    return fan->direction;
}

fanStatus_t fanToggleDirection(fanHandler_t *fan)
{
    fanStatus_t rslt = FAN_OK;

    if(fan->direction == FAN_DIR_CCW)
    {
        rslt = fanSetDirection(fan, FAN_DIR_CW);
    }else
    {
        rslt = fanSetDirection(fan, FAN_DIR_CCW);
    }

    return rslt;
}
```

```c
/*
 * uart.h
 *
 * Created: 1/20/2024 4:47:16 PM
 *  Author: Szymon
 */


#ifndef UART_H_
#define UART_H_

/*
    INCLUDES
*/

#include <avr/io.h>
#include <avr/pgmspace.h>

/*
    FUNCTIONS
*/

void uartInit();
void uartSendByte(uint8_t byte);
void uartSendString(const __memx uint8_t *str, uint8_t len);

#endif /* UART_H_ */
```

```c
/*
 * uart.c
 *
 * Created: 1/20/2024 4:47:09 PM
 *  Author: Szymon
 */

/*
    INCLUDES
*/

#include "uart.h"

/*
    DEFINES
*/

#ifndef F_CPU
    #define  F_CPU 11059200UL
#endif

#define UART_MAX_INT_LENGTH 3u

/*
    FUNCTIONS
*/

static void uart9600()
{
    #define BAUD 9600UL
    #include <util/setbaud.h>

    UBRR0H = UBRRH_VALUE;
    UBRR0L = UBRRL_VALUE;

    #if USE_2X
        UCSR0A |= (1 << U2X0);
    #else
        UCSR0A &= ~(1 << U2X0);
    #endif
}

static inline void uartWait()
{
    while(!(UCSR0A & (1 << UDRE0)));
}

void uartInit()
{
    uart9600();

    UCSR0B |= (1 << RXEN0) | (1 << TXEN0) | (1 << RXCIE0);
```

```
    UCSR0C |= (1 << UCSZ00) | (1 << UCSZ01);
}

void uartSendByte(uint8_t byte)
{
    uartWait();
    UDR0 = byte;
}

void uartSendString(const __memx uint8_t *str, uint8_t len)
{
    for(uint8_t i = 0; i < len; i++)
    {
        uartSendByte(str[i]);
    }
}
```

```c
/*
 * lcd.h
 *
 * Created: 1/17/2024 11:06:43 PM
 *  Author: Szymon
 */


#ifndef LCD_H_
#define LCD_H_

/*
    INCLUDES
*/

#include <avr/io.h>

/*
    DEFINES
*/

#define LCD_OK    0u
#define LCD_ERROR 1u

/*
    TYPE DEFINITIONS
*/

typedef uint8_t lcdStatus_t;

/*
    FUNCTIONS
*/

void lcdInit(void);
void lcdWriteCmd(uint8_t cmd);
void lcdWriteData(uint8_t data);
void lcdClear(void);
void lcdHome(void);
void lcdGoto(uint8_t x, uint8_t y);
void lcdWriteString(char *str);
int  lcdPrintf(const __memx char *format, ...);
lcdStatus_t lcdWriteInt(int num);

/*
    EXTERNS
*/

extern volatile lcdStatus_t lcdStatus;


#endif /* LCD_H_ */
```

```c
/*
 * lcd.c
 *
 * Created: 1/17/2024 11:06:36 PM
 *  Author: Szymon
 */

/*
    INCLUDES
*/

#ifndef F_CPU
    #define F_CPU 11059200UL
#endif

#include <stdarg.h>
#include <stdio.h>
#include <util/delay.h>
#include <stdlib.h>
#include "lcd.h"
#include "uartControl.h"

/*
    DEFINES
*/

#define LCD_MAX_INT_LENGTH 4u

#define LCD_EN_DDR  DDRC
#define LCD_EN_PORT PORTC
#define LCD_EN_PIN  PC1

#define LCD_RS_DDR  DDRC
#define LCD_RS_PORT PORTC
#define LCD_RS_PIN  PC0

#define LCD_DATA_DDR  DDRB
#define LCD_DATA_PORT PORTB

#define LCD_CMD  LCD_RS_PORT &= ~(1 << LCD_RS_PIN)
#define LCD_DATA LCD_RS_PORT |=  (1 << LCD_RS_PIN)

#define LCD_SET_EN   LCD_EN_PORT |= (1 << LCD_EN_PIN)
#define LCD_CLEAR_EN LCD_EN_PORT &= ~(1 << LCD_EN_PIN)

#define LCD_PUT_TO_BUFFER(data) LCD_DATA_PORT = (LCD_DATA_PORT & 0xE1) | ((data << 1)
  & 0x1E)

/*
    FUNCTIONS
*/
```

```c
static void lcdSendNibble(uint8_t nibble)
{
    LCD_SET_EN;
    LCD_PUT_TO_BUFFER(nibble);
    _delay_ms(1);
    LCD_CLEAR_EN;
    _delay_ms(1);
}

static void lcdSendByte(uint8_t byte)
{
    lcdSendNibble(byte >> 4);
    lcdSendNibble(byte);
}

void lcdWriteCmd(uint8_t cmd)
{
    LCD_CMD;
    lcdSendByte(cmd);
    _delay_ms(2);
}

void lcdWriteData(uint8_t data)
{
    LCD_DATA;
    lcdSendByte(data);
    _delay_ms(2);
}

void lcdInit(void)
{
    LCD_EN_DDR   |= (1 << LCD_EN_PIN);
    LCD_RS_DDR   |= (1 << LCD_RS_PIN);
    LCD_DATA_DDR |= (0x0F << 1);

    LCD_CMD;

    _delay_ms(5);
    lcdSendNibble(3);
    _delay_ms(1);
    lcdSendNibble(3);
    _delay_ms(1);
    lcdSendNibble(3);
    _delay_ms(1);
    lcdSendNibble(2);
    _delay_ms(1);

    lcdWriteCmd(0x28); // 4-bit, 2 lines
    _delay_ms(1);
    lcdWriteCmd(0x01); // clear display
    _delay_ms(1);
    lcdWriteCmd(0x06); // increment DDRAM, no shift
```

```
    _delay_ms(1);
    lcdWriteCmd(0x0C); // turn display on, no cursor and no blinking
    _delay_ms(1);

    lcdPrintf("Szymon");
    lcdGoto(0,1);
    lcdPrintf("Hrehorowicz");
    _delay_ms(3000);
}

void lcdClear(void)
{
    lcdWriteCmd(0x01);
    lcdWriteCmd(0x02);
}

void lcdHome(void)
{
    lcdWriteCmd(0x02);
    _delay_ms(2);
}

void lcdGoto(uint8_t x, uint8_t y)
{
    lcdWriteCmd(128 | ((y == 1) ? 64 : 0) | x);
}

void lcdWriteString(char *str)
{
    uint16_t i = 0;

    while(str[i] && i < 1000)
    {
        lcdWriteData(str[i]);
        i++;
    }
}

lcdStatus_t lcdWriteInt(int num)
{
    lcdStatus_t rslt = LCD_OK;
    uint8_t str[LCD_MAX_INT_LENGTH];
    uint8_t len;

    if(num > 9999)
    {
        rslt = LCD_ERROR;
        return rslt;
    }

    intToString(num, str, &len);
```

```
    for(uint8_t i = 0; i < len; i++)
    {
        lcdWriteData(str[i]);
    }

    return rslt;
}

int lcdPrintf(const char *format, ...)
{
    va_list argptr;
    char str[40];
    int n;

    va_start(argptr, format);

    n = sprintf(str, format, argptr);

    lcdWriteString(str);
    va_end(argptr);

    return n;
}
```

```c
/*
 * buttons.h
 *
 * Created: 1/20/2024 3:42:08 PM
 *  Author: Szymon
 */


#ifndef BUTTONS_H_
#define BUTTONS_H_

/*
    INCLUDES
*/

#include <avr/io.h>

/*
    DEFINES
*/

#define BTN_NEXT  PC2
#define BTN_ENTER PC3

#define BUTTONS_COUNTER_LIMIT 512u

/*
    FUNCTIONS
*/

void buttonsInit(void);
void buttonsIntHandler(void);
void checkButtons(void);
uint8_t buttonNextClicked(void);
void buttonNextClear(void);
uint8_t buttonEnterClicked(void);
void buttonEnterClear(void);

/*
    EXTERNS
*/

extern volatile uint8_t buttons;

#endif /* BUTTONS_H_ */
```

```c
/*
 * buttons.c
 *
 * Created: 1/20/2024 3:41:56 PM
 *  Author: Szymon
 */

/*
    INCLUDES
*/

#include "buttons.h"
#include "menu.h"

/*
    FUNCTIONS
*/

void buttonsInit(void)
{
    DDRC   &= ~(1 << BTN_NEXT) & ~(1 << BTN_ENTER);

    OCR1AH |= ((uint16_t)BUTTONS_COUNTER_LIMIT >> 8);
    OCR1AL |= (uint8_t)BUTTONS_COUNTER_LIMIT;
    TIMSK1 |= (1 << OCIE1A);
    TCCR1B |= (1 << CS12) | (1 << CS10);  // Normal mode, 1024 prescaler
}

void buttonsIntHandler(void)
{
    static uint8_t prevButtonsStatus = 0xFF;
    uint8_t currentButtonsStatus = PINC;

    if(!(currentButtonsStatus & (1 << BTN_NEXT)) && (prevButtonsStatus & (1 <<
      BTN_NEXT)))
    {
        buttons |= (1 << BTN_NEXT);
    }

    if(!(currentButtonsStatus & (1 << BTN_ENTER)) && (prevButtonsStatus & (1 <<
      BTN_ENTER)))
    {
        buttons |= (1 << BTN_ENTER);
    }

    prevButtonsStatus = currentButtonsStatus;
}

void checkButtons(void)
{
    if(buttonNextClicked())
    {
```

```
        buttonNextClear();
        menuNext();
    }

    if(buttonEnterClicked())
    {
        buttonEnterClear();
        menuEnter();
    }
}

uint8_t buttonNextClicked(void)
{
    return buttons & (1 << BTN_NEXT);
}

void buttonNextClear(void)
{
    buttons &= ~(1 << BTN_NEXT);
}

uint8_t buttonEnterClicked(void)
{
    return buttons & (1 << BTN_ENTER);
}

void buttonEnterClear(void)
{
    buttons &= ~(1 << BTN_ENTER);
}
```

```
/*
 * menu.h
 *
 * Created: 1/18/2024 10:54:08 AM
 *  Author: Szymon Hrehorowicz
 */


#ifndef MENU_H_
#define MENU_H_

/*
    TYPE DEFINITIONS
*/

typedef void (*menuAction)();

typedef struct MenuItem
{
    const __flash char * const     label;
    menuAction                     action;
    const __flash struct MenuItem *parent;
    const __flash struct MenuItem *submenu;
    const __flash struct MenuItem *next;
} menuItem_t;

/*
    FUNCTIONS
*/

void menuDisplay();
void menuNext();
void menuPrev();
void menuEnter();
void menuBack();

/*
    EXTERNS
*/

extern menuItem_t const __flash menu;

#endif /* MENU_H_ */
```

```c
/*
 * menu.c
 *
 * Created: 1/18/2024 10:54:01 AM
 *  Author: Szymon
 */

/*
    INCLUDES
*/

#include <avr/pgmspace.h>
#include <stddef.h>
#include <string.h>
#include "menu.h"
#include "lcd.h"

/*
    DEFINES
*/

#define MENU_ROWS 2u

/*
    GLOBAL VARIABLES
*/

static const __flash menuItem_t *currentMenu_p = &menu;

static uint8_t menuIdx;
static uint8_t menuFirstPosition;

/*
    FUNCTIONS
*/

static uint8_t menuGetItemsNo(void)
{
    const __flash menuItem_t *tempMenuItem = currentMenu_p;
    uint8_t idx = 0;

    while(tempMenuItem)
    {
        tempMenuItem = tempMenuItem->next;
        idx++;
    }

    return idx;
}

static const __flash menuItem_t *menuGetMenuItem(uint8_t idx)
{
```

```
    const __flash menuItem_t *tempMenuItem = currentMenu_p;

    while((tempMenuItem) && (idx > 0))
    {
        tempMenuItem = tempMenuItem->next;
        idx--;
    }

    return tempMenuItem;
}

void menuDisplay()
{
    const __flash menuItem_t *tempMenuItem = menuGetMenuItem(menuFirstPosition);

    uint8_t noOfItems = menuGetItemsNo();

    lcdClear();

    for(uint8_t i = 0; i < MENU_ROWS; i++)
    {
        lcdGoto(0,i);

        if(menuIdx == (menuFirstPosition + i) % noOfItems)
        {
            lcdPrintf(">");
        }else
        {
            lcdPrintf(" ");
        }

        lcdPrintf(tempMenuItem->label);

        tempMenuItem = tempMenuItem->next;
        if(tempMenuItem == NULL)
        {
            if(menuGetItemsNo() > MENU_ROWS)
            {
                tempMenuItem = currentMenu_p;
            }else
            {
                break;
            }
        }
    }
}

void menuNext()
{
    uint8_t no = menuGetItemsNo();
    menuIdx++;
```

```
    if(no > MENU_ROWS)
    {
        uint8_t distance;

        if(menuIdx < menuFirstPosition)
        {
            distance = no - menuFirstPosition + menuIdx;
        }else
        {
            distance = menuIdx - menuFirstPosition;
        }

        if(distance >= MENU_ROWS)
        {
            menuFirstPosition++;
        }
    }

    menuIdx           %= no;
    menuFirstPosition %= no;

    menuDisplay();
}

void menuPrev()
{
    if(menuIdx > 0)
    {
        if(menuIdx == menuFirstPosition)
        {
            menuFirstPosition--;
        }
        menuIdx--;
    }else
    {
        if(menuFirstPosition == 0)
        {
            menuIdx = menuGetItemsNo() - 1;
            if(menuGetItemsNo() > MENU_ROWS)
            {
                menuFirstPosition = menuIdx;
            }
        }else
        {
            menuIdx = menuGetItemsNo() - 1;
        }
    }

    menuDisplay();
}

void menuEnter()
```

```
{
    const __flash menuItem_t *tempMenuItem = menuGetMenuItem(menuIdx);
    const __flash menuItem_t *submenu      = tempMenuItem->submenu;

    menuAction action = tempMenuItem->action;

    if(action)
    {
        (*action)();
    }

    if(submenu)
    {
        currentMenu_p     = submenu;
        menuIdx           = 0;
        menuFirstPosition = 0;
    }

    menuDisplay();
}

void menuBack()
{
    menuFirstPosition = 0;
    menuIdx           = 0;
    currentMenu_p     = currentMenu_p->parent;
}
```

```c
/*
 * menuDefines.c
 *
 * Created: 1/18/2024 11:20:55 AM
 *  Author: Szymon
 */

/*
 * menudef.c
 *
 * Created: 1/18/2024 10:58:05 AM
 *  Author: Szymon Hrehorowicz
 */

/*
    INCLUDES
*/

#ifndef F_CPU
    #define F_CPU 11059200UL
#endif

#include <util/delay.h>
#include <avr/pgmspace.h>
#include "menu.h"
#include "lcd.h"
#include "fan.h"
#include "adc.h"
#include "buttons.h"

/*
    DEFINES
*/

#define PGM_STR(X) ((const __flash char[]) { X })

/*
    GLOBAL VARIABLES
*/

fanHandler_t         fan;
volatile fanStatus_t fanStatus;
volatile lcdStatus_t lcdStatus;
volatile uint8_t     buttons;

/*
    FUNCTIONS 1
*/

void menuStartStopFan();
void menuSetNewSpeed();
void menuChangeDirection();
```

```
void menuGetCurrentSpeed();

/*
    GLOBAL VARIABLE DEFINITIONS
*/

menuItem_t const __flash menu;
menuItem_t const __flash speedMenuCurrentSpeed;

menuItem_t const __flash speedMenuReturn       = {PGM_STR("<return>"), menuBack,
  &speedMenuCurrentSpeed, 0, 0};
menuItem_t const __flash speedMenuSetSpeed     = {PGM_STR("Set new speed"),
  menuSetNewSpeed, &speedMenuCurrentSpeed, 0, &speedMenuReturn};
menuItem_t const __flash speedMenuCurrentSpeed = {PGM_STR("Current speed"),
  menuGetCurrentSpeed, &menu, 0, &speedMenuSetSpeed};

menuItem_t const __flash directionMenu = {PGM_STR("Change dir"), menuChangeDirection,
  &menu, 0, 0};
menuItem_t const __flash speedMenu     = {PGM_STR("Change speed"), 0, &menu,
  &speedMenuCurrentSpeed, &directionMenu};

menuItem_t const __flash menu = {PGM_STR("START/STOP"), menuStartStopFan, 0, 0,
  &speedMenu};

/*
    FUNCTIONS 2
*/

void menuStartStopFan()
{
    uint8_t fanMode = fanGetMode(&fan);

    if(fanMode == FAN_ON)
    {
        fanStatus = fanStop(&fan);
    }else
    {
        fanStatus = fanStart(&fan);
    }
}

void menuGetCurrentSpeed()
{
    lcdClear();
    lcdStatus = lcdWriteInt(fanGetSpeed(&fan));
    lcdPrintf(" \%");
    _delay_ms(2000);
}

void menuSetNewSpeed()
{
    _delay_ms(200);
```

```
    adcOn();
    uint16_t newSpeed = 0;

    while(!buttonEnterClicked())
    {
        newSpeed = adcMap(adcMeasure(), 0, 100);
        lcdClear();
        lcdStatus = lcdWriteInt(newSpeed);
        _delay_ms(100);
    }

    buttonEnterClear();

    fanStatus = fanSetSpeed(&fan, newSpeed);
}

void menuChangeDirection()
{
    fanStatus = fanToggleDirection(&fan);
}
```

```c
/*
 * dataQueue.h
 *
 * Created: 1/20/2024 8:55:12 PM
 *  Author: Szymon
 */

#ifndef DATAQUEUE_H_
#define DATAQUEUE_H_

/*
    INCLUDES
*/

#include <avr/io.h>
#include <stdbool.h>

/*
    DEFINES
*/

#define MAX_INT_LENGTH 4u
#define MAX_CMD_LENGTH 4u

/*
    TYPE DEFINITIONS
*/

typedef struct DataQueue
{
    void   (*action)(void);
    struct DataQueue *next;
} dataQueue_t;

/*
    FUNCTIONS
*/

void dataQueueInit(void);
bool intToString(int num, uint8_t strOut[MAX_INT_LENGTH], uint8_t *strLength);
bool stringToInt(uint8_t str[MAX_INT_LENGTH], uint8_t strLength, int *num);
void printWelcome(void);
void printMode(void);
void printSpeed(void);
void printDirection(void);
void translateCmd(void);
void commandIntHandler(void);

/*
    EXTERNS
*/
```

```
extern volatile uint8_t     statusRequest;
extern          dataQueue_t dataQueueStart;
extern          uint8_t     cmdError;
extern volatile uint8_t     cmdBuff[MAX_CMD_LENGTH];
extern volatile uint8_t     cmdLength;
extern volatile uint8_t     cmdToExec;

#endif /* DATAQUEUE_H_ */
```

```c
/*
 * dataQueue.c
 *
 * Created: 1/20/2024 8:55:00 PM
 *  Author: Szymon
 */

/*
    INCLUDES
*/

#include <util/atomic.h>
#include "uart.h"
#include "uartControl.h"
#include "fan.h"

/*
    DEFINES
*/

#define NO_CMDS        3u
#define CMD            0u
#define CMD_FIRST_BIT  1u
#define CR             13u

/*
    GLOBAL VARIABLES
*/

fanHandler_t fan;

dataQueue_t dataQueueMode;
dataQueue_t dataQueueSpeed;
dataQueue_t dataQueueDirection;

uint8_t cmds[NO_CMDS] = {'v', 'd', 'm'};

dataQueue_t dataQueueStart =
{
    .action =  printWelcome,
    .next   = &dataQueueMode,
};

/*
    FUNCTIONS
*/

void dataQueueInit(void)
{
    dataQueueMode.action =  printMode;
    dataQueueMode.next   = &dataQueueSpeed;
```

```
    dataQueueSpeed.action =  printSpeed;
    dataQueueSpeed.next   = &dataQueueDirection;

    dataQueueDirection.action =  printDirection;
    dataQueueDirection.next   = &dataQueueStart;
}

bool intToString(int num, uint8_t strOut[MAX_INT_LENGTH], uint8_t *strLength)
{
    uint8_t size = MAX_INT_LENGTH;
    uint8_t idx  = 0;
    uint8_t str[MAX_INT_LENGTH];

    if(num > 9999)
    {
        return false;
    }

    str[3] =  num % 10;
    str[2] = (num / 10) % 10;
    str[1] = (num / 100) % 10;
    str[0] = (num / 1000) % 10;

    if(str[0] == 0)
    {
        idx++;
    }

    if((str[0] == 0) && (str[1] == 0))
    {
        idx++;
    }

    if((str[0] == 0) && (str[1] == 0) && (str[2] == 0))
    {
        idx++;
    }

    for(int i = idx; i < size; i++)
    {
        strOut[i - idx] = '0' + str[i];
    }

    *strLength = size - idx;

    return true;
}

bool stringToInt(uint8_t str[MAX_INT_LENGTH], uint8_t strLength, int *num)
{
    if(strLength > MAX_INT_LENGTH)
    {
```

```
        return false;
    }

    uint8_t numerical[MAX_INT_LENGTH];

    numerical[0] = (str[0] == '0') ? 0 : (str[0] - '0');
    numerical[1] = (str[1] == '0') ? 0 : (str[1] - '0');
    numerical[2] = (str[2] == '0') ? 0 : (str[2] - '0');

    switch(strLength)
    {
        case 1:
            *num = numerical[0];
            break;
        case 2:
            *num = (numerical[0] * 10) + numerical[1];
            break;
        case 3:
            *num = (numerical[0] * 100) + (numerical[1] * 10) + numerical[2];
            break;
        default:
            return false;
    }

    return true;
}

void printWelcome(void)
{
    static const __flash uint8_t str[] = {"Fan controller\r\n"};

    for(uint8_t i = 0; i < sizeof(str)/sizeof(str[0]); i++)
    {
        uartSendByte(str[i]);
    }
}

void printMode(void)
{
    uint8_t mode = fanGetMode(&fan);
    static const __flash uint8_t onStr[] =  {"Mode: ON\r\n"};
    static const __flash uint8_t offStr[] = {"Mode: OFF\r\n"};

    if(mode == FAN_ON)
    {
        uartSendString(onStr, sizeof(onStr)/sizeof(onStr[0]));
    }else
    {
        uartSendString(offStr, sizeof(offStr)/sizeof(offStr[0]));
    }
}
```

```
void printSpeed(void)
{
    static const __flash uint8_t speedStr[] = {"Speed: "};
    static const __flash uint8_t endStr[]   = {" %\r\n"};
    uint8_t speed = fanGetSpeed(&fan);
    uint8_t str[MAX_INT_LENGTH];
    uint8_t len;

    intToString((int)speed, str, &len);

    uartSendString(speedStr, sizeof(speedStr)/sizeof(speedStr[0]));
    uartSendString(str, len);
    uartSendString(endStr, sizeof(endStr)/sizeof(endStr[0]));
}
void printDirection(void)
{
    static const __flash uint8_t cwStr[]  = {"Direction: CW\r\n"};
    static const __flash uint8_t ccwStr[] = {"Direction: CCW\r\n"};
    static const __flash uint8_t endStr[] = {"\r\n"};

    uint8_t direction = fanGetDirection(&fan);

    if(direction == FAN_DIR_CW)
    {
        uartSendString(cwStr, sizeof(cwStr)/sizeof(cwStr[0]));
    }else
    {
        uartSendString(ccwStr, sizeof(ccwStr)/sizeof(ccwStr[0]));
    }

    uartSendString(endStr, sizeof(endStr)/sizeof(endStr[0]));

    ATOMIC_BLOCK(ATOMIC_RESTORESTATE)
    {
        if(statusRequest)
        {
            statusRequest = 0;
        }
    }
}

void translateCmd(void)
{
    uint8_t tempCmdLength = 0;
    uint8_t tempCmdBuff[MAX_CMD_LENGTH];

    ATOMIC_BLOCK(ATOMIC_RESTORESTATE)
    {
        tempCmdLength = cmdLength;
        for(uint8_t i = 0; i < cmdLength; i++)
        {
            tempCmdBuff[i] = cmdBuff[i];
```

```
        }
        cmdToExec = 0;
    }

    switch(tempCmdBuff[CMD])
    {
        case 'v' :
        cmdError = 0;
        int newSpeed = 0;
        uint8_t noNumError = stringToInt(tempCmdBuff + 1, tempCmdLength - 1,
          &newSpeed);

        if(!noNumError)
        {
            cmdError = 1;
            break;
        }

        fanStatus = fanSetSpeed(&fan, (uint8_t)newSpeed);

        break;
        case 'd' :
        cmdError = 0;

        switch(tempCmdBuff[CMD_FIRST_BIT])
        {
            case '0' :
            fanStatus = fanSetDirection(&fan, FAN_DIR_CW);
            break;
            case '1' :
            fanStatus = fanSetDirection(&fan, FAN_DIR_CCW);
            break;
            default:
            cmdError = 1;
            break;
        }
        break;
        case 'm' :
        cmdError = 0;

        switch(tempCmdBuff[CMD_FIRST_BIT])
        {
            case '0' :
            fanStatus = fanStop(&fan);
            break;
            case '1' :
            fanStatus = fanStart(&fan);
            break;
            default:
            cmdError = 1;
            break;
        }
```

```
        break;
        default:
        cmdError = 1;
        break;
    }
}

void commandIntHandler(void)
{
    static uint8_t idx  = 0;
    uint8_t byte = UDR0;

    if(byte == 's')
    {

        if(!statusRequest)
        {
            statusRequest = 1;
        }
    }else if(idx)
    {
        if(byte == CR)
        {
            cmdLength = idx;
            idx = 0;
            cmdToExec = 1;
        }else
        {
            cmdBuff[idx++] = byte;
        }
    }else
    {
        for(uint8_t i = 0; i < NO_CMDS; i++)
        {
            if(byte == cmds[i])
            {
                cmdBuff[0] = byte;
                idx++;
                break;
            }
        }
    }
}
```