Contents

1 steering/usart.c

```c
/*
 * uart.c
 *
 * Created: 2019-11-07 11:29:43
 *  Author: osklu414
 */

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

#include <stdlib.h>

#include "usart.h"
#include "steering.h"

#define BAUD_PRESCALE 25

void
usart_init()
{
	UBRR0H = (unsigned char)(BAUD_PRESCALE>>8);
	UBRR0L = (unsigned char)BAUD_PRESCALE;

	// set double speed operation
	UCSR0A |= (1<<U2X0);

	// enable receiver and transmitter
	UCSR0B = (1<<RXEN0)|(1<<TXEN0);

	// Set frame format: 8 data, 1 stop bit
	UCSR0C = (0<<USBS0)|(3<<UCSZ00);

	// Enable the USART receive complete interrupt (USART_RXC)
	UCSR0B |= (1 << RXCIE0);
}

void usart_transmit(uint8_t data)
{
	// Wait for empty transmit buffer
	while ( !( UCSR0A & (1<<UDRE0)) );
	// Put data into buffer, sends the data
	UDR0 = data;
}

/*
unsigned char usart_receive(void)
{
	// Wait for data to be received
	while (!(UCSR0A & (1<<RXC0)));
	// Get and return received data from buffer
	return UDR0;
}
*/

// USART rx state
```

```
struct rx_state
{
        enum
        {
                NONE = 0,
                PWM = 1,
                DIR = 2,
                IDENTIFIED = 3,

        } current_type;
        uint8_t current_field;
        uint8_t field_buffer[6]; // largest transmission is 6 bytes
};


ISR(USART0_RX_vect)
{
        // code to be executed when the USART receives a byte here
        static struct rx_state state =
        {
                .current_type = NONE,
                .current_field = 0,
        };

        uint8_t rx_byte;
        rx_byte = UDR0; // fetch received byte value into variable
        // UDR0 = rx_byte; // echo back received byte to its transmitter

        switch(state.current_type)
        {
                case NONE:
                {
                        // identify transmission type
                        if(rx_byte == IDENTIFIED)
                        {
                                steering_identified();
                        }
                        else
                        {
                                state.current_type = rx_byte;
                        }
                        break;
                }
                case PWM:
                {
                        state.field_buffer[state.current_field++] = rx_byte; //
                            fill field buffer
                        // check if last field reached
                        if(state.current_field == 2)
                        {
                                // call callback with new pwm
                                steering_pwm(state.field_buffer[0], state.
                                    field_buffer[1]);
                                // reset state
                                state.current_type = NONE;
                                state.current_field = 0;
                        }
```

```
            break;
        }
        case DIR:
        {
            state.field_buffer[state.current_field++] = rx_byte; //
                fill field buffer
            // check if last field reached
            if(state.current_field == 2)
            {
                // call callback with new dir
                steering_dir((bool)state.field_buffer[0], (bool)
                    state.field_buffer[1]);
                // reset state
                state.current_type = NONE;
                state.current_field = 0;
            }
            break;
        }
        case IDENTIFIED:
        {
            state.current_type = NONE;
            state.current_field = 0;
            break;
        }
    }
}
```

2 steering/steering.c

```c
/*
 * steering.c
 *
 * Created: 2019-11-11 14:13:17
 *  Author: osklu414
 */

#include "motors.h"
#include "usart.h"
#include "steering.h"

#include <stdlib.h>
#include <math.h>

#include <avr/interrupt.h>
#include <util/delay.h>


static struct steering_state state;

void
steering_init()
{
        motors_init();
        usart_init();
        state.identified = false;

        // enable interrupts
        sei();

        // transmit module id
        while(!state.identified)
        {
                usart_transmit(STEERING_ID);
                _delay_ms(100);
        }


}

void
steering_tick()
{
        /* OLD CODE (kept for reference)
        int16_t rot_delta = acos((cos(state.r)*(state.tgt_x - state.x) + sin(
            state.r)*(state.tgt_y - state.y)) / (sqrt(pow(cos(state.r), 2) + pow(
            sin(state.r), 2)) * sqrt(pow(state.tgt_x - state.x, 2) + pow(state.
            tgt_y - state.y, 2))));

        // dist controller
        int16_t dist_e = sqrt(pow(state.tgt_x - state.x, 2) + pow(state.tgt_y -
            state.y, 2));
        int16_t dist_de = dist_e - state.dist_e_last;
        state.dist_e_last = dist_e;
        uint8_t dist_u = state.dist_kp*dist_e + state.dist_kd*dist_de;
```

```
	// turn controller
	int16_t turn_e = rot_delta;
	int16_t turn_de = turn_e - state.turn_e_last;
	state.turn_e_last = turn_e;
	int16_t turn_u = state.turn_kp*turn_e + state.turn_kd*turn_de;

	// TODO: limit u(t):s to values between 0 and 255
	// What is the maximum values they can be? Maybe the only viable
	    solution is to tune P values?

	if(abs(rot_delta) > 5)
	{
		uint8_t speed = turn_u;
		// rotate in correct direction
		if(rot_delta >= 0)
		{
			// rotate left
			dir_set(false, true);
			pwm_set(speed, speed);
		}
		else
		{
			// rotate right
			dir_set(true, false);
			pwm_set(speed, speed);
		}
	}
	else
	{
		// max speed depends on distance to target
		uint8_t speed_1 = dist_u;
		uint8_t speed_2 = dist_u - turn_u;

		// move towards destination
		dir_set(true, true);
		if(rot_delta >= 0)
		{
			// turn left
			pwm_set(speed_2, speed_1);
		}
		else
		{
			// turn right
			pwm_set(speed_1, speed_2);
		}
	}
	*/
}

void
steering_pwm(uint8_t left, uint8_t right)
{
	pwm_set(left, right);
}

void
```

```
steering_dir(bool left_forward, bool right_forward)
{
	dir_set(left_forward, right_forward);
}

void
steering_identified()
{
	state.identified = true;
}
```

3 steering/motors.c

```c
/*
 * steering.c
 *
 * Created: 2019-11-06 15:16:41
 *  Author: marno874
 */


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

#include "motors.h"


void
pwm_init()
{
	// set PWM port pins as outputs
	DDRD |= (1 << DDD4) | (1 << DDD5);

	// make timers reset to BOTTOM when they reach TOP (255)
	TCCR1A |= (1 << COM1A1) | (0 << COM1A0);
	TCCR1A |= (1 << COM1B1) | (0 << COM1B0);

	// enable fast PWM mode
	TCCR1A |= (1 << WGM10) | (0 << WGM11);
	TCCR1B |= (1 << WGM12) | (0 << WGM13);

	// configure timers (no prescaling)
	TCCR1B |= (0 << CS12) | (0 << CS11) | (1 << CS10);

	//set pmw to 0
	OCR1A = 0;
	OCR1B = 0;
}

void
dir_init()
{
	// set direction port pins as outputs
	DDRD |= (1 << DDD2) | (1 << DDD3);
}

void
pwm_set(uint8_t left, uint8_t right)
{
	OCR1A = right;
	OCR1B = left;
}

void
dir_set(bool left_forward, bool right_forward)
{
```

```
        if(left_forward)
        {
                PORTD |= (1 << 2);
        }
        else
        {
                PORTD &= ~(1 << 2);
        }

        if(right_forward)
        {
                PORTD |= (1 << 3);
        }
        else
        {
                PORTD &= ~(1 << 3);
        }
}

void
motors_init()
{
        pwm_init();
        dir_init();
}
```

4 steering/main.c

```c
/*
 * main.c
 *
 * Created: 2019-11-07 11:27:36
 *  Author: osklu414
 */

#include "steering.h"


int
main(void)
{
	steering_init();
	while(1) steering_tick();
	return 0;
}
```

5 sensor/adc.c

```
/*
 * adc.c
 *
 * Created: 2019-11-07 13:24:20
 *  Author: marno874
 */


#include "adc.h"
#include <math.h>

void adc_init(){
	// AREF = AVcc
	ADMUX = (1<<REFS0);

	// ADC Enable and prescaler of 128
	// 16000000/128 = 125000
	ADCSRA = (1<<ADEN)|(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0);
}

uint16_t adc_read(uint8_t ch)
{
	// select the corresponding channel 0~7
	// ANDing with '7' will always keep the value
	// of 'ch' between 0 and 7
	ch &= 0b00000111;  // AND operation with 7
	ADMUX = (ADMUX & 0xF8)|ch;     // clears the bottom 3 bits before ORing

	// start single conversion
	// write '1' to ADSC
	ADCSRA |= (1<<ADSC);

	// wait for conversion to complete
	// ADSC becomes '0' again
	// till then, run loop continuously
	while(ADCSRA & (1<<ADSC));

	return (ADC);
}

uint16_t voltage_to_dist(uint16_t voltage){
	volatile float inverse = (3*pow(10, -5)*voltage) + 0.4204;   // Linear
	    relationship between voltage and (1/distance) -0,42.
	volatile uint16_t distance = (uint16_t)(1/(inverse - 0.42));

	if(distance < LOWER_LIMIT || distance > UPPER_LIMIT){
		return 0;
	}
	return distance;
}
```

6 sensor/usart.c

```c
/*
 * uart.c
 *
 * Created: 2019-11-07 11:29:43
 *  Author: osklu414
 */

#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>

#include <stdlib.h>

#include "usart.h"



#define BAUD_PRESCALE 25

void
init_usart()
{
        UBRR0H = (unsigned char)(BAUD_PRESCALE>>8);
        UBRR0L = (unsigned char)BAUD_PRESCALE;

        // set double speed operation
        UCSR0A |= (1<<U2X0);

        // enable receiver and transmitter
        UCSR0B = (1<<RXEN0)|(1<<TXEN0);

        // Set frame format: 8 data, 1 stop bit
        UCSR0C = (0<<USBS0)|(3<<UCSZ00);

        // Enable the USART receive complete interrupt (USART_RXC)
        UCSR0B |= (1 << RXCIE0);
        // set the global interrupt enable flag
        sei();
}

void usart_transmit(uint8_t data)
{
        // Wait for empty transmit buffer
        while ( !( UCSR0A & (1<<UDRE0)) );
        // Put data into buffer, sends the data
        UDR0 = data;
}

unsigned char usart_receive(void)
{
        // Wait for data to be received
        while (!(UCSR0A & (1<<RXC0)));
        // Get and return received data from buffer
        return UDR0;
}
```

```c
void usart_transmit_competition()
{
        usart_transmit(COMPETITION);
}


struct rx_state
{
        enum
        {
                NONE = 0,
                IDENTIFIED = 1
        } current_type;
        uint8_t current_field;
        uint8_t field_buffer[6]; // largest transmission size 6 (TBD for sensor)
};


ISR(USART0_RX_vect)
{
        // code to be executed when the USART receives a byte here
        static struct rx_state state =
        {
                .current_type = NONE,
                .current_field = 0,
        };

        uint8_t rx_byte;
        rx_byte = UDR0; // fetch received byte value into variable
        // UDR0 = rx_byte; // echo back received byte to its transmitter

        switch(state.current_type)
        {
                case NONE:
                {
                        // identify transmission type
                        if(rx_byte == IDENTIFIED)
                        {
                                sensor_identified();
                        }
                        else
                        {
                                state.current_type = rx_byte;
                        }
                        break;
                }

                default:
                        break;
        }
}

void usart_transmit_int(int n, int size){
```

```
        char str[size];
        sprintf(str, "%d", n);

        for (int i = 0; i < size; i++){
                usart_transmit(str[i]);
                _delay_ms(1);
        }
}
```

7 sensor/twimaster.c

```
/*************************************************************************
* Title:    I2C master library using hardware TWI interface
* Author:   Peter Fleury <pfleury@gmx.ch>  http://jump.to/fleury
* File:     $Id: twimaster.c,v 1.4 2015/01/17 12:16:05 peter Exp $
* Software: AVR-GCC 3.4.3 / avr-libc 1.2.3
* Target:   any AVR device with hardware TWI
* Usage:    API compatible with I2C Software Library i2cmaster.h
**************************************************************************/
#include <inttypes.h>
#include <compat/twi.h>

#include "i2cmaster.h"


/* define CPU frequency in hz here if not defined in Makefile */
#ifndef F_CPU
#define F_CPU 8000000UL
#endif

/* I2C clock in Hz */
#define SCL_CLOCK  100000UL


/*************************************************************************
 Initialization of the I2C bus interface. Need to be called only once
*************************************************************************/
void i2c_init(void)
{
  /* initialize TWI clock: 100 kHz clock, TWPS = 0 => prescaler = 1 */

  TWSR = 0;                         /* no prescaler */
  TWBR = ((F_CPU/SCL_CLOCK)-16)/2;  /* must be > 10 for stable operation */

}/* i2c_init */


/*************************************************************************
  Issues a start condition and sends address and transfer direction.
  return 0 = device accessible, 1= failed to access device
*************************************************************************/
unsigned char i2c_start(unsigned char address)
{
    uint8_t   twst;

        // send START condition
        TWCR = (1<<TWINT) | (1<<TWSTA) | (1<<TWEN);

        // wait until transmission completed
        while(!(TWCR & (1<<TWINT)));

        // check value of TWI Status Register. Mask prescaler bits.
        twst = TW_STATUS & 0xF8;
        if ( (twst != TW_START) && (twst != TW_REP_START)) return 1;

        // send device address
```

```c
        TWDR = address;
        TWCR = (1<<TWINT) | (1<<TWEN);

        // wail until transmission completed and ACK/NACK has been received
        while(!(TWCR & (1<<TWINT)));

        // check value of TWI Status Register. Mask prescaler bits.
        twst = TW_STATUS & 0xF8;
        if ( (twst != TW_MT_SLA_ACK) && (twst != TW_MR_SLA_ACK) ) return 1;

        return 0;

}/* i2c_start */


/*************************************************************************
 Issues a start condition and sends address and transfer direction.
 If device is busy, use ack polling to wait until device is ready

 Input:   address and transfer direction of I2C device
*************************************************************************/
void i2c_start_wait(unsigned char address)
{
    uint8_t   twst;


    while ( 1 )
    {
            // send START condition
            TWCR = (1<<TWINT) | (1<<TWSTA) | (1<<TWEN);

        // wait until transmission completed
        while(!(TWCR & (1<<TWINT)));

        // check value of TWI Status Register. Mask prescaler bits.
        twst = TW_STATUS & 0xF8;
        if ( (twst != TW_START) && (twst != TW_REP_START)) continue;

        // send device address
        TWDR = address;
        TWCR = (1<<TWINT) | (1<<TWEN);

        // wail until transmission completed
        while(!(TWCR & (1<<TWINT)));

        // check value of TWI Status Register. Mask prescaler bits.
        twst = TW_STATUS & 0xF8;
        if ( (twst == TW_MT_SLA_NACK )||(twst ==TW_MR_DATA_NACK) )
        {
            /* device busy, send stop condition to terminate write operation */
                TWCR = (1<<TWINT) | (1<<TWEN) | (1<<TWSTO);

                // wait until stop condition is executed and bus released
                while(TWCR & (1<<TWSTO));

            continue;
        }
```

```
        //if( twst != TW_MT_SLA_ACK) return 1;
        break;
     }

}/* i2c_start_wait */


/*************************************************************************
 Issues a repeated start condition and sends address and transfer direction

 Input:   address and transfer direction of I2C device

 Return:  0 device accessible
          1 failed to access device
*************************************************************************/
unsigned char i2c_rep_start(unsigned char address)
{
    return i2c_start( address );

}/* i2c_rep_start */


/*************************************************************************
 Terminates the data transfer and releases the I2C bus
*************************************************************************/
void i2c_stop(void)
{
    /* send stop condition */
        TWCR = (1<<TWINT) | (1<<TWEN) | (1<<TWSTO);

        // wait until stop condition is executed and bus released
        while(TWCR & (1<<TWSTO));

}/* i2c_stop */


/*************************************************************************
  Send one byte to I2C device

  Input:    byte to be transfered
  Return:   0 write successful
            1 write failed
*************************************************************************/
unsigned char i2c_write( unsigned char data )
{
    uint8_t   twst;

        // send data to the previously addressed device
        TWDR = data;
        TWCR = (1<<TWINT) | (1<<TWEN);

        // wait until transmission completed
        while(!(TWCR & (1<<TWINT)));

        // check value of TWI Status Register. Mask prescaler bits
        twst = TW_STATUS & 0xF8;
        if( twst != TW_MT_DATA_ACK) return 1;
```

```c
        return 0;

}/* i2c_write */


/*************************************************************************
 Read one byte from the I2C device, request more data from device

 Return:  byte read from I2C device
*************************************************************************/
unsigned char i2c_readAck(void)
{
        TWCR = (1<<TWINT) | (1<<TWEN) | (1<<TWEA);
        while(!(TWCR & (1<<TWINT)));

    return TWDR;

}/* i2c_readAck */


/*************************************************************************
 Read one byte from the I2C device, read is followed by a stop condition

 Return:  byte read from I2C device
*************************************************************************/
unsigned char i2c_readNak(void)
{
        TWCR = (1<<TWINT) | (1<<TWEN);
        while(!(TWCR & (1<<TWINT)));

    return TWDR;

}/* i2c_readNak */
```

8 sensor/sensor.c

```c
/*
 * sensor.c
 *
 * Created: 2019-11-07 13:24:20
 *  Author: marno874, felli675, edwjo109, matlj387
 */


#include "sensor.h"


#define LEFT_SENSOR 0
#define RIGHT_SENSOR 1
#define TRANSMIT_DELAY 200


//Fields
int overflow_count = 0;
uint16_t prev_time = 0;
static struct sensor_data data;
static struct sensor_state state;

/*-------Interupt handlers--------------------
--------------------------------------------*/
ISR(TIMER1_OVF_vect){
        overflow_count += 1;
}


ISR(PCINT1_vect){
        static bool rising_edge = true;
        if(rising_edge) usart_transmit_competition();
        rising_edge = !rising_edge;
}


/*-------Functions---------------------------
--------------------------------------------*/
void init_timer(){
        TCCR1A = 0x00;
        TCCR1B = (1<<CS10) | (1<<CS12);  // Timer mode with 1024 prescaling.
        TIMSK1 = (1 << TOIE1) ;   // Enable timer1 overflow interrupt(TOIE1).
}


void read_gyro(float *rotation) {
        i2c_start(GYRO_SAD + I2C_WRITE);
        i2c_write(GYRO_REGS);
        i2c_rep_start(GYRO_SAD+I2C_READ);

        uint8_t gyro_lo = i2c_readAck();
        uint8_t gyro_hi = i2c_readNak();
        i2c_stop();
```

```
        int16_t gyro = gyro_hi;
        gyro <<= 8;
        gyro |= gyro_lo;

        *rotation = (float)gyro;
}


void calc_offset(float *gyro, int n) {
        float sum_gyro = 0;

        for (int i = 0; i < n; i++) {
                read_gyro(gyro);
                sum_gyro += *gyro;
        }

        *gyro = sum_gyro / n;
}


void init_btn(){
        PCICR |= (1 << PCIE1);
        PCMSK1 |= (1 << PORTB0);
}

void
init_i2c(){
    i2c_init();
        write_to_reg(GYRO_SAD, 0x0F, 0x20);
        write_to_reg(GYRO_SAD, 0x00, 0x23);
}

void hw_init(){
        init_usart();
        init_i2c();
        adc_init();
        init_btn();
        sei();
        init_timer();
        sensor_init();
}


void update_angle(float offset, float *angle){
        float gyro_rotation;

        // read data
        read_gyro(&gyro_rotation);

        //Calculate dt
        uint16_t temp_dt = overflow_count * 65536 + TCNT1 - prev_time;
        prev_time = TCNT1;
        float dt = (PRESCALE * temp_dt) / F_CPU;
        overflow_count = 0;

        // integrate angle velocity
        *angle += (gyro_rotation - offset) * GYRO_SENSITIVITY_250DPS * dt;
```

```
}


uint16_t get_dist(int n){
        if(n == 0){
                uint16_t right_sensor_voltage = adc_read(0);
                return voltage_to_dist(right_sensor_voltage);
        } else{
                uint16_t left_sensor_voltage = adc_read(1);
                return voltage_to_dist(left_sensor_voltage);
        }
}


void sensor_tick(float offset, float angle, struct sensor_data* d){

        uint16_t right_dist = get_dist(0);
        uint16_t left_dist = get_dist(1);
        //angle = (int)angle;   //skips two decimals, just uses integers
        int16_t gyro_angle = (int16_t)angle;
        if(right_dist != d->right_distance || left_dist != d->left_distance ||
            gyro_angle != d->gyro_angle){
                d->right_distance = right_dist;
                d->left_distance = left_dist;
                d->gyro_angle = gyro_angle;

                send_data(d);
        }
}


void send_data(struct sensor_data* d){
        /*Data is sent in the following order every time:
                Header
                Angle(gyro)
                Right distance
                Left distance
        */
        //Header
        cli();
        usart_transmit(MEASUREMENT);
        sei();
        _delay_us(TRANSMIT_DELAY);
        //Data
        uint8_t g_hi = (int8_t)(d->gyro_angle >> 8);
        uint8_t g_lo = (int8_t)(d->gyro_angle);
        cli();
        usart_transmit(g_hi);
        sei();
        _delay_us(TRANSMIT_DELAY);
        cli();
        usart_transmit(g_lo);
        sei();
        _delay_us(TRANSMIT_DELAY);

        uint8_t r_hi = (uint8_t)(d->right_distance >> 8);
        uint8_t r_lo = (uint8_t)(d->right_distance);
```

```c
        cli();
        usart_transmit(r_hi);
        sei();
        _delay_us(TRANSMIT_DELAY);
        cli();
        usart_transmit(r_lo);
        sei();
        _delay_us(TRANSMIT_DELAY);
        uint8_t l_hi = (uint8_t)(d->left_distance >> 8);
        uint8_t l_lo = (uint8_t)(d->left_distance);
        cli();
        usart_transmit(l_hi);
        sei();
        _delay_us(TRANSMIT_DELAY);
        cli();
        usart_transmit(l_lo);
        sei();
        _delay_us(TRANSMIT_DELAY);
}


void sensor_init()
{
        state.identified = false;

        while(!state.identified)
        {
                usart_transmit(SENSOR_ID);
                _delay_ms(100);
        }
}


void sensor_identified()
{
        state.identified = true;
}


int main()
{
        hw_init();
        // calculate mean offsets
        float offset_gyro;
        calc_offset(&offset_gyro, 5000);
        //static struct sensor_data data;
        static float angle = 0;
        while(1) {
                update_angle(offset_gyro, &angle);
                _delay_ms(5);
                sensor_tick(offset_gyro, angle, &data);
        }

        return 0;
}
```

9 communication/src/communication.cpp

```
/*

file: communication.cpp
author: marno874, osklu414, felli675, edwjo109.
created: 2019-11-13

Main runner class.

Calculates robot behaviour, keeps track of robot position,
creates a representation of the map, sends data to pc program.

*/

#include <unistd.h>
#include <cmath>
#include <ctime>
#include <iostream>
#include <functional>
#include <algorithm>
#include <vector>
#include <memory>
#include <thread>
#include "communication.hpp"
#include "serial.hpp"
#include "sensor.hpp"
#include "logging.hpp"


using json = nlohmann::json;


#define ROT_OFFSET 1

//Tune this constants depending on battery power
#define STOP_DIST 225
#define ROT_RIGHT_1_DIST 150
#define ROT_RIGHT_3_DIST 275


Direction Communication::left_turn(Direction dir){
        switch (dir) {
            case Direction::UP:         return Direction::LEFT;
            case Direction::RIGHT:  return Direction::UP;
            case Direction::DOWN:       return Direction::RIGHT;
            case Direction::LEFT:       return Direction::DOWN;
        }
}

Direction Communication::right_turn(Direction dir) {
        switch (dir) {
            case Direction::UP:         return Direction::RIGHT;
            case Direction::RIGHT:  return Direction::DOWN;
            case Direction::DOWN:       return Direction::LEFT;
            case Direction::LEFT:       return Direction::UP;
        }
```

```
}

Communication::Communication
(
    const std::string& sensor_file,
    const std::string& steering_file,
    const std::string& rplidar_file
):
    map(),
    sensor(sensor_file),
    steering(steering_file),
    rplidar(rplidar_file),
    target_rot(0),
    mode(Mode::MOVING),
    direction(Direction::UP),
    x_pos(0),
    y_pos(0),
        target_dist(0),
    prev_dist(0),
    pc(std::make_shared<PC>()),
    robot_mode(RobotMode::AUTONOMOUS)

{
    rplidar.start_scanning();
    pc->on_command([this](SteeringCommand command){if(this->robot_mode ==
        RobotMode::MANUAL) this->steering.command(command);});
    pc->on_calibration([this](float kp, float kd){this->steering.calibrate(kp,
        kd);});
    sensor.set_pc(pc);
    sensor.on_competition([this](){
        if(this->robot_mode == RobotMode::MANUAL)this->robot_mode = RobotMode::
            AUTONOMOUS;
        else this->robot_mode = RobotMode::MANUAL;

        // Wait 1 second to start autonomous mode
        auto timer = std::clock();
        while((std::clock() - timer)/CLOCKS_PER_SEC < 1);
        }
    );
    steering.set_pc(pc);
}


Communication::~Communication(){
    rplidar.stop_motor();
}


bool
Communication::update() {
    pc->update();
    sensor.update();

    /* Separate manual and autonoumus mode and
    init autonomous mode if it not has been done. */
    static bool inited_auto = false;
```

```
    if(robot_mode == RobotMode::MANUAL) {
        if(inited_auto) inited_auto = false;
        return true;
    }
    else {
        if(!inited_auto) {
            autonomous_init();
            inited_auto = true;
        }
    }


    //Get measurements from sensors and rplidar.
    SensorMeasurement measurement = sensor.measurement();
    std::vector<ScanNode> curr_nodes;
    bool new_data = false;
    get_rplidar_scan(curr_nodes, new_data);

    //Calculate robot behaviour.
    bool done = calc_inst(measurement, curr_nodes);
    if (done) return false;

    steering.update();


    //Pc communication
    if (new_data) {
        pc->rplidar(curr_nodes);
        pc->robot((float)x_pos/400.0f + 0.5f, (float)y_pos/400.0f + 0.5f,
            measurement.rot * M_PI / 180.0f);
        std::thread(&Communication::update_map, this, curr_nodes, measurement).
            detach();
        pc->map(map);
    }

    return true;
}


void
Communication::get_rplidar_scan(std::vector<ScanNode>& curr_nodes, bool&
    new_data){
    static std::vector<ScanNode> old_nodes;

    //Set new rplidar measurement if any, otherwise take most recent ones.
    curr_nodes = rplidar.get_scan();
    if (curr_nodes.empty()){
        curr_nodes = old_nodes;
    }
    else{
        old_nodes = curr_nodes;
        new_data = true;
    }
}


void
```

```
Communication::autonomous_init(){

    //Wait for rplidar to return values
    std::vector<ScanNode> curr_nodes;
    bool dummy = false;
    while(curr_nodes.empty()) get_rplidar_scan(curr_nodes,dummy);

    //Wait for side sensor to return values
    while(true){
        sensor.update();
        if(sensor.measurement().right != 0) break;
    }

    //Init pos and gyro
    x_pos = 0;
    y_pos = 0;
    sensor.init_gyro(sensor.measurement().rot);
}


float
Communication::get_distance_at(float angle, vector<ScanNode>& curr_nodes, float
    rot){
    //Calculate rplidar angle relative to robot rotation
    angle += rot-target_rot;

    //Make sure angle is between 0 and 360.
    if(angle > 360) angle -= 360;
    if(angle < 0) angle += 360;


    //static constants
    static const float offset = 1.0;
    static const float LOWER_LIMIT = angle - offset;
    static const float UPPER_LIMIT = angle + offset;


    //First measurment is sometimes more then 1 deg, therefore return first
        measurement in list.
    if(angle < 1) return curr_nodes[0].dist;

    //Go through nodes until the node with the given angle is found.
    for (const ScanNode &node: curr_nodes) {
        if (LOWER_LIMIT < node.angle && node.angle < UPPER_LIMIT) return node.
            dist;
    }

    // returns 0.0 if no node at the given angle was found.
    return 0.0;
}


void
Communication::update_pos(vector<ScanNode>& curr_nodes, float rot){
    int dist = get_distance_at(0.0, curr_nodes, rot);

    //Only update pos when we know the distance to the front.
```

```
    if(dist != 0.0 ){
        //If this is the first measurement then we have no referens point to
            compare with
        if (prev_dist == 0) prev_dist = dist;
        else {
            float dist_delta = prev_dist - dist;
            /*If the previous distance is bigger then the new one,
             then set the new one as referens point and do not update posistion
                 */
            if(dist_delta < 0) WARN("New␣distance:␣",dist,"␣was␣bigger␣than␣the␣
                previous:␣", prev_dist);
            /*If distance changed more then 10 cm since last measurement,
            then ignore last measurement and set the new one as referense point.
                */
            if(abs(dist_delta) < 100){
                //Update target distance relative to new position.
                if (target_dist > 0) target_dist -= prev_dist - dist;
                //Update pos depending on direction.
                switch(direction){
                    case Direction::UP: {
                        y_pos += prev_dist - dist;
                        break;
                    }
                    case Direction::RIGHT: {
                        x_pos += prev_dist - dist;
                        break;
                    }
                    case Direction::DOWN: {
                        y_pos -= prev_dist - dist;
                        break;
                    }
                    case Direction::LEFT: {
                        x_pos -= prev_dist - dist;
                        break;
                    }
                }
            }
            //Save distance so that we can calculate the position delta next
                time we update the position.
            prev_dist = dist;
        }
    }
}


bool
Communication::calc_inst(SensorMeasurement& sensor_measurements, vector<ScanNode
    >& curr_nodes){
    static bool started = false;

    //Check if we reached the end of the map
    if(-300 < x_pos && x_pos < 300 && -300 < y_pos && y_pos < 300 && started &&
        (direction == Direction::UP)) {
        return true;
    }

    //If robot drove one tile then we have started.
```

```
if((x_pos >= 400 || y_pos >= 400 || x_pos <= -400 || y_pos <= -400) && !
    started) started = true;

static float rot = 0;
    static uint16_t left = 0;
    static uint16_t right = 0;
static bool regulate = true;
static bool adjust_right = false;
static bool adjust_left = false;
static float prev_rot = 0;

rot = sensor_measurements.rot;
left = sensor_measurements.left;
right = sensor_measurements.right;

float dist_front = get_distance_at(0.0, curr_nodes, rot);
float dist_right = get_distance_at(90.0, curr_nodes, rot);
float dist_down = get_distance_at(180.0, curr_nodes, rot);
float dist_left = get_distance_at(270.0, curr_nodes, rot);


//Calculate robot behaviour depending on current mode and sensor
    measurements.
switch(mode){
    case Mode::MOVING: {
        update_pos(curr_nodes, rot);

        //Check if we went passed the end of the wall to the right, then
            turn right.
        if(right == 0){
            target_dist = ROT_RIGHT_1_DIST;
            regulate = false;
            mode = Mode::ROTATING_RIGHT_1;
        }
        //Check if there is a wall in front of the robot, then turn left
        else if(dist_front != 0.0 && dist_front < STOP_DIST){;
            correct_position();
            mode = Mode::ROTATING_LEFT;
            direction = left_turn(direction);
            prev_dist = dist_left;
            target_rot += 90;
            steering.set_rotation(Rotation::LEFT);
        }

        steering.update_regulation(right, (rot-target_rot), regulate,
            get_distance_at(0.0, curr_nodes, rot));
        break;
    }
    case Mode::ROTATING_LEFT: {
        //If robot rotated into correct interval.
        if (rot >= target_rot - ROT_OFFSET && rot <= target_rot + ROT_OFFSET
            ) {
            //If the reason behind the rotation was an over rotation to the
                right
            if(adjust_left){
                mode = Mode::ROTATING_RIGHT_2;
                adjust_left = false;
```

```
                steering.set_rotation(Rotation::RIGHT);
            }
            else {
                mode = Mode::MOVING;
                steering.set_rotation(Rotation::NONE);
            }
        }
        // If robot over rotated to the left.
        else if (rot >= target_rot + ROT_OFFSET) {
            WARN("Turned␣too␣far,␣adjusting", rot);
            mode = Mode::ROTATING_RIGHT_2;
            steering.set_rotation(Rotation::RIGHT);
            adjust_right = true;
        }
        // If robot has not reached correct rotation, then continue rotating
        else {
            if (prev_rot != rot) steering.rotate_regulated(abs(target_rot-
                rot));
        }
        break;
    }
    case Mode::ROTATING_RIGHT_1:{
        update_pos(curr_nodes, rot); // Update pos only when moving forward
        //Check if rotation was initiated by a bad sensor value
        if(right != 0){
            WARN("Right␣not␣zero:␣", right);
            mode = Mode::MOVING;
            regulate = true;
        }
        //Check if we drove out enough from the corner.
        else if(target_dist <= 0){
            correct_position();
            mode = Mode::ROTATING_RIGHT_2;
            direction = right_turn(direction);
            target_rot -=  90;
            steering.set_rotation(Rotation::RIGHT);
        }
        break;
    }
    case Mode::ROTATING_RIGHT_2:{
        //Check if robot rotation is in correct interval.
        if (rot >= target_rot - ROT_OFFSET && rot <= target_rot + ROT_OFFSET
            ) {
            //Check if the reason behind the rotation is because the robot
                over rotated to the left.
            if(adjust_right){
                mode = Mode::MOVING;
                adjust_right = false;
                steering.set_rotation(Rotation::NONE);
                break;
            }
            mode = Mode::ROTATING_RIGHT_3;
            target_dist = ROT_RIGHT_3_DIST;
            prev_dist = dist_front;
            steering.set_rotation(Rotation::NONE);
        }
        //Check if the robot over rotated.
```

```
            else if(rot <= target_rot - ROT_OFFSET){
                WARN("Rotation␣went␣to␣far,␣adjusting", rot);
                mode = Mode::ROTATING_LEFT;
                steering.set_rotation(Rotation::LEFT);
                adjust_left = true;
            }
            // If the robot has not yet reached the correct rotation, the
                continue rotating
            else
            {
                if (prev_rot != rot) steering.rotate_regulated(abs(target_rot-
                    rot));
            }
            break;
        }
        case Mode::ROTATING_RIGHT_3:{
            update_pos(curr_nodes, rot);
            //Check if robot drove in towards the wall enough after right turn.
            if(target_dist <= 0){ // || dist_front < STOP_DIST){
                regulate = true;
                mode = Mode::MOVING;
            }
            break;
        }
    }

    //Save prev rotation to be able to know if it changed since last time an
        instruction was calculated.
    prev_rot = rot;

    return false;
}


void
Communication::correct_position(){
    y_pos = round(y_pos/400.0f)*400;
    x_pos = round(x_pos/400.0f)*400;
}


void
Communication::update_map(const std::vector<ScanNode>& nodes, const
    SensorMeasurement& measurement) {
    // update internal map
    for(const ScanNode& node : nodes){
        // delta vector between robot and hit tile
        float d_x = -(float)node.dist / (float)Map::TILE_SIZE * cos((-node.angle
             + measurement.rot - 90) * M_PI / 180.0f);
        float d_y = -(float)node.dist / (float)Map::TILE_SIZE * sin((-node.angle
             + measurement.rot - 90) * M_PI / 180.0f);

        // calculate coordinates, src = robot position, dst = hit position
        float src_x = (float)x_pos/400.0f + 0.5f;
        float src_y = (float)y_pos/400.0f + 0.5f;
        float dst_x = d_x + src_x;
        float dst_y = d_y + src_y;
```

```
        // create a small delta vector to use for iterating through points
            between src and dst
        float d = sqrt(pow(d_x, 2) + pow(d_y, 2));
        d_x /= d;
        d_y /= d;
        d_x *= 0.25f;
        d_y *= 0.25f;

        // keep track of distance between src and dst
        float current_distance = d;
        float last_distance = d + 1.0f;

        // go through tiles between src and dst and set them to empty
        int last_col = Map::MAP_SIZE, last_row = Map::MAP_SIZE;

        // keep setting to empty until distance between src and dst gets greater
             (meaning that src has passed dst)
        while(current_distance <= last_distance)
        {
            // col and row of tile closest to src (has to be floored since
                (-0.5, -0.5) corresponds to tile (-1, -1)))
            int empty_col = floor(src_x + Map::ORIGIN);
            int empty_row = floor(src_y + Map::ORIGIN);

            // only update every time a new tile is reached
            if(empty_col != last_col || empty_row != last_row)
            {
                map.update(empty_col, empty_row, Tile::EMPTY);
            }

            // move src by delta
            src_x += d_x;
            src_y += d_y;

            // update last distance and current distance
            last_distance = current_distance;
            current_distance = sqrt(pow(src_x - dst_x, 2) + pow(src_y - dst_y,
                2));

            // update last col and row
            last_col = empty_col;
            last_row = empty_row;
        }

        int wall_col = floor(src_x + (float)Map::ORIGIN);
        int wall_row = floor(src_y + (float)Map::ORIGIN);

        // set tile at dst to wall (is this guaranteed to be a new tile from
            last?)
        map.update(wall_col, wall_row, Tile::WALL);
    }
}
```

10 communication/src/socket.cpp

```
/*

file: socket.cpp
author: juska933
created: 2019-11-13

Wrapper class for socket communication.

*/

#include <stdio.h>
#include <string.h>   //strlen
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>   //close
#include <arpa/inet.h>    //close
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/time.h> //FD_SET, FD_ISSET, FD_ZERO macros
#include <vector>
#include <string>
#include <iostream>

#include <json/json.hpp>

#include "socket.hpp"
#include "logging.hpp"


using json = nlohmann::json;


static std::vector<std::string>
split_str(const std::string &text, std::string delim)
{
    std::vector<std::string> res;
    size_t i = 0, len = delim.length();
    size_t prev = 0;
    while (i+len <= text.length()) {
                if (text.substr(i, len) == delim) {
                        res.push_back(text.substr(prev, i-prev));
                        i += len;
                        prev = i;
                } else {
                        i++;
                }
        }
        res.push_back(text.substr(prev, text.size()-prev));
        return res;
}


Socket::Socket(int max_clients)
{
```

```
    this->max_clients = max_clients;
    client_sockets.assign(max_clients, 0);
}

void
Socket::send_to_clients_json(json msg)
{
    send_to_clients(msg.dump());
}

void Socket::send_to_clients_json(std::string route, json msg)
{
    json patch = {{"route", route}};
    msg.merge_patch(patch);
    send_to_clients(msg);
}


void Socket::send_to_clients(std::string msg){
    //check_incoming_message();
    msg = msg + "__MSG_END__";
    for (unsigned i = 0; i < client_sockets.size(); i++) {
        int client = client_sockets[i];
        if (client == 0)
            continue;
        if( send(client, msg.c_str(), msg.length(), 0) != (ssize_t)msg.length()
            ){
            WARN("Could not send message to client");
            client_sockets[i] = 0;
        }
    }
}

void Socket::emit_message(int sd, std::string msg){
    std::string entire_msg = last_msg_buffer + msg;
    std::vector<std::string> packets = split_str(entire_msg, "__MSG_END__");
    std::string last_msg = packets[packets.size()-1];
    if (last_msg.size() > 0) {
        last_msg_buffer = last_msg; // Save partial message
    }
    packets.pop_back();
    for (MessageHandler message_handler: message_handlers) {
        for (std::string packet: packets)  {
            message_handler(packet, sd);
        }

    }
    std::vector<json> json_packets;
    for (std::string packet: packets) {
        try {
            json data = json::parse(packet);
            json_packets.push_back(data);
        } catch (const nlohmann::detail::parse_error& e) {
            WARN("Incomplete json message from socket: " + packet);
        }
    }
    for (JsonHandler json_handler: json_handlers) {
```

```
        for (json packet: json_packets)  {
            json_handler(packet, sd);
        }

    }
}

void Socket::on_message(MessageHandler message_handler) {
    this->message_handlers.push_back(message_handler);
}

void Socket::on_json(JsonHandler json_handler){
    this->json_handlers.push_back(json_handler);
}

void Socket::start_socket(){
    //create a master socket
    WARN("STARTING SOCKET");
    master_socket = socket(AF_INET , SOCK_STREAM , 0);
    if (!master_socket) {
        perror("socket failed");
        exit(EXIT_FAILURE);
    }

    //set master socket to allow multiple connections ,
    //this is just a good habit, it will work without this
    int multiple_socket_res = setsockopt(master_socket, SOL_SOCKET, SO_REUSEADDR
        , (char *)&opt, sizeof(opt));
    if(multiple_socket_res < 0) {
        perror("setsockopt");
        exit(EXIT_FAILURE);
    }

    //type of socket created
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons( PORT );

    //bind the socket to localhost port 8000
    if (bind(master_socket, cast_sock_addr(), sizeof(address))<0)
    {
        perror("bind failed");
        exit(EXIT_FAILURE);
    }
    printf("Listener on port %d \n", PORT);

    // try to specify maximum of 3 pending connections for the master socket
    if (listen(master_socket, 3) < 0)
    {
        perror("listen");
        exit(EXIT_FAILURE);
    }

    //accept the incoming connection
    addrlen = sizeof(address);
    puts("Waiting for connections ...");
}
```

```
void Socket::check_activity(){
    //clear the socket set
    FD_ZERO(&readfds);

    //add master socket to set
    FD_SET(master_socket, &readfds);
    int max_sd = master_socket;

    //add child sockets to set
    for (int i = 0 ; i < max_clients ; i++)
    {
        //socket descriptor
        int sd = client_sockets[i];

        //if valid socket descriptor then add to read list
        if(sd > 0)
            FD_SET( sd , &readfds);

        //highest file descriptor number, need it for the select function
        if(sd > max_sd)
            max_sd = sd;
    }

    //wait for an activity on one of the sockets , timeout is NULL (last arg),
    //so wait indefinitely
    // NOTE: Might be problem when integrating bcz of stalling
    timeval timeout = {0, ACTIVITY_DELAY_MICRO_SECONDS};
    activity = select( max_sd + 1 , &readfds , NULL , NULL , &timeout);

    if ((activity < 0) && (errno!=EINTR)) {
        return;
    }
    try_accept_client();
    check_incoming_message();
}

sockaddr* Socket::cast_sock_addr(){
    return (struct sockaddr *) &address;
}


void Socket::try_accept_client(){
    //If something happened on the master socket ,
    //then its an incoming connection
    if (FD_ISSET(master_socket, &readfds))
    {
        if ((new_socket = accept(master_socket,  cast_sock_addr(), (socklen_t*)&
            addrlen))<0)
        {
            perror("accept");
            exit(EXIT_FAILURE);
        }

        //inform user of socket number - used in send and receive commands
        printf("New connection , socket fd is %d , ip is : %s , port : %d  \n" ,
             new_socket , inet_ntoa(address.sin_addr) , ntohs(address.sin_port));
```

```
        //send new connection greeting message


        //add new socket to array of sockets
        for (int i = 0; i < max_clients; i++)
        {
            //if position is empty
            if( client_sockets[i] == 0 )
            {
                client_sockets[i] = new_socket;
                break;
            }
        }
    }
}

void Socket::check_incoming_message(){
    for (int i = 0; i < max_clients; i++)
    {
        int sd = client_sockets[i];

        if (FD_ISSET( sd , &readfds))
        {
            //Check if it was for closing , and also read the
            //incoming message
            int num_bytes_read = read( sd , buffer, 1024);
            if (num_bytes_read <= 0)
            {
                //Somebody disconnected , get his details and print
                getpeername(sd , cast_sock_addr(), (socklen_t*)&addrlen);
                printf("Host disconnected , ip %s , port %d \n", inet_ntoa(
                    address.sin_addr) , ntohs(address.sin_port));

                //Close the socket and mark as 0 in list for reuse
                close(sd);
                client_sockets[i] = 0;
            }

            //Echo back the message that came in
            else
            {
                //set the string terminating NULL byte on the end
                //of the data read
                //buffer[num_bytes_read] = '\0';



                //printf("msg incoming: %s\n", buffer);
                emit_message(sd, std::string(buffer, num_bytes_read));
            }
        }
    }
}
```

11 communication/src/map.cpp

```
/*

file: map.cpp
author: osklu414
created: 2019-11-15

Map model class.

*/


#include "map.hpp"


// robot starts in the middle of tile (0, 0)
Robot::Robot() : x(0.5f), y(0.5f), r(0) {}


Map::Map()
{
    // each tile starts of as unknown
    for(int r = 0; r < MAP_SIZE; r++)
    {
        for(int  c = 0; c < MAP_SIZE; c++)
        {
            tiles[r][c] = Tile::UNKNOWN;
            confidence_empty[r][c] = 0;
            confidence_wall[r][c] = 0;
        }
    }

    // the starting position of the robot is empty
    //tiles[ORIGIN][ORIGIN] = Tile::EMPTY;
}

Map::~Map()
{

}


void Map::set(const int col, const int row, Tile tile)
{
    tiles[row][col] = tile;
}


Tile Map::get(const int col, const int row) const
{
    return tiles[row][col];
}


void Map::update(const int col, const int row, const Tile tile)
{
```

```
    switch(tile)
    {
        case Tile::EMPTY:
        {
            if(++confidence_empty[row][col] > confidence_wall[row][col])
            {
                 if(confidence_empty[row][col] >= CONFIDENCE_MIN) tiles[row][col
                     ] = Tile::EMPTY;
            }
            break;
        }
        case Tile::WALL:
        {
            if(++confidence_wall[row][col] > confidence_empty[row][col])
            {
                if(confidence_wall[row][col] >= CONFIDENCE_MIN) tiles[row][col]
                    = Tile::WALL;
            }
            break;
        }
        case Tile::UNKNOWN:
        {
            break;
        }
    }
}


void Map::clean()
{
    // start from origin and find outer walls
    int c = ORIGIN;
    int r = ORIGIN;


}
```

12 communication/src/main.cpp

```
/*

file: main.hpp
author: osklu414
created: 2019-11-14

Program entry point. Identifies modules connected via UART and creates
    communication object.

*/


#include <iostream>
#include <atomic>

#include <unistd.h>
#include <signal.h>

#include "logging.hpp"
#include "serial.hpp"
#include "communication.hpp"
#include "rplidar.hpp"

static std::atomic<bool> quit(false);

void signal_callback(int) { quit.store(true); }

void identify_modules(std::string& sensor_file, std::string& steering_file, std
    ::string& rplidar_file);

int main(int argc, char* argv[])
{
    TRACE("communication␣module␣started");

    // make sure destructors are called normally
    struct sigaction sa;
    memset(&sa, 0, sizeof(sa));
    sa.sa_handler = signal_callback;
    sigfillset(&sa.sa_mask);
    sigaction(SIGINT, &sa, NULL);

    // identify modules
    std::string sensor_file, steering_file, rplidar_file;
    identify_modules(sensor_file, steering_file, rplidar_file);

    // start communication module and update it until signal or update returns
        false
    Communication communication(sensor_file, steering_file, rplidar_file);
    while(communication.update() && !quit.load());

    TRACE("communication␣module␣stopped");
    return 0;
}
```

```cpp
void identify_modules
(
    std::string& sensor_file,
    std::string& steering_file,
    std::string& rplidar_file
)
{
    // determine which port is belongs to which device/module
    TRACE("identifying␣modules");

    Serial serials[3];
    serials[0].open("/dev/ttyUSB0");
    serials[1].open("/dev/ttyUSB1");
    serials[2].open("/dev/ttyUSB2");


    // identify steering module
    TRACE("identifying␣steering␣module...");
    bool steering_identified = false;
    while(!steering_identified)
    {
        // make sure program quits when sending signal
        if(quit.load()) break;

        for(int s = 0; s < 3; s++)
        {
            uint8_t module_id;
            if(serials[s].read(&module_id, 1) == 1 && (ModuleId)module_id ==
                STEERING)
            {
                steering_identified = true;
                std::swap(serials[s], serials[2]); // move identified serial to
                    back
                steering_file = serials[2].get_file();
                break;
            }
        }
    }
    TRACE("steering␣module␣identified␣at␣", steering_file);


    // identify sensor module
    TRACE("identifying␣sensor␣module...");
    bool sensor_identified = false;
    while(!sensor_identified)
    {
        // make sure program quits when sending signal
        if(quit.load()) break;

        for(int s = 0; s < 2; s++)
        {
            uint8_t module_id;
            if(serials[s].read(&module_id, 1) == 1 && (ModuleId)module_id ==
                SENSOR)
            {
                sensor_identified = true;
                std::swap(serials[s], serials[1]); // move identified serial to
```

```
                    back
                sensor_file = serials[1].get_file();
                break;
            }
        }
    }
    TRACE("sensor␣module␣identified␣at␣", sensor_file);

    // identify rplidar device
    TRACE("identifying␣rplidar␣device...");
    rplidar_file = serials[0].get_file();
    TRACE("rplidar␣device␣identified␣at␣", rplidar_file);

    serials[0].close();
    serials[1].close();
    serials[2].close();

}
```

13 communication/src/serial.cpp

```
/*

file: serial.cpp
author: osklu414, juska933
created: 2019-11-14

Wrapper class for serial I/O.

*/

#include <stdint.h>
#include <stdio.h>   /* Standard input/output definitions */
#include <string.h>  /* String function definitions */
#include <unistd.h>  /* UNIX standard function definitions */
#include <fcntl.h>   /* File control definitions */
#include <errno.h>   /* Error number definitions */
#include <termios.h> /* POSIX terminal control definitions */

#include <iostream>

#include "logging.hpp"
#include "serial.hpp"



Serial::Serial() : fd(-1), file() {}

Serial::~Serial() {}


void Serial::open(const std::string& file)
{
    //TRACE("serial open: ", file);
    fd = ::open(file.data(), O_RDWR | O_NOCTTY);
    if(fd < 0) {
        ERROR("serial open ", file);
        return;
    }

    //TRACE("setting options");
    struct termios options;
    tcgetattr(fd, &options);
    cfsetispeed(&options, BAUD);
        cfsetospeed(&options, BAUD);

        options.c_cflag |= (CLOCAL | CREAD);
        options.c_cflag &= ~CSIZE; /* Mask the character size bits */
        options.c_cflag |= CS8; /* Select 8 data bits */
        options.c_lflag &= ~(ICANON | ECHO | ECHOE | ISIG); /* Raw input */
        options.c_oflag &= ~OPOST; /* Raw output */

        tcsetattr(fd, TCSANOW, &options);

    if(!set_blocking(false))
    {
```

```
        return;
    }

    this->file = file;
}

void Serial::close()
{
    //TRACE("serial close");
    ::close(fd);
    this->file = "";
    this->fd = -1;
}


int Serial::write(const uint8_t* bytes, unsigned int size)
{
    //TRACE("serial write ", size, " bytes to ", get_file());
    int written = ::write(fd, bytes, size);
    if(written < size) WARN("wrote␣", written, '/', size, "␣bytes");
    return written;
}

int Serial::read(uint8_t* bytes, unsigned int size)
{
    //TRACE("serial read ", size, " bytes from ", get_file());
    int read = ::read(fd, bytes, size);
    if(read < size); //WARN("read ", read, '/', size, " bytes");
    if(read < 0) return 0;
    return read;
}


bool Serial::set_blocking(bool block)
{
    int flags = fcntl(fd, F_GETFL, 0);
    if (flags == -1)
    {
        WARN("serial␣set_blocking:␣unable␣to␣get␣flags");
        return false;
    }

    if (block)
        flags &= ~O_NONBLOCK;
    else
        flags |= O_NONBLOCK;

    if(fcntl(fd, F_SETFL, flags) == -1)
    {
        WARN("serial␣set_blocking:␣unable␣to␣set␣blocking");
        return false;
    }

    return true;
}
```

```
int Serial::get_fd()
{
    return fd;
}

std::string Serial::get_file()
{
    return file;
}
```

14 communication/src/sensor.cpp

```
/*

file: sensor.cpp
author: osklu414
created: 2019-11-14

Interface to sensor module.

*/

#include "sensor.hpp"
#include "logging.hpp"
#include "pc.hpp"
#include <bitset>


Sensor::Sensor(const std::string& file) : Module(file), rx_type(SensorRx::NONE),
     rx_field(0), latest_measurement(), start_rot(0)
{
    //TRACE("sensor constructor: called with file ", file);
    transmit_identified();
}

Sensor::~Sensor() {}

void Sensor::update()
{
        //TRACE("sensor update:");
        // read transmissions from sensor module
        bool read = true;
        while(read)
        {
                if(rx_type == SensorRx::NONE)
                {
                        //TRACE("checking for rx type");
                        SensorRx new_rx_type;
                        if(serial.read((uint8_t*)&new_rx_type, 1) == 1)
                        {
                                //TRACE("new rx from sensor module with id ", (
                                    int)new_rx_type);
                                rx_type = new_rx_type;
                                rx_field = 0;
                        }
                }

                switch(rx_type)
                {
                        case SensorRx::MEASUREMENT:
                        {
                                //TRACE("receiving measurement byte(s)");
                                // new measurement
                                const uint8_t n_fields = 6;
                                //TRACE("afwpwokdwpodkwpo ", (int)n_fields, " -
                                    ", (int)rx_field, " = ", n_fields - rx_field)
                                    ;
```

```
		rx_field += serial.read(rx_field_buffer +
		    rx_field, n_fields - rx_field);
		if(rx_field == n_fields)
		{
			int16_t rot_raw = (rx_field_buffer[0] <<
			     8) | rx_field_buffer[1];
			float rot = rot_raw + 720;
			rot -= start_rot;
			uint16_t right = (rx_field_buffer[2] <<
			    8) | rx_field_buffer[3];
			uint16_t left = (rx_field_buffer[4] <<
			    8) | rx_field_buffer[5];

			/*
			TRACE("1: ", (bitset<8>)rx_field_buffer
			    [0]," 2: ", (bitset<8>)
			    rx_field_buffer[1]," 3: ",
			(bitset<8>)rx_field_buffer[2]," 4: ", (
			    bitset<8>)rx_field_buffer[3]," 5: ",
			    (bitset<8>)rx_field_buffer[4]," 6: ",
			     (bitset<8>)rx_field_buffer[5]);
			*/


			// store new measurement
			SensorMeasurement measurement{rot, left,
			     right};
			pc->sensor(measurement);
			latest_measurement = measurement;

			//TRACE("received measurement from
			    sensor module");
			//TRACE("rot: ", rot, ", left: ", left,
			    ", right: ", right);

			// there might be more to read
			rx_type = SensorRx::NONE;
			rx_field = 0;
		}
		else
		{
			read = false;
		}
		break;
	}
	case SensorRx::COMPETITION:
	{
		// competition mode button pressed
		//TRACE("received competition button press from
		    sensor module");
		// there might be more to read
		competition_callback();
		TRACE("competition␣button␣pressed");
		rx_type = SensorRx::NONE;
		break;
	}
	case SensorRx::NONE:
```

```
                        default:
                                read = false;
                                rx_type = SensorRx::NONE; // in case received
                                    byte is not "correct"
                                break;
                }
        }
}


SensorMeasurement Sensor::measurement()
{
        return latest_measurement;
}

void Sensor::transmit_identified()
{
    //TRACE("sensor transmit identified: ");
    uint8_t bytes[1];
    bytes[0] = static_cast<uint8_t>(SensorTx::IDENTIFIED);
    //TRACE("transmitting identified");
    serial.write(bytes, 1);
    //TRACE("transmitting identified done");
}

void Sensor::on_competition(Sensor::CompetitionCallback callback)
{
        competition_callback = callback;
}

void Sensor::init_gyro(float rot){
        start_rot = rot;
}
```

15 communication/src/rplidar.cpp

```
/*

file: rplidar.cpp
author: juska933
created: 2019-11-17

RPLIDAR driver wrapper class.

*/


#include "rplidar.hpp"
#include <vector>
#include <signal.h>
#include "stdio.h"
#include "logging.hpp"

RPLidar::RPLidar(const std::string& port_name) {
    INFO("Rplidar constructor, port: ", port_name);
    opt_com_path = port_name;
    driver = RPlidarDriver::CreateDriver(DRIVER_TYPE_SERIALPORT);
    if (!driver) {
        status = INSUFFICIENT_MEMORY;
        print_err("Insufficient memory.\n");
    }
    int baudrate = try_set_baudrate();
    if (baudrate < 0) {
        status = NO_DRIVER_CONNECTION;
        print_err("Could not connect to driver.\n");
    }
    if(!check_health()){
        status = BAD_HEALTH;
        print_err("Bad health.\n");
    }
}

RPLidar::~RPLidar() {
    on_finish();
}

bool RPLidar::is_ok(){
    return status >= 0;
}

void RPLidar::print_err(const char* msg) {
    ERROR(msg);
}

void RPLidar::start_scanning(){
    if (status < 0){
        return;
    }
    driver->startMotor();
    driver->startScan(0, 1);
    status = SCANNING;
```

```
}

void RPLidar::print_scan(){
    vector<ScanNode> nodes = get_scan();
    for (const ScanNode &node: nodes) {
        node.print();
    }
}
vector<ScanNode> RPLidar::get_scan(){
    vector<ScanNode> res;

    if (status == OK) {
        start_scanning();
    } else if (status < 0) {
        return res;
    }
    size_t count = 8192;
    rplidar_response_measurement_node_hq_t nodes[8192];

    // Timeout = 0 -> No waiting
    op_result = driver->grabScanDataHq(nodes, count, 0);
    // cout is now equal to how many nodes were fetched
    if (IS_OK(op_result)) {
        driver->ascendScanData(nodes, count);
        for (int pos = 0; pos < (int)count ; ++pos) {
            res.push_back({
                nodes[pos].dist_mm_q2/4,
                nodes[pos].angle_z_q14 * 90.f / (1 << 14),
                nodes[pos].quality
            });

        }
    }
    return res;
}

void RPLidar::print_node(rplidar_response_measurement_node_hq_t node){
    printf("%s theta: %03.2f Dist: %08.2f Q: %d \n",
    (node.flag & RPLIDAR_RESP_MEASUREMENT_SYNCBIT) ?"S ":"  ",
    (node.angle_z_q14 * 90.f / (1 << 14)),
    node.dist_mm_q2/4.0f,
    node.quality);
}

void RPLidar::stop_motor(){
    driver->stop();
    driver->stopMotor();
}

void RPLidar::on_finish(){
    RPlidarDriver::DisposeDriver(driver);
    driver = NULL;
}

int RPLidar::try_set_baudrate() {
    rplidar_response_device_info_t devinfo;
```

```cpp
    vector<uint32_t> baudrates = {115200, 256000};
    for(uint32_t baudrate: baudrates)
    {
        if(!driver)
            driver = RPlidarDriver::CreateDriver(DRIVER_TYPE_SERIALPORT);
        if(IS_OK(driver->connect(opt_com_path.data(), baudrate)))
        {
            op_result = driver->getDeviceInfo(devinfo);

            if (IS_OK(op_result))
            {
                return baudrate;
            }
            else
            {
                delete driver;
                driver = NULL;
            }
        }
    }
    return -1;
}

bool RPLidar::check_health(){
    rplidar_response_device_health_t healthinfo;
    op_result = driver->getHealth(healthinfo);
    if (IS_OK(op_result)) { // the macro IS_OK is the preperred way to judge
        whether the operation is succeed.
        printf("RPLidar␣health␣status␣:␣%d\n", healthinfo.status);
        if (healthinfo.status == RPLIDAR_STATUS_ERROR) {
            print_err("Error,␣rplidar␣internal␣error␣detected.␣Please␣reboot␣the
                ␣device␣to␣retry.\n");
            // enable the following code if you want rplidar to be reboot by
                software
            // driver->reset();
            return false;
        } else {
            return true;
        }

    } else {
        printf("RPLidar:␣Cannot␣retrieve␣the␣lidar␣health␣code\n");
        return false;
    }
}
```

16 communication/src/pc.cpp

```
/*

file: pc.cpp
author: osklu414, juska933
created: 2019-11-25

Interface to PC client.

*/


#include <vector>
#include <string>

#include <json/json.hpp>
#include <deque>
#include "logging.hpp"
#include "rplidar.hpp"
#include "pc.hpp"


using json = nlohmann::json;


PC::PC() : socket(30), command_callback(), calibration_callback(), map_clock(std
    ::clock())
{
    // route all received data here
    socket.on_json([this](json data, int sd)
    {
        // json must have id
        if (!data.contains("id")) return;

        std::string id = data["id"];
        if(id == "command" && this->command_callback)
        {
            TRACE("received␣command␣from␣pc");
            SteeringCommand command = (SteeringCommand)data["type"].get<int>();
            this->command_callback(command);
        }
        else if(id == "calibration")
        {
            TRACE("received␣calibration␣from␣pc");
            float kp = data["kp"].get<float>();
            float kd = data["kd"].get<float>();
            this->calibration_callback(kp, kd);
        }
        else
        {
            TRACE("received␣unknown␣id␣from␣pc");
        }
    });
    socket.start_socket();
}
```

```cpp
PC::~PC()
{

}


void PC::update()
{
    socket.check_activity();
}


void PC::send_json(const json& data)
{
    TRACE("sending json to pc");
    socket.send_to_clients_json(data);
}

void PC::message(const std::string& text)
{
    //TRACE("sending message to pc");
    socket.send_to_clients_json
    ({
        {"id", "message"},
        {"text", text}
    });
}

void PC::tile(const int col, const int row, Tile tile)
{
    //TRACE("sending tile to pc");
    socket.send_to_clients_json
    ({
        {"id", "tile"},
        {"col", col},
        {"row", row},
        {"type", (int)tile}
    });
}

void PC::map(const Map& map)
{
    std::clock_t now = std::clock();
    if((now - map_clock) / CLOCKS_PER_SEC >= 1)
    {
        std::vector<Tile> tiles;
        for(int r = 0; r < Map::MAP_SIZE; r++)
        {
            for(int c = 0; c < Map::MAP_SIZE; c++)
            {
                tiles.push_back(map.get(c, r));
            }
        }
        socket.send_to_clients_json
        ({
            {"id", "map"},
            {"tiles", tiles}
```

```
        });
        map_clock = std::clock();
    }
    //TRACE("sending map to pc");
}

void PC::robot(const float x, const float y, const float r)
{
    //TRACE("sending robot state to pc");
    socket.send_to_clients_json
    ({
        {"id", "robot"},
        {"x", x},
        {"y", y},
        {"r", r}
    });
}


void PC::rplidar(const std::vector<ScanNode>& nodes)
{
    std::vector<json> json_nodes;
    for (const ScanNode &node: nodes)
    {
        json_nodes.push_back
        ({
            {"dist", node.dist},
            {"angle", node.angle},
            {"quality", node.quality}
        });
    }
    socket.send_to_clients_json({
        {"id", "rplidar"},
        {"nodes", json_nodes}
    });
}


void PC::point(const float col, const float row)
{
    socket.send_to_clients_json
    ({
        {"id", "point"},
        {"col", col},
        {"row", row}
    });
}


void PC::sensor(const SensorMeasurement& measurement)
{
    //TRACE("sending sensor measurement to pc");
    socket.send_to_clients_json
    ({
        {"id", "sensor"},
        {"left", measurement.left},
        {"right", measurement.right},
```

```
        {"rot", measurement.rot}
    });
}


void PC::steering(const SteeringControl& control)
{
    //TRACE("sending steering control to pc");
    socket.send_to_clients_json
    ({
        {"id", "steering"},
        {"left_speed", control.left_speed},
        {"right_speed", control.right_speed},
        {"left_forward", control.left_forward},
        {"right_forward", control.right_forward}
    });
}


void PC::on_command(CommandCallback callback)
{
    command_callback = callback;
}

void PC::on_calibration(CalibrationCallback callback)
{
    calibration_callback = callback;
}
```

17 communication/src/module.cpp

```cpp
/*

file: module.cpp
author: osklu414
created: 2019-11-14

Base class for modules (steering and sensor).

*/


#include "logging.hpp"
#include "module.hpp"


Module::Module(const std::string& file) : serial()
{
    serial.open(file);
}

Module::~Module()
{
    serial.close();
}


void
Module::set_pc(const std::shared_ptr<PC>& pc)
{
    this->pc = pc;
}
```

18 communication/src/steering.cpp

```
/*

file: steering.cpp
author: marno874, osklu414, felli675, edwjo109.
created: 2019-11-14

Choose what instructions that should be sent to steering module (AVR).

*/

#include "steering.hpp"
#include "logging.hpp"
#include "pc.hpp"
#include <ctime>

//Tune this depending on battery power
#define MAX_SPEED 0.15f
#define NEAR_WALL_SPEED 0.03f
#define NEAR_WALL_DIST 650
#define ROT_SPEED 0.15f
#define FORWARD_SPEED 0.1f


Steering::Steering(const std::string& file) :
        Module(file),
        kp(1.5f),
        kd(1.0f),
        latest_control(),
        rotation(Rotation::NONE),
        prev_rotation(Rotation::NONE),
    clock_steering(std::clock()),
    side_dist(0),
    front_dist(0),
    d_rot(0),
    regulate(true)
{
    transmit_identified();
}


Steering::~Steering() {
    command(SteeringCommand::HALT);
}


void
Steering::set_rotation(Rotation rot) {
    rotation = rot;
}


void
Steering::update() {
    //Check if changed state
    if(prev_rotation != rotation){
```

```
            control_speed(0.0f, 0.0f);
        control_direction(false, false);

            switch (rotation) {
                case Rotation::NONE:
                    control_direction(true, true);
            control_speed(FORWARD_SPEED, FORWARD_SPEED);
                    break;
                case Rotation::LEFT:
                    control_direction(false, true);
            control_speed(ROT_SPEED, ROT_SPEED);
                    break;
                case Rotation::RIGHT:
                    control_direction(true, false);
            control_speed(ROT_SPEED, ROT_SPEED);
                    break;
            }
    }
    // If robot is moving forward and regulation should be applied.
    else if (((std::clock() - clock_steering)/(float)CLOCKS_PER_SEC > 0.01f) &&
        (rotation == Rotation::NONE)){
        clock_steering = std::clock();
        move_forward();
    }
    //Save rotation to be able to know if rotation has been changed.
        prev_rotation = rotation;
}


void
Steering::move_forward(){
    static const float PREF_SIDE_DIST = 130;
    static const float SIDE_DIST_RANGE = 80;
    static const float GYRO_RANGE = 5;
    static float max_speed = 0.15;

    //If no regulation should be applied then just move straight forward with
        low speed.
    if(!regulate){
        latest_control.left_forward = true;
        latest_control.right_forward = true;

        latest_control.left_speed = 0.0001;
        latest_control.right_speed = 0.0001;

        control(latest_control);
        return;
    }

    //Make robot slow down when approching a wall.
    max_speed = (front_dist < NEAR_WALL_DIST) ? NEAR_WALL_SPEED : MAX_SPEED;

    //Clamp side dist to interval.
    if(side_dist == 0) side_dist = 299;
    side_dist = std::clamp(side_dist, PREF_SIDE_DIST - SIDE_DIST_RANGE,
        PREF_SIDE_DIST + SIDE_DIST_RANGE);
    //Clamp rotation to interval.
```

```
    d_rot = std::clamp(d_rot, -GYRO_RANGE, GYRO_RANGE);

    //Calculate regulation constants.
    float _kp = kp*(max_speed/SIDE_DIST_RANGE);
    float _kd = kd*(max_speed/GYRO_RANGE);

    //Proportional Term
    float e = _kp*(side_dist - (float)PREF_SIDE_DIST);
    //Rotation from wanted angle
    float u = _kd*d_rot;
    //Output from regulation
    float out = e + u;
    //Bind to min/max
    out = std::clamp(out, (-max_speed), max_speed);

    //Set motor rotation direction
    latest_control.left_forward = true;
    latest_control.right_forward = true;

    //Set speed of wheels, the small term prevent the wheels for stop spinning.
    latest_control.left_speed = max_speed + out + 0.000001;
    latest_control.right_speed = max_speed - out + 0.000001;

    control(latest_control);
}


void
Steering::update_regulation(float right, float d_rotation, bool reg, float front
    ){
    side_dist = right;
    front_dist = front;
    d_rot = d_rotation;
    regulate = reg;
}


void
Steering::rotate_regulated(float rot){
    float max = 0.3;
    float n_rot = rot/90.0f;
    float out = std::clamp(max*n_rot, 0.1f, max);
    control_speed(out, out);
}


void
Steering::command(const SteeringCommand command)
{
    const float speed_max = 0.5f;
    const float speed_min = 0.5f;

    //Set direction and speed of wheels depending on steering command.
    switch(command)
    {
        case SteeringCommand::ROTATE_LEFT:
            control_direction(false, true);
```

```
            control_speed(0.15f, 0.15f);
            break;
        case SteeringCommand::ROTATE_RIGHT:
            control_direction(true, false);
            control_speed(0.15f, 0.15f);
            break;
        case SteeringCommand::DRIVE_FORWARD:
            control_direction(true, true);
            control_speed(speed_max, speed_max);
            break;
        case SteeringCommand::DRIVE_BACKWARD:
            control_direction(false, false);
            control_speed(speed_max, speed_max);
            break;
        case SteeringCommand::DRIVE_LEFT:
            control_direction(true, true);
            control_speed(speed_min, speed_max);
            break;
        case SteeringCommand::DRIVE_RIGHT:
            control_direction(true, true);
            control_speed(speed_max, speed_min);
            break;
        case SteeringCommand::HALT:
        default:
            control_direction(true, true);
            control_speed(0.0f, 0.0f);
            break;
    }
}


void
Steering::control(const SteeringControl& control)
{
    control_speed(control.left_speed, control.right_speed);
    control_direction(control.left_forward, control.right_forward);
}


void
Steering::control_speed(float left_speed, float right_speed)
{
    //Clamp right and left spped to interval.
    if(left_speed > 1.0f) left_speed = 1.0f;
    if(right_speed > 1.0f) right_speed = 1.0f;

    // map from [0.0f, 1.0f] to [100, 255]
    uint8_t left_pwm = 0, right_pwm = 0;
    if(left_speed != 0.0f) left_pwm = left_speed * (255 - 100) + 100;
    if(right_speed != 0.0f) right_pwm = right_speed * (255 - 100) + 100;

    transmit_pwm(left_pwm, right_pwm);

    latest_control.left_speed = left_speed;
    latest_control.right_speed = right_speed;

    pc->steering(latest_control);
```

```
}


void
Steering::control_direction(bool left_forward, bool right_forward)
{
    transmit_dir(left_forward, right_forward);

    latest_control.left_forward = left_forward;
    latest_control.right_forward = right_forward;
    pc->steering(latest_control);
}


void
Steering::calibrate(float kp, float kd)
{
    this->kp = kp;
    this->kd = kd;
}


void
Steering::transmit_pwm(uint8_t left_pwm, uint8_t right_pwm) {
    uint8_t bytes[3];
    bytes[0] = (uint8_t)SteeringTx::PWM;
    bytes[1] = left_pwm;
    bytes[2] = right_pwm;

    serial.write(bytes, 3);
}


void
Steering::transmit_dir(bool left_forward, bool right_forward)
{
    uint8_t bytes[3];
    bytes[0] = (uint8_t)SteeringTx::DIR;
    bytes[1] = (uint8_t)left_forward;
    bytes[2] = (uint8_t)right_forward;

    serial.write(bytes, 3);
}


void
Steering::transmit_identified() {
    uint8_t bytes[1];
    bytes[0] = (uint8_t)SteeringTx::IDENTIFIED;
    serial.write(bytes, 1);
}
```

19 pc/client.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
file: client.py
authors: juska933, osklu414

Client module. PC program entry-point and main loop.
"""


import pygame

import const
import entity
import remote_control
from interface import communication

class Client:
    """Client class."""

    def __init__(self):
        """Initialize client."""
        pygame.init()
        self.screen = pygame.display.set_mode((const.SCREEN_WIDTH, const.
            SCREEN_HEIGHT), pygame.HWSURFACE | pygame.DOUBLEBUF)
        self.clock = pygame.time.Clock()
        self.entities = [
            entity.Map(),
            entity.Interface(),
        ]
        self.remote_handler = remote_control.RemoteHandler()
        self.running = True

        @communication.on_receive("message")
        def on_message(text):
            print("received message: ", text)
            pass

        communication.connect()
        while self.running:
            for event in pygame.event.get():
                self.on_event(event)
            self.on_loop()
            self.screen.fill(const.SCREEN_FILL)
            self.on_render()
            self.clock.tick(const.SCREEN_FRAMERATE)

        communication.close()
        pygame.quit()

    def on_event(self, event):
        """Handle events."""
        if event.type == pygame.QUIT:
            # quit when window is closed
```

```
                self.running = False
            elif event.type == pygame.KEYDOWN:
                self.remote_handler.key_down(event.key)
            elif event.type == pygame.KEYUP:
                self.remote_handler.key_up(event.key)

    def on_loop(self):
        """Update client."""
        # make sure any callbacks are called
        communication.on_loop()
        for e in self.entities:
            e.loop()

    def on_render(self):
        """Render client entities."""
        for e in self.entities:
            e.render(self.screen)
        pygame.display.flip()


if __name__ == "__main__":
    import traceback
    try:
        client = Client()
    except:
        traceback.print_exc()
        communication.close()
```

20 pc/remote_control.py

```python
# -*- coding: utf-8 -*-

"""
file: remote_control.py
author: juska933

Remote control module. Translates key presses to steering commands sent to the
communication module.
"""


from pygame.constants import *
import json

from interface import communication


MOTORS_ROTATE_LEFT              = 0
MOTORS_ROTATE_RIGHT             = 1
MOTORS_DRIVE_FORWARD    = 2
MOTORS_DRIVE_BACKWARD   = 3
MOTORS_DRIVE_LEFT               = 4
MOTORS_DRIVE_RIGHT              = 5
MOTORS_HALT                                     = 6

valid_keys = {K_LEFT, K_DOWN, K_UP, K_RIGHT}

# in order of "biggest" states
state_checks = [
    (MOTORS_DRIVE_LEFT, lambda keys_down: keys_down == {K_UP, K_LEFT}),
    (MOTORS_DRIVE_RIGHT, lambda keys_down: keys_down == {K_UP, K_RIGHT}),
    (MOTORS_DRIVE_FORWARD, lambda keys_down: keys_down == {K_UP}),
    (MOTORS_DRIVE_BACKWARD, lambda keys_down: keys_down == {K_DOWN}),
    (MOTORS_ROTATE_LEFT, lambda keys_down: keys_down == {K_LEFT}),
    (MOTORS_ROTATE_RIGHT, lambda keys_down: keys_down == {K_RIGHT}),
    (MOTORS_HALT, lambda keys_down: True),
]


class RemoteHandler:
    def __init__(self):
        self.keys_down = set()
        self.is_remote = True
        self.state = MOTORS_HALT

    def check_state(self):
        for state, check in state_checks:
            if check(self.keys_down):
                return state

    def maybe_update_state(self):
        state = self.check_state()
        if state != self.state:
            self.state = state
            self.send_remote_command()
```

```
def send_remote_command(self):
    data = {
        "id": "command",
        "type": int(self.state)
    }
    data = json.dumps(data)
    if self.is_remote:
        communication.send(data)

def key_down(self, key):
    if key in valid_keys:
        self.keys_down.add(key)
    self.maybe_update_state()

def key_up(self, key):
    if key in self.keys_down:
        self.keys_down.remove(key)
    self.maybe_update_state()
```

21 pc/entity.py

```python
# -*- coding: utf-8 -*-

"""
Entity module.
file: entity.py
authors: juska933, osklu414

Entity module. The various entities drawn the client updates and draws.
"""


import os
import math
import time
import pygame

import const
from interface import communication


def load_sprite(img_name, size=None):
    """Load a sprite from file."""
    file_path = os.path.join(const.RESOURCES_PATH, img_name)
    surface = pygame.image.load(file_path)
    if size:
        surface = pygame.transform.scale(surface, size)
    return surface.convert_alpha()


class Entity:
    """Entity class."""

    def __init__(self, position, rotation=0, parent=None):
        """Initialize entity."""
        self.position = position
        self.rotation = rotation
        self.parent = parent
        if parent:
            parent.children.append(self)
        self.children = []

    def on_render(self, screen, screen_position, screen_rotation):
        """Custom render code here."""
        pass

    def on_loop(self):
        """Custom update code here."""
        pass

    def render(self, screen, parent_position=(0, 0), parent_rotation=0):
        """Render entity and its children, recursively."""
        position = (self.position[0] + parent_position[0], self.position[1] +
            parent_position[1])
        rotation = self.rotation + parent_rotation
```

```
        for child in self.children:
            child.render(screen, parent_position=position, parent_rotation=
                rotation)

        # call entity draw callback
        self.on_render(screen, position, rotation)

    def loop(self):
        """Loop code common for all entities."""
        self.on_loop()
        for child in self.children:
            child.loop()


class Robot(Entity):
    """Robot class. Has a DotCloud, side distance sensors and gyro."""

    SPRITE = None

    def __init__(self, map):
        """Initialize robot."""
        if not Robot.SPRITE:
            Robot.SPRITE = load_sprite("robot.png", size=const.ROBOT_SIZE)
        position = (const.TILE_WIDTH * const.ROBOT_ORIGIN[0], const.TILE_HEIGHT
            * const.ROBOT_ORIGIN[1])
        super().__init__(position, parent=map)
        self.dot_cloud = DotCloud(self)
        self.left_distance = 1000.0
        self.right_distance = 1000.0

        @communication.on_receive("robot")
        def on_robot(x, y, r):
            self.position = ((x + const.ROBOT_ORIGIN[0]) * const.TILE_SIZE[0], (
                y + const.ROBOT_ORIGIN[1]) * const.TILE_SIZE[1])
            self.rotation = r

        @communication.on_receive("sensor")
        def on_sensor(left, right, rot):
            self.left_distance = left
            self.right_distance = right

    def on_render(self, screen, screen_position, screen_rotation):
        """Draw robot."""
        # draw robot sprite TODO: rotate around center
        screen.blit(pygame.transform.rotate(Robot.SPRITE, screen_rotation), (
            screen_position[0] - Robot.SPRITE.get_width()/2, screen_position[1] -
             Robot.SPRITE.get_height()/2))
        # draw side distance sensors
        left_distance_start = (screen_position[0] + const.ROBOT_SIZE[0] // 2,
            screen_position[1] + const.ROBOT_SIZE[1] // 2)
        left_distance_end = (
            left_distance_start[0] + self.left_distance*(const.TILE_SIZE[0]/
                const.TILE_MM)*math.cos(-(screen_rotation+180)*math.pi/180),
            left_distance_start[1] + self.left_distance*(const.TILE_SIZE[0]/
                const.TILE_MM)*math.sin(-(screen_rotation+180)*math.pi/180)
        )
        pygame.draw.line(screen, (0, 0, 255), left_distance_start,
```

```
            left_distance_end, 1)

        right_distance_start = (screen_position[0] + const.ROBOT_SIZE[0] // 2,
            screen_position[1] + const.ROBOT_SIZE[1] // 2)
        right_distance_end = (
            right_distance_start[0] + self.right_distance*(const.TILE_SIZE[0]/
                const.TILE_MM)*math.cos(-screen_rotation*math.pi/180),
            right_distance_start[1] + self.right_distance*(const.TILE_SIZE[0]/
                const.TILE_MM)*math.sin(-screen_rotation*math.pi/180)
        )
        pygame.draw.line(screen, (0, 0, 255), right_distance_start,
            right_distance_end, 1)

    def on_loop(self):
        r = self.rotation
        #self.rotation = r + 1
        #self.position = (self.position[0] + 1, self.position[1])


class Dot(Entity):
    """Dot entity for RPlidar dotcloud."""

    COLOR = (255, 0, 0)

    def __init__(self, robot, angle, dist, quality):
        """Initialize dot entity."""
        origin_x, origin_y = robot.position
        origin_r = robot.rotation
        angle = (-angle - origin_r - 90)*(math.pi/180)
        x, y = origin_x + (dist*const.TILE_WIDTH/const.TILE_MM)*math.cos(angle),
             origin_y + (dist*const.TILE_HEIGHT/const.TILE_MM)*math.sin(angle)
        super().__init__((int(x), int(y)))
        self.quality = quality
        self.created = time.time()

    def on_render(self, screen, screen_position, screen_rotation):
        """Draw dot."""
        pygame.draw.circle(screen, Dot.COLOR, self.position, 1)


class DotCloud(Entity):
    """DotCloud entity for RPlidar dotcloud."""

    def __init__(self, robot):
        """Initialize dotcloud."""
        super().__init__((0, 0), parent=robot)
        self.dots = []
        self.dot_lifetime = 0.1
        self.robot = robot

        @communication.on_receive("rplidar")
        def on_rplidar(nodes):
            for node in nodes:
                dist = node["dist"]
                angle = node["angle"]
                quality = node["quality"]
                self.dots.append(Dot(self.robot, angle, dist, quality))
```

```
    def on_loop(self):
        """Remove old dots."""
        new_dots = []
        now = time.time()
        for dot in self.dots:
            lived = now - dot.created
            if lived <= self.dot_lifetime:
                new_dots.append(dot)
        self.dots = new_dots

    def on_render(self, screen, screen_position, screen_rotation):
        """Render the dots (they are not attached as children)."""
        for dot in self.dots:
            dot.on_render(screen, screen_position, screen_rotation)


class Tile(Entity):
    """Tile class."""

    SPRITES = []

    UNKNOWN = 0
    EMPTY = 1
    WALL = 2

    def __init__(self, map, col, row, tile_type=UNKNOWN):
        """Initialize tile."""
        if not Tile.SPRITES:
            Tile.SPRITES = [
                load_sprite("unknown_tile.png", size=const.TILE_SIZE),
                load_sprite("empty_tile.png", size=const.TILE_SIZE),
                load_sprite("wall_tile.png", size=const.TILE_SIZE)
            ]

        super().__init__((col*const.TILE_WIDTH, row*const.TILE_HEIGHT), parent=
            map)
        self.type = tile_type
        self.col = col
        self.row = row

    def set_type(self, tile_type):
        """Update tile type."""
        self.type = tile_type

    def on_render(self, screen, screen_position, screen_rotation):
        """Draw correct tile sprite."""
        screen.blit(Tile.SPRITES[self.type], screen_position)


class Map(Entity):
    """Map class. Has tiles and robot."""

    def __init__(self):
        """Initialize map."""
        super().__init__(const.MAP_POSITION)
        self.tiles = [[Tile(self, c, r) for c in range(const.MAP_COLS)] for r in
```

```
            range(const.MAP_ROWS)]
        self.robot = Robot(self)
        self.points = []

        @communication.on_receive("tile")
        def on_tile(col, row, type):
            self.set_tile(col, row, type)

        @communication.on_receive("map")
        def on_map(tiles):
            for r in range(const.MAP_ROWS):
                for c in range(const.MAP_COLS):
                    self.set_tile(c, r, tiles[r * const.MAP_COLS + c])

        @communication.on_receive("point")
        def on_point(col, row):
            if not col or not row:
                return
            self.points.append((col, row, time.time()))

    def set_tile(self, col, row, tile_type):
        """Update tile type."""
        self.tiles[row][col].set_type(tile_type)

    def on_render(self, screen, screen_position, screen_rotation):
        """Draw points."""
        new_points = []
        for p in self.points:
            pygame.draw.circle(screen, (0, 255, 0), (int(screen_position[0] + p
                [0]*const.TILE_SIZE[0]), int(screen_position[1] + p[1]*const.
                TILE_SIZE[1])), 1)
            if time.time() - p[2] < 0.1:
                new_points.append(p)
        self.points = new_points

class Interface(Entity):
    """Interface class. Contains buttons and text inputs."""

    def __init__(self):
        """Initialize interface."""
        super().__init__(const.INTERFACE_POSITION)

        x = const.INTERFACE_PADDING
        y = const.INTERFACE_PADDING
        w = const.INTERFACE_WIDTH - 2*const.INTERFACE_PADDING
        h = const.INTERFACE_FONT_SIZE

        self.steering_text = Text((x, y), self, text="steering␣data:")
        y += h
        self.steering_left_speed = Text((x, y), self)
        y += h
        self.steering_right_speed = Text((x, y), self)
        y += h
        self.steering_left_forward = Text((x, y), self)
        y += h
        self.steering_right_forward = Text((x, y), self)
```

```
        y += 2*h

        self.sensor_text = Text((x, y), self, text="sensor␣data:")
        y += h
        self.sensor_left = Text((x, y), self)
        y += h
        self.sensor_right = Text((x, y), self)
        y += h
        self.sensor_rot = Text((x, y), self)
        y += h
        self.robot_pos = Text((x, y), self)

        @communication.on_receive("sensor")
        def on_sensor(left, right, rot):
            self.sensor_left.set_text("left␣distance:␣{:.2f}".format(left))
            self.sensor_right.set_text("right␣distance:␣{:.2f}".format(right))
            self.sensor_rot.set_text("gyro␣rotation:␣{:.2f}␣deg".format(rot))

        @communication.on_receive("steering")
        def on_steering(left_speed, right_speed, left_forward, right_forward):
            self.steering_left_speed.set_text("left_speed:␣{:.2f}".format(
                left_speed))
            self.steering_right_speed.set_text("right_speed:␣{:.2f}".format(
                right_speed))
            self.steering_left_forward.set_text("left␣direction:␣{}".format("
                forward" if left_forward else "backward"))
            self.steering_right_forward.set_text("right␣direction:␣{}".format("
                forward" if right_forward else "backward"))

        @communication.on_receive("robot")
        def on_robot(x, y, r):
            self.robot_pos.set_text("robot␣position:␣({:.2f},␣{:.2f})".format(x,
                 y))


class Text(Entity):
    """Text class. Renders text."""

    FONT = None

    def __init__(self, position, interface, text=""):
        """Initialize text."""
        if not Text.FONT:
            Text.FONT = pygame.font.SysFont(const.INTERFACE_FONT_NAME, const.
                INTERFACE_FONT_SIZE)
        super().__init__(position, parent=interface)
        self.text = text

    def on_render(self, screen, screen_position, screen_rotation):
        """Draw text."""
        text_surface = Text.FONT.render(self.text, False, const.INTERFACE_COLOR)
        screen.blit(text_surface, screen_position)

    def set_text(self, text):
        """Update text."""
        self.text = text
```

22 pc/const.py

```
# -*- coding: utf-8 -*-

"""
Constants module.
file: const.py
authors: juska933, osklu414

Constants module. Various constants used throughout the client program.
"""

import os


# screen constants
SCREEN_FRAMERATE = 30
SCREEN_WIDTH = 1280
SCREEN_HEIGHT = 720
SCREEN_SIZE = (SCREEN_WIDTH, SCREEN_HEIGHT)
SCREEN_FILL = (0, 0, 0)

# rendering constants
MAP_POSITION = (0, 0)
MAP_WIDTH = SCREEN_HEIGHT
MAP_HEIGHT = SCREEN_HEIGHT
MAP_SIZE = (MAP_WIDTH, MAP_HEIGHT)
MAP_ROWS = MAP_COLS = 2 * 10_000 // 400 + 1

TILE_WIDTH = MAP_WIDTH // MAP_COLS
TILE_HEIGHT = MAP_HEIGHT // MAP_ROWS
TILE_SIZE = (TILE_WIDTH, TILE_HEIGHT)
TILE_MM = 400

ROBOT_ORIGIN = (MAP_ROWS // 2, MAP_COLS // 2)
ROBOT_SIZE = TILE_SIZE

INTERFACE_POSITION = (MAP_WIDTH, 0)
INTERFACE_WIDTH = SCREEN_WIDTH - MAP_WIDTH
INTERFACE_HEIGHT = SCREEN_HEIGHT
INTERFACE_SIZE = (INTERFACE_WIDTH, INTERFACE_HEIGHT)

INTERFACE_PADDING = 20
INTERFACE_COLOR = (255, 255, 255)
INTERFACE_FONT_NAME = "Comic␣Sans␣MS"
INTERFACE_FONT_SIZE = 30



# socket setting
LOCALHOST = "127.0.0.1"
OSKARHOST = "192.168.43.125"
FELIXHOST = "172.20.10.6"
HOST = OSKARHOST  # or IP address

PORT = 8000
OFFLINE = False
```

```
# resource settings
RESOURCES_PATH = os.path.join(os.path.dirname(os.path.abspath(__file__)), "
    resources")
```

23 pc/interface.py

```
# -*- coding: utf-8 -*-

"""
file: interface.py
authors: juska933, osklu414

Interface module. Used to communicate with the communication module over its
WiFi access point.
"""


import requests
import json
import socket

import const


# communication constants
MSG_END_HEADER = "__MSG_END__"


class Communication:
    """Communication module interface class."""

    def __init__(self):
        """Init communication module interface object."""
        self.s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.callbacks = {}
        self.last_read_buffer = ""

    def connect(self):
        """Connect socket."""
        try:
            self.s.connect((const.HOST, const.PORT))
        except:
            print("error when connecting to communication module")

    def close(self):
        """Close socket."""
        try:
            self.s.close()
        except:
            print("error when closing connection to communication module")

    def read(self):
        """Read socket message."""
        buf_size = 1024
        self.s.setblocking(False)
        msg = ""
        try:
            while True:
                cur = self.s.recv(buf_size)
                msg += cur.decode("utf-8")
                if len(cur) < buf_size:
```

```
                return msg
    except:
        return ""

def send(self, msg):
    """Send socket message."""
    try:
        msg = msg + MSG_END_HEADER
        self.s.send(msg.encode())
    except:
        print("error when sending message to communication module")

def on_loop(self):
    """Read socket messages and handle packets."""
    data = self.last_read_buffer + self.read()
    if data:
        packets = data.split(MSG_END_HEADER)
        self.last_read_buffer = packets[-1]
        packets.pop()
        for packet in packets:
            self.handle_packet(packet)

def handle_packet(self, data):
    """Handle a single packet."""
    try:
        data = json.loads(data)
        if "id" not in data:
            return
        id = data.pop("id", None)
        if id and id in self.callbacks:
            for callback in self.callbacks[id]:
                callback(**data)
        else:
            print("unhandled packet id " + data["id"])

    except:
        print("Error decoding json req {}".format(data))

# RX

def on_receive(self, id):
    """Register callback for messages from communication module."""
    def wrapper(fn):
        if not id in self.callbacks:
            self.callbacks[id] = []
        self.callbacks[id].append(fn)

        return fn
    return wrapper


# TX

def transmit_command(self, command):
    """Send command."""
    data = {
        "id": "command",
```

```
            "type": command
        }
        data = json.dumps(data)
        self.send(data)

    def transmit_calibration(self, kp, kd):
        """Send calibration."""
        data = {
            "id": "calibration",
            "kp": float(kp),
            "kd": float(kd)
        }
        data = json.dumps(data)
        self.send(data)


# global communication object, can and should be imported to interface with
# the communication module
communication = Communication()
```