Proabot Program -Spring18 final

Boat Program

Classes:

1. Motor
2. Rudder

Library:

1. <SPI.h>
2. <RF24.h>
3. <nRF24L01.h>
4. <PID_v1.h>

Class Motor:

```
/*
 * Motor.h
 *  This class is an implementation of all the methods required
 *  to control a motor using an L298N H-bridge motor driver.
 *  >> Please reference "Boat Circuit: H-bridge motor driver"
 *      for the circuit diagram.
 *
 *  Created: January 5, 2018
 *  Updated: June 6, 2018
 *      Author: Steven Hu
 *  >>Project Orthogonal  -Proabot
 *
 *  Known Issues:
 *   1. Current sensing values are different for different
 *      motor/H-bridge. A universal current_threshold
 *      is used because the current sensing values we got from
 *      the two motors/H-bridges are similar and a single
 *      threshold works just fine.
 *
 */

#ifndef MOTOR_H
#define MOTOR_H

#define DIRECT 0        // These are the encodings of:
#define REVERSE 1       //  motor direction
```

```
#define STOP 5          //  motor status: stop
#define HIGH_CURRENT 7  // motor high current warning

#define CURRENT_THRESHOLD 190    //define the cut off point of current
warning
                              //It is an analog value (0-255). Any value
                              // bigger than 190 will trigger the safety
                              // mechanism to kick in.
class Motor {
public:

    /*
     * constructor:
     *    pwm_pin number, in1_pin #, in2_pin #, current_sensing # is passed
     *   in and initialized.
     *    pwm_p, in1_p, in2_p are set to be OUTPUT, current_p is set to be
     *   INPUT
     */
    Motor(int pwm_p, int in1_p, int in2_p, int current_p);

    /*
     * Note that pwm value is not stored in motor object as knowing the
     * information doesn't benefit any motor operators. However, rudder
     * does store an output value which is the pwm value for motor.
     *
     * Exception: when motor is in HIGH_CURRENT mode, this method will
     * not work. ask_reboot() is the only back door to the safety
     * mechanism.
     */
    void set_pwm(int pwm);

    /*
     * change_to_direct and change_to_reverse changes motor directions.
     * Note that there is no predefined direction of motor. The two
     * methods only guarantee to give opposite directions. Motors need
     * to be calibrated and directions has to be defined before using
     * these two methods.
     */
    void change_to_direct();
    void change_to_reverse();

    void stop_motor();

    /*
```

```
     * sense_current() checks if the current sensing analog value is
     * larger than CURRENT_THRESHOLD. if it is, safety mechanism kicks
     * in and stops the motor and set motor status to HIGH_CURRENT.
     *
     * ask_reboot() is the only way to reset the motor after a
     * HIGH_CURRENT is issued. It monitors serial port until a 'R' is
detected.
     *
     */
    void sense_current();
    void ask_reboot();

    int get_direction();


private:
  int in1_p;
  int in2_p;
  int pwm_p;
  int direction;
  int current_p;

};

#endif /* MOTOR_MOTOR_H */
```

```
/*
 * Motor.cpp
 *
 *  Created: January 5, 2018
 *  Updated: June 6, 2018
 *      Author: Steven Hu
 *  >>Project Orthogonal  -Proabot
 *
 */

#include <Arduino.h>
#include "Motor.h"



Motor::Motor(int pwm_p,int in1_p,int in2_p,int current_p)
    :in1_p(in1_p),in2_p(in2_p),pwm_p(pwm_p),direction(STOP),
```

```
current_p(current_p)
{
  //set up
  pinMode(pwm_p,OUTPUT);
  pinMode(in1_p,OUTPUT);
  pinMode(in2_p,OUTPUT);
  pinMode(current_p,INPUT);
}

void Motor::set_pwm(int pwm){
    if (direction!=HIGH_CURRENT)
        analogWrite(pwm_p,pwm);
}

void Motor::change_to_direct(){
    if(direction!=HIGH_CURRENT){
          digitalWrite(in1_p,HIGH);
          digitalWrite(in2_p,LOW);
          direction = DIRECT;
    }

}

void Motor::change_to_reverse(){
    if (direction!=HIGH_CURRENT){
        digitalWrite(in1_p,LOW);
        digitalWrite(in2_p,HIGH);
        direction = REVERSE;
    }
}

int Motor::get_direction(){
  return direction;
}

void Motor::stop_motor(){
  set_pwm(0);
  digitalWrite(in2_p,LOW);
  digitalWrite(in1_p,LOW);
  direction = STOP;
}

void Motor::sense_current(){
    int current = analogRead(current_p);
```

```
    Serial.print("\t# Current: ");
    Serial.println(current);
    if(direction!= HIGH_CURRENT && current>CURRENT_THRESHOLD){
        stop_motor();
        direction = HIGH_CURRENT;
        Serial.println("WARNING: HIGH_CURRENT[Motor Stopped.]");
    }

}

void Motor::ask_reboot()
{
    Serial.println("[R]eboot? ");
    if(Serial.read()=='R')
        direction = STOP;
}
```

Class Rudder:

```
/*
 * Rudder.h
 *   Rudder class gives an implementation of all the methods required to
control
 *  a motor-controlled rudder. The rudder can be operated by PID
controller.
 *
 *  Created: January 7, 2018
 *  Update: June 6, 2018
 *      Author: Steven Hu
 *  >>Project Orthogonal  -Proabot
 *
 *  Include classes:
 *     "Motor.h"  --Steven Hu
 *     <PID_v1.h> --br3ttb
 *        http://brettbeauregard.com/blog/2011/04/improving-the-beginners-
pid-introduction/
 *
 *  Known Issues:
 *    1. position is given a voltage value which is not idea for display,
should
 *       add a mapping function to return degrees. However, analog values
for
 *       different potentiometers are different and the values vary from
time to
```

```
 *      time so it does raise problems in the testing stage.
 *    2. If NRF24 doesn't receive anything, it will slow the whole system
down.
 *    3. motor and pid should be private. making them public gives access
allows
 *      outside code to fail the whole system. but it's difficult to
calibrate and
 *   experiment things with those two being private. They will stay public
for
 *   testing purposes.
 *    4. Running two rudders under the current setting (SampleTime, double,
parameters)
 *      works fine. It still remains a question if the Mega can handle more
objects
 *      of this class run simultaneously.
 */

#ifndef RUDDER_H
#define RUDDER_H

#include "Motor.h"
#include <PID_v1.h>

class Rudder {
private:
    int position_p;
    double setpoint; //the rudder position commanded by the controller
    double output;   // pwm value for motor
    double position; // current position(angle) of the rudder
    int position_lower_limit;  //position limits of rudder
    int position_upper_limit;
public:
    /*
     * Constructor initializes all the pin numbers. These pins are all
     * used to control to motor. a circuit diagram is shown in the
     * documentation.
     *
     */

    Rudder (int position_pin, int pwm_p, int in1_p, int in2_p, int
current_p);

    void set_limits(int lower, int upper);
    int choose_direction();
```

```
/*
 * update_setpoint(double) is called after setpoint is received by the
 * boat. setpoint should be updated before Compute_and_Drive()
 */
void update_setpoint(double setpoint);
void update_position();

/*
 * Compute_and_Drive() calculates the output and operates the motor
 * using PID controller.
 *
 * If the difference between setpoint and position is smaller than
 * the TOLERATE range (see #define), motor is stopped.
 *
 *  POSITION_UPPER_LIMIT gives the analog value range of the motor
 * potentiometer. Any position (value) that exceeds the range
established
 * by UPPER and LOWER LIMIT will cause the motor to stop.
 *
 * HIGH_CURRENT will be given to dir if during the last current
 * sensing process, a high current is detected. Serial monitor
 * will give an option to reboot the motor. Otherwise, the motor
 * will not move under any circumstances.
 *
 */
void Compute_and_Drive();

/*
 * get_xxx() methods returns the stored value of different private
members.
 */
int get_position();
int get_output();
int get_stat();

/*
 * set_direction(int) is called by Compute_and_Drive() to choose a dir.
 */
void set_direction(int direction);

/*
 * print() prints all information about the rudder object.
 */
```

```
    void print();

    /*
     * motor and pid should be kept private (more of it in the known issues
part).
     *   These two points will be initialized to a motor object and a PID
object
     *   at runtime.
     * destructor makes sure the dynamically allocated motor and pid are
freed.
     */
    ~Rudder();
    Motor * motor;
    PID * pid;


};

#endif /* RUDDER_H */
```

```
/*
 * Rudder.cpp
 *
 *  Created: January 7, 2018
 *  Update:  June 6, 2018
 *      Author: Steven Hu
 *  >>Project Orthogonal  -Proabot
 *
 */

#include "Rudder.h"
#include <Arduino.h>

#define GOOD 15
#define CURRENT_SENSING_DOWN 20
#define POSITION_SENSING_DOWN 21
#define TOLERATE 8

// These values are only for testing purposes. When used in real
// application, they should be overridden right after a rudder is
// constructed.
#define POSITION_LOWER_LIMIT  279
#define POSITION_UPPER_LIMIT  575
```

```
/*
 * These values are convenient for testing purposes but in real application
 * pid->SetTunings(double,double,double) is called right after the
construction
 * to overide them.
 *
 *
 * Notes for r1 tuning: For our purpose, Kp should be in range(10,15)
 * Kp = 8 is accurate for a long distance, but short distance takes a long
time
 * final values for R1: 4/28 Kp = 4, Kd = 3, Ki = 1
 */
double Kp=4, Kd = 3, Ki = 1;


/*
 * setpoint is set to 0 in the code. But 0 is actually not in the range of
 * possible values for the position. This value will be updated to a
 * correct value in the first loop.
 * SampleTime, OutputLimits are two settings that can be customized.
 *
 * > pid->SetOutputLimits(-135,135);
 *     OutputLimits should be well chosen for different rudders(motors)
 *     so a more logical way is to adjust this setting every time a rudder
 *     is constructed.
 *
 * pid has two modes: AUTOMATIC, MANUAL
 *     AUTOMATIC mode is used when we use PID to calculate motor output
 *     MANUAL mode is used when we manually define the output
 *       If we don't switch to MANUAL when we define the output, the PID
 *       formula will give inaccurate output because of the wrong assumption
 *       which is that PID's output is being used. More information on this
 *       issue can be found:
 *       http://brettbeauregard.com/blog/2011/04/improving-the-beginners-
pid-introduction/
 *
 * motor and pid are dynamically allocated here. The memory gets freed in
~Rudder()
 */
Rudder::Rudder(int position_p, int pwm_p, int in1_p, int in2_p, int
current_p)
    :position_p(position_p), setpoint(0), output(0),
position(0),position_lower_limit(200),position_upper_limit(800),
```

```
     motor(new Motor(pwm_p,in1_p,in2_p,current_p)),
     pid(new PID(&position, &output, &setpoint, Kp,Ki,Kd,DIRECT))

{
    update_position();
    pid->SetSampleTime(1000);
    pid->SetMode(AUTOMATIC);

}

void Rudder::set_limits(int lower, int upper)
{
    position_lower_limit = lower;
    position_upper_limit = upper;
}

/*
 * analog value will not be stable. We could smooth it out
 * by assigning an average value of a consecutive 10 msec. But
 * since we are only checking it every thousand msec, that might
 * not be necessary. Note: delay cannot be used anywhere in the
 * program because it will cause the NRF module to stop working.
 * which essentially stops the program.
 */
void Rudder::update_position(){
    position = analogRead(position_p);
}


void Rudder::update_setpoint(double setpoint_received){
    setpoint = setpoint_received;
}


void Rudder::Compute_and_Drive(){
    int dir = motor->get_direction();
    if (dir==HIGH_CURRENT){
        pid->SetMode(MANUAL);
        output = 0;
        motor->ask_reboot();
    }
       //compute_and_drive only operates when NOT(HIGH_CURRENT)
    else{
        update_position();
```

```
        if (abs(setpoint-position) < TOLERATE){
            motor->stop_motor();
            pid->SetMode(MANUAL);
            output = 0;
        }

        else{
            pid->SetMode(AUTOMATIC);
            pid->Compute(); //output is computed
            if (output<0){
                if(position<position_lower_limit){
                    motor->stop_motor();
                    pid->SetMode(MANUAL);
                    output = 0;
                }
                else{
                    motor->change_to_reverse();
                    motor->set_pwm(-output);
                }
            }
            else{ //logical error
                if(position>position_upper_limit){
                    motor->stop_motor();
                    pid->SetMode(MANUAL);
                    output = 0;
                }
                else{
                    motor->change_to_direct();
                    motor->set_pwm(output);
                }
            }
        }
    }
}

void Rudder::print(){
    Serial.print("\tSetpoint: ");
    Serial.println(setpoint);
    Serial.print("\tPosition: ");
    Serial.println(position);
    Serial.print("\tOutput: ");
    Serial.println(output);
    Serial.print("\t# Direction: ");
    Serial.println(motor->get_direction());
```

```
    // current_sensing is printed inside it's method. It should be called right after this so that you can see which current it's showing. Read issues for more detail.
}

int Rudder::get_position(){
    return position;
}

int Rudder::get_output(){
  return output;
}

int Rudder::get_stat(){
    return motor->get_direction();
}

Rudder::~Rudder() {
    delete motor;
    delete pid;
}
```

Boat Program:

```
/*
 * boat_program_spring2018.ino
 *   Created: March 2018
 *   Updated: June 6, 2018
 *   Author: Steven, Asis
 *
 * >> Implemented:
 *   1.nrf  2.PID   3. Two rudders
 *
 *   <Final version for Spring2018>
 *    The script uses PID controller to drive two rudders on the
 *    radio-controlled proabot.
 */

#include "Arduino.h"
#include <SPI.h>
#include <RF24.h>
#include <nRF24L01.h>
#include <PID_v1.h>
#include "Rudder.h"
```

```
#include "Motor.h"

/* [0] := r1_position, [1] := r2_position,
 * [2] := r1_current_sensing_status, [3] := r2_current_sensing_status
 *  current_sensing_condition can either be 0 or HIGH_CURRENT(=7 defined in
 *  motor.h).
 */
double data_to_send[4] = {0,0, 1,1};

//received_data[0] = r1_setpoint, [1] = r2_setpoint
double received_data[2];

RF24 radio(9,53);  //CE, CSN pins
byte addresses[][6] = {"00001","00002"};
  //names of the two communications(2 directions)

/* construct two rudder objects:
 *   Rudder(position_p, pwm_p, in1_p, in2_p, current_p)
 */
Rudder r1(15,5,28,29,3);
Rudder r2(8,4,26,27,2);

void radioSetup()
{
    //initiate the radio object
    radio.begin();

    /*
     * Set the transmit power to lowest available to prevent power supply
related
     * If using a higer level, a bypass capacitor across GND and 3.3V
should be
     * used to smooth out the voltage.
     */
    radio.setPALevel(RF24_PA_MIN);

    //Set the speed of the transmission to the quickest available
    radio.setDataRate(RF24_2MBPS);

    //Use a channel unlikely to be used by Wifi, Microwave ovens etc
    radio.setChannel(124);

    //Open a writing and reading pipe on each radio, with opposite
addresses
```

```
    radio.openWritingPipe(addresses[0]);
    radio.openReadingPipe(1,addresses[1]);
}

/*
 * OutputLimits and Tunings were chosen for our application only.
 * set_limits(int,int) sets the position limits for the two rudders.
 */
void pidSetup()
{
    r1.pid->SetOutputLimits(-135,135);
    r2.pid->SetOutputLimits(-185,185);
    r1.pid->SetTunings(4,1,2.5);
    r2.pid->SetTunings(4,1,3);
    r1.set_limits(422,610);
    r2.set_limits(279,575);
}

void setup()
{
    Serial.begin(9600);

    radioSetup();
    pidSetup();
}

/*
 * update_data() 1. passes the received setpoints to the two rudder
objects.
 * 2. get the positions of the rudders and update the data_to_send array.
 * 3. check current sensing result. If HIGH_CURRENT, update data_to_send
 */
void update_data(){
    r1.update_setpoint(received_data[0]);
    r2.update_setpoint(received_data[1]);
    data_to_send[0] = r1.get_position();
    data_to_send[1] = r2.get_position();
    if (r1.get_stat()==HIGH_CURRENT)
        data_to_send[2] = HIGH_CURRENT;
    if (r2.get_stat() == HIGH_CURRENT)
        data_to_send[3] = HIGH_CURRENT;

}
void operate_rudder(Rudder & r){
```

```
    r.update_position();
    r.Compute_and_Drive();
    r.print();
    r.motor->sense_current();
}

void loop()
{
    //start listening to the radio. wait until it receives a signal or
timeout
    radio.startListening();
    unsigned long start_wait_time = millis();
    while(!radio.available()){
        if(millis() - start_wait_time > 5){
            Serial.println("Nothing received!");
            return;
        }
    }

    radio.read(&received_data, sizeof(received_data));
    Serial.print("Received: ");
    Serial.print(received_data[0]);
    Serial.print("\t");
    Serial.println(received_data[1]);

    update_data(); //update setpoint and data_to_send

    radio.stopListening();

    operate_rudder(r1);
    operate_rudder(r2);

    //send the rudder positions and current_sensing results
    radio.write(&data_to_send, sizeof(data_to_send));
    Serial.print("Sent: ");
    Serial.println(data_to_send[0]);
}
```

Controller program:

```
/*
 * controller_program_spring2018.ino
 *   Created: March 2018
 *   Updated: June 6, 2018
```

```
 *   Author: Steven, Asis
 *
 * >> Implemented:
 *   1.nrf  2.LCD screen
 *
 *   <Final version for Spring2018>
 *     The script implements the controller part of the radio-controlled
 *    proabot.
 */


#include "Arduino.h"
#include <SPI.h>
#include <RF24.h>
#include "LiquidCrystal_I2C.h"

#define HIGH_CURRENT 7    // The same encoding is used in "motor.h"


// Hardware configuration: Set up nRF24L01 radio on SPI bus (pins 10, 11,
12, 13) plus pins 7 & 8
RF24 radio(9, 53);  //CE, CSN pins
byte addresses[][6] = {"00001", "00002"};  //names of the two
communications(2 directions)

//[0] setpoint_1; [1] setpoint_2
double data_to_send[2];
//[0] rudder_position_1; [1] rudder_position_2; [2] rudder_current_1; [3]
rudder_current_2
double data_receive[4];

LiquidCrystal_I2C lcd(0x3F,16,2);

void radioSetup_controller()
{
    // Initiate the radio object
    radio.begin();
    // Set the transmit power to lowest available to prevent power supply
related issues
    radio.setPALevel(RF24_PA_MIN);
    // Set the speed of the transmission to the quickest available
    radio.setDataRate(RF24_2MBPS);
    // Use a channel unlikely to be used by Wifi, Microwave ovens etc
    radio.setChannel(124);
```

```
    // Open a writing and reading pipe on each radio, with opposite
addresses
    radio.openWritingPipe(addresses[1]);
    radio.openReadingPipe(1, addresses[0]);
}


void setup() {
  Serial.begin(9600);

  radioSetup_controller();
  /* legacy code: These two pins were used to give potentiometer 5V
  pinMode(31,OUTPUT);
  pinMode(33,OUTPUT);
  digitalWrite(31,HIGH);
  digitalWrite(33,LOW);
  */


  lcd.begin();
  lcd.backlight();
  pinMode(A1, INPUT);// A1 will read the pot. to control rudder position.
  pinMode(A2, INPUT);
}

void show_info(){
    lcd.setCursor(0, 0);
    lcd.print(data_receive[1]);
    lcd.setCursor(7,0);
    lcd.print(data_receive[0]);
    if (data_receive[3]==HIGH_CURRENT){
        lcd.setCursor(6,0);
        lcd.print("H");
    }
    if(data_receive[2]==HIGH_CURRENT){
        lcd.setCursor(15,0);
        lcd.print("H");
    }

    lcd.setCursor(0,1);
    lcd.print(data_to_send[1]);
    lcd.setCursor(7,1);
    lcd.print(data_to_send[0]);
```

```
}

void update_setpoint(){
    data_to_send[0] = analogRead(A1); //r1_setpoint
    data_to_send[1] = analogRead(A2); //r2_setpoint
}

void radio_write(){
    // Ensure we have stopped listening (even if we're not) or we won't be
able to transmit
    radio.stopListening();

    // Did we manage to SUCCESSFULLY transmit that (by getting an
acknowledgement back from the other Arduino)?
    // Even we didn't we'll continue with the sketch, you never know, the
radio fairies may help us
    if (!radio.write( &data_to_send, sizeof(data_to_send) )) {
        //Serial.println("No acknowledgement of transmission - receiving
radio device connected?");
    }
    radio.write(&data_to_send,sizeof(data_to_send));
}

void radio_read(){
    // Now listen for a response
    radio.startListening();

    // But we won't listen for long, 200 milliseconds is enough
    unsigned long started_waiting_at = millis();

    // Loop here until we get indication that some data is ready for us to
read (or we time out)
    while ( ! radio.available() ) {

    // Oh dear, no response received within our timescale
        if (millis() - started_waiting_at > 100 ) {
            Serial.println("No response received - timeout!");
            return;
        }
    }

    // Now read the data that is waiting for us in the nRF24L01's buffer

    radio.read( &data_receive, sizeof(data_receive) );
```

```
}

void monitor_print(){
    // Show user what we sent and what we got back
    Serial.print("Sent setpoint: ");
    Serial.print(data_to_send[0]);
    Serial.print("\t");
    Serial.println(data_to_send[1]);

    Serial.print("Received:\n");
    Serial.print("\tr1 position: ");
    Serial.println(data_receive[0]);
    Serial.print("\tr2 position: ");
    Serial.println(data_receive[1]);
    if (data_receive[2]==7)
        Serial.println("\t r1 stopped due to HIGH_CURRENT_ERROR");
    if (data_receive[3]==7)
        Serial.println("\t r2 stopped due to HIGH_CURRENT_ERROR");
}

void loop() {

    show_info();
    update_setpoint();

    radio_write();
    radio_read();

    monitor_print();
}
```