src/global.h

```c
#ifndef GLOBAL_H
#define GLOBAL_H

/*
////        Defining constants
*/

#define F_CPU 16000000UL
#define __AVR_ATmega328P__


/*
////        Defining bit functions
*/

#define bitIsSet(macro, bit) ((macro & _BV(bit)))
#define bitIsClear(macro, bit) (!(macro & _BV(bit)))
#define loopUntilBitIsSet(macro, bit) do { } while (bitIsSet(macro, bit))
#define loopUntilBitIsClear(macro, bit) do { } while (bitIsClear(macro, bit))



/*
////        Including liberaries
*/

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

#endif
```

```cpp
/*
////        including liberaries
*/

#include "global.h"
#include "motor.h"
#include "servo.h"
#include "serial.h"


/*
////        Defining constants
*/

#define NUM_OF_COMMANDS 11
#define LEFT_SENSOR PC1
#define LEFT_START 120
#define RIGHT_SENSOR PC3
#define REACH_DIFF 30
#define LB 0
#define RB 2
#define WB 4


/*
////        Initializing variables
*/

Motor weapon(1);
Motor left(2);
Motor right(3);
Motor motors[3] = {left, right, weapon};
Servo servo(9);
bool manuallyOn = false;
bool reachIncrements[3] = {true, true, false};
uint8_t reaches[3] = {0, 0, 60};


/*
////        List of rx commands
*/

```

```cpp
uint8_t rxCommands[NUM_OF_COMMANDS] = {
//  (left     << LB) | (right    << RB) | (weapon   << WB),          // Title         Index     Alphabet Character
    (FORWARD  << LB) | (FORWARD  << RB) | (IGNORE   << WB),          // Forward        0         @
    (FORWARD  << LB) | (RELEASE  << RB) | (IGNORE   << WB),          // Right:         1         A
    (RELEASE  << LB) | (FORWARD  << RB) | (IGNORE   << WB),          // Left:          2         B
    (BACKWARD << LB) | (BACKWARD << RB) | (IGNORE   << WB),          // Back:          3         C
    (RELEASE  << LB) | (RELEASE  << RB) | (IGNORE   << WB),          // Stop wheels:   4         D
    (IGNORE   << LB) | (IGNORE   << RB) | (FORWARD  << WB),          // Weapon on:     5         E
    (IGNORE   << LB) | (IGNORE   << RB) | (BACKWARD << WB),          // Weapon back:   6         F
    (IGNORE   << LB) | (IGNORE   << RB) | (RELEASE  << WB),          // Weapon off:    7         G
    (RELEASE  << LB) | (RELEASE  << RB) | (RELEASE  << WB),          // STOP ALL:      8         H
    (FORWARD  << LB) | (BACKWARD << RB) | (IGNORE   << WB),          // Super right    9         I
    (BACKWARD << LB) | (FORWARD  << RB) | (IGNORE   << WB),          // Super left     10        J
};

/*
////        Prototypes
*/
void putOff(int direction, uint8_t* reach, bool* reachIncrement);


/*
////        Main
*/

int main() {

  /*
    Start serial monitor
  */
  Serial::begin();

  /*
    Interrupts
  */

  // Enable global interrupts
  sei();

  // Enabling rx interrupt
  UCSR0B |= _BV(RXCIE0);
```

```


    /*
      Main loop
    */


    while (true)
    {
      []()
      {

        // For every sensor of the sensors
        for (char sensor = LEFT_SENSOR, start = LEFT_START; sensor <= RIGHT_SENSOR; sensor++, start -= 60)
        {

          // return if no fire detected from sensor digital input
          PORTC |= _BV(sensor); // set a pullover on pin 0 to read it by pinb
          // insert a nop
          _NOP();

          // if fire detected from that senosr
          if (bitIsClear(PINC, sensor))
          {

            uint8_t index = sensor - LEFT_SENSOR;
            if (!servo.manual)
              // put off that fire then return
              putOff(start, &reaches[index], &reachIncrements[index]);
            return;
          }
        }

        // If no fire is detected and also the weapon not turned on manually
        if (!manuallyOn)
          // make sure the weapon is not running
          weapon.run(RELEASE);

        if (!servo.manual)
          // make the weapon look forward
          servo.write(CENTER);

          // Return reach to zero
        for (uint8_t i = 0; i < 2; i++) {
          reaches[i] = 0;
          reachIncrements[i] = true;
        }
        reaches[2] = 60;
        reachIncrements[2] = false;

      }();
      }


  }

  ISR(USART_RX_vect) {
    // Get the operation code {index} by getting only the first five bits
    uint8_t read = Serial::read();

    switch (read)
    {
      // Toggle enabling servo
      // 0
      case '0':
        servo.enable = !servo.enable;
        return;

      // Rotate servo to the right
      // 1
      case '1':
        servo.increment(-1);
        return;

      // Rotate servo to the left
      // 2
      case '2':
        servo.increment(1);
        return;

      // Stop rotation of servo
      // 3
      case '3':
        servo.done = true;
        return;

      // Turn manual mode off
      // 4
      case '4':
        servo.manual = false;
        return;
    }

    uint8_t index = read & 0x0f;
    if (index > NUM_OF_COMMANDS - 1) {
      return;
    }
    uint8_t command;

    // For every motor on the shield, execute the appropriate command
    for (uint8_t i = 0, shift = LB; i < 3; i++, shift+= 2) {
      command = (rxCommands[index] >> shift) & 0b11;
      motors[i].run(command);
    }

    // For the weapon, make sure to update values of manuallyOn
    switch (command)
    {
    case IGNORE:
      return;
    case RELEASE:
      manuallyOn = false;
      break;
    default:
      manuallyOn = true;
      break;
    }
  }

  void putOff(int direction, uint8_t* reach, bool* reachIncrement) {

    // Turn on the fan
    weapon.run(FORWARD);

    switch (*reachIncrement) {
      case true:
        if (*reach == 60) {
          *reachIncrement = false;
          return;
        }
        *reach += REACH_DIFF;
        break;
      default:
        if (*reach == 0) {
          *reachIncrement = true;
          return;
        }
        *reach -= REACH_DIFF;
        break;
    }

    // Turn the servo to direction + reach
    servo.write(direction + *reach);
  }



  bool servoOn = false;

  ISR(TIMER1_COMPA_vect) {

    switch (servoOn)
    {
    case false:
      TCNT1 = 0;
      OCR1A = TCNT1 + servo.ticks;
      if (servo.enable){
        PORTB |= _BV(servo.pin);
        servoOn = true;
      }
      break;
    case true:
      PORTB &= ~_BV(servo.pin);
      if (TCNT1 + 4 < usToTicks(REFRESH_INTERVAL))
        OCR1A = usToTicks(REFRESH_INTERVAL);
      else
        OCR1A = TCNT1 + 4;
      servoOn = false;
      break;
    }

  }
```

```cpp
#include "global.h"
#include "motor.h"
#include "serial.h"

static uint8_t latchState{0};
static uint8_t MOTORS_A[4] = {2, 1, 5, 0};
static uint8_t MOTORS_B[4] = {3, 4, 7, 6};

void Motor::latch_tx() {

    LATCH_AND_DATA_PORT &= ~MOTORLATCH & ~MOTORDATA;

    for (uint8_t i = 0; i < 8; i++) {
        ENABLE_AND_CLK_PORT &= ~MOTORCLK;

        if (latchState & _BV(7-i))
            LATCH_AND_DATA_PORT |= MOTORDATA;
        else
            LATCH_AND_DATA_PORT &= ~MOTORDATA;

        ENABLE_AND_CLK_PORT |= MOTORCLK;
    }

    LATCH_AND_DATA_PORT |= MOTORLATCH;
}

Motor::Motor(uint8_t motorNum) {

    // set the motor num
    this->motorNum = motorNum;

    /*
    ////    Enable
    */
    [&]()
    {
        LATCH_AND_DATA_DRR |= MOTORLATCH | MOTORDATA;
        ENABLE_AND_CLK_DRR |= MOTORENABLE | MOTORCLK;

        latch_tx(); // Reset latch

        ENABLE_AND_CLK_PORT &= ~MOTORENABLE;

    }();

    uint8_t i = motorNum - 1;
    latchState &= ~_BV(MOTORS_A[i]) & ~_BV(MOTORS_B[i]);
    latch_tx();

    switch (motorNum)
    {
    case 1:
         // use PWM from timer2A on PB3 (Arduino pin #11)
        TCCR2A |= _BV(COM2A1) | _BV(WGM20) | _BV(WGM21); // fast PWM, turn on oc2a
        TCCR2B = 0x7;
        OCR2A = SPEED;
        DDRB |= _BV(DDB3);
        break;
    case 2:
        // use PWM from timer2A on PB3 (Arduino pin #11)
        TCCR2A |= _BV(COM2B1) | _BV(WGM20) | _BV(WGM21); // fast PWM, turn on oc2b
        TCCR2B = 0x7;
        OCR2B = SPEED;
        DDRD |= _BV(DDD3);
        break;
    case 3:
        // use PWM from timer0A / PD6 (pin 6)
        TCCR0A |= _BV(COM0A1) | _BV(WGM00) | _BV(WGM01); // fast PWM, turn on OC0A
        TCCR0B = 0x7;
        OCR0A = SPEED;
        DDRD |= _BV(DDD6);
        break;
    case 4:
        // use PWM from timer0B / PD5 (pin 5)
        TCCR0A |= _BV(COM0B1) | _BV(WGM00) | _BV(WGM01); // fast PWM, turn on oc0a
        TCCR0B = 0x7;
        OCR0B = SPEED;
        DDRD |= _BV(DDD5);
        break;
    }


}

```

```cpp
void Motor::run(uint8_t direction) {

    if (direction < FORWARD) {
        return;
    }

    uint8_t i = this->motorNum - 1;
    switch (direction)
    {
    case FORWARD:
        latchState |= _BV(MOTORS_A[i]);
        latchState &= ~_BV(MOTORS_B[i]);
        break;
    case BACKWARD:
        latchState &= ~_BV(MOTORS_A[i]);
        latchState |= _BV(MOTORS_B[i]);
        break;
    case RELEASE:
        latchState &= ~_BV(MOTORS_A[i]);
        latchState &= ~_BV(MOTORS_B[i]);
        break;
    }
    latch_tx();
}
```

```
#ifndef MOTOR_H
#define MOTOR_H

#include <stdint.h>

#define FREQ _BV(CS01)
#define MOTORLATCH _BV(DDB4)
#define MOTORDATA _BV(DDB0)
#define MOTORENABLE _BV(DDD7)
#define MOTORCLK _BV(DDD4)
#define SPEED 0xff
#define IGNORE 0
#define FORWARD 1
#define BACKWARD 2
#define RELEASE 3

#define LATCH_AND_DATA_DRR DDRB
#define LATCH_AND_DATA_PORT PORTB
#define ENABLE_AND_CLK_DRR DDRD
#define ENABLE_AND_CLK_PORT PORTD

class Motor
{
private:
    uint8_t motorNum;
    void latch_tx();
public:
    Motor(uint8_t motorNum);
    void run(uint8_t direction);
};

#endif
```

src/serial.cpp

```
#include "global.h"
#include "serial.h"
// #include <string.h>

#define BAUD 9600
#define MYUBRR F_CPU/(long(16) * BAUD) -1

void Serial::begin() {
    // Set baud rate
    UBRR0H =(MYUBRR >> 8);
    UBRR0L = MYUBRR;
    // Enable receiver and transmitter
    UCSR0B = _BV(RXEN0) | _BV(TXEN0);
    // Set frame format: 8data, 1stop bit
    UCSR0C = _BV(UCSZ01) | _BV(UCSZ00);
}

uint8_t Serial::read() {
    return UDR0;
}

// void Serial::print(char* text) {

//     for (uint8_t i = 0, length = strlen(text); i < length; i++) {
//         loopUntilBitIsClear(UCSR0A, UDRE0);
//         UDR0 = text[i];
//     }

// }

// void Serial::print(char c) {
//     char character[2] = {c, 0};
//     print(character);
// }

// void Serial::print(uint8_t n) {
//     char chars[3] = {48, 48, 48};

//     for (char i = 0; i < 3; i++){
//         if (n <= 0) {
//             break;
//         }
//         chars[i] = (n % 10) + 48;
//         n = n / 10;
//     }

//     for (char i = 2; i >= 0; i--) {
//         print(chars[i]);
//     }

// }

// void Serial::println(char* text) {
//     Serial::print(text);
//     Serial::print('\n');
// }

// bool Serial::available() {
//     return bitIsSet(UCSR0A, RXC0);
// }
```

src/serial.h

```cpp
#ifndef SERIAL_H
#define SERIAL_H

#include <stdint.h>

class Serial
{
public:
    Serial() {};
    static void begin();
    // static void print(char* text);
    // static void print(uint8_t n);
    // static void print(char c);
    // static void println(char *text);
    static uint8_t read();
    // static bool available();
};

#endif
```

```cpp
#include "global.h"
#include "servo.h"
#include "serial.h"

Servo::Servo(uint8_t pinNum) {

    // Set the pin to output
    this->pin = pinNum - 8;
    DDRB |= _BV(pin);

    // Initializing the timer
    TCCR1A = 0;                 // normal counting mode
    TCCR1B = _BV(CS11);         // Set prescaler of 8
    TCNT1 = 0;                  // Clear the timer count

    TIFR1 |= _BV(OCF1A);        // Clear any pending interrupts
    TIMSK1 |= _BV(OCIE1A);      // enable the output compare interrupt

}

void Servo::write(uint8_t angle) {

    if (angle == this->servoAngle) {
        return;
    }

    char increment = this->servoAngle > angle ? -1 : 1;

    // For in in the range start to finish
    for (uint8_t i = this->servoAngle + increment, r = angle + increment; i != r;i += increment) {

        // Write the angle of the servo as the current value
        _write(i);

        // Delay 5 seconds
        _delay_ms(SERVO_DELAY);
    }

}

bool Servo::_write(uint8_t angle) {

    if (!this->enable) {
        return false;
    }

    bool returnValue = true;
    if (angle == START) {
        returnValue = false;
    }
    else if (angle > END) {
        angle = END;
        returnValue = false;
    }
    // Convert the angle to microseconds
    this->servoAngle = angle;
    long value = angleToPulse((unsigned long)angle);

    // convert the microseconds to ticks
    value = value - TRIM_DURATION;
    value = usToTicks(value);           // convert to ticks after compensating for interrupt overhead

    // stop interrupts  while overwriting the value of ticks
    // uint8_t oldSREG = SREG;
    // cli();
    this->ticks = value;
    // SREG = oldSREG;

    return returnValue;
}

void Servo::increment(uint8_t increment) {
    sei();
    this->manual = true;
    this->done = false;
    while (_write(this->servoAngle + increment) && !this->done && this->manual)
    {
        _delay_ms(SERVO_DELAY);
    }
    this->done = true;
}
```

```cpp
#ifndef SERVO_H
#define SERVO_H

#include <stdint.h>

#define MIN_PULSE_WIDTH       544     // the shortest pulse sent to a servo
#define MAX_PULSE_WIDTH      2400     // the longest pulse sent to a servo
#define TRIM_DURATION 0               // compensation ticks to trim adjust for digitalWrite delays
#define SERVO_DELAY 10
#define DEFAULT_PULSE_WIDTH 1500
#define REFRESH_INTERVAL    20000     // minimum time to refresh servos in microseconds
#define clockCyclesPerMicrosecond (F_CPU / 1000000L)
#define usToTicks(_us) ((_us * clockCyclesPerMicrosecond ) / 8)             // Converts microseconds to ticks
#define ticksToUs(_ticks) ((_ticks * 8) / clockCyclesPerMicrosecond)        // From ticks to microseconds
#define DEFAULT_TICKS usToTicks(DEFAULT_PULSE_WIDTH)
#define angleToPulse(x) x * (MAX_PULSE_WIDTH - MIN_PULSE_WIDTH) / 180  + MIN_PULSE_WIDTH


#define CENTER 90
#define START 0
#define END 180

class Servo
{
public:
    bool manual{false};
    bool done{true};
    bool enable {true};
    uint8_t servoAngle {CENTER};
    uint8_t pin;
    long ticks {angleToPulse(CENTER)};
    Servo(uint8_t pin);
    void write(uint8_t angle);
    bool _write(uint8_t angle);
    void increment(uint8_t increment);
};

#endif
```