

ZUMO LINE FOLLOWING ROBOT



Group Members:

FAHEEM-UL-QUDDUS	(180857)
MUHAMMAD GHOUS IQBAL	(180869)
OSAMA ABID MUGHAL	(180872)

BE MECHATRONICS (Session 2018-2022)

Project Supervisor: ENGR. OMER FAROOQ

Supervisor Name: ENGR. OMER FAROOQ

Designation: COURSE INSTRUCTOR

DEPARTMENT OF MECHATRONICS ENGINEERING

FACULTY OF ENGINEERING

AIR UNIVERSITY, ISLAMABAD

Table of Contents

GROUP MEMBERS	2
CHAPTER: 01	3
PRELIMINARIES	3
1.1 Proposal:	3
1.2 Initial Feasibility:	4
1.3 Team Roles and Details:	4
1.4 Work Breakdown:	4
CHAPTER: 02	5
PROJECT CONCEPTION	5
2.1 Introduction:	5
2.2 Literature Background:	5
2.2.1 About Line Following and Maze Solving:	5
2.3 Features and Operational Specifications:	5
2.4 Basic Block Diagram:	6
CHAPTER: 03	7
SOFTWARE AND FIRMWARE DESIGN	7
3.1 Main Code:	7
3.2 Zumo Motors Code:	21
3.3 Zumo Buzzer Code:	23
3.4 Push Button Code:	35
3.5 Reflector Sensor Array Code:	39
3.6 QTR Sensors Code:	40
3.7 SD Library Code:	49
CHAPTER: 04	58
SIMULATION AND IMPLEMENTATION	58
4.1 Schematic:	58
4.2 Simulation:	59
4.3 Simulation Result:	60
CHAPTER: 05	61
FINDINGS AND CONCLUSION	61
5.1 Findings and Results:	61
5.2 Conclusion and Deductions:	62

GROUP MEMBERS

Team Member Name	Role	Word count	Signature
M. GHOU S IQBAL	Team Lead	398	<i>Cloud Salazar</i>
OSAMA ABID MUGHAL	Project Manager	415	<i>Osama Abid</i>
FAHEEM-UL-QUDDUS	Technical	398	<i>Tiger Zatti</i>

CHAPTER: 01

PRELIMINARIES

1.1 Proposal:

It is a project based off a very core object in robotics – Automation and Mobility of Robots. We focus on moving a robot from one position to another with the help of various sensors and a microcontroller.

Goals:

- To develop a system that utilizes a community-wide available microcontroller.
- To develop a self-navigating mobile robot that can get from point A to point B.
- To develop a solid understanding of programming Arduino IDE using C++ and Objective-C.
- To be able to interface various sensors and microcontroller with each other with ease.

Components:

- Arduino UNO 328P
- Zumo Reflector Sensor Array
- SD Card
- SD Card Reader
- Zumo Line Follower Robot Kit

Working:

The Robot utilizes various sensors planted at various parts of the robot that sense and send data back to the microcontroller. Once the microcontroller has all the data about obstacles and its own surroundings, it can decide its next step. The robot carries on this task over and over again until it reaches its target.

The working of this system is simple and straight-forward. It all comes down to the ability of microcontroller to detect changes in its environment and act accordingly.

Inputs and Outputs:

- Input is taken from Reflector Sensor Array to read the track.
- Output is the motor speed of the Zumo Line Follower Kit.

1.2 Initial Feasibility:

Taking a brief look at the project, its usage, and its ability to serve a function in robotics evolution, one can rest assured to say that the project is feasible and can stand its chance when it comes to labelling a worth on its forehead.

1.3 Team Roles and Details:

The team comprised of three individuals. One of them took the role of management, the other took the role of technical specialists, while the last one took the lead role to keep everything in order. The roles in the team were as follows:

- **Osama Abid Mughal** – Management and Logistics
- **Muhammad Ghous Iqbal** – Project Overview and Programming
- **Faheem Ul Quddus** – Technical troubleshoots.
- **Faheem Ul Quddus** – Technical troubleshoots and Assemblies.

1.4 Work Breakdown:

- **Osama Abid Mughal:** He was responsible for making sure everything was in order and check and had to make sure that the project goes smoothly.
- **Muhammad Ghous Iqbal:** He was responsible for working of the project. He had to make sure that the components worked properly and were in the right order by programming.
- **Faheem Ul Quddus:** He was responsible for the management of technical details like part compatibility and assembly.

CHAPTER: 02

PROJECT CONCEPTION

2.1 Introduction:

It is a project based off a very core object in robotics – Automation and Mobility of Robots. We focus on moving a robot from one position to another with the help of various sensors and a microcontroller.

The instructions were given from the project manager. The project was done accordingly. Then we faced several challenges which we tackled till end and made our project finally work.

2.2 Literature Background:

We took help from several forums and from official documentation of *Pololu*.

2.2.1 About Line Following and Maze Solving:

Line following is a great introduction to robot programming, and it makes a great contest: it's easy to build a line-following course, the rules are simple to understand, and it's not hard to program a robot to follow a line. Optimizing your program to make your robot zoom down the line at the highest speed possible, however, is a challenge that can introduce you to some advanced programming concepts.

When you're ready for a more difficult task, build a line maze for your robot to solve. You can start by solving the maze with a simple "left hand on the wall" strategy and move up to more and more advanced techniques to make your robot faster.

2.3 Features and Operational Specifications:

Following are the specifications or personal specifications of the project:

- The robot can make turns in every direction and make its decision accordingly.
- The robot can make a complete turn when it reaches endpoint.
- Data logging is done using SD Card.

2.4 Basic Block Diagram:

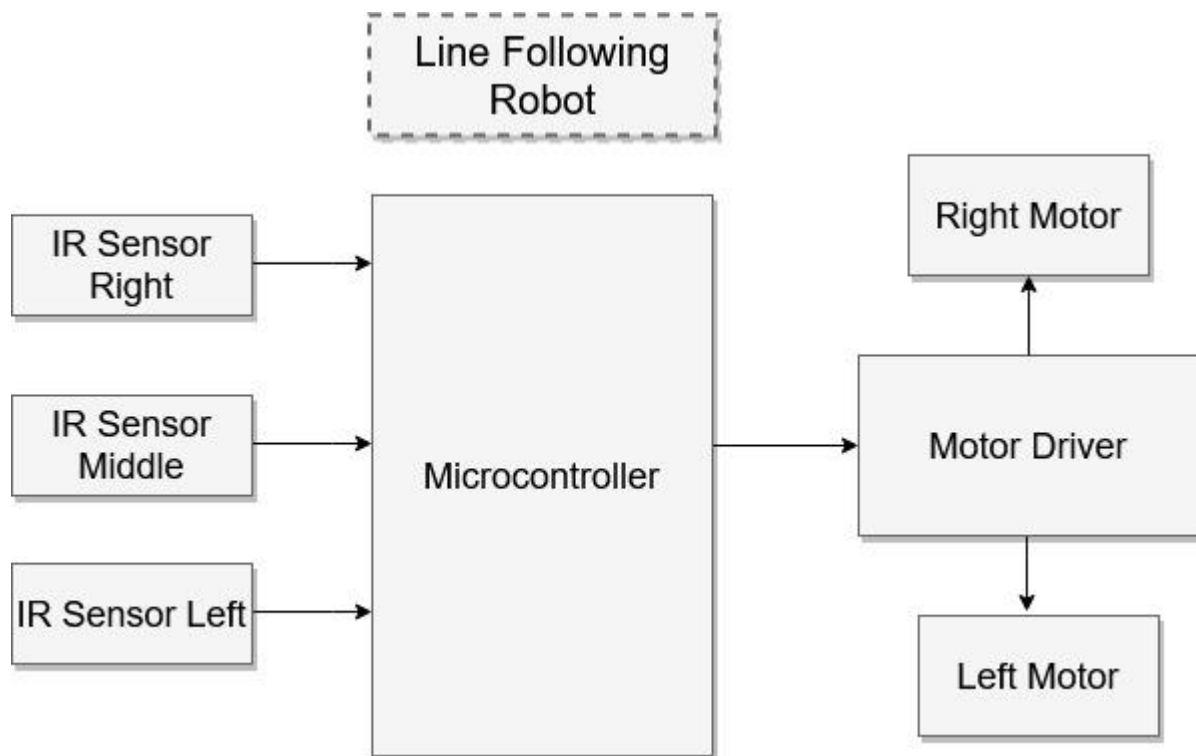


Fig 2.1: Block Diagram

CHAPTER: 03

SOFTWARE AND FIRMWARE DESIGN

3.1 Main Code:

We devised the following code with the logic that we have already developed and discussed over the past few chapters.

```
#include <QTRSensors.h>
#include <ZumoReflectanceSensorArray.h>
#include <ZumoMotors.h>
#include <ZumoBuzzer.h>
#include <Pushbutton.h>
#include <SPI.h>
#include <SD.h>

/* This example uses the Zumo Reflectance Sensor Array
 * to navigate a black line maze with no loops. This program
 * is based off the 3pi maze solving example which can be
 * found here:
 *
 * http://www.pololu.com/docs/0J21/8.a
 *
 * The Zumo first calibrates the sensors to account
 * for differences of the black line on white background.
 * Calibration is accomplished in setup().
 *
 * In loop(), the function solveMaze() is called and navigates
 * the Zumo until it finds the finish line which is defined as
 * a large black area that is thick and wide enough to
 * cover all six sensors at the same time.
 *
 * Once the Zumo reaches the finishing line, it will stop and
 * wait for the user to place the Zumo back at the starting
 * line. The Zumo can then follow the shortest path to the finish
 * line.
 *
 * The macros SPEED, TURN_SPEED, ABOVE_LINE(), and LINE_THICKNESS
 * might need to be adjusted on a case by case basis to give better
 * line following results.
 */

// SENSOR_THRESHOLD is a value to compare reflectance sensor
// readings to to decide if the sensor is over a black line
#define SENSOR_THRESHOLD 300
```



```

// ABOVE_LINE is a helper macro that takes returns
// 1 if the sensor is over the line and 0 if otherwise
#define ABOVE_LINE(sensor)((sensor) > SENSOR_THRESHOLD)

// Motor speed when turning. TURN_SPEED should always
// have a positive value, otherwise the Zumo will turn
// in the wrong direction.
#define TURN_SPEED 200

// Motor speed when driving straight. SPEED should always
// have a positive value, otherwise the Zumo will travel in the
// wrong direction.
#define SPEED 200

// Thickness of your line in inches
#define LINE_THICKNESS .75

// When the motor speed of the zumo is set by
// motors.setSpeeds(200,200), 200 is in ZUNITS/Second.
// A ZUNIT is a fictitious measurement of distance
// and only helps to approximate how far the Zumo has
// traveled. Experimentally it was observed that for
// every inch, there were approximately 17142 ZUNITS.
// This value will differ depending on setup/battery
// life and may be adjusted accordingly. This value
// was found using a 75:1 HP Motors with batteries
// partially discharged.
#define INCHES_TO_ZUNITS 17142.0*4.0

// When the Zumo reaches the end of a segment it needs
// to find out three things: if it has reached the finish line,
// if there is a straight segment ahead of it, and which
// segment to take. OVERSHOOT tells the Zumo how far it needs
// to overshoot the segment to find out any of these things.
#define OVERSHOOT(line_thickness)((INCHES_TO_ZUNITS * (line_thickness)) / SPEED))

unsigned char buttonPin = 4;
ZumoBuzzer buzzer;
ZumoReflectanceSensorArray reflectanceSensors;
ZumoMotors motors;
Pushbutton button(buttonPin);

// SD Card Variables
const int chipSelect = 10;
File myFile;
File logFile;

// path[] keeps a log of all the turns made

```

```

// since starting the maze
char path[100] = "";
unsigned char path_length = 0; // the length of the path

void setup()
{

    Serial.begin(9600);
    Serial.println("||--- Initializing Zumo Maze Solver");
    Serial.println("");

    unsigned int sensors[6];
    unsigned short count = 0;
    unsigned short last_status = 0;
    int turn_direction = 1;

    buzzer.play(">g32>>c32");

    reflectanceSensors.init();
    motors.setSpeeds(0, 0);

    delay(500);
    //pinMode(13, OUTPUT);
    digitalWrite(13, HIGH);           // turn on LED to indicate we are in calibration mode

    button.waitForButton();

    Serial.println("||--- Initializing SD Card");

    pinMode(chipSelect, OUTPUT);

    if(SD.begin()){
        Serial.println("||-- SD Card Initialization Complete");
    } else {
        Serial.println("||-- SD Card Initialization Failed");
    }

    myFile = SD.open("random.txt", FILE_WRITE);

    if(myFile){

        Serial.println("||-- Writing To File Initialized");
        myFile.println("||-- A Random Test");
        myFile.close();
        Serial.println("||-- Writing To File Completed");

    } else {

```

```

    Serial.println("||-- File Initialization Failed");

}

logFile = SD.open("zumoPath.txt", FILE_WRITE);

if(logFile){

    logFile.println("||--- Start of the Path");
    logFile.println("");
    logFile.close();

}

Serial.println("");
Serial.println("");

// Calibrate the Zumo by sweeping it from left to right
for(int i = 0; i < 4; i++)
{
    // Zumo will turn clockwise if turn_direction = 1.
    // If turn_direction = -1 Zumo will turn counter-clockwise.
    turn_direction *= -1;

    // Turn direction.
    motors.setSpeeds(turn_direction * TURN_SPEED, -1*turn_direction * TURN_SPEED);

    // This while loop monitors line position
    // until the turn is complete.
    while(count < 2)
    {
        reflectanceSensors.calibrate();
        reflectanceSensors.readLine(sensors);
        if(turn_direction < 0)
        {
            // If the right most sensor changes from (over white space -> over
            // line or over line -> over white space) add 1 to count.
            count += ABOVE_LINE(sensors[5]) ^ last_status;
            last_status = ABOVE_LINE(sensors[5]);
        }
        else
        {
            // If the left most sensor changes from (over white space -> over
            // line or over line -> over white space) add 1 to count.
            count += ABOVE_LINE(sensors[0]) ^ last_status;
            last_status = ABOVE_LINE(sensors[0]);
        }
    }
}

```

```

    count = 0;
    last_status = 0;
}

// Turn Left.
turn('L');

motors.setSpeeds(0, 0);

// Sound off buzzer to denote Zumo is finished calibrating
buzzer.play("L16 cdegre4");

// Turn off LED to indicate we are through with calibration
digitalWrite(13, LOW);
}

void loop()
{

    // solveMaze() explores every segment
    // of the maze until it finds the finish
    // line.
    solveMaze();

    // Sound off buzzer to denote Zumo has solved the maze
    buzzer.play(">>a32");

    Serial.println("||--- Maze Solved");

    // Logging the info into the SD Card
    logFile = SD.open("zumoPath.txt", FILE_WRITE);

    if(logFile){

        logFile.println("");
        logFile.println("");
        logFile.println("||--- Maze Solved");
        logFile.println("");
        logFile.println("");
        logFile.close();
        Serial.println("||-- Written To File");

    } else {

        Serial.println("||-- Not Logged To SD");

    }
}

```

```

// The maze has been solved. When the user
// places the Zumo at the starting line
// and pushes the Zumo button, the Zumo
// knows where the finish line is and
// will automatically navigate.
while(1)
{
    button.waitForButton();

    goToFinishLine();
    Serial.println("||-- Reached FinishLine");

    // Sound off buzzer to denote Zumo is at the finish line.
    buzzer.play(">a32");

}
}

// Turns according to the parameter dir, which should be
// 'L' (left), 'R' (right), 'S' (straight), or 'B' (back).
void turn(char dir)
{
    // count and last_status help
    // keep track of how much further
    // the Zumo needs to turn.
    unsigned short count = 0;
    unsigned short last_status = 0;
    unsigned int sensors[6];

    // dir tests for which direction to turn
    switch(dir)
    {

        // Since we're using the sensors to coordinate turns instead of timing them,
        // we can treat a left turn the same as a direction reversal: they differ only
        // in whether the zumo will turn 90 degrees or 180 degrees before seeing the
        // line under the sensor. If 'B' is passed to the turn function when there is a
        // left turn available, then the Zumo will turn onto the left segment.
        case 'L':
        case 'B':
            // Turn Left.
            motors.setSpeeds(-TURN_SPEED, TURN_SPEED);

```

```

// This while loop monitors line position
// until the turn is complete.
while(count < 2)
{
    reflectanceSensors.readLine(sensors);

    // Increment count whenever the state of the sensor changes
    // (white->black and black->white) since the sensor should
    // pass over 1 line while the robot is turning, the final
    // count should be 2
    count += ABOVE_LINE(sensors[1]) ^ last_status;
    last_status = ABOVE_LINE(sensors[1]);
}

break;

case 'R':
    // Turn right.
    motors.setSpeeds(TURN_SPEED, -TURN_SPEED);

    // This while loop monitors line position
    // until the turn is complete.
    while(count < 2)
    {
        reflectanceSensors.readLine(sensors);
        count += ABOVE_LINE(sensors[4]) ^ last_status;
        last_status = ABOVE_LINE(sensors[4]);
    }

    break;

case 'S':
    // Don't do anything!
    break;
}
}

// This function decides which way to turn during the learning phase of
// maze solving. It uses the variables found_left, found_straight, and
// found_right, which indicate whether there is an exit in each of the
// three directions, applying the "left hand on the wall" strategy.
char selectTurn(unsigned char found_left, unsigned char found_straight,
    unsigned char found_right)
{
    // Make a decision about how to turn. The following code
    // implements a left-hand-on-the-wall strategy, where we always
    // turn as far to the left as possible.

```

```
if(found_left){

    Serial.println(">> Going Left");

    logFile = SD.open("zumoPath.txt", FILE_WRITE);

    if(logFile){

        logFile.println(">> Going Left");
        logFile.close();
        Serial.println("||-- Written To File");
        Serial.println("");

    } else {

        Serial.println("||-- Not Logged To SD");

    }

    return 'L';

}
else if(found_straight){

    Serial.println(">> Going Straight");

    logFile = SD.open("zumoPath.txt", FILE_WRITE);

    if(logFile){

        logFile.println(">> Going Straight");
        logFile.close();
        Serial.println("||-- Written To File");
        Serial.println("");

    } else {

        Serial.println("||-- Not Logged To SD");

    }

    return 'S';

}
else if(found_right){

    Serial.println(">> Going Right");
```

```

    logFile = SD.open("zumoPath.txt", FILE_WRITE);

    if(logFile){

        logFile.println(">> Going Right");
        logFile.close();
        Serial.println("||-- Written To File");
        Serial.println("");

    } else {

        Serial.println("||-- Not Logged To SD");

    }

    return 'R';

}
else{

    Serial.println(">> Going Back");

    logFile = SD.open("zumoPath.txt", FILE_WRITE);

    if(logFile){

        logFile.println(">> Going Back");
        logFile.close();
        Serial.println("||-- Written To File");
        Serial.println("");

    } else {

        Serial.println("||-- Not Logged To SD");

    }

    return 'B';

}
}

// The maze is broken down into segments. Once the Zumo decides
// which segment to turn on, it will navigate until it finds another
// intersection. followSegment() will then return after the
// intersection is found.
void followSegment()
{

```



```

unsigned int position;
unsigned int sensors[6];
int offset_from_center;
int power_difference;

while(1)
{
    // Get the position of the line.
    position = reflectanceSensors.readLine(sensors);

    // The offset_from_center should be 0 when we are on the line.
    offset_from_center = ((int)position) - 2500;

    // Compute the difference between the two motor power settings,
    // m1 - m2. If this is a positive number the robot will turn
    // to the left. If it is a negative number, the robot will
    // turn to the right, and the magnitude of the number determines
    // the sharpness of the turn.
    power_difference = offset_from_center / 3;

    // Compute the actual motor settings. We never set either motor
    // to a negative value.
    if(power_difference > SPEED)
        power_difference = SPEED;
    if(power_difference < -SPEED)
        power_difference = -SPEED;

    if(power_difference < 0)
        motors.setSpeeds(SPEED + power_difference, SPEED);
    else
        motors.setSpeeds(SPEED, SPEED - power_difference);

    // We use the inner four sensors (1, 2, 3, and 4) for
    // determining whether there is a line straight ahead, and the
    // sensors 0 and 5 for detecting lines going to the left and
    // right.

    if(!ABOVE_LINE(sensors[0]) && !ABOVE_LINE(sensors[1]) && !ABOVE_LINE(sensors[2]) && !ABOVE_LINE(sensors[3]) && !ABOVE_LINE(sensors[4]) && !ABOVE_LINE(sensors[5]))
    {
        // There is no line visible ahead, and we didn't see any
        // intersection. Must be a dead end.
        return;
    }
    else if(ABOVE_LINE(sensors[0]) || ABOVE_LINE(sensors[5]))
    {
        // Found an intersection.
        return;
    }
}

```

```

    }

}
}

// The solveMaze() function works by applying a "left hand on the wall" strategy:
// the robot follows a segment until it reaches an intersection, where it takes the
// leftmost fork available to it. It records each turn it makes, and as long as the
// maze has no loops, this strategy will eventually lead it to the finish. Afterwards,
// the recorded path is simplified by removing dead ends. More information can be
// found in the 3pi maze solving example.
void solveMaze()
{
    while(1)
    {
        // Navigate current line segment
        followSegment();

        // These variables record whether the robot has seen a line to the
        // left, straight ahead, and right, while examining the current
        // intersection.
        unsigned char found_left = 0;
        unsigned char found_straight = 0;
        unsigned char found_right = 0;

        // Now read the sensors and check the intersection type.
        unsigned int sensors[6];
        reflectanceSensors.readLine(sensors);

        // Check for Left and right exits.
        if(ABOVE_LINE(sensors[0]))
            found_left = 1;
        if(ABOVE_LINE(sensors[5]))
            found_right = 1;

        // Drive straight a bit more, until we are
        // approximately in the middle of intersection.
        // This should help us better detect if we
        // have left or right segments.
        motors.setSpeeds(SPEED, SPEED);
        delay(OVERSHOOT(LINE_THICKNESS)/2);

        reflectanceSensors.readLine(sensors);

        // Check for Left and right exits.
        if(ABOVE_LINE(sensors[0]))
            found_left = 1;
        if(ABOVE_LINE(sensors[5]))

```

```

        found_right = 1;

        // After driving a little further, we
        // should have passed the intersection
        // and can check to see if we've hit the
        // finish line or if there is a straight segment
        // ahead.
        delay(OVERSHOOT(LINE_THICKNESS)/2);

        // Check for a straight exit.
        reflectanceSensors.readLine(sensors);

        // Check again to see if left or right segment has been found
        if(ABOVE_LINE(sensors[0]))
            found_left = 1;
        if(ABOVE_LINE(sensors[5]))
            found_right = 1;

        if(ABOVE_LINE(sensors[1]) || ABOVE_LINE(sensors[2]) || ABOVE_LINE(sensors[3]) || ABOVE_LINE(sensors[4]))
            found_straight = 1;

        // Check for the ending spot.
        // If all four middle sensors are on dark black, we have
        // solved the maze.
        if(ABOVE_LINE(sensors[1]) && ABOVE_LINE(sensors[2]) && ABOVE_LINE(sensors[3]) && ABOVE_LINE(sensors[4]))
        {
            motors.setSpeeds(0,0);
            break;
        }

        // Intersection identification is complete.
        unsigned char dir = selectTurn(found_left, found_straight, found_right);

        // Make the turn indicated by the path.
        turn(dir);

        // Store the intersection in the path variable.
        path[path_length] = dir;
        path_length++;

        // You should check to make sure that the path_length does not
        // exceed the bounds of the array. We'll ignore that in this
        // example.

        // Simplify the learned path.
        simplifyPath();

```

```

    }
}

// Now enter an infinite loop - we can re-run the maze as many
// times as we want to.
void goToFinishLine()
{
    unsigned int sensors[6];
    int i = 0;

    // Turn around if the Zumo is facing the wrong direction.
    if(path[0] == 'B')
    {
        turn('B');
        i++;
    }

    for(;i<path_length;i++)
    {

        followSegment();

        // Drive through the intersection.
        motors.setSpeeds(SPEED, SPEED);
        delay(OVERSHOOT(LINE_THICKNESS));

        // Make a turn according to the instruction stored in
        // path[i].
        turn(path[i]);
    }

    // Follow the last segment up to the finish.
    followSegment();

    // The finish line has been reached.
    // Return and wait for another button push to
    // restart the maze.
    reflectanceSensors.readLine(sensors);
    motors.setSpeeds(0,0);

    return;
}

// simplifyPath analyzes the path[] array and reduces all the
// turns. For example: Right turn + Right turn = (1) Back turn.
void simplifyPath()
{

```

```

// only simplify the path if the second-to-last turn was a 'B'
if(path_length < 3 || path[path_length - 2] != 'B')
return;

int total_angle = 0;
int i;

for(i = 1; i <= 3; i++)
{
    switch(path[path_length - i])
    {
        case 'R':
            total_angle += 90;
            break;
        case 'L':
            total_angle += 270;
            break;
        case 'B':
            total_angle += 180;
            break;
    }
}

// Get the angle as a number between 0 and 360 degrees.
total_angle = total_angle % 360;

// Replace all of those turns with a single one.
switch(total_angle)
{
    case 0:
        path[path_length - 3] = 'S';
        break;
    case 90:
        path[path_length - 3] = 'R';
        break;
    case 180:
        path[path_length - 3] = 'B';
        break;
    case 270:
        path[path_length - 3] = 'L';
        break;
}

// The path is now two steps shorter.
path_length -= 2;
}

```

3.2 Zumo Motors Code:

We were provided with the following code with the logic that we have already understood and discussed over the past few chapters.

```
#include "ZumoMotors.h"

#define PWM_L 5
#define PWM_R 6
#define DIR_L 8
#define DIR_R 7

static boolean flipLeft = false;
static boolean flipRight = false;

// constructor (doesn't do anything)
ZumoMotors::ZumoMotors()
{
}

// initialize timer1 to generate the proper PWM outputs to the motor drivers
void ZumoMotors::init2()
{
    pinMode(PWM_L, OUTPUT);
    pinMode(PWM_R, OUTPUT);
    pinMode(DIR_L, OUTPUT);
    pinMode(DIR_R, OUTPUT);

#ifdef USE_20KHZ_PWM

    TCCR0A = 0b10100000;
    TCCR0B = 0b00010001;
    ICR0 = 400;
#endif
}

// enable/disable flipping of left motor
void ZumoMotors::flipLeftMotor(boolean flip)
{
    flipLeft = flip;
}

// enable/disable flipping of right motor
void ZumoMotors::flipRightMotor(boolean flip)
{
    flipRight = flip;
}
```

```

// set speed for left motor; speed is a number between -400 and 400
void ZumoMotors::setLeftSpeed(int speed)
{
    init(); // initialize if necessary

    boolean reverse = 0;

    if (speed < 0)
    {
        speed = -speed; // make speed a positive quantity
        reverse = 1;    // preserve the direction
    }
    if (speed > 400) // Max
        speed = 400;

#ifdef USE_20KHZ_PWM
    OCR0B = speed;
#else
    analogWrite(PWM_L, speed * 51 / 80); // default to using analogWrite, mapping 400 to 255
#endif

    if (reverse ^ flipLeft) // flip if speed was negative or flipLeft setting is active, but
not both
        digitalWrite(DIR_L, HIGH);
    else
        digitalWrite(DIR_L, LOW);
}

// set speed for right motor; speed is a number between -400 and 400
void ZumoMotors::setRightSpeed(int speed)
{
    init(); // initialize if necessary

    boolean reverse = 0;

    if (speed < 0)
    {
        speed = -speed; // Make speed a positive quantity
        reverse = 1;    // Preserve the direction
    }
    if (speed > 400) // Max PWM dutycycle
        speed = 400;

#ifdef USE_20KHZ_PWM
    OCR0A = speed;
#else
    analogWrite(PWM_R, speed * 51 / 80); // default to using analogWrite, mapping 400 to 255
#endif
}

```

```

    if (reverse ^ flipRight) // flip if speed was negative or flipRight setting is active, but not both
        digitalWrite(DIR_R, HIGH);
    else
        digitalWrite(DIR_R, LOW);
}

// set speed for both motors
void ZumoMotors::setSpeeds(int leftSpeed, int rightSpeed)
{
    setLeftSpeed(leftSpeed);
    setRightSpeed(rightSpeed);
}

```

3.3 Zumo Buzzer Code:

We were provided with the following code with the logic that we have already understood and discussed over the past few chapters.

```

#ifndef F_CPU
#define F_CPU 16000000UL // Standard Arduinos run at 16 MHz
#endif //!F_CPU

#include <avr/interrupt.h>
#include <avr/pgmspace.h>
#include "ZumoBuzzer.h"

#ifdef __AVR_ATmega32U4__

// PD7 (OC4D)
#define BUZZER_DDR DDRD
#define BUZZER      (1 << PORTD7)

#define TIMER4_CLK_8 0x4 // 2 MHz

#define ENABLE_TIMER_INTERRUPT() TIMSK4 = (1 << TOIE4)
#define DISABLE_TIMER_INTERRUPT() TIMSK4 = 0

#else // 168P or 328P

// PD3 (OC2B)
#define BUZZER_DDR DDRD
#define BUZZER      (1 << PORTD3)

#define TIMER2_CLK_32 0x3 // 500 kHz

```



```

static const unsigned int cs2_divider[] = {0, 1, 8, 32, 64, 128, 256, 1024};

#define ENABLE_TIMER_INTERRUPT()    TIMSK2 = (1 << TOIE2)
#define DISABLE_TIMER_INTERRUPT()  TIMSK2 = 0

#endif

unsigned char buzzerInitialized = 0;
volatile unsigned char buzzerFinished = 1; // flag: 0 while playing
const char *buzzerSequence = 0;

static volatile unsigned int buzzerTimeout = 0; // tracks buzzer time limit
static char play_mode_setting = PLAY_AUTOMATIC;

extern volatile unsigned char buzzerFinished; // flag: 0 while playing
extern const char *buzzerSequence;

static unsigned char use_program_space; // boolean: true if we should
// use program space

// music settings and defaults
static unsigned char octave = 4; // the current octave
static unsigned int whole_note_duration = 2000; // the duration of a whole note
static unsigned int note_type = 4; // 4 for quarter, etc
static unsigned int duration = 500; // the duration of a note in ms
static unsigned int volume = 15; // the note volume
static unsigned char staccato = 0; // true if playing staccato

// staccato handling
static unsigned char staccato_rest_duration; // duration of a staccato
// rest, or zero if it is time
// to play a note

static void nextNote();

#ifdef __AVR_ATmega32U4__

// Timer4 overflow interrupt
ISR (TIMER4_OVF_vect)
{
    if (buzzerTimeout-- == 0)
    {
        DISABLE_TIMER_INTERRUPT();
        sei(); // re-
enable global interrupts (nextNote() is very slow)
        TCCR4B = (TCCR4B & 0xF0) | TIMER4_CLK_8; // select I/O clock
        unsigned int top = (F_CPU/16) / 1000; // set TOP for freq = 1 kHz:

```

```

    TC4H = top >> 8;           // top 2 bits... (TC4H temporarily stores top
2 bits of 10-bit accesses)
    OCR4C = top;               // and bottom 8 bits
    TC4H = 0;                  // 0% duty cycle: top 2 bits...
    OCR4D = 0;                  // and bottom 8 bits
    buzzerFinished = 1;
    if (buzzerSequence && (play_mode_setting == PLAY_AUTOMATIC))
        nextNote();
}
}

#else

// Timer2 overflow interrupt
ISR (TIMER2_OVF_vect)
{
    if (buzzerTimeout-- == 0)
    {
        DISABLE_TIMER_INTERRUPT();
        sei();                 // re-
enable global interrupts (nextNote() is very slow)
        TCCR2B = (TCCR2B & 0xF8) | TIMER2_CLK_32; // select I/O clock
        OCR2A = (F_CPU/64) / 1000; // set TOP for freq = 1 kHz
        OCR2B = 0;              // 0% duty cycle
        buzzerFinished = 1;
        if (buzzerSequence && (play_mode_setting == PLAY_AUTOMATIC))
            nextNote();
    }
}

#endif

// constructor

ZumoBuzzer::ZumoBuzzer()
{
}

// this is called by playFrequency()
inline void ZumoBuzzer::init()
{
    if (!buzzerInitialized)
    {
        buzzerInitialized = 1;
        init2();
    }
}

```

```

// initializes timer4 (32U4) or timer2 (328P) for buzzer control
void ZumoBuzzer::init2()
{
    DISABLE_TIMER_INTERRUPT();

#ifdef __AVR_ATmega32U4__
    TCCR4A = 0x00; // bits 7 and 6 clear: normal port op., OC4A disconnected

    TCCR4B = 0x04; // bit 7 clear: disable PWM inversion

    TCCR4C = 0x09; // bits 7 and 6 clear: normal port op., OC4A disconnected

    TCCR4D = 0x01; // bit 7 clear: disable fault protection interrupt

    unsigned int top = (F_CPU/16) / 1000; // set TOP for freq = 1 kHz:
    TC4H = top >> 8; // top 2 bits...
    OCR4C = top; // and bottom 8 bits
    TC4H = 0; // 0% duty cycle: top 2 bits...
    OCR4D = 0; // and bottom 8 bits
#else
    TCCR2A = 0x21; // bits 7 and 6 clear: normal port op., OC4A disconnected

    TCCR2B = 0x0B; // bit 7 clear: no force output compare for channel A

    OCR2A = (F_CPU/64) / 1000; // set TOP for freq = 1 kHz
    OCR2B = 0; // 0% duty cycle
#endif

    BUZZER_DDR |= BUZZER; // buzzer pin set as an output
    sei();
}

void ZumoBuzzer::playFrequency(unsigned int freq, unsigned int dur,
                               unsigned char volume)
{
    init(); // initializes the buzzer if necessary
    buzzerFinished = 0;

    unsigned int timeout;
    unsigned char multiplier = 1;

    if (freq & DIV_BY_10) // if frequency's DIV_BY_10 bit is set
    {
        // then the true frequency is freq/10
        multiplier = 10; // (gives higher resolution for small freqs)
        freq &= ~DIV_BY_10; // clear DIV_BY_10 bit
    }
}

```

```

    unsigned char min = 40 * multiplier;
    if (freq < min) // min frequency allowed is 40 Hz
        freq = min;
    if (multiplier == 1 && freq > 10000)
        freq = 10000; // max frequency allowed is 10kHz

#ifdef __AVR_ATmega32U4__
    unsigned long top;
    unsigned char dividerExponent = 0;

    // calculate necessary clock source and counter top value to get freq
    top = (unsigned int)(((F_CPU/2 * multiplier) + (freq >> 1))/ freq);

    while (top > 1023)
    {
        dividerExponent++;
        top = (unsigned int)((((F_CPU/2 >> (dividerExponent)) * multiplier) + (freq >> 1))/ freq);
    }
#else
    unsigned int top;
    unsigned char newCS2 = 2; // try prescaler divider of 8 first (minimum necessary for 10 kHz)
    unsigned int divider = cs2_divider[newCS2];

    // calculate necessary clock source and counter top value to get freq
    top = (unsigned int)(((F_CPU/16 * multiplier) + (freq >> 1))/ freq);

    while (top > 255)
    {
        divider = cs2_divider[++newCS2];
        top = (unsigned int)(((F_CPU/2/divider * multiplier) + (freq >> 1))/ freq);
    }
#endif

    // set timeout (duration):
    if (multiplier == 10)
        freq = (freq + 5) / 10;

    if (freq == 1000)
        timeout = dur; // duration for silent notes is exact
    else
        timeout = (unsigned int)((long)dur * freq / 1000);

    if (volume > 15)
        volume = 15;

```

```

DISABLE_TIMER_INTERRUPT();           // disable interrupts while writing to registers

#ifdef __AVR_ATmega32U4__
    TCCR4B = (TCCR4B & 0xF0) | (dividerExponent + 1); // select timer 4 clock prescaler: divider = 2^n if CS4 = n+1
    TC4H = top >> 8;                               // set timer 1 pwm frequency: top 2 bits...
    OCR4C = top;                                     // and bottom 8 bits
    unsigned int width = top >> (16 - volume);       // set duty cycle (volume):
    TC4H = width >> 8;                               // top 2 bits...
    OCR4D = width;                                    // and bottom 8 bits
    buzzerTimeout = timeout;                         // set buzzer duration

    TIFR4 |= 0xFF; // clear any pending t4 overflow int.
#else
    TCCR2B = (TCCR2B & 0xF8) | newCS2; // select timer 2 clock prescaler
    OCR2A = top;                          // set timer 2 pwm frequency
    OCR2B = top >> (16 - volume);          // set duty cycle (volume)
    buzzerTimeout = timeout;              // set buzzer duration

    TIFR2 |= 0xFF; // clear any pending t2 overflow int.
#endif

ENABLE_TIMER_INTERRUPT();
}

void ZumoBuzzer::playNote(unsigned char note, unsigned int dur,
                          unsigned char volume)
{
    unsigned int freq = 0;
    unsigned char offset_note = note - 16;

    if (note == SILENT_NOTE || volume == 0)
    {
        freq = 1000; // silent notes => use 1kHz freq (for cycle counter)
        playFrequency(freq, dur, 0);
        return;
    }

    if (note <= 16)
        offset_note = 0;
    else if (offset_note > 95)
        offset_note = 95;

    unsigned char exponent = offset_note / 12;

    switch (offset_note - exponent * 12) // equivalent to (offset_note % 12)

```

```

{
    case 0:          // note E1 = 41.2 Hz
        freq = 412;
        break;
    case 1:          // note F1 = 43.7 Hz
        freq = 437;
        break;
    case 2:          // note F#1 = 46.3 Hz
        freq = 463;
        break;
    case 3:          // note G1 = 49.0 Hz
        freq = 490;
        break;
    case 4:          // note G#1 = 51.9 Hz
        freq = 519;
        break;
    case 5:          // note A1 = 55.0 Hz
        freq = 550;
        break;
    case 6:          // note A#1 = 58.3 Hz
        freq = 583;
        break;
    case 7:          // note B1 = 61.7 Hz
        freq = 617;
        break;
    case 8:          // note C2 = 65.4 Hz
        freq = 654;
        break;
    case 9:          // note C#2 = 69.3 Hz
        freq = 693;
        break;
    case 10:         // note D2 = 73.4 Hz
        freq = 734;
        break;
    case 11:         // note D#2 = 77.8 Hz
        freq = 778;
        break;
}

if (exponent < 7)
{
    freq = freq << exponent; // frequency *= 2 ^ exponent
    if (exponent > 1)        // if the frequency is greater than 160 Hz
        freq = (freq + 5) / 10; // we don't need the extra resolution
    else
        freq += DIV_BY_10;    // else keep the added digit of resolution
}
else

```

```

    freq = (freq * 64 + 2) / 5; // == freq * 2^7 / 10 without int overflow

    if (volume > 15)
        volume = 15;
    playFrequency(freq, dur, volume); // set buzzer this freq/duration
}

// Returns 1 if the buzzer is currently playing, otherwise it returns 0
unsigned char ZumoBuzzer::isPlaying()
{
    return !buzzerFinished || buzzerSequence != 0;
}

void ZumoBuzzer::play(const char *notes)
{
    DISABLE_TIMER_INTERRUPT(); // prevent this from being interrupted
    buzzerSequence = notes;
    use_program_space = 0;
    staccato_rest_duration = 0;
    nextNote(); // this re-enables the timer1 interrupt
}

void ZumoBuzzer::playFromProgramSpace(const char *notes_p)
{
    DISABLE_TIMER_INTERRUPT(); // prevent this from being interrupted
    buzzerSequence = notes_p;
    use_program_space = 1;
    staccato_rest_duration = 0;
    nextNote(); // this re-enables the timer1 interrupt
}

// stop all sound playback immediately
void ZumoBuzzer::stopPlaying()
{
    DISABLE_TIMER_INTERRUPT(); // disable interrupts

#ifdef __AVR_ATmega32U4__
    TCCR4B = (TCCR4B & 0xF0) | TIMER4_CLK_8; // select IO clock
    unsigned int top = (F_CPU/16) / 1000; // set TOP for freq = 1 kHz:
    TC4H = top >> 8; // top 2 bits... (TC4H temporarily stores top 2
bits of 10-bit accesses)
    OCR4C = top; // and bottom 8 bits
    TC4H = 0; // 0% duty cycle: top 2 bits...
    OCR4D = 0; // and bottom 8 bits
#else
    TCCR2B = (TCCR2B & 0xF8) | TIMER2_CLK_32; // select IO clock

```

```

OCR2A = (F_CPU/64) / 1000;           // set TOP for freq = 1 kHz
OCR2B = 0;                           // 0% duty cycle
#endif

    buzzerFinished = 1;
    buzzerSequence = 0;
}

static char currentCharacter()
{
    char c = 0;
    do
    {
        if(use_program_space)
            c = pgm_read_byte(buzzerSequence);
        else
            c = *buzzerSequence;

        if(c >= 'A' && c <= 'Z')
            c += 'a'-'A';
    } while(c == ' ' && (buzzerSequence ++));

    return c;
}

static unsigned int getNumber()
{
    unsigned int arg = 0;

    // read all digits, one at a time
    char c = currentCharacter();
    while(c >= '0' && c <= '9')
    {
        arg *= 10;
        arg += c-'0';
        buzzerSequence ++;
        c = currentCharacter();
    }

    return arg;
}

static void nextNote()
{
    unsigned char note = 0;
    unsigned char rest = 0;
    unsigned char tmp_octave = octave; // the octave for this note
    unsigned int tmp_duration; // the duration of this note

```



```

    unsigned int dot_add;

    char c; // temporary variable

    // if we are playing staccato, after every note we play a rest
    if(staccato && staccato_rest_duration)
    {
        ZumoBuzzer::playNote(SILENT_NOTE, staccato_rest_duration, 0);
        staccato_rest_duration = 0;
        return;
    }

    parse_character:

    // Get current character
    c = currentCharacter();
    buzzerSequence ++;

    // Interpret the character.
    switch(c)
    {
    case '>':
        // shift the octave temporarily up
        tmp_octave ++;
        goto parse_character;
    case '<':
        // shift the octave temporarily down
        tmp_octave --;
        goto parse_character;
    case 'a':
        note = NOTE_A(0);
        break;
    case 'b':
        note = NOTE_B(0);
        break;
    case 'c':
        note = NOTE_C(0);
        break;
    case 'd':
        note = NOTE_D(0);
        break;
    case 'e':
        note = NOTE_E(0);
        break;
    case 'f':
        note = NOTE_F(0);
        break;
    case 'g':

```

```

    note = NOTE_G(0);
    break;
case 'l':
    // set the default note duration
    note_type = getNumber();
    duration = whole_note_duration/note_type;
    goto parse_character;
case 'm':
    // set music staccato or legato
    if(currentCharacter() == 'l')
        staccato = false;
    else
    {
        staccato = true;
        staccato_rest_duration = 0;
    }
    buzzerSequence ++;
    goto parse_character;
case 'o':
    // set the octave permanently
    octave = getNumber();
    tmp_octave = octave;
    goto parse_character;
case 'r':
    // Rest - the note value doesn't matter.
    rest = 1;
    break;
case 't':
    // set the tempo
    whole_note_duration = 60*400/getNumber()*10;
    duration = whole_note_duration/note_type;
    goto parse_character;
case 'v':
    // set the volume
    volume = getNumber();
    goto parse_character;
case '!':
    // reset to defaults
    octave = 4;
    whole_note_duration = 2000;
    note_type = 4;
    duration = 500;
    volume = 15;
    staccato = 0;
    // reset temp variables that depend on the defaults
    tmp_octave = octave;
    tmp_duration = duration;
    goto parse_character;

```

```

default:
    buzzerSequence = 0;
    return;
}

note += tmp_octave*12;

// handle sharps and flats
c = currentCharacter();
while(c == '+' || c == '#')
{
    buzzerSequence ++;
    note ++;
    c = currentCharacter();
}
while(c == '-')
{
    buzzerSequence ++;
    note --;
    c = currentCharacter();
}

// set the duration of just this note
tmp_duration = duration;

// If the input is 'c16', make it a 16th note, etc.
if(c > '0' && c < '9')
    tmp_duration = whole_note_duration/getNumber();

dot_add = tmp_duration/2;
while(currentCharacter() == '.')
{
    buzzerSequence ++;
    tmp_duration += dot_add;
    dot_add /= 2;
}

if(staccato)
{
    staccato_rest_duration = tmp_duration / 2;
    tmp_duration -= staccato_rest_duration;
}

// this will re-enable the timer1 overflow interrupt
ZumoBuzzer::playNote(rest ? SILENT_NOTE : note, tmp_duration, volume);
}

void ZumoBuzzer::playMode(unsigned char mode)

```

```

{
    play_mode_setting = mode;

    if(mode == PLAY_AUTOMATIC)
        playCheck();
}

unsigned char ZumoBuzzer::playCheck()
{
    if(buzzerFinished && buzzerSequence != 0)
        nextNote();
    return buzzerSequence != 0;
}

```

3.4 Push Button Code:

We were provided with the following code with the logic that we have already understood and discussed over the past few chapters.

```

#include "Pushbutton.h"

Pushbutton::Pushbutton(unsigned char pin, unsigned char pullUp, unsigned char defaultState)
{
    _pin = pin;
    _pullUp = pullUp;
    _defaultState = defaultState;
    gsdpState = 0;
    gsdrState = 0;
    gsdpPrevTimeMillis = 0;
    gsdrPrevTimeMillis = 0;
    initialized = false;
}

// wait for button to be pressed
void Pushbutton::waitForPress()
{
    init(); // initialize if necessary

    do
    {
        while (!_isPressed()); // wait for button to be pressed
        delay(10); // debounce the button press
    }
    while (!_isPressed()); // if button isn't still pressed, loop
}

```

```

// wait for button to be released
void Pushbutton::waitForRelease()
{
    init(); // initialize if necessary

    do
    {
        while (!_isPressed()); // wait for button to be released
        delay(10);             // debounce the button release
    }
    while (!_isPressed()); // if button isn't still released, loop
}

// wait for button to be pressed, then released
void Pushbutton::waitForButton()
{
    waitForPress();
    waitForRelease();
}

// indicates whether button is pressed
boolean Pushbutton::isPressed()
{
    init(); // initialize if necessary

    return _isPressed();
}

boolean Pushbutton::getSingleDebouncedPress()
{
    unsigned long timeMillis = millis();

    init(); // initialize if necessary

    switch (gsdpState)
    {
        case 0:
            if (!_isPressed()) // if button is released
            {
                gsdpPrevTimeMillis = timeMillis;
                gsdpState = 1; // proceed to next state
            }
            break;

        case 1:
            if ((timeMillis - gsdpPrevTimeMillis >= 15) && !_isPressed()) // if 15 ms or longer has elapsed and button is still released
                gsdpState = 2; // proceed to next state
    }
}

```

```

        else if (!_isPressed())
            gsdpState = 0; // button is pressed or bouncing, so go back to p
previous (initial) state
            break;

    case 2:
        if (!_isPressed()) // if button is now pressed
        {
            gsdpPrevTimeMillis = timeMillis;
            gsdpState = 3; // proceed to next state
        }
        break;

    case 3:
        if ((timeMillis - gsdpPrevTimeMillis >= 15) && _isPressed()) // if 15 ms or longer h
as elapsed and button is still pressed
        {
            gsdpState = 0; // next state becomes initial state
            return true; // report button press
        }
        else if (!_isPressed())
            gsdpState = 2; // button is released or bouncing, so go back to
previous state
            break;
    }

    return false;
}

boolean Pushbutton::getSingleDebouncedRelease()
{
    unsigned int timeMillis = millis();

    init(); // initialize if necessary

    switch (gsdrState)
    {
        case 0:
            if (!_isPressed()) // if button is pressed
            {
                gsdrPrevTimeMillis = timeMillis;
                gsdrState = 1; // proceed to next state
            }
            break;

        case 1:
            if ((timeMillis - gsdrPrevTimeMillis >= 15) && _isPressed()) // if 15 ms or longer h
as elapsed and button is still pressed

```

```

        gsdrState = 2;                                // proceed to next state
    else if (!_isPressed())
        gsdrState = 0;                                // button is released or bouncing, so go back to
previous (initial) state
        break;

    case 2:
        if (!_isPressed())                            // if button is now released
        {
            gsdrPrevTimeMillis = timeMillis;
            gsdrState = 3;                            // proceed to next state
        }
        break;

    case 3:
        if ((timeMillis - gsdrPrevTimeMillis >= 15) && !_isPressed()) // if 15 ms or longer h
as elapsed and button is still released
        {
            gsdrState = 0;                            // next state becomes initial state
            return true;                              // report button release
        }
        else if (_isPressed())
            gsdrState = 2;                            // button is pressed or bouncing, so go back to p
revious state
            break;
    }

    return false;
}

// initializes I/O pin for use as button inputs
void Pushbutton::init2()
{
    if (_pullUp == PULL_UP_ENABLED)
        pinMode(_pin, INPUT_PULLUP);
    else
        pinMode(_pin, INPUT); // high impedance

    delayMicroseconds(5); // give pull-up time to stabilize
}

// button is pressed if pin state differs from default state
inline boolean Pushbutton::_isPressed()
{
    return (digitalRead(_pin) == LOW) ^ (_defaultState == DEFAULT_STATE_LOW);
}

```

3.5 Reflector Sensor Array Code:

We were provided with the following code with the logic that we have already understood and discussed over the past few chapters.

```
#ifndef ZumoReflectanceSensorArray_h
#define ZumoReflectanceSensorArray_h

#include <QTRSensors.h>
#include <Arduino.h>

#if defined(__AVR_ATmega32U4__)
    // Arduino Leonardo
    #define ZUMO_SENSOR_ARRAY_DEFAULT_EMITTER_PIN A4
#else
    // Arduino UNO and other ATmega328P/168 Arduinos
    #define ZUMO_SENSOR_ARRAY_DEFAULT_EMITTER_PIN 2
#endif

class ZumoReflectanceSensorArray : public QTRSensorsRC
{
public:

    ZumoReflectanceSensorArray() {}

    ZumoReflectanceSensorArray(unsigned char emitterPin)
    {
        init(emitterPin);
    }

    ZumoReflectanceSensorArray(unsigned char * pins, unsigned char numSensors, unsigned int timeout = 2000,
        unsigned char emitterPin = ZUMO_SENSOR_ARRAY_DEFAULT_EMITTER_PIN)
    {
        QTRSensorsRC::init(pins, numSensors, timeout, emitterPin);
    }

    void init(unsigned char emitterPin = ZUMO_SENSOR_ARRAY_DEFAULT_EMITTER_PIN)
    {
        unsigned char sensorPins[] = { A4, A3, A1, A0, A2, 9};
        QTRSensorsRC::init(sensorPins, sizeof(sensorPins), 2000, emitterPin);
    }

    void init(unsigned char * pins, unsigned char numSensors, unsigned int timeout = 2000,
        unsigned char emitterPin = ZUMO_SENSOR_ARRAY_DEFAULT_EMITTER_PIN)
    {
        QTRSensorsRC::init(pins, numSensors, timeout, emitterPin);
    }
}
```



```
};

#endif
```

3.6 QTR Sensors Code:

We were provided with the following code with the logic that we have already understood and discussed over the past few chapters.

```
#include <stdlib.h>
#include "QTRSensors.h"
#include <Arduino.h>

// Base class data member initialization (called by derived class init())
void QTRSensors::init(unsigned char *pins, unsigned char numSensors,
    unsigned char emitterPin)
{
    calibratedMinimumOn=0;
    calibratedMaximumOn=0;
    calibratedMinimumOff=0;
    calibratedMaximumOff=0;

    if (numSensors > QTR_MAX_SENSORS)
        _numSensors = QTR_MAX_SENSORS;
    else
        _numSensors = numSensors;

    if (_pins == 0)
    {
        _pins = (unsigned char*)malloc(sizeof(unsigned char)*_numSensors);
        if (_pins == 0)
            return;
    }

    unsigned char i;
    for (i = 0; i < _numSensors; i++)
    {
        _pins[i] = pins[i];
    }

    _emitterPin = emitterPin;
}

void QTRSensors::read(unsigned int *sensor_values, unsigned char readMode)
{
```

```

    unsigned int off_values[QTR_MAX_SENSORS];
    unsigned char i;

    if(readMode == QTR_EMITTERS_ON || readMode == QTR_EMITTERS_ON_AND_OFF)
        emittersOn();
    else
        emittersOff();

    readPrivate(sensor_values);
    emittersOff();

    if(readMode == QTR_EMITTERS_ON_AND_OFF)
    {
        readPrivate(off_values);

        for(i=0;i<_numSensors;i++)
        {
            sensor_values[i] += _maxValue - off_values[i];
        }
    }
}

void QTRSensors::emittersOff()
{
    if (_emitterPin == QTR_NO_EMITTER_PIN)
        return;
    pinMode(_emitterPin, OUTPUT);
    digitalWrite(_emitterPin, LOW);
    delayMicroseconds(200);
}

void QTRSensors::emittersOn()
{
    if (_emitterPin == QTR_NO_EMITTER_PIN)
        return;
    pinMode(_emitterPin, OUTPUT);
    digitalWrite(_emitterPin, HIGH);
    delayMicroseconds(200);
}

// Resets the calibration.
void QTRSensors::resetCalibration()
{
    unsigned char i;
    for(i=0;i<_numSensors;i++)
    {
        if(calibratedMinimumOn)
            calibratedMinimumOn[i] = _maxValue;
    }
}

```

```

        if(calibratedMinimumOff)
            calibratedMinimumOff[i] = _maxValue;
        if(calibratedMaximumOn)
            calibratedMaximumOn[i] = 0;
        if(calibratedMaximumOff)
            calibratedMaximumOff[i] = 0;
    }
}

void QTRSensors::calibrate(unsigned char readMode)
{
    if(readMode == QTR_EMITTERS_ON_AND_OFF || readMode == QTR_EMITTERS_ON)
    {
        calibrateOnOrOff(&calibratedMinimumOn,
                        &calibratedMaximumOn,
                        QTR_EMITTERS_ON);
    }

    if(readMode == QTR_EMITTERS_ON_AND_OFF || readMode == QTR_EMITTERS_OFF)
    {
        calibrateOnOrOff(&calibratedMinimumOff,
                        &calibratedMaximumOff,
                        QTR_EMITTERS_OFF);
    }
}

void QTRSensors::calibrateOnOrOff(unsigned int **calibratedMinimum,
                                unsigned int **calibratedMaximum,
                                unsigned char readMode)
{
    int i;
    unsigned int sensor_values[16];
    unsigned int max_sensor_values[16];
    unsigned int min_sensor_values[16];

    // Allocate the arrays if necessary.
    if(*calibratedMaximum == 0)
    {
        *calibratedMaximum = (unsigned int*)malloc(sizeof(unsigned int)*_numSensors);

        // If the malloc failed, don't continue.
        if(*calibratedMaximum == 0)
            return;

        // Initialize the max and min calibrated values to values that
        // will cause the first reading to update them.

        for(i=0;i<_numSensors;i++)

```

```

        (*calibratedMaximum)[i] = 0;
    }
    if(*calibratedMinimum == 0)
    {
        *calibratedMinimum = (unsigned int*)malloc(sizeof(unsigned int)*_numSensors);

        // If the malloc failed, don't continue.
        if(*calibratedMinimum == 0)
            return;

        for(i=0;i<_numSensors;i++)
            (*calibratedMinimum)[i] = _maxValue;
    }

    int j;
    for(j=0;j<10;j++)
    {
        read(sensor_values,readMode);
        for(i=0;i<_numSensors;i++)
        {
            // set the max we found THIS time
            if(j == 0 || max_sensor_values[i] < sensor_values[i])
                max_sensor_values[i] = sensor_values[i];

            // set the min we found THIS time
            if(j == 0 || min_sensor_values[i] > sensor_values[i])
                min_sensor_values[i] = sensor_values[i];
        }
    }

    // record the min and max calibration values
    for(i=0;i<_numSensors;i++)
    {
        if(min_sensor_values[i] > (*calibratedMaximum)[i])
            (*calibratedMaximum)[i] = min_sensor_values[i];
        if(max_sensor_values[i] < (*calibratedMinimum)[i])
            (*calibratedMinimum)[i] = max_sensor_values[i];
    }
}

void QTRSensors::readCalibrated(unsigned int *sensor_values, unsigned char readMode)
{
    int i;

    // if not calibrated, do nothing
    if(readMode == QTR_EMITTERS_ON_AND_OFF || readMode == QTR_EMITTERS_OFF)
        if(!calibratedMinimumOff || !calibratedMaximumOff)
            return;

```

```

    if(readMode == QTR_EMITTERS_ON_AND_OFF || readMode == QTR_EMITTERS_ON)
        if(!calibratedMinimumOn || !calibratedMaximumOn)
            return;

    // read the needed values
    read(sensor_values, readMode);

    for(i=0; i<_numSensors; i++)
    {
        unsigned int calmin, calmax;
        unsigned int denominator;

        // find the correct calibration
        if(readMode == QTR_EMITTERS_ON)
        {
            calmax = calibratedMaximumOn[i];
            calmin = calibratedMinimumOn[i];
        }
        else if(readMode == QTR_EMITTERS_OFF)
        {
            calmax = calibratedMaximumOff[i];
            calmin = calibratedMinimumOff[i];
        }
        else // QTR_EMITTERS_ON_AND_OFF
        {
            if(calibratedMinimumOff[i] < calibratedMinimumOn[i]) // no meaningful signal
                calmin = _maxValue;
            else
                calmin = calibratedMinimumOn[i] + _maxValue - calibratedMinimumOff[i]; // this won't go past _maxValue

            if(calibratedMaximumOff[i] < calibratedMaximumOn[i]) // no meaningful signal
                calmax = _maxValue;
            else
                calmax = calibratedMaximumOn[i] + _maxValue - calibratedMaximumOff[i]; // this won't go past _maxValue
        }

        denominator = calmax - calmin;

        signed int x = 0;
        if(denominator != 0)
            x = (((signed long)sensor_values[i]) - calmin)
                * 1000 / denominator;
        if(x < 0)
            x = 0;
        else if(x > 1000)

```

```

        x = 1000;
        sensor_values[i] = x;
    }

}

int QTRSensors::readLine(unsigned int *sensor_values,
    unsigned char readMode, unsigned char white_line)
{
    unsigned char i, on_line = 0;
    unsigned long avg; // this is for the weighted total, which is long
                        // before division
    unsigned int sum; // this is for the denominator which is <= 64000
    static int last_value=0; // assume initially that the line is left.

    readCalibrated(sensor_values, readMode);

    avg = 0;
    sum = 0;

    for(i=0;i<_numSensors;i++) {
        int value = sensor_values[i];
        if(white_line)
            value = 1000-value;

        // keep track of whether we see the line at all
        if(value > 200) {
            on_line = 1;
        }

        // only average in values that are above a noise threshold
        if(value > 50) {
            avg += (long)(value) * (i * 1000);
            sum += value;
        }
    }

    if(!on_line)
    {
        // If it last read to the left of center, return 0.
        if(last_value < (_numSensors-1)*1000/2)
            return 0;

        // If it last read to the right of center, return the max.
        else
            return (_numSensors-1)*1000;
    }
}

```

```

    last_value = avg/sum;

    return last_value;
}

// Derived RC class constructors
QTRSensorsRC::QTRSensorsRC()
{
    calibratedMinimumOn = 0;
    calibratedMaximumOn = 0;
    calibratedMinimumOff = 0;
    calibratedMaximumOff = 0;
    _pins = 0;
}

QTRSensorsRC::QTRSensorsRC(unsigned char* pins,
    unsigned char numSensors, unsigned int timeout, unsigned char emitterPin)
{
    calibratedMinimumOn = 0;
    calibratedMaximumOn = 0;
    calibratedMinimumOff = 0;
    calibratedMaximumOff = 0;
    _pins = 0;

    init(pins, numSensors, timeout, emitterPin);
}

void QTRSensorsRC::init(unsigned char* pins,
    unsigned char numSensors, unsigned int timeout, unsigned char emitterPin)
{
    QTRSensors::init(pins, numSensors, emitterPin);

    _maxValue = timeout;
}

void QTRSensorsRC::readPrivate(unsigned int *sensor_values)
{
    unsigned char i;

    if (_pins == 0)
        return;

    for(i = 0; i < _numSensors; i++)
    {
        sensor_values[i] = _maxValue;
        digitalWrite(_pins[i], HIGH); // make sensor line an output
    }
}

```

```

        pinMode(_pins[i], OUTPUT);    // drive sensor line high
    }

    delayMicroseconds(10);            // charge lines for 10 us

    for(i = 0; i < _numSensors; i++)
    {
        pinMode(_pins[i], INPUT);    // make sensor line an input
        digitalWrite(_pins[i], LOW); // important: disable internal pull-up!
    }

    unsigned long startTime = micros();
    while (micros() - startTime < _maxValue)
    {
        unsigned int time = micros() - startTime;
        for (i = 0; i < _numSensors; i++)
        {
            if (digitalRead(_pins[i]) == LOW && time < sensor_values[i])
                sensor_values[i] = time;
        }
    }
}

// Derived Analog class constructors
QTRSensorsAnalog::QTRSensorsAnalog()
{
    calibratedMinimumOn = 0;
    calibratedMaximumOn = 0;
    calibratedMinimumOff = 0;
    calibratedMaximumOff = 0;
    _pins = 0;
}

QTRSensorsAnalog::QTRSensorsAnalog(unsigned char* pins,
    unsigned char numSensors, unsigned char numSamplesPerSensor,
    unsigned char emitterPin)
{
    calibratedMinimumOn = 0;
    calibratedMaximumOn = 0;
    calibratedMinimumOff = 0;
    calibratedMaximumOff = 0;
    _pins = 0;

    init(pins, numSensors, numSamplesPerSensor, emitterPin);
}

void QTRSensorsAnalog::init(unsigned char* pins,

```



```

    unsigned char numSensors, unsigned char numSamplesPerSensor,
    unsigned char emitterPin)
{
    QTRSensors::init(pins, numSensors, emitterPin);

    _numSamplesPerSensor = numSamplesPerSensor;
    _maxValue = 1023; // this is the maximum returned by the A/D conversion
}

void QTRSensorsAnalog::readPrivate(unsigned int *sensor_values)
{
    unsigned char i, j;

    if (_pins == 0)
        return;

    // reset the values
    for(i = 0; i < _numSensors; i++)
        sensor_values[i] = 0;

    for (j = 0; j < _numSamplesPerSensor; j++)
    {
        for (i = 0; i < _numSensors; i++)
        {
            sensor_values[i] += analogRead(_pins[i]); // add the conversion result
        }
    }

    // get the rounded average of the readings for each sensor
    for (i = 0; i < _numSensors; i++)
        sensor_values[i] = (sensor_values[i] + (_numSamplesPerSensor >> 1)) /
            _numSamplesPerSensor;
}

// the destructor frees up allocated memory
QTRSensors::~QTRSensors()
{
    if (_pins)
        free(_pins);
    if(calibratedMaximumOn)
        free(calibratedMaximumOn);
    if(calibratedMaximumOff)
        free(calibratedMaximumOff);
    if(calibratedMinimumOn)
        free(calibratedMinimumOn);
    if(calibratedMinimumOff)
        free(calibratedMinimumOff);
}

```

3.7 SD Library Code:

We were provided with the following code with the logic that we have already understood and discussed over the past few chapters.

```
#include "SD.h"

namespace SDLib {

    // Used by `getNextPathComponent`
#define MAX_COMPONENT_LEN 12 // What is max length?
#define PATH_COMPONENT_BUFFER_LEN MAX_COMPONENT_LEN+1

    bool getNextPathComponent(const char *path, unsigned int *p_offset,
                             char *buffer) {

        int bufferOffset = 0;

        int offset = *p_offset;

        // Skip root or other separator
        if (path[offset] == '/') {
            offset++;
        }

        // Copy the next next path segment
        while (bufferOffset < MAX_COMPONENT_LEN
               && (path[offset] != '/')
               && (path[offset] != '\0')) {
            buffer[bufferOffset++] = path[offset++];
        }

        buffer[bufferOffset] = '\0';

        // Skip trailing separator so we can determine if this
        // is the last component in the path or not.
        if (path[offset] == '/') {
            offset++;
        }

        *p_offset = offset;

        return (path[offset] != '\0');
    }

    boolean walkPath(const char *filepath, SdFile& parentDir,
```

```

        boolean(*callback)(SdFile& parentDir,
                           const char *filePathComponent,
                           boolean isLastComponent,
                           void *object),
        void *object = NULL) {

    SdFile subfile1;
    SdFile subfile2;

    char buffer[PATH_COMPONENT_BUFFER_LEN];

    unsigned int offset = 0;

    SdFile *p_parent;
    SdFile *p_child;

    SdFile *p_tmp_sdfile;

    p_child = &subfile1;

    p_parent = &parentDir;

    while (true) {

        boolean moreComponents = getNextPathComponent(filepath, &offset, buffer);

        boolean shouldContinue = callback((*p_parent), buffer, !moreComponents, object);

        if (!shouldContinue) {
            // TODO: Don't repeat this code?
            // If it's one we've created then we
            // don't need the parent handle anymore.
            if (p_parent != &parentDir) {
                (*p_parent).close();
            }
            return false;
        }

        if (!moreComponents) {
            break;
        }

        boolean exists = (*p_child).open(*p_parent, buffer, O_RDONLY);

        // If it's one we've created then we
        // don't need the parent handle anymore.
        if (p_parent != &parentDir) {
            (*p_parent).close();
        }
    }
}

```

```

    }

    // Handle case when it doesn't exist and we can't continue...
    if (exists) {
        // We alternate between two file handles as we go down
        // the path.
        if (p_parent == &parentDir) {
            p_parent = &subfile2;
        }

        p_tmp_sdfile = p_parent;
        p_parent = p_child;
        p_child = p_tmp_sdfile;
    } else {
        return false;
    }
}

if (p_parent != &parentDir) {
    (*p_parent).close(); // TODO: Return/ handle different?
}

return true;
}

boolean callback_pathExists(SdFile& parentDir, const char *filePathComponent,
                           boolean /* isLastComponent */, void * /* object */) {
    /*

    Callback used to determine if a file/directory exists in parent
    directory.

    Returns true if file path exists.

    */
    SdFile child;

    boolean exists = child.open(parentDir, filePathComponent, O_RDONLY);

    if (exists) {
        child.close();
    }

    return exists;
}

boolean callback_makeDirPath(SdFile& parentDir, const char *filePathComponent,

```

```

        boolean isLastComponent, void *object) {

    /*

        Callback used to create a directory in the parent directory if
        it does not already exist.

        Returns true if a directory was created or it already existed.

    */
    boolean result = false;
    SdFile child;

    result = callback_pathExists(parentDir, filePathComponent, isLastComponent, object);
    if (!result) {
        result = child.mkdir(parentDir, filePathComponent);
    }

    return result;
}

boolean callback_remove(SdFile& parentDir, const char *filePathComponent,
                        boolean isLastComponent, void * /* object */) {
    if (isLastComponent) {
        return SdFile::remove(parentDir, filePathComponent);
    }
    return true;
}

boolean callback_rmdir(SdFile& parentDir, const char *filePathComponent,
                       boolean isLastComponent, void * /* object */) {
    if (isLastComponent) {
        SdFile f;
        if (!f.open(parentDir, filePathComponent, O_READ)) {
            return false;
        }
        return f.rmdir();
    }
    return true;
}

/* Implementation of class used to create `SDCard` object. */

boolean SDClass::begin(uint8_t csPin) {
    if (root.isOpen()) {
        root.close();
    }
}

```

```

    }

    /*
       Performs the initialisation required by the sdfatlib library.

       Return true if initialization succeeds, false otherwise.

    */
    return card.init(SPI_HALF_SPEED, csPin) &&
           volume.init(card) &&
           root.openRoot(volume);
}

boolean SDClass::begin(uint32_t clock, uint8_t csPin) {
    if (root.isOpen()) {
        root.close();
    }

    return card.init(SPI_HALF_SPEED, csPin) &&
           card.setSpiClock(clock) &&
           volume.init(card) &&
           root.openRoot(volume);
}

//call this when a card is removed. It will allow you to insert and initialise a new card
.

void SDClass::end() {
    root.close();
}

// this little helper is used to traverse paths
SdFile SDClass::getParentDir(const char *filepath, int *index) {
    // get parent directory
    SdFile d1;
    SdFile d2;

    d1.openRoot(volume); // start with the mostparent, root!

    // we'll use the pointers to swap between the two objects
    SdFile *parent = &d1;
    SdFile *subdir = &d2;

    const char *origpath = filepath;

    while (strchr(filepath, '/')) {

        // get rid of leading '/'s

```

```

    if (filepath[0] == '/') {
        filepath++;
        continue;
    }

    if (! strchr(filepath, '/')) {
        // it was in the root directory, so leave now
        break;
    }

    // extract just the name of the next subdirectory
    uint8_t idx = strchr(filepath, '/') - filepath;
    if (idx > 12) {
        idx = 12; // don't let them specify long names
    }
    char subdirname[13];
    strncpy(subdirname, filepath, idx);
    subdirname[idx] = 0;

    // close the subdir (we reuse them) if open
    subdir->close();
    if (! subdir->open(parent, subdirname, O_READ)) {
        // failed to open one of the subdirectories
        return SdFile();
    }
    // move forward to the next subdirectory
    filepath += idx;

    // we reuse the objects, close it.
    parent->close();

    // swap the pointers
    SdFile *t = parent;
    parent = subdir;
    subdir = t;
}

*index = (int)(filepath - origpath);
// parent is now the parent directory of the file!
return *parent;
}

File SDClass::open(const char *filepath, uint8_t mode) {

    int pathidx;

    // do the interactive search
    SdFile parentdir = getParentDir(filepath, &pathidx);

```

```

    // no more subdirs!

    filepath += pathidx;

    if (! filepath[0]) {
        // it was the directory itself!
        return File(parentdir, "/");
    }

    // Open the file itself
    SdFile file;

    // failed to open a subdir!
    if (!parentdir.isOpen()) {
        return File();
    }

    if (! file.open(parentdir, filepath, mode)) {
        return File();
    }
    // close the parent
    parentdir.close();

    if ((mode & (O_APPEND | O_WRITE)) == (O_APPEND | O_WRITE)) {
        file.seekSet(file.fileSize());
    }
    return File(file, filepath);
}

boolean SDClass::exists(const char *filepath) {
    /*
        Returns true if the supplied file path exists.

    */
    return walkPath(filepath, root, callback_pathExists);
}

boolean SDClass::mkdir(const char *filepath) {
    /*
        Makes a single directory or a hierarchy of directories.

        A rough equivalent to `mkdir -p`.

    */
    return walkPath(filepath, root, callback_makeDirPath);
}

```



```

boolean SDClass::rmdir(const char *filepath) {
    /*
        Remove a single directory or a hierarchy of directories.

        A rough equivalent to `rm -rf`.

    */
    return walkPath(filepath, root, callback_rmdir);
}

boolean SDClass::remove(const char *filepath) {
    return walkPath(filepath, root, callback_remove);
}

// allows you to recurse into a directory
File File::openNextFile(uint8_t mode) {
    dir_t p;

    //Serial.print("\t\treading dir...");
    while (_file->readDir(&p) > 0) {

        // done if past last used entry
        if (p.name[0] == DIR_NAME_FREE) {
            //Serial.println("end");
            return File();
        }

        // skip deleted entry and entries for . and ..
        if (p.name[0] == DIR_NAME_DELETED || p.name[0] == '.') {
            //Serial.println("dots");
            continue;
        }

        // only list subdirectories and files
        if (!DIR_IS_FILE_OR_SUBDIR(&p)) {
            //Serial.println("not a file");
            continue;
        }

        // print file name with possible blank fill
        SdFile f;
        char name[13];
        _file->dirName(p, name);
        //Serial.print("try to open file ");
        //Serial.println(name);
    }
}

```

```
    if (f.open(_file, name, mode)) {  
        //Serial.println("OK!");  
        return File(f, name);  
    } else {  
        //Serial.println("ugh");  
        return File();  
    }  
}  
  
//Serial.println("nothing");  
return File();  
}  
  
void File::rewindDirectory(void) {  
    if (isDirectory()) {  
        _file->rewind();  
    }  
}  
  
SDClass SD;  
  
};
```

CHAPTER: 04

SIMULATION AND IMPLEMENTATION

4.1 Schematic:

We devised the following schematic with the logic that we have already developed and discussed over the past few chapters.

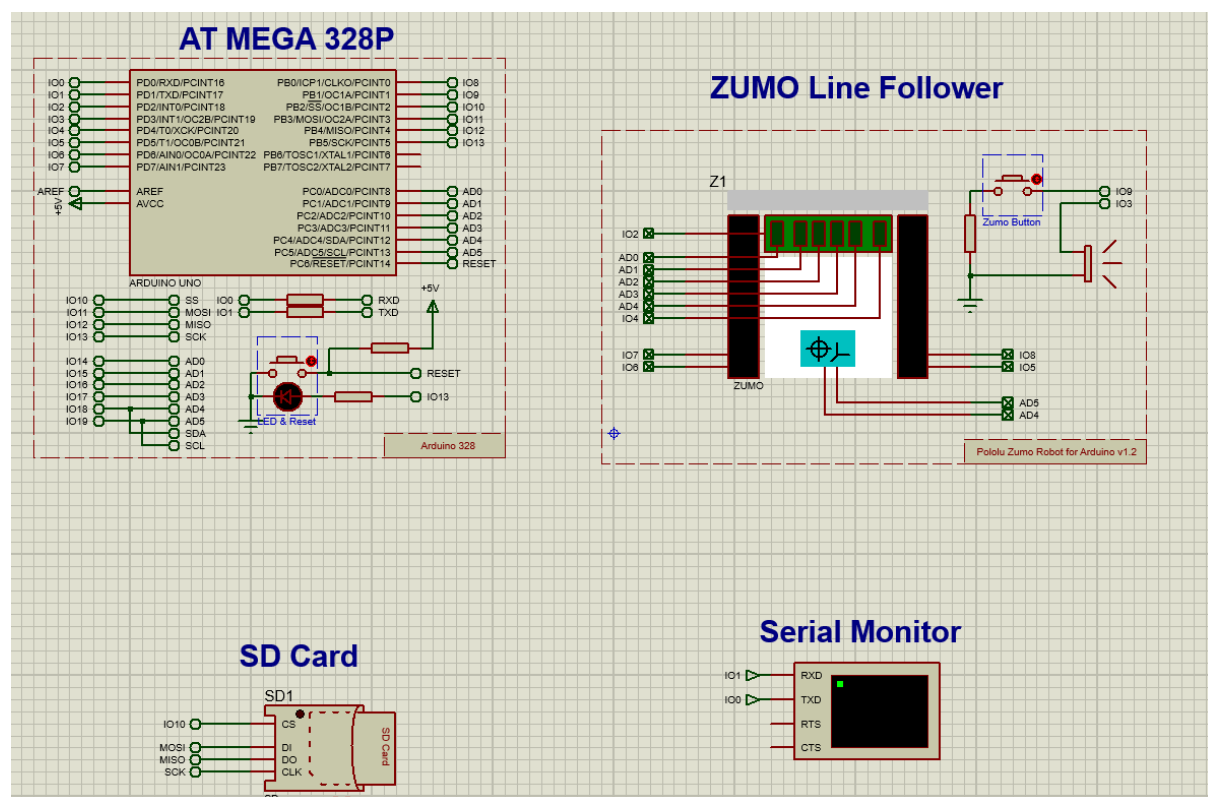


Fig 4.1: Schematic of the Zumo Maze Solver

4.2 Simulation:

We performed the following simulation with the schematic that we have already discussed in the last section of the current chapter.

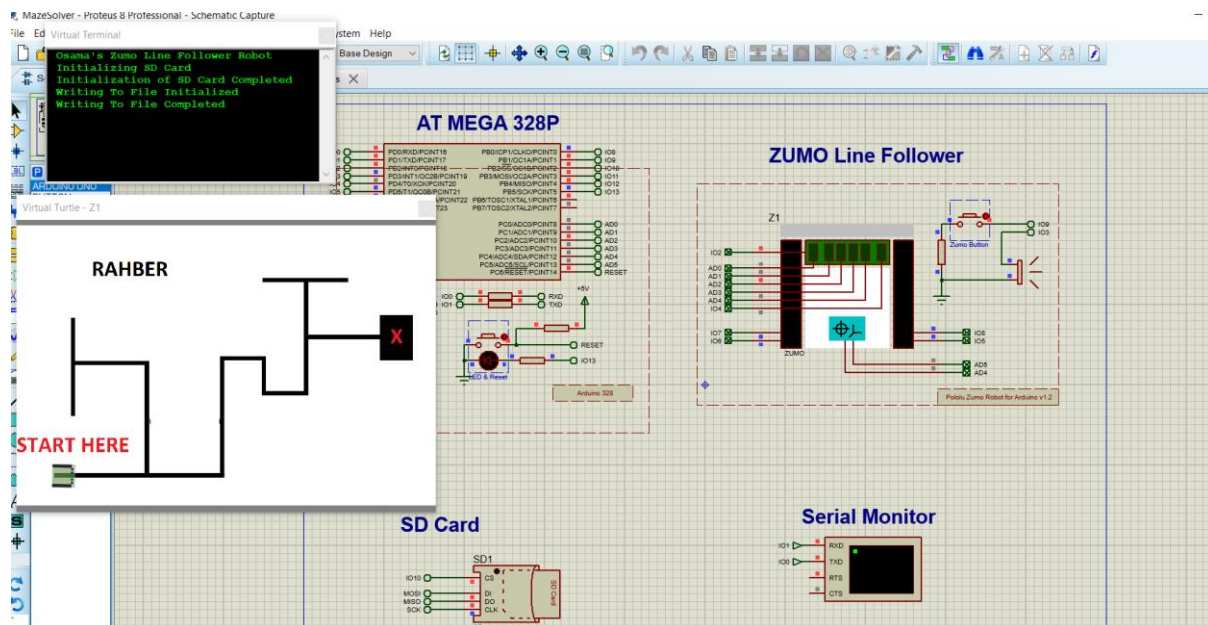
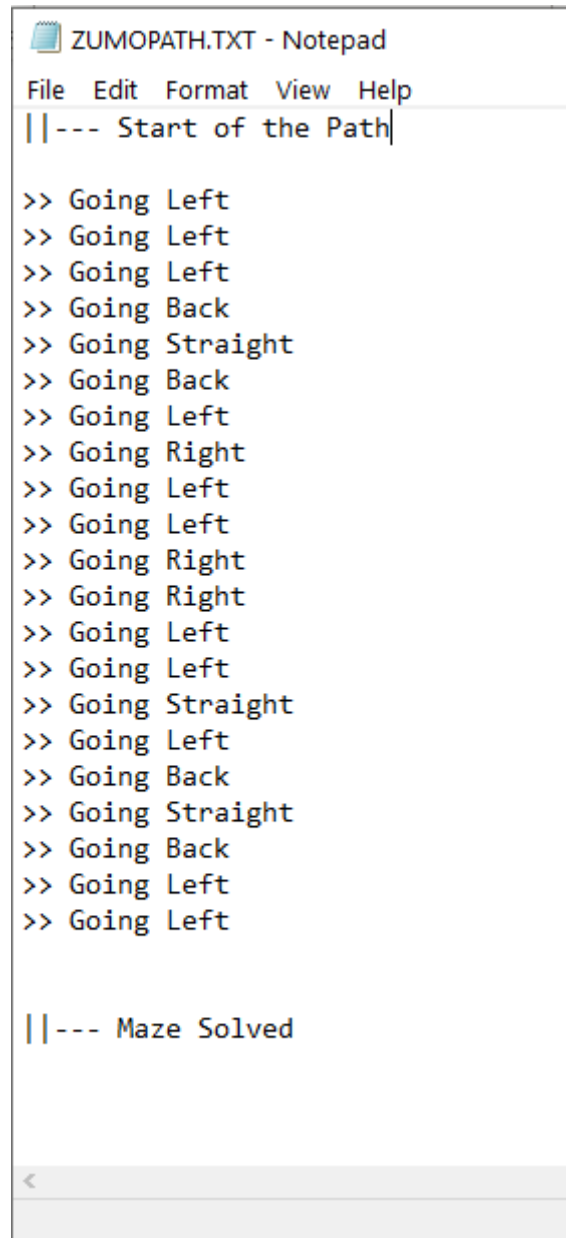


Fig 4.2: Simulation of the Zumo Maze Solver

4.3 Simulation Result:

We reached the following result from the simulation that we have already performed in the last section of the current chapter.



```
ZUMOPATH.TXT - Notepad
File Edit Format View Help
||--- Start of the Path|

>> Going Left
>> Going Left
>> Going Left
>> Going Back
>> Going Straight
>> Going Back
>> Going Left
>> Going Right
>> Going Left
>> Going Left
>> Going Right
>> Going Right
>> Going Left
>> Going Left
>> Going Straight
>> Going Left
>> Going Back
>> Going Straight
>> Going Back
>> Going Left
>> Going Left

||--- Maze Solved
```

Fig 4.3: The Result of our Simulation of Zumo Maze Solver

CHAPTER: 05

FINDINGS AND CONCLUSION

5.1 Findings and Results:

From our experiment, we came to an understanding with the following findings and results:

- We can not use GPIO pins for more than one purpose simultaneously.
- We understood the working and conceptual background of different kinds of sensors, including but not limited to the following:
 - Reflectance Sensor Array
- We understood the working and usage of PWM to control the speed of the motors and the direction of rotation.
- We were able to understand the usage of built-in clock timers of AVR to set duty cycles for the PWM.
- We utilized SPI Protocols for the communication between the virtual terminal and the microcontroller.
- Regarding the SD Card implementation, we performed the following tasks:
 - Understood the usage.
 - Implemented proper communication between microcontroller and the SD Card controller.
 - Implemented C++ File Management techniques to log the information of the sensors and the path taken by the Zumo Maze Solver.
 - We were able to display and extract the logged information in a file from the virtual SD Card.

5.2 Conclusion and Deductions:

We came to a proper understanding the following entities:

- AVR
- AVR Timers
- AVR PWM
- Serial Communication
- Data Logging
- Creating Virtual SD Card Images
- Extracting Logged Data from SD Card Images

We have learnt the usage of the aforementioned techniques and technologies with a proper understanding of their basics. We tried our best to develop a strong practical and theoretical knowledge of the techniques and algorithms formed in the undertaken activity.