Robotic Marble Launcher - Project Report

Prepared By:

Sehaj Cheema

Dimitrios Christopoulos

Owen Henry

Connor Hughes

Table of Contents

Table of Figures

Table of Tables

Introduction

For the final project our group decided to navigate the EV3 robot around a track and use a Lego ball launcher to shoot at targets placed around the track.

The track was designed similarly to the TRON days one as we had a starting point for the robot in the bottom corner and it would follow the black line around the track. The surroundings were painted orange so the colour sensor could distinguish the difference in the black tape, whereas when having a brown background, the colour sensor identified the black tape and brown surroundings as the same colour value. Therefore, we painted the surroundings orange as it was easily able to determine that it was a different colour value when compared to the black tape.

When following the black line, there were instances where the robot had to make sharp turns. This is where we made the robot distinguish the instantaneous differences in colour values as the robot would redirect itself back onto the black line. The robot would create an oscillating motion where it would turn on one motor at a time depending on whether the robot is on the black line or not. Therefore, this made our robot move along the boundary of the black taped lining.

Throughout the track, different coloured tape markers were placed on the track to signal the robot to perform a specific task. The robot was able to read these signals using the colour sensor.

When encountering the green tape, the gyro sensor is then used to turn the robot at a specified angle where the shoot launcher is aligned directly across from the shoot target. Next, the ultrasonic sensor is used to adjust the distance between the ball launcher and target. Once the robot was correctly aligned, the motor attached to the ball launcher was turned on, therefore pushing and launching the ball through the target. Once the robot has successfully shot through the target, it would reverse the motor and reload the ball launcher for the next target, then set the motor power to zero.

When encountering the red tape, the robot uses timers along with motor encoders and gyro sensor to drive straight for a specific period of time. The gyro sensor will ensure that the robot is aligned straight at an exact 90-degree angle. Then the encoders will work along with the timers to drive for a certain period of time.

When encountering the blue tape, it would signal the robot sensor that there is a sharp turn using the colour sensor. The robot is then able to turn at a specific angle using the gyro sensor to eventually start following the black line again.

When encountering the white tape, the robot will start spinning in circles while playing a sound, to signify that the robot has fully completed the track.

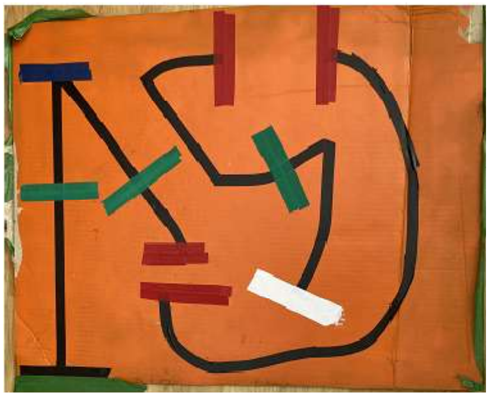

Figure 1: The track that the robot followed

Software Design and Implementation

Dividing the Task

When dividing our program into smaller parts, we segmented different parts of code that would be needed and continued to segment finer details of these parts of code until each segment was a small aspect of the overall program.

We decided to take this approach because of the complexity of the problem which we were trying to solve. The program had many different elements to it, which would have made it difficult to single out specific, small tasks that would eventually make up the entire program.

We first divided the program into two smaller segments, the code for shooting the balls, and the code for following the line.

The code for following the line was then broken down into what happens when the robot is on the line, what happens when the robot is off the line, and what happens when the robot is on a patch of colored tape.

When the robot is on the line, it simply drives forward. When the robot is off the line, it turns left and right until it finds the line again. When on a patch of color, the robot did various things depending on which colour it detected. This part of code was broken down into even more smaller sections, each color needing a section of its own.

The code for shooting the balls was broken down into smaller parts as well. Firstly, the code that detected that the robot needed to launch a ball, then the code that aligned the robot with the target, then the code which shot the ball, and finally the code that realigned the robot with the track.

Task List

Table 1: Task list

Task	Changes Made
Start up, wait 5 seconds, begin driving	This was changed to not include the 5 second wait time during testing, as the start up time was very tedious while running multiple tests one after another. However, in the demonstration the five second wait

	time was included. Originally, this task also waited for the bumper to be pressed before the robot started driving, however this was taken out since it did not serve a purpose.
Follow the line path	Although this task was optimized and altered slightly throughout the course of the project, the nature of the task always stayed the same. Changes included altering the motor speed for more accurate measurements and using experimentally determined values to help the robot turn at various points throughout the track.
Detect various colours and perform the associated action	No changes were made to this task. The actions performed stayed the same throughout the course of the project. Only slight optimizations were made for when the robot had to turn at a steep angle to stay on the track. We altered the program to use the RGB colour values to minimize false reading and increase consistency.
Shoot targets	No changes were made to this task.
Reach the end zone	No changes were made to this task.
Perform a victory dance in the end zone	No changes were made to this task.
Stop the program if the bumper is pressed	Although no changes were made to this task, it was implemented into more parts of the code as time went on. This is because we wanted to be able to stop the program during testing, and it is also a good way to ensure that the robot does not break any hardware if it unexpectedly collided with an object.

Functions

All functions below in the table that use boolean return values return whether the code ran without the interruption of the touch sensor. This was so the program could end should an interruption have been detected.

Table 2: List of functions

Function name	Parameters	Return type	Description	Written by:
getAbsoluteAngle	N/A	int	The purpose of this function is to determine what direction the robot is facing, in terms of an angle between 0 and 360.	Connor Hughes
getAbsoluteAngle	(int angle)	int	The purpose of this function is to determine the equivalent angle of in the input between 0 and 360.	Connor Hughes
turnToAngle	(int angle, int power, int & index)	bool	This function uses the getAbsoluteAngle function to turn and face the new desired direction.	Dimitrios Christopoulos

turnToAnlge	(int angle, int power)	bool	This function is the same as the other turnToAngle, but does not reference the index so it could be better used inside other functions	Connor Hughes
driveStraight	(int timeInMsec, int power, int & index)	bool	This function uses a timer to drive forward a specified amount. The specified amount comes from a 2d array which will be covered in the “Data Storing” section.	Dimitrios Christopoulos
shoot	(int angle, int power, int shooterPower, int & index)	bool	The shoot function is responsible for making the robot turn to a specified angle, drive to a fixed distance from the wall, shoot the ball into the target, and then realign itself back onto the track.	Connor Hughes
runShooter	(int power, int dir)	bool	This function was designed to run the shooter motor specifically, and use the direction variable to invert motion when needed.	Connor Hughes
calibrateSensors	N/A	void	This function calibrates all the sensors that are used in the program.	Sehaj Cheema
followLine	(int power)	void	The follow line function is responsible for ensuring that the robot is following the designated path, and that the robot can find its way back onto the path if it goes off it.	Sehaj Cheema
victoryDance	N/A	void	The victory dance function makes the robot spin in circles and play a sound until the front bumper is pressed, which ends the program.	Owen Henry

The program was meant to be ran in a loop, with various conditions to trigger during the process of following the line that would call the other fuctions at their matching colour code, as seen in the figure below. This way, the program was dynamic and would not require the hard coding of all the actions the robot would take. A slight chagne to the data array and a new motion could be added or removed easily.

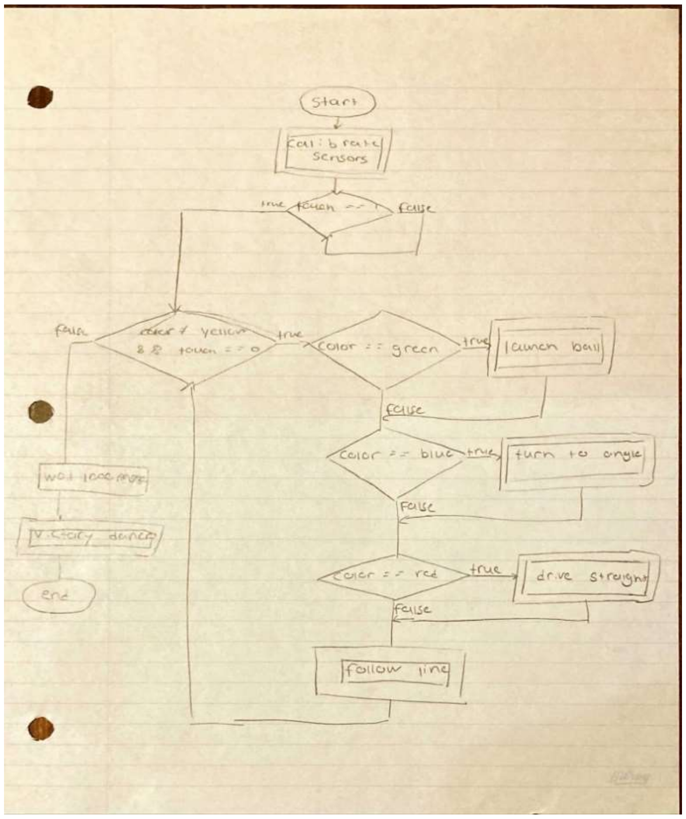

Figure 2: A flow chart of the overall program flow

Data Storing

For our program, we used regular data storing methods with the addition of a 2d array. The “regular methods” we used were standard variables such as integers, floats, strings and booleans. These were used as both parameters and return types for our functions.

Additionally, we used a 2d array to store pre-determined values which we used to ensure that the robot was able to realign itself when needed. This included when the robot reached the strips of blue tape (which meant it needed to make a steep turn), and with the shoot function (to realign itself with the target), and before and after completing various tasks after reaching strips of colored tape.

A 2d array was needed because there were multiple data points associated with a single action. In other words, if the robot reached a point where it needed to turn and move a given distance, there were multiple data points which were needed to do this. This led us to using a 2d array for all these angles and drive times.

Finally, we used a pass by reference for the variable which we used to index our 2d array. This was important because it allowed us to increment the index after using it, which helped us ensure that the value of the index was correct.

Design Decisions and Trade-offs

One important design decision was to 3d print a part that would hold the launcher. We decided to do this because we needed to provide stability and consistency to the launcher. The 3d printed part was designed to hold everything together, so that the launcher parts wouldn't be pulled down by gravity or be moved while the robot was in motion.

This 3d printed part ensured that the launcher could be used across many days of testing with consistency. This was important because it meant that our calibrations for one day were accurate and could be used in the following days.

The 3d printed part also protected the mechanical design from being damaged during operation. This was a concern due to the speeds at which the robot would be turning, which could have caused the parts to fall off the robot. Additionally, the robot was programmed to stop if it ever unexpectedly encountered a wall. This was to ensure that the robot would not continue to drive into the wall, potentially damaging the robot.

Another important design decision was the use of raw RGB values when working with the color sensor. This led to us using ranges of values associated with the raw values we got from the color sensor, which were not as accurate as we would like. The reason why we chose to use raw RGB values was because the reflected mode of the color sensor detected colors to be the same, when they should have been different.

After realizing this flaw in our design, we decided to switch to using raw RGB values and ranges, as previously mentioned. This came with a significant trade-off, as the color sensor's raw RGB values were inconsistent as it allowed for a mixing of colours which the robot would sometimes think was a completely different colour. This led to the robot detecting the wrong colors, which threw off our robot and caused many unfixable errors.

Another trade-off was associated with the use of a 3d printed part. This meant that a significant time investment was needed to design and implement the part. However, the benefits far outweighed the time investment, as the part provided consistency for our design.

Testing

Testing our program was made easy by the nature of our design idea. Since the point of our robot was to follow a designated track and shoot designated targets, we were able to meaningfully test our program with the track and targets that we made.

During testing, we were able to identify any errors, as we had a very clear idea of what the robot was supposed to do, and we could easily compare it to what it was doing during testing. We ensured that the robot was following the track and hitting the targets as needed, and if it wasn't, we were able to calibrate our angles and driving distances to ensure that the track was followed and that the targets were hit.

One useful addition to our program which helped us during testing was the addition of a test program. This program made the robot perform many of the tasks which it needed to perform, and we could pinpoint what functions were causing errors. For example, if the robot was supposed to realign itself onto the track, we could use the test function to see where it was going wrong and tweak the values as needed. This could be done for all our functions, which helped us pinpoint the source of errors during testing.

Another useful implementation during testing was printing out the values that were obtained from the color sensor. During testing, our robot was reading in color values which were different from what we expected them to be, and it took a while to realize that the color sensor itself was the source of error.

After realizing the source of the problem, our decision to print out the color values helped us determine what errors were coming from what aspects of the design.

For example, if the robot was reading incorrect color values, we could deal with that problem by altering the ranges we were using or by altering the physical design of the track. If the robot was reading in the correct color values, then we knew that the error was caused by the software design.

Significant Problems and Resolutions

The first significant problem we encountered was that the colour sensor was not able to determine the differences in colour values between green and blue, black and blue, and the colour white wouldn't register within the code. This is where our program started performing tasks such as shooting when sensing the blue line when it is only supposed to shoot when sensing the green line. It would follow the blue line and lead the robot into a constant loop where it would spin non-stop, which was an obvious problem within our colour sensor.

Firstly, we tested the code again using a different colour sensor as we thought there may have been a mechanical problem within our specific colour sensor. After testing, we realized that there was no difference as the colour sensor was still misreading the colour values. This is where we used exact RGB values and ranges (RGB values and ranges are shown at the beginning of the code in Appendix A). When using these specific values instead of the default colour value for each colour, our robot was able to identify the changes within the colour and perform the correct task for each colour.

The second significant problem we encountered was that the third motor wasn't able to successfully launch the ball. When the axle pushed the ball it would go little to no distance, which introduced a very complex problem.

This is where we designed an additional 3D part to help resolve the problem. The blue arrow points at the 3D part and the red arrow points at the ball launcher. The additional motor attached onto the 3D part used encoders to spin the gear. The gear is connected to a worm gear at the bottom which pushes the rack gear forward to successfully launch the ball. Additionally, testing was required to determine the exact value for the motor encoder, in order to successfully launch the ball through the target.

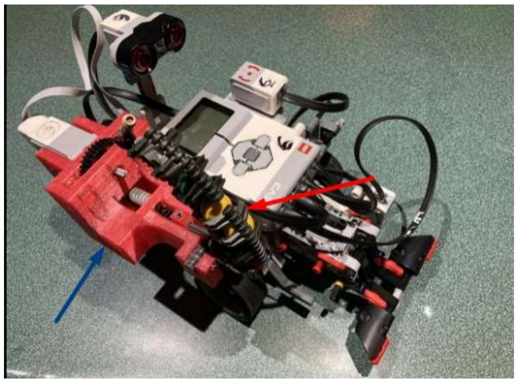
Figure 3: An image of the completed robot

The third significant problem we encountered was playing a sound through the robot. We tried converting many files to try to make robot play sounds, but we encountered many errors and realized that playing a specific song would be nearly impossible.

This is where we decided to compromise by using the function "playTone" (as shown in the victoryDance function in Appendix A). As we still wanted our robot to play some type of sound notifying us that it has successfully completed the track, we used "playTone" to create a beat sounding like an alarm. Therefore, this is how we resolved the song playing issue by implementing a function to play a sound using "playTone".

Conclusions and Recommendations

Our initial goal of completing the track and overcoming many obstacles such as taking a sharp turn, shooting through a hole, driving straight when there's no line to follow and doing the victory dance was achieved. Our group collaboratively worked on the code by creating flowcharts and splitting tasks amongst ourselves.

Specific RGB values and ranges were used within the code to help the colour sensor distinguish differences between different colours as the default values led to the colour sensor misreading values and performing the incorrect tasks. Our functions included within the code were used to calibrate sensors, get angle values, turn to a specific angle, drive straight, follow the black line, shoot the launcher, and finally perform the victory dance.

Our most significant problem was encountered when trying to shoot the balls through the target as the balls would simply not project forward and get stuck within the launcher with the other balls. This is where we collaboratively came up with a solution by using a gear which is attached to a worm gear, which is then able to push the rack gear forward and launch the shooter. We designed a 3D part as discussed earlier, which was able to hold the gears, axles, and the additional motor.

After completing the code and launcher portion, we ran through many tests. As a group, we realized that we had some mechanical and software issues specifically related to the colour sensing. We went through many potential solutions and finally solved the issue by using exact RGB values.

If our group had more time for this project, we would've used different colours for the track and a larger board. Using different colours could've prevented us from encountering the colour sensing issue and having a larger board would allow the robot to complete more tasks instead of cramming all the tasks into a smaller board. Additionally, we would've added a point system for the launcher system. This would create a greater challenge for us as a group to get the launcher to be very accurate and precise. Later, this could have been formed into a game where you can compete against your friends.

If this project was used in industry, we would change the turning angle portion of the software because the robot tends to spin around a few times before finding the exact angle value to shoot at. This can be implemented by having a list of exact angles to rotate at and resetting the gyro sensor every time it has completed a task. Testing would also be required here to determine the exact angle of the targets. By implementing this change into our code, our robot can be more efficient by completing the track faster and shooting through each target successfully instead of waiting for the robot to rotate a few times to find the specific angle.

Appendices

Appendix A:

```
//Sensor constants

const tSensors COLOR = S1;

const tSensors SONIC = S2;

const tSensors GYRO = S3;

const tSensors TOUCH = S4;


//Function prototypes

void calibrateSensors();

int getAbsoluteAngle();

int getAbsoluteAngle(int angle);

bool turnToAngle(int angle, int power, int & index);

bool turnToAngle(int angle, int power);

bool driveStraight(int time, int angle, int power, int & index);

void followLine(int power);

bool shoot(int angle, int power, int shootPower, int & index);

void victoryDance();



//RGB values and ranges

      //11-15

const int SHOOT_R = 15;

      //40-46

const int SHOOT_G = 50;

      //40-50

const int SHOOT_B = 55;


/* unused colors
```

```
      //10-13
const int TURN_R = 12;
      //41-49
const int TURN_G = 45;
*/
      //20-25
const int TURN_B = 30;

      //87-112
const int DRIVE_R = 110;
      //14-21
const int DRIVE_G = 18;
      //12-17
const int DRIVE_B = 18;

      //17-115
const int LINE_R = 30;
      //20-29
const int LINE_G = 30;
      //20-30
const int LINE_B = 30;

      //160-175
const TLegoColors FINISH = colorWhite;

      //100-130
const int BACKGROUND_R = 100;

/* unused colors
      //20-37
const int BACKGROUND_G = 28;
```

```
	//25-45
const int BACKGROUND_B = 40;
*/



//Movement constants
const int SHOOT_DIST = 60;
const int SONIC_TOL = 5;
const int SHOT_LG = 790;
const int ANGLE_TOL = 2;
const int FWD = 1;
const int BACK = -1;
const int TURN_SPEED_FACTOR = 2;

const int DATA_LG = 7;
const int DATA_ROW = 2;

//Sets all sensors to the correct ports and modes
void calibrateSensors()
{
	SensorType[COLOR]=sensorEV3_Color;
	wait1Msec(50);

	SensorMode[COLOR]=modeEV3Color_Color;

	wait1Msec(50);
	SensorType[SONIC]=sensorEV3_Ultrasonic;
	wait1Msec(50);
	SensorType[GYRO]=sensorEV3_Gyro;
	wait1Msec(50);
	SensorMode[GYRO]=modeEV3Gyro_Calibration;
	wait1Msec(50);

	SensorMode[GYRO]=modeEV3Gyro_RateAndAngle;

	wait1Msec(50);

	SensorType[TOUCH]=sensorEV3_Touch;

	wait1Msec(50);

}


//Takes in the gryo degrees and returns that angle in terms of 1 to 360 
degrees

int getAbsoluteAngle()

{

	int angle = getGyroDegrees(GYRO);


	angle %= 360;


	if(angle < 0)

		angle += 360;


	return angle;

}


//Takes a given angle and returns that angle in terms of 1 to 360 degrees

int getAbsoluteAngle(int angle)

{

	angle %= 360;


	if(angle < 0)

		angle += 360;


	return angle;

}
```

```
//Given an angle, a motor power and the index
//it will turn the robot to that angle at the given power
//and then increment the index
bool turnToAngle(int angle, int power, int & index)
{
      angle = getAbsoluteAngle(angle);

      motor[motorA] = -power;
      motor[motorD] = power;
      bool fail = false;
      while (abs(getAbsoluteAngle() - angle) > ANGLE_TOL && !fail)
      {
            if (SensorValue[TOUCH] == 1)
            {
                  fail = true;
            }
      }

      motor[motorA] = motor[motorD] = 0;
      if(!fail)
            index++;
      return !fail;
}

//Given an angle, a motor power and the index
//it will turn the robot to that angle at the given power
bool turnToAngle(int angle, int power)
{
      angle = getAbsoluteAngle(angle);
      bool fail = false;
```

```
        motor[motorA] = -power;
        motor[motorD] = power;
        while (abs(getAbsoluteAngle() - angle) > ANGLE_TOL && !fail)
        {
                if (SensorValue[TOUCH] == 1)
                {
                        fail = true;
                }
        }

        motor[motorA] = motor[motorD] = 0;

        return !fail;
}

//Given a time angle, power and index the robot will turn to the
//given angle, drive for the given time in miliseconds at the given power
//and then increment the index
bool driveStraight(int time, int angle , int power, int & index)
{
        angle = getAbsoluteAngle(angle);
        bool fail = false;

        if(time < 0)
                fail = true;

        if(!fail)
        {
                turnToAngle(angle, power / TURN_SPEED_FACTOR);

                motor[motorA] = motor[motorD] = power;
```

```
            clearTimer(T1);

            while (time1[T1] < time && !fail)
            {
                  if (SensorValue[TOUCH] == 1)
                        fail = true;
            }

      motor[motorA] = motor[motorD] = 0;
      turnToAngle(angle, power / TURN_SPEED_FACTOR);
      if(!fail)
            index++;
      }
      return !fail;
}

//Given the motor power and direction
//the function will drive the shooter
//motor at the given power either
//forward or backwards
bool runShooter(int power, int dir)
{
      nMotorEncoder[motorB] = 0;
      motor[motorB] = -power * dir;
      bool fail = false;

      while(abs(nMotorEncoder[motorB]) < SHOT_LG && !fail)
      {
            if (SensorValue[TOUCH] == 1)
                  fail = true;
```

```
		}
		motor[motorB] = 0;
		return !fail;
	}

	//Given the target angle, power, shooter power and index
	//the function will record its start position, turn to
	//the given angle, move to the set distance, fire and then
	//return to the track, and increment the index
	bool shoot(int angle, int power, int shootPower, int & index)
	{
		angle = getAbsoluteAngle(angle);
		bool fail = false;
		bool reverse = false;
		int startAng = getAbsoluteAngle();
		displayString(5, "%d", angle);

		turnToAngle(angle, power / TURN_SPEED_FACTOR);

		nMotorEncoder[motorA] = nMotorEncoder[motorD] = 0;

		playTone(400, 15);
		wait10Msec(15);

		if(SensorValue(SONIC) < SHOOT_DIST)
		{
			power *= -1;
			reverse = true;
		}

		motor[motorA] = motor[motorD] = power;
```

```
while(abs(SensorValue(SONIC) - SHOOT_DIST) > SONIC_TOL && !fail)
{
      displayString(4, "%d : %d : %d", SensorValue(SONIC), SHOOT_DIST,
      SensorValue(SONIC) - SHOOT_DIST);
      if (SensorValue[TOUCH] == 1)
            fail = true;

}

motor[motorA] = motor[motorD] = 0;

int dist = nMotorEncoder[motorA];

if(!runShooter(shootPower, FWD))
      return false;

if(!runShooter(shootPower, BACK))
      return false;

turnToAngle(getGyroDegrees(GYRO) + 180, power / TURN_SPEED_FACTOR);

nMotorEncoder[motorA] = 0;
if(!reverse)
{
      motor[motorA] = motor[motorD] -power;
      while(nMotorEncoder[motorA] < dist && !fail)
      {
            if (SensorValue[TOUCH] == 1)
                  fail = true;

      }
```

```
	} else
	{
		playTone(400, 15);
		wait10Msec(15);
		motor[motorA] = motor[motorD] = power;
		while(nMotorEncoder[motorA] > dist - 5 && !fail)
		{
			if (SensorValue[TOUCH] == 1)
				fail = true;

		}
	}

	turnToAngle(startAng, power / TURN_SPEED_FACTOR);
	if(!fail)
		index++;
	return !fail;
}

//Given the motor power, the function
//will decide whether to move left or right
//based on whether the line boundry is detected
void followLine(int power)
{

	int red = 0, blue = 0, green = 0;
	getColorRawRGB(COLOR, red, blue, green);
		if (red < BACKGROUND_R && blue < LINE_B)
		{
			motor[motorA]= 0;
			motor[motorD]= power;
```

```
            }
            else
            {
                  motor[motorA] = power;
                  motor[motorD] = 0;
    }

    wait1Msec(220);

    motor[motorA] = motor[motorD] = 0;
}

//The robot will play a repeating sound
//then generate a sound of increase pitch
//and spin untill the bumper is pressed
void victoryDance()
{
      bool fail = false;

      for(int x = 0; x < 10 && !fail; x++)
      {
            if(SensorValue(TOUCH) == 1)
                        fail = true;

            playTone(250, 15);
            wait10Msec(16);

            playTone(500, 20);
            wait10Msec(21);
      }
```

```
        for(int x = 0; x <20000 && !fail; x += 4)
        {
                playTone(x, 1);
                motor[motorA] = -100;
                motor[motorD] = 100;
                wait10Msec(1);
                if(SensorValue[TOUCH] == 1)
                        fail = true;
        }
        motor[motorA] = motor[motorD] = 0;
}

int const power = 20;

task main()
{
        calibrateSensors();

        int moveData[DATA_ROW][DATA_LG] = {{270, 145, 325, 170, 270, 150, 45},
                                         {1,2,3,800,5000, 6, 7}};

        int index = 0;
        int shooterPower = 100;

        bool fail = false;
        while(!SensorValue(TOUCH) &&
                        SensorValue(COLOR) != (int)FINISH && !fail && index <
10)
        {
                int red= 0 , blue = 0, green = 0;
```

```
            followLine(power);

            getColorRawRGB(COLOR,red,blue,green);
            displayString(3, "%d          %d          %d", red, green, blue);



            if(blue < SHOOT_B && red < SHOOT_R && green < SHOOT_G && blue >
TURN_B)
            {
                  if(index == 2)
                        turnToAngle(135, power);

                  motor[motorA] = motor[motorD] = 20;
                  wait1Msec(750);
                  motor[motorA] = motor[motorD] = 0;

                  displayString(1, "SHOOT GREEN");
                  if(!shoot(moveData[0][index], power, shooterPower, index))
                        fail = true;



            } else

            if(red < LINE_R && blue < TURN_B && green > LINE_G)
            {

                  displayString(1, "TURN BLUE");
                  if(!turnToAngle(moveData[0][index], power /
TURN_SPEED_FACTOR, index))
                        fail = true;

                  motor[motorA] = motor[motorD] = -20;
```

```
                    wait1Msec(500);
                    motor[motorA] = motor[motorD] = 0;

              } else

        if(blue < DRIVE_B && green < DRIVE_G && red < DRIVE_R && red > LINE_R)
        {
              motor[motorA] = motor[motorD] = 20;
              wait1Msec(500);
              motor[motorA] = motor[motorD] = 0;

              displayString(1, "DRIVE RED %d", index);
              if(!driveStraight(moveData[1][index], moveData[0][index], power,
index))
                    fail = true;

        }

              eraseDisplay();

        }

        victoryDance();
}
```

Appendix B:

```
//Sensor constants
const tSensors COLOR = S1;
const tSensors SONIC = S2;
const tSensors GYRO = S3;
const tSensors TOUCH = S4;
```

```
//Function prototypes
void calibrateSensors();
int getAbsoluteAngle();
int getAbsoluteAngle(int angle);
bool turnToAngle(int angle, int power, int & index);
bool turnToAngle(int angle, int power);
bool driveStraight(int time, int angle, int power, int & index);
void followLine(int power);
bool shoot(int angle, int power, int shootPower, int & index);
void victoryDance();

//Movement constants
const int SHOOT_DIST = 60;
const int SONIC_TOL = 5;
const int SHOT_LG = 790;
const int ANGLE_TOL = 2;
const int FWD = 1;
const int BACK = -1;
const int TURN_SPEED_FACTOR = 2;

//20-30
const int LINE_B = 30;
      //100-130
const int BACKGROUND_R = 100;

//Sets all sensors to the correct ports and modes
void calibrateSensors()
{
      SensorType[COLOR]=sensorEV3_Color;
      wait1Msec(50);
```

```
	SensorMode[COLOR]=modeEV3Color_Color;


	wait1Msec(50);
	SensorType[SONIC]=sensorEV3_Ultrasonic;
	wait1Msec(50);
	SensorType[GYRO]=sensorEV3_Gyro;
	wait1Msec(50);
	SensorMode[GYRO]=modeEV3Gyro_Calibration;
	wait1Msec(50);
	SensorMode[GYRO]=modeEV3Gyro_RateAndAngle;
	wait1Msec(50);
	SensorType[TOUCH]=sensorEV3_Touch;
	wait1Msec(50);
}


//Takes in the gryo degrees and returns that angle in terms of 1 to 360
degrees
int getAbsoluteAngle()
{
	int angle = getGyroDegrees(GYRO);


	angle %= 360;


	if(angle < 0)
		angle += 360;


	return angle;
}


//Takes a given angle and returns that angle in terms of 1 to 360 degrees
```

```
int getAbsoluteAngle(int angle)
{
      angle %= 360;

      if(angle < 0)
            angle += 360;

      return angle;
}

//Given an angle, a motor power and the index
//it will turn the robot to that angle at the given power
//and then increment the index
bool turnToAngle(int angle, int power, int & index)
{
      angle = getAbsoluteAngle(angle);

      motor[motorA] = -power;
      motor[motorD] = power;
      bool fail = false;
      while (abs(getAbsoluteAngle() - angle) > ANGLE_TOL && !fail)
      {
            if (SensorValue[TOUCH] == 1)
            {
                  fail = true;
            }
      }

      motor[motorA] = motor[motorD] = 0;
      if(!fail)
            index++;
```

```
	return !fail;
}

//Given an angle, a motor power and the index
//it will turn the robot to that angle at the given power
bool turnToAngle(int angle, int power)
{
	angle = getAbsoluteAngle(angle);
	bool fail = false;
	motor[motorA] = -power;
	motor[motorD] = power;
	while (abs(getAbsoluteAngle() - angle) > ANGLE_TOL && !fail)
	{
		if (SensorValue[TOUCH] == 1)
		{
			fail = true;
		}
	}

	motor[motorA] = motor[motorD] = 0;

	return !fail;
}

//Given a time angle, power and index the robot will turn to the
//given angle, drive for the given time in miliseconds at the given power
//and then increment the index
bool driveStraight(int time, int angle , int power, int & index)
{
	angle = getAbsoluteAngle(angle);
	bool fail = false;
```

```
    if(time < 0)
        fail = true;

    if(!fail)
    {
        turnToAngle(angle, power / TURN_SPEED_FACTOR);

        motor[motorA] = motor[motorD] = power;

        clearTimer(T1);

        while (time1[T1] < time && !fail)
        {
            if (SensorValue[TOUCH] == 1)
                fail = true;
        }

    motor[motorA] = motor[motorD] = 0;
    turnToAngle(angle, power / TURN_SPEED_FACTOR);
    if(!fail)
        index++;
    }
    return !fail;
}

//Given the motor power and direction
//the function will drive the shooter
//motor at the given power either
//forward or backwards
bool runShooter(int power, int dir)
```

```
{
      nMotorEncoder[motorB] = 0;
      motor[motorB] = -power * dir;
      bool fail = false;

      while(abs(nMotorEncoder[motorB]) < SHOT_LG && !fail)
      {
            if (SensorValue[TOUCH] == 1)
                  fail = true;
      }
      motor[motorB] = 0;
      return !fail;
}

//Given the target angle, power, shooter power and index
//the function will record its start position, turn to
//the given angle, move to the set distance, fire and then
//return to the track, and increment the index
bool shoot(int angle, int power, int shootPower, int & index)
{
      angle = getAbsoluteAngle(angle);
      bool fail = false;
      bool reverse = false;
      int startAng = getAbsoluteAngle();
      displayString(5, "%d", angle);

      turnToAngle(angle, power / TURN_SPEED_FACTOR);

      nMotorEncoder[motorA] = nMotorEncoder[motorD] = 0;

      playTone(400, 15);
```

```
wait10Msec(15);

if(SensorValue(SONIC) < SHOOT_DIST)
{
      power *= -1;
      reverse = true;
}

motor[motorA] = motor[motorD] = power;
while(abs(SensorValue(SONIC) - SHOOT_DIST) > SONIC_TOL && !fail)
{
      displayString(4, "%d : %d : %d", SensorValue(SONIC), SHOOT_DIST,
      SensorValue(SONIC) - SHOOT_DIST);
      if (SensorValue[TOUCH] == 1)
            fail = true;

}

motor[motorA] = motor[motorD] = 0;

int dist = nMotorEncoder[motorA];

if(!runShooter(shootPower, FWD))
      return false;

if(!runShooter(shootPower, BACK))
      return false;

turnToAngle(getGyroDegrees(GYRO) + 180, power / TURN_SPEED_FACTOR);

nMotorEncoder[motorA] = 0;
```

```
    if(!reverse)
    {
        motor[motorA] = motor[motorD] -power;
        while(nMotorEncoder[motorA] < dist && !fail)
        {
            if (SensorValue[TOUCH] == 1)
                fail = true;

        }
    } else
    {
        playTone(400, 15);
        wait10Msec(15);
        motor[motorA] = motor[motorD] = power;
        while(nMotorEncoder[motorA] > dist - 5 && !fail)
        {
            if (SensorValue[TOUCH] == 1)
                fail = true;

        }
    }

    turnToAngle(startAng, power / TURN_SPEED_FACTOR);
    if(!fail)
        index++;
    return !fail;
}

//Given the motor power, the function
//will decide whether to move left or right
//based on whether the line boundry is detected
```

```
void followLine(int power)
{

      int red = 0, blue = 0, green = 0;
      getColorRawRGB(COLOR, red, blue, green);
            if (red < BACKGROUND_R && blue < LINE_B)
            {
                  motor[motorA]= 0;
                  motor[motorD]= power;
            }
            else
            {
                  motor[motorA] = power;
                  motor[motorD] = 0;
    }

    wait1Msec(220);

    motor[motorA] = motor[motorD] = 0;
}

//The robot will play a repeating sound
//then generate a sound of increase pitch
//and spin untill the bumper is pressed
void victoryDance()
{
      bool fail = false;

      for(int x = 0; x < 10 && !fail; x++)
      {
            if(SensorValue(TOUCH) == 1)
```

```
                        fail = true;

                playTone(250, 15);
                wait10Msec(16);

                playTone(500, 20);
                wait10Msec(21);
        }

        for(int x = 0; x <20000 && !fail; x += 4)
        {
                playTone(x, 1);
                motor[motorA] = -100;
                motor[motorD] = 100;
                wait10Msec(1);
                if(SensorValue[TOUCH] == 1)
                        fail = true;
        }
        motor[motorA] = motor[motorD] = 0;
}

task main()
{
        calibrateSensors();

        int index = 0;
        while(SensorValue(TOUCH) == 0)
        {}

        while(SensorValue(TOUCH) == 1)
        {}
```

```
time1[T1] = 0;
displayString(1, "Following the line");
while(time1[T1] < 20000)
{
followLine(20);
}
motor[motorA] = motor[motorD] = 0;
displayString(1, "Turn to some angles");
while(getButtonPress(buttonEnter) != 0)
{}
while(getButtonPress(buttonEnter) == 0)
{}
turnToAngle(90, 10);
      wait1Msec(1000);
turnToAngle(180, 10);
      wait1Msec(1000);
turnToAngle(-90, 10);

displayString(1, "Drive in some straight lines");
while(getButtonPress(buttonEnter) != 0)
{}
while(getButtonPress(buttonEnter) == 0)
{}
driveStraight(2000, 90, 10, index);
      wait1Msec(1000);
driveStraight(2000, 270, 10, index);

displayString(1, "Shoot at some targets for practice");
while(getButtonPress(buttonEnter) != 0)
{}
```

```
    while(getButtonPress(buttonEnter) == 0)
    {}
    wait1Msec(2000);
    shoot(0, 20, 100, index);
        wait1Msec(1000);
    shoot(90, 20, 100, index);
        wait1Msec(1000);
    shoot(180, 20, 100, index);
        wait1Msec(1000);
    shoot(270, 20, 100, index);
        wait1Msec(1000);

    victoryDance();
}
```