UNIVERSITY OF

Waterloo

Department of Mechanical and Mechatronics Engineering

MTE 121 Final Report

Prepared By Group 1:
Chelsea Dmytryk 20932778
Dan Huynh 20896965
Josh Blayone 20968135
Sung-En Huang 20932488

MTE 121 Stream 4 (Tuesday 8:30am)
24th November 2021

Table of Contents

List of Tables

Table of Figures

Introduction
The goal of our project was to program the Lego EV3 robot to drive until an open parking space is detected, then perform a parallel park in that spot. Once the robot is turned on, it will be on standby until the “car key” triggers the program to start. Once the robot has started driving, a 30-second timer will be running in the background which will shut down the robot if no open spot is found within that time frame. The robot should be checking for cars parked to the right of it and the distance between them. If the space between two cars is sufficient for the robot to park, it should line itself up with the car at the front of the parking space then manoeuvre itself into the spot. The robot should be rotating, reversing, straightening itself out, then moving forward if it is too close to the car behind it. If the robot does a parallel park, it should raise the banner upon completion.

Software Design and Implementation
When creating the main function and other non-trivial functions, reference was made to a flowchart (Appendix B) which outlined the logical aspects of the robots’ decision-making processes in an abstracted and high-level manner.

Task List
Table 1: A table outlining the list for testing the robot's functionality

Function Being Tested	**How It Will Be Tested**	**Updated Changes from Previous List**
findParkingSpaceAndPark	Test if the robot accurately detects parking spots with sufficient space	N/A
waitForGreenCard	Test different coloured cards Test on different surfaces to ensure no false detection	N/A
searchForCar	Testing a variety of distances from the “sidewalk” Scenario when there are no cars to search for Scenario when the cars are consecutively lined up Test when it is sufficient and lack of space to park	Added the additional scenario where cars were lined up consecutively
parallelPark	Test that inner functions successfully call it Test if robot will parallel park in space when given sufficient space	Additionally tested if the robot would parallel park when given a big parking spot
parkingLineUp	Assess if the robot aligns itself in front of the spot to allow for sufficient backing space Test for wiggle room when fully parked	N/A

reverseIntoSpot	Assess if robot parks without contacting other vehicles Assure that parallelPark function is called after completing the reverse	N/A
pullForward	Test if the robot pulls forward when reversing too close to the rear car Assess if the safety distance is sufficient and triggers this function	New function created to prevent backing incidences and sufficient space for the user to drive away
toggleBannger	Test if the function raises the banner only if the robot successfully parked Assess if the banner hits outside obstacles when being raised Test if the function drops the banner when completed the raise	Added the drop banner command when the banner is successfully raised

Non-Trivial Functions

Table 2: A table describing our non-trivial functions

Function Name	**Parameters**	**Return Type**	**Author**
findParkingSpaceAndPark	N/A	bool	Josh Blayone
waitForGreenCard	N/A	void	Chelsea Dmytryk
searchForCar	Bool lookingForCar	void	Collaboration
parallelPark	N/A	void	Josh Blayone
parkingLineUp	N/A	void	Dan Huynh
reverseIntoSpot	Float distanceToCar	void	Chelsea Dmytryk
pullForward	N/A	void	Josh Blayone
toggleBannger	Bool isRaised	void	Sam Huang / Dan

Descriptions

findParkingSpaceAndPark

The primary objective of “findParkingSpaceAndPark” is to locate a suitable parking space for the robot under several constraints. Once a parking space is found, the “parallelPark” function is called which satisfies its secondary objective. This function initiates by declaring several variables that contain; the gap distance between two cars (float gapDistance), the number of failed parallel parking attempts (int carCount), and a boolean that determines if the parallel parking attempt was successful (bool finished).

A constant integer is also declared which helps convert angular encoder measurements to centimetres (const int encToCM).

A while loop is then initialized which loops if the number of parallel parking attempts is less than the predefined constant of max attempts (Appendix B, step 4) and if “finished” has a value of “false”. The motors are then turned on and the “searchForCar” function is called with the parameter “true” to look for the first car of which the robot will parallel park in front (Appendix B, step 6). Once the first car is located, the “searchForCar” function is called with the parameter “false” such that an empty space may be searched for (Appendix B, step 8). The motor encoders are then reset such that the gap distance between the two vehicles may be measured and the “searchForCar” function is called once more with the parameter “true” (Appendix B, step 10). Once the second car is sensed, the motors are turned off and the gap distance is set equal to the absolute value of the encoder value of motorA multiplied by “encToCM” (Appendix B, step 11).

A number of end conditions are then checked. If gapDistance is greater than the predefined constant “MINIMUM_GAP_NEEDED” (Appendix B, step 12), the parallelPark function is called and “finished” is set to “true” (Appendix B, step 16). If “TIMER” becomes greater than the constant “MAX_TIME_FOR_ATTEMPTS”, “finished” is returned with the value “false”. This exits the process of searching for a parking space if the robot “runs out of fuel” in this fictitious scenario without interfering with the maximum attempt counter. If both preceding statements are false, “carCount” is incremented and “finished” is returned with value “false”.

waitForGreenCard

This function waits for the presence of the green “key” before the robot begins to search for a parallel parking spot (Appendix B, step 2). A while loop loops infinitely whilst the colour green is not sensed by the colour sensor.

searchForCar

The “searchForCar” function is called to both actively look for a car, or to search for empty space by inverting the search condition. This function accepts a boolean parameter “lookingForCar”. If “lookingForCar” is true, a while loop is declared which loops until the ultrasonic sensor located on the side of the robot returns a value that is less than the constant integer “VALID_DISTANCE”. The value of “TIMER” must also be less than the “MAX_TIME_FOR_ATTEMPT” to stay within the while loop. Essentially, this function is receiving the value of the side ultrasonic sensor. Once the value of this particular sensor is less than the value “VALID_DISTANCE”, a car is found. If “lookingForCar” is false, a while loop is declared which loops until the side ultrasonic sensor returns a value that is greater than “VALID_DISTANCE. Once again, the value of “TIMER” must be less than the “MAX_TIME_FOR_ATTEMPT” to stay within the loop. If “lookingForCar” is false, the function effectively operates as the reverse of how it would operate if “lookingForCar” was true. That is to say, if “lookingForCar” is false, the function searches for empty space beyond the value of “VALID_DISTANCE” instead of a car. Once the value of the side ultrasonic sensor is greater than “VALID_DISTANCE”, empty space is found.

The value of “TIMER” is then checked. If the value of “TIMER” is greater than “MAX_TIME_FOR_ATTEMPT”, the motors are turned off (Appendix B, step 7).

parallelPark

“parallelPark” calls several functions (parkingLineUp, reverseIntoSpot, pullForward), to simulate parallel parking the vehicle (Appendix B, step 16).

parkingLineUp:

“parkingLineUp” is a function that drives the robot forwards a distance equal to one-third of the robot body (measured in centimetres) such that the robot is positioned to reverse into the parking spot when the “reverseIntoSpot” function is called. The function turns on the motors, resets the encoder of motorA and calls the “driveDistanceCM” function. The parameters passed to the “driveDistanceCM” function are the distance one desires to move the robot (one-third of the robot body), and the direction (forwards).

reverseIntoSpot

“reverseIntoSpot” is the function that facilitates the parallel parking action of the robot. This function takes a parameter “distanceToCar” which measures the distance between the adjacent side of the robot to the second car detected. The function first rotates the robot clockwise 45 degrees and drives the robot backwards a distance equal to the sum of the robot width and “distanceToCar” in centimetres, to position the robot between two cars. The robot will then rotate counterclockwise 45 degrees such that the robot will be positioned parallel to the cars it is parking between. The function then calls the “pullForward” function, which drives the robot forwards such that the robot is an appropriate distance away from the rear car.

pullForward

“pullForward” is called within the “reverseIntoSpot” function to position the robot an appropriate distance away from the rear car. This distance was determined to be 5 centimetres, which was given the constant “DISTANCE_FROM_CAR”. The function turns on the motors, and drives forwards until the rear ultrasonic sensor returns a value greater than “DISTANCE_FROM_CAR”. If the rear ultrasonic sensor already returns a value greater than “DISTANCE_FROM_CAR”, the robot will not move forwards as was already an appropriate distance away from the rear car. Finally, the function turns off the motors.

toggleBanner

“toggleBanner” is responsible for both raising the successful completion flag, as well as lowering the flag. The function accepts an integer parameter called “isRaised” which dictates whether “toggleBanner” raises or lowers the flag. The parameter “isRaised” can be a value of “1”, or “-1”. The “FLAG_MOTOR” encoder is reset, where “FLAG_MOTOR” is the defined term given to the motor attached to the flag shaft. Power is then provided to “FLAG_MOTOR”, with its value being the product of “isRaised” and the constant integer “BANNER_SPEED”. In this case, “BANNER_SPEED” is the constant integer determined to be the speed in which the banner is raised. By multiplying “isRaised” by “BANNER_SPEED”, both clockwise and counterclockwise rotations are able to be facilitated without using decisions. In both cases, “FLAG_MOTOR” is rotated until the absolute value of the encoder value of the “FLAG_MOTOR” is equal to the constant “BANNER_ANGLE” which was defined to be 90 degrees. By calling “toggleBanner” twice in succession, one can raise the flag and lower the flag consecutively by altering the value of the parameter of the function (Appendix B, section 17-18).

Trivial Functions

Table 3: A table describing our trivial functions

Function Name	**Parameters**	**Return Type**	**Description**
configureSensors	N/A	void	Would configure all the sensors on the robot and set them into the right mode
resetEncoder	tMotor selectedMotor	void	Resets the recorded encoder values of the selected motor
resetAllEncoders	N/A	void	Resets the recorded encoder values of all motors
getEncoder	tMotor selectedMotor	int	Returns the current amount of degrees the selected motor has turned.
getBackUltra	N/A	int	Returns the current distance to the closest object behind the robot.
getSideUltra	N/A	int	Returns the current distance to the closest object beside the robot.
getColor	N/A	int	Returns the current colour that the colour sensor can sense.
getGyroAngle	N/A	int	Returns the current angle of the robot in degrees.
moveRobot	int speed	void	Will move the robot with the desired speed. If the speed is positive the robot will move forwards, otherwise, it will move backwards.
stopRobot	N/A	void	Will stop all movement of the robot.
rotateRobot	int angle	void	Will rotate the robot to the desired angle.
driveDistanceCM	float distanceCM, int direction	void	Will move the robot by the desired distance measured in cm.

Tests

Table 4: A table outlining the test cases

Environment (Test Case)	**Desired Outcome**	**What Aspects Will Be Tested**
For every test	N/A	Waiting for Green Card Driving Functions Looking For First Car
One car (object) placed as a vehicle on the side of the road.	The robot will eventually timeout at 30 seconds since it is unable to detect the next vehicle.	Timeout conditions
Two cars (objects) placed as vehicles on the side of the road with enough space to park.	The robot parallel parks between the two cars.	Vehicle Gap Detection Parallel Parking Functions Raise Banner Function
Three cars (objects) placed as vehicles on the side of the road with the gap between the first and second car being too small, and the gap between the second and third being more than enough.	The robot will pass by the first spot and drive to the second spot where it will parallel park between them.	Confirming that the Gap between vehicles is too small Making sure the searching for car loop properly loops back
Four cars (objects) placed as vehicles on the side of the road with the gap between all the cars being too small to fit in between.	The robot will stop by the last car and display on the screen that it was unable to find a spot.	The maximum number of spots the robot attempts to detect (which is 3)

Data Storage

Data is organized using variables for both simplicity, and a lack of requirement for more complex data storage due to the scope of the project. Defined constants are also utilized to improve the readability of code. A series of functions were implemented for a greater level of abstraction as well as to further improve readability. By organizing data using functions, multiple individuals were able to work on the main function and other non-trivial functions without needing to consider the fine details with respect to the contents of each function.

Software Design Decisions

Function Design

There were three tiers of functions in our program, where the last tier is written with all the below it. The third tier of functions in our code were very easy to read as it was written at a higher level. For example, the findParkingSpaceAndPark function was high level in the sense that anytime something were to occur in it, it would be calling another function from the tier below, such as searchForCar. This made it very easy to read as these complex functions were broken down into steps that were clearly labeled with function names. In contrast, the first tier of functions all deal with interfacing with the hardware level of the robot, such as retrieving sensor information, or moving the robot.

Since we decided to structure our code in this format, the code that is contained in the main task is very simple and resembles a pseudocode like structure. It also makes the code easier to read and understand what it's doing when broken down this way.

Other Design Decisions

Our group made a variety of software design decisions when debugging and facing technical issues. Initially, we implemented a driveAway function which would be called when the robot successfully parallel parked and would prompt the robot to drive away to a certain distance and stop indicating the end of the program and a successful parallel park. Although the function prototype and the logic behind the function was already formulated, the robot was previously having issues completing a parallel park that would perfectly align its body with the robot in front and behind. This would result in an inconsistent angle that the robot would need to turn in order to successfully pull away from the spot, this issue would've been resolved with the gyro sensor but we lacked sufficient time to complete the main functions, so we dropped the idea of pulling away after a successful park.

Additionally, we tinkered with the reverse angle when the robot reversed into the spot. During multiple occurrences, the robot would either reverse at an angle too sharp and back into the front vehicle or it would reverse at an angle too flat that it would hit the rear vehicle when reversing. The finalized angle value was based on trial and error as the gyro value had some inaccuracies.

Full System test

Our full system test will test our start and shutdown program and the three different scenarios that may occur when trying to parallel park in a real-life situation. The three different scenarios include when there are no cars on the sidewalk, multiple parking spots with insufficient space, and multiple parking spots with sufficient space. Based on the scenarios, the robot should either continue driving, compare car and spot distance and continue driving if spot distance smaller than car distance, or parallel park when car length fits into spot. The robot should then shutdown in the three given scenarios, when the robot has spent a minute searching for spots, when the robot has located three parking spots that have insufficient space, and when the robot successfully parks.

Unexpected Occurrences and Issues Encountered

The most common issue that occurred when testing the robot was it having inconsistent motor speeds when the battery was low. The left and right motor would be set at the same speed but the robot would either travel in a non-linear direction or be rotating in place based on its battery life. Initially, we thought the lack of linear driving was due to our motor values being inconsistent, therefore we adjusted our motor values to accommodate for the "difference in speed". As expected, this solution did not work as adjusting the motor speed value by just one greatly affected the direction of the robot. Through experience, we realized this issue was simply resolved with a quick battery recharge and both motors would be back on track after being supplied sufficient power.

Another situation that we encountered was the robot choosing to parallel park even though the parking space was too small. We initially thought that the minimum parking space distance value we inputted was too low so we jacked the number up and tested but the robot still looked to park when there was insufficient space. While editing our code, we noticed that our encoder distance value did not account for the rotation of the wheel and simply assumed the distance value was the distance of the parking space. This would explain why the robot was constantly looking to park even when the parking space was too narrow.

Conclusion and Recommendations

In conclusion, this report outlined the software design and implementation of the Lego EV3 robot performing a parallel park. The problem at hand was for the robot to perform a park between two cars that it senses through the ultrasonic sensor. In addition, we included important, real-life situation features into our program. These include the use of a “car key” to start the program, and a timer that acts as the fuel tank. Once the timer runs out, this represents the car running out of fuel and the robot will end the program.   This feature also ensures that the program will not infinitely be looking for a parking space in the case that there are no cars parked on the side of the street. Therefore, after 30 seconds of searching, and no cars found to park between, the robot will shut down.

If more time was allotted for this project, the utilization of user input, better response to unexpected environments and use of more sensors could have been implemented.

Firstly, the use of button input could have been added as a form of feedback from the user to indicate whether the robot should park in the valid parking spot that is found or continue looking. For example, the robot may have found a parking spot that fits the conditions necessary for it to be able to park, however, the user may want to keep driving to find a spot closer to their destination. In this case, the user could press a specific button so that the robot will continue driving and look for another spot.

Secondly, at the moment, our robot is only programmed to advance until it is 10 cm from the back car to help center itself. Nonetheless, it would be preferred that the robot centers itself based on the size of the parking spot. By taking the length of the open spot and subtracting the length of the robot, the amount of empty space is given. Using this value, the program could divide it by two and have that value stored as the distance from the back of the robot to the front of the car behind it. The robot could then use the ultrasonic sensor at the back of the robot to position itself correctly.

Thirdly, an additional feature that could be implemented would be to either have the car always self-aligning to move straight or, to follow the yellow centerline on the pavement. The first option would assure that the robot will still move in a straight motion despite hardware issues such as having a slight tilt in a tire. The second option would allow the robot to still function in an environment with twisting roads as well as straight ones.

Lastly, if we had more time, we could have reworked the program to immediately check for open space beside it and track the distance before it reaches the first car. If there is sufficient space it will parallel park. This would allow the robot to park behind the first car it finds even when there is no car behind it. This way, the robot does not have to find the first car, then keep looking for a second car to park between. This would better simulate real-life situations and allow for the robot to react more efficiently in various environments.
When looking at the software of our program, there are a few ways we could make it more durable for industry. To make our software more robust or easier to maintain, we could improve the waitForGreenCard and moveRobot functions.

For our project, the waitForGreenCard function is used to start the program once it senses the colour green under the colour sensor. This function is configured to use the colour sensor in the colour mode. In colour mode, the LED lamp emits red, green, and blue light and uses the colour sensor to determine any one of seven colours (black, blue, green, yellow, red, white, brown) [1]. In this case, a multitude of shades of green could trigger the sensor. This may be disadvantageous since the robot could be on any coloured surface but should not start until the exact colour of the key is placed under the sensor. Therefore, if the robot is on a dark green surface, the robot should not detect the floor as the key. In this scenario, we could change the function to be testing the key colour using two colour sensors: one in colour mode, and one in reflected light intensity mode. Hence, we would be able to check for the correct colour as well as the intensity of it. Therefore, only the exact colour settings of a specific key will trigger the sensor and allow the robot to start driving.

As mentioned above, the implementation of a self-aligning drive function would assure the robot is more adaptable in various conditions and environments. Therefore, the moveRobot function could be redesigned to constantly check the gyro sensor values and change the motor speeds depending on the direction it is deviating. The moveRobot function would make the robot easier to maintain since it will be able to automatically handle minor deviations in its path due to an imbalance within the motors, for example.

References

[1] G. Turnbull, "Lego Mindstorms EV3: Color Sensor In Detail," Unknown, 17 September 2019. [Online]. Available: https://www.funcodeforkids.com/lego-mindstorms-ev3-color-sensor-deep-dive/. [Accessed 24 November 2021].

Appendix A

```
/* -------------------------- *\
    MTE 121 Project
    Parallel Parking Robot
    November 13, 2021

    Group 1
        Chelsea Dmytryk
        Dan Huynh
        Josh Blayone
        Sung-En Huang
\* -------------------------- */


// ----- Port Definitions -----
#define BACK_ULRTA_SENSOR S2
#define SIDE_ULTRA_SENSOR S3
#define GYRO_SENSOR S1
#define COLOR_SENSOR S4

#define LEFT_MOTOR motorA
#define RIGHT_MOTOR motorB
#define FLAG_MOTOR motorC

#define TIMER T1


// ----- Constants -----
const int LEFT_MOTOR_OFFSET = 0;
const int RIGHT_MOTOR_OFFSET = 0;

// Motor Speeds
const int DRIVING_SPEED = 20;
const int PARKING_SPEED = 10;
const int BANNER_SPEED = 10;
const int TURNING_SPEED = 10;

// Robot Constants
const int ROBOT_BODY_CM = 20;
const int ROBOT_WIDTH_CM = 14;
const float WHEEL_RADIUS = 2.75;

const int FORWARD = 1;
const int BACKWARD = -1;

const int RAISE = -1;
const int LOWER = 1;

// Program Settings
const int MAX_ATTEMPTS = 4;
const int MAX_TIME_FOR_ATTEMPT = 30 * 1000; // in milliseconds
const int MINIMUM_GAP_NEEDED = 20;          // in cm
const int DISTANCE_FROM_CAR = 10;           // in cm
const int BANNER_ANGLE = 90;                // in degrees
const int VALID_DISTANCE = 15;              // in cm


// ----- Robot Function Prototypes -----
void configureSensors();
void resetEncoder(tMotor selectedMotor);
void resetAllEncoders();
int getBackUltra();
int getSideUltra();
int getGyroAngle();
int getColor();
int getEncoder(tMotor selectedMotor);
void moveRobot(int speed); // Negative speed will move robot backwards
void stopRobot();
void rotateRobot(int angle); // Negative angle will move robot counter-clockwise
void driveDistanceCM(float distanceCM, int direction);


// ----- Non-Trivial Robot Function Prototypes -----
bool findParkingSpaceAndPark();
void waitForGreenCard();
void searchForCar(bool lookingForCar);
void parallelPark();
void parkingLineUp();
void reverseIntoSpot(float distanceToCar);
void pullForward();
void toggleBanner(int isRaised);


// ----- Main Program -----
task main()
{
    // Setup Robot
    displayString(4, "MTE 121 Project - Group 1");
    displayString(5, "Parallel Parking Robot");
    displayString(7, "Chelsea Dmytryk, Dan Huynh");
    displayString(8, "Josh Blayone, Sung-En Huang");

    configureSensors();
    resetAllEncoders();
    time1[TIMER] = 0;

    // Program
    waitForGreenCard();

    bool successfulParking = findParkingSpaceAndPark();

    if (successfulParking)
    {
        toggleBanner(RAISE);
        wait1Msec(2000);
        toggleBanner(LOWER);
    }
    else
    {
        eraseDisplay();
        displayString(5, "Unable to Find Parking Spot");
    }
    wait1Msec(1000);
}


// ----- Non-Trivial Robot Functions -----
bool findParkingSpaceAndPark()
{
    const float encToCM = (WHEEL_RADIUS * PI);
    float gapDistance = 0;

    int carCount = 0;
    bool finished = false;

    clearTimer(TIMER);

    while (carCount < MAX_ATTEMPTS && !finished)
    {
        moveRobot(DRIVING_SPEED);

        searchForCar(true);

        if (time1[TIMER] < MAX_TIME_FOR_ATTEMPT)
        {
            searchForCar(false);

            resetAllEncoders();

            searchForCar(true);

            stopRobot();
            gapDistance = (abs(getEncoder(RIGHT_MOTOR)) / 360.0) * encToCM;
        }

        if (gapDistance > MINIMUM_GAP_NEEDED)
        {
            parallelPark();
            finished = true;
        }
        else if (time1[TIMER] > MAX_TIME_FOR_ATTEMPT)
        {
            return finished;
        }
        else
        {
            carCount++;
            wait1Msec(1000);
        }
    }

    return finished;
}

void waitForGreenCard()
{
    while (getColor() != (int)colorGreen)
    {}
}

void searchForCar(bool lookingForCar)
{
    if (lookingForCar)
    {
        while (getSideUltra() > VALID_DISTANCE && time1[TIMER] < MAX_TIME_FOR_ATTEMPT)
        {}
    }
    else
    {
        while (!(getSideUltra() > VALID_DISTANCE) && time1[TIMER] < MAX_TIME_FOR_ATTEMPT)
        {}
    }
    if (time1[TIMER] > MAX_TIME_FOR_ATTEMPT)
    {
        stopRobot();
    }
}

void parallelPark()
{
    float distanceToCar = getSideUltra();

    parkingLineUp();
    wait1Msec(1000);
    reverseIntoSpot(distanceToCar);
    pullForward();
}

void parkingLineUp()
{
    moveRobot(20);
    resetEncoder(LEFT_MOTOR);

    driveDistanceCM(ROBOT_BODY_CM / 3, FORWARD);
    moveRobot(0);
}

void reverseIntoSpot(float distanceToCar)
{
    rotateRobot(45);
    wait1Msec(1000);

    driveDistanceCM(ROBOT_WIDTH_CM + distanceToCar, BACKWARD);
    stopRobot();
    wait1Msec(1000);

    rotateRobot(-45);
    pullForward();
}

void pullForward()
{
    moveRobot(PARKING_SPEED);
    while (getBackUltra() < DISTANCE_FROM_CAR)
    {}
    stopRobot();
}

void toggleBanner(int isRaised)
{
    resetEncoder(FLAG_MOTOR);

    motor[FLAG_MOTOR] = isRaised * BANNER_SPEED;

    while (abs(getEncoder(FLAG_MOTOR)) < abs(BANNER_ANGLE))
    {}

    motor[FLAG_MOTOR] = 0;
```

```
}


// ----- Robot Functions -----
void moveRobot(int speed)
{
    // "-" speed because motors are backwards
    motor[LEFT_MOTOR] = -(speed + LEFT_MOTOR_OFFSET);
    motor[RIGHT_MOTOR] = -(speed + RIGHT_MOTOR_OFFSET);
}

void stopRobot()
{
    moveRobot(0);
}

void driveDistanceCM(float distanceCM, int direction)
{
    float cmToDeg = distanceCM * 180 / (PI * 2.75);
    resetAllEncoders();

    moveRobot(direction * PARKING_SPEED);

    while (abs(getEncoder(LEFT_MOTOR)) < cmToDeg)
    {}
    stopRobot();
}

void rotateRobot(int angle)
{
    resetGyro(GYRO_SENSOR);
    if (angle > 0)
    {
        motor[LEFT_MOTOR] = TURNING_SPEED;
    }
    else
    {
        motor[RIGHT_MOTOR] = TURNING_SPEED;
    }

    while (abs(getGyroAngle()) < abs(angle))
    {}
    stopRobot();
}

void configureSensors()
{
    SensorType[BACK_ULRTA_SENSOR] = sensorEV3_Ultrasonic;
    SensorType[SIDE_ULTRA_SENSOR] = sensorEV3_Ultrasonic;
    SensorType[GYRO_SENSOR] = sensorEV3_Gyro;
    SensorType[COLOR_SENSOR] = sensorEV3_Color;
    SensorMode[COLOR_SENSOR] = modeEV3Color_Color;

    wait1Msec(50);
    SensorMode[GYRO_SENSOR] = modeEV3Gyro_Calibration;
    wait1Msec(50);
    SensorMode[GYRO_SENSOR] = modeEV3Gyro_RateAndAngle;
    wait1Msec(50);

    SensorMode[COLOR_SENSOR] = modeEV3Color_Color;
    wait1Msec(50);
}

void resetEncoder(tMotor selectedMotor)
{
    nMotorEncoder[selectedMotor] = 0;
}

void resetAllEncoders()
{
    resetEncoder(LEFT_MOTOR);
    resetEncoder(RIGHT_MOTOR);
    resetEncoder(FLAG_MOTOR);
}

int getBackUltra()
{
    return SensorValue[BACK_ULRTA_SENSOR];
}

int getSideUltra()
{
    return SensorValue[SIDE_ULTRA_SENSOR];
}

int getGyroAngle()
{
    return SensorValue[GYRO_SENSOR];
}

int getColor()
{
    return SensorValue[COLOR_SENSOR];
}

int getEncoder(tMotor selectedMotor)
```

```
 {
     return nMotorEncoder[selectedMotor];
 }
```

Figure 1: Robot C source code

Appendix B

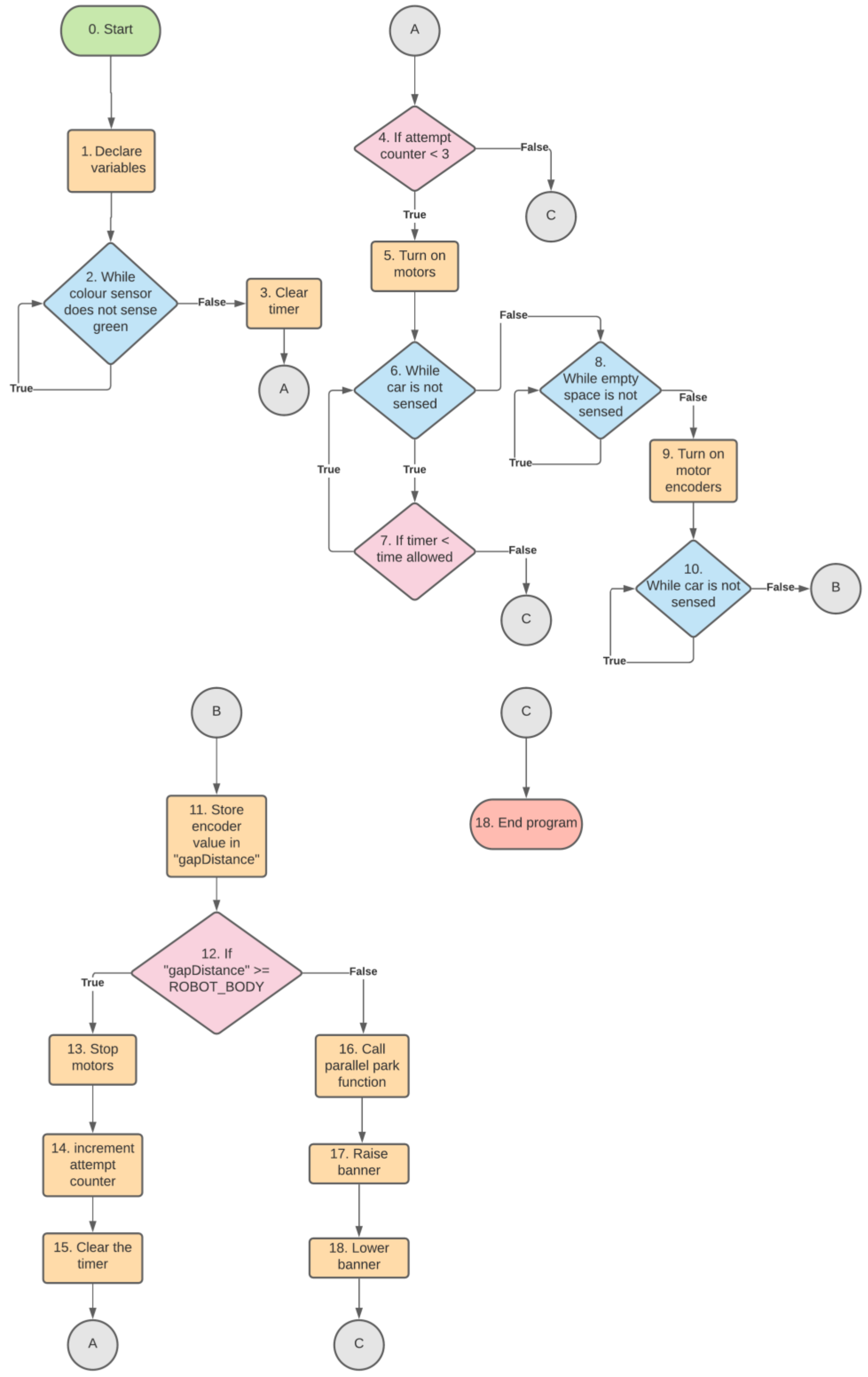
0. Start
1. Declare variables
2. While colour sensor does not sense green
True
False
3. Clear timer
A
A
4. If attempt counter < 3
False
C
True
5. Turn on motors
6. While car is not sensed
False
True
True
7. If timer < time allowed
False
C
8. While empty space is not sensed
True
False
9. Turn on motor encoders
10. While car is not sensed
True
False
B
B
11. Store encoder value in "gapDistance"
12. If "gapDistance" >= ROBOT_BODY
True
False
13. Stop motors
14. increment attempt counter
15. Clear the timer
A
16. Call parallel park function
17. Raise banner
18. Lower banner
C
C
18. End program

Figure 2: The flowchart that was referenced when coding logical action