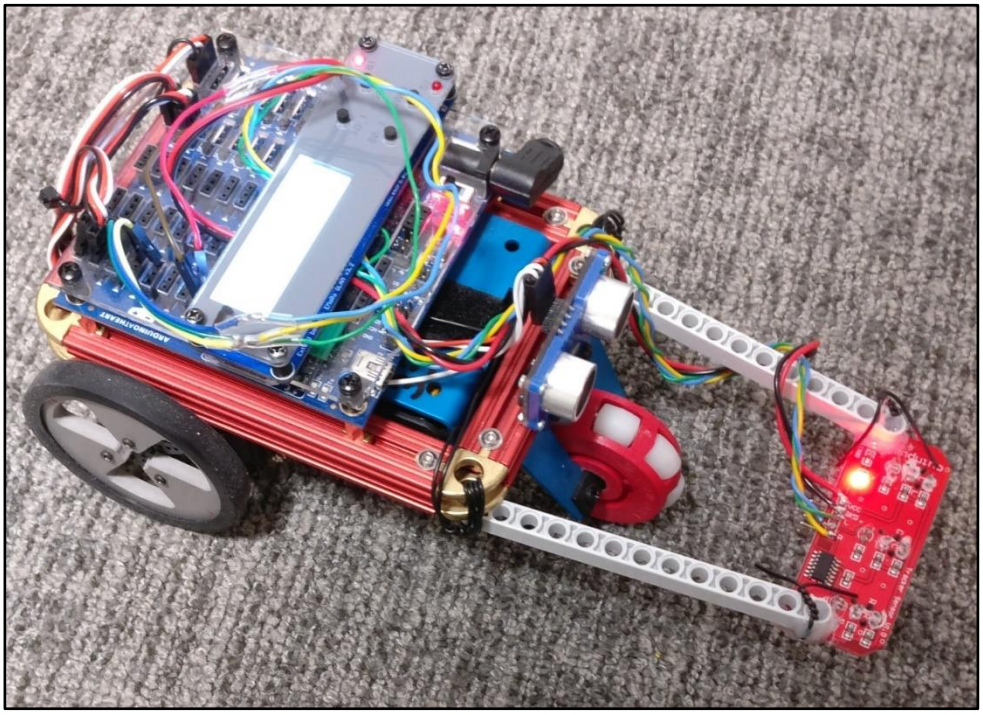
2019



ASSIGNMENT 2

CE243 C PROGRAMMING AND EMBEDDED SYSTEMS

BOROWSKI, MICHAL (1904535)

Table of Contents

Introduction

Objectives

The goal of the assignment is to create Arduino based C program to control EMoRo 2560 robot including its' GLAM module which provides display, 4 buttons and bluetooth module.

Although programming the bluetooth module is not required by the assignment specification, doing it allows for convenient debugging and configuration of the robot.

Buttons of the GLAM module supposed to provide 3 modes such as: obstacle avoidance, ellipse track following, rectangle track following. The 4th button should serve as a reset for the other modes.

The display of the GLAM module should display different messages depending on the button pressed. Additionally, in obstacle avoidance and square track following modes it should display notifications about certain events.

Button – Mode	Immediate display message	Event-triggered message
1 – Obstacle avoidance	moving forward	obstacle detected
2 – Circle track following	Circle track following.	
3 – Square track following	Square track following.	Sharp turning *** degrees
4 - Reset	1. Obst 2. Circle 3. Square 4. Reset	

Overview

Report structure

For the sake of clarity, each mode has its' own section where its' characteristics, flowchart and review are presented.

Program structure

The program has modular design, with functionality distributed among multiple files, each responsible for separate aspect of design. Each mode has its' own class and implements "Init, Update and Reset" abstract methods of their parent class (called Mode). Thanks to polymorphism, the switching between modes (and the behaviour of the modes themselves) was implemented in a concise way. It can be noticed by looking at how simple the loop function is:

```
// Assignment2.ino file

/*  FollowRect parameters are dimensions of the track  */
Mode *modes[] = { new AvoidObstacles(), new FollowCircle(), new FollowRect(67,
42), new ResetMode() };

/*  Irrelevant part was removed  */

void loop() {
  bt->Update();
```

```
  /*  Declaring the array as static prevents unnecessary overwriting of memory at
every iteration of the loop.
      The array is initialized only once. It's better than using global variable
because it's close to where it's used.  */
  static int switch_id[] = { SW_1, SW_2, SW_3, SW_4 };
  if(ReadEmoroHardware() & SW_AVAILABLE)    // if switches are available
    for (int i = 0; i < 4; ++i)
      if (ReadSwitch(switch_id[i]))
         /*  static Set method below sets the currently active mode.  */
         Mode::Set(modes[i]);

  /* Each of the modes has its' own implementation of the Update function.
     static Get() method below returns the currently active mode.  */
  Mode::Get()->Update();
}
```

File structure

Assignment2.ino is the main file that includes all the rest of the files and includes high-level instructions to operate the program. The table below roughly describes the responsibility of each file included in the project.

Filename (.cpp and .h)	Description

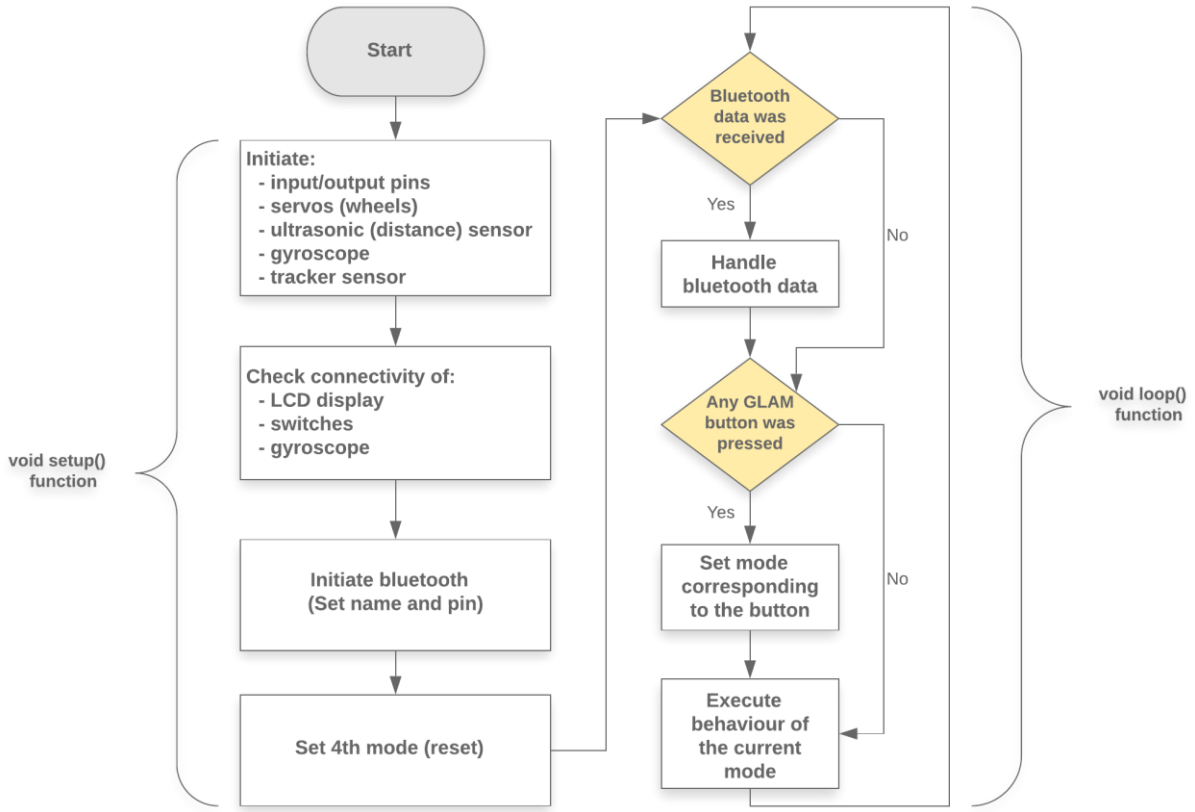
bluetooth	Files that implement functionality of sensors/modules and provide simple interface to operate them, usually consisting of single-line instructions such as:
	`BT_SERIAL.println("Hello smartphone, I'm Emoro.");`
	`display.Msg("top-left", "top-right", "bot-left", "bot-right");`
	`drive.Forward();`

display	
drive	
mode	Abstract class containing currently used mode with static methods to get/set it. It constitutes interface that must be implemented by each subclass/mode. Each subclass/mode must implement the following 3 methods: - Init - Update - Reset
mode_AvoidObstacles	Subclasses of the Mode abstract class. Each provides different functionality for its' respective button.
mode_FollowCircle	
mode_FollowRect	
mode_ResetMode	
tracker_sensor	Contains TrackerSensor class which holds functions to get state of the 3 infra-red sensors.

includes	includes.h file consists of a set of “#include “filename” instructions. This allows all the other files in the project to use a single line at the top in order to include all the necessary objects, all at once: `#include "includes.h"`

Slightly more detailed description of some files (e.g. bluetooth) is provided in the comments of includes.h file.

High level flowchart

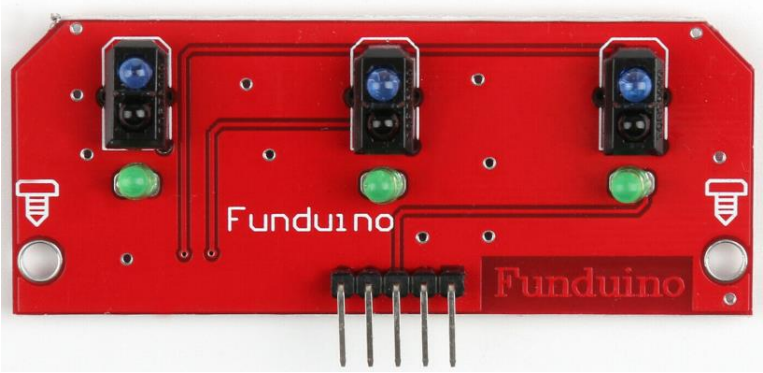


Setting of the mode involves execution of the “Reset” method of the previous mode and the “Init” method of the newly chosen mode as shown below.

```
// mode.cpp file

void Mode::Set(Mode *m) {
  if (current)
    current->Reset();

  current = m;
  current->Init();
}
```

Hardware

The module presented below (containing 3 infra-red sensors) was used instead of the original 2 sensors underneath the robot:

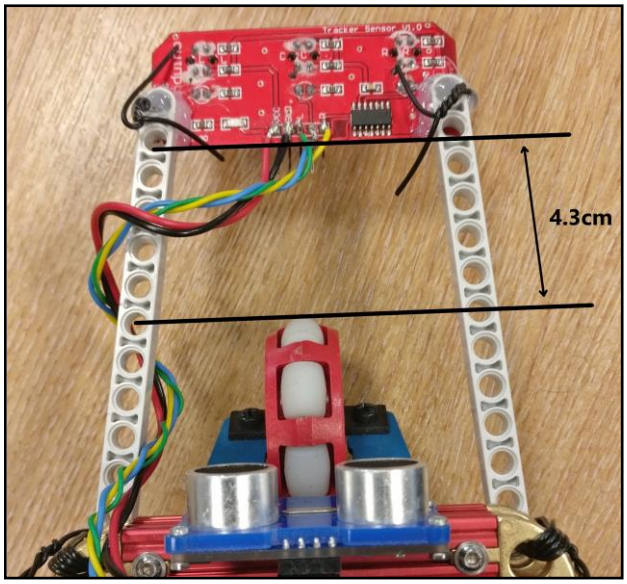


Source (of image and the listing where the module was purchased): https://www.ebay.co.uk/itm/4Pcs-3-Way-Infrared-IR-Line-Tracking-Sensor-Module-For-Raspberry-Pi-U3/193184039763

The fact of having 3 sensors at the front of the robot instead of 2 sensors directly underneath it, enabled the robot to apply relatively simple rules to follow the circular track. It also allowed implementation of effective technique of following the rectangular track. Hardware modifications were consulted with the module tutor to make sure they are acceptable.

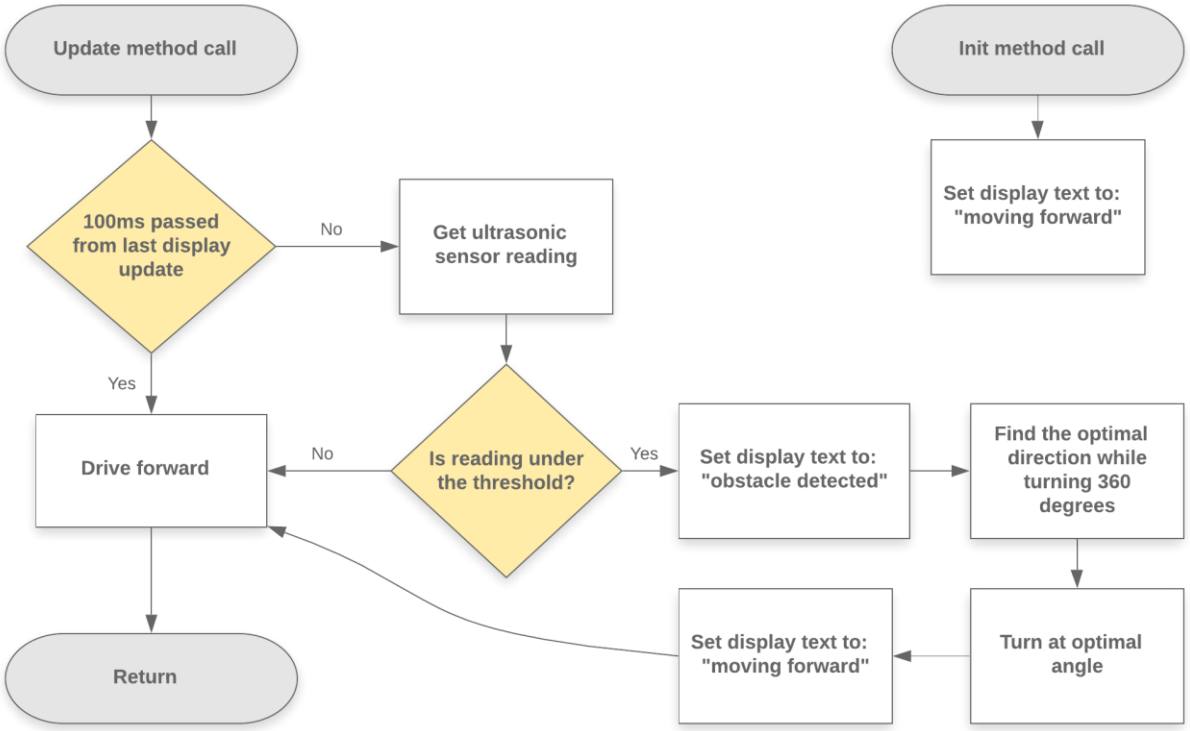
Wiring used	
EMoRo2560	**Tracker sensor module**
GND	GND
VCC	VCC
IO_2	L (left)
IO_3	C (center)
IO_4	R (right)

The red module containing 3 sensors is referred to as “the front module” in the remaining part of this report.



Obstacle avoidance

Flowchart



The purpose of 100ms delay is explained in the "Display update delay" section below which addresses how it does not negatively affect the performance.

Detection method

Ultrasonic sensor at the front of the robot is used to detect obstacles. The detection threshold is defined at the top of “mode_AvoidObstacles.cpp” file and is set to 50cm.

Behaviour upon detection

The robot stops, gives audio notification and does 360deg turn scanning for the optimal direction to choose. The scanning involves reading of 2 sensors repetitively:

1. Ultrasonic sensor – to compare the current empty space length with the previously saved longest empty space detected.

2. Gyroscope – to know when 360deg turn is finished and keep reference to an angle where the longest empty space was detected.

Every time the new longest empty space is detected, the robot beeps (with a higher pitch as compared to the one used upon detection of obstacle). This provides insight and verification

method that is better than the use of Serial/display output messages (that would be difficult to see unless it was recorded with camera and slowed down).

After the scanning is completed the robot rotates to the angle where the longest empty distance was detected and begins the forward movement again.

Display update delay

Aside from "moving forward" message, the program additionally displays the value received from ultrasonic sensor reading (on the 2nd row of the display). Too frequent update of display text results in unreadable text. The delay ensures that the ultrasonic sensor reading can be read. Considering relatively small maximum speed of the robot, the delay is insignificant for the performance of the obstacle detection mode.

That statement is supported by the speed tests performed which showed that the robot reaches 70cm distance within around 3.5s, which is around 2cm per 100ms. The obstacle detection range threshold is set to 50cm. 2cm added/subtracted from 50cm gap is not a significant difference.

Performance of optimal angle detection

The 360deg turn is done with maximum speed. For that reason a couple of techniques were applied to verify how well it works, techniques such as:

- Sound notification when new optimal position is found.
- Counting how many distinct readings are used within a single 360deg turn and outputting it through Bluetooth.

It was observed that the readings are collected around 6000 times, of which around 20 are distinct and around 350 indicated out-of-range value coming from the ultrasonic sensor. The low number of distinct readings was caused by the limitation of the ultrasonic sensor distance (officially it is 4m) and the fact that there was a lot of empty space around the robot, hence the high number of out-of-range readings.

Observation of the robot movement and decision making, show that the optimal angle detection method works well.

Performance of obstacle avoidance mode

The robot successfully avoids flat objects standing perpendicularly to it, however if the object stands over certain angle then the ultrasonic sensor does not recognize the distance properly. Appropriate notifications are displayed on the display at the right times ("moving forward", "obstacle detected").

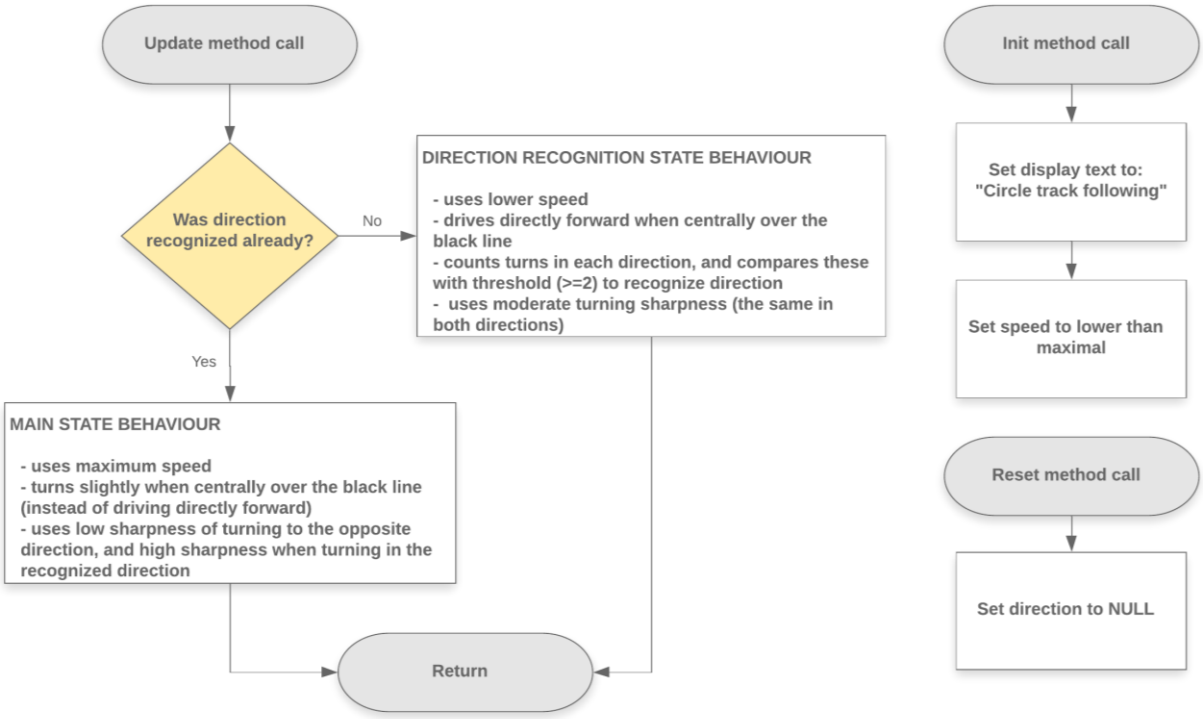
Potential improvements

Placing the ultrasonic sensor on a servo attached at the top of the robot would allow to rotate the direction of the measurements and avoid the issue mentioned in the section above. Alternatively, more ultrasonic sensors could be attached to the robot pointing to the sides of the robot (e.g. 45deg angle from the front direction)

Circle track following

States

Trying to move the robot as fast as possible, the initial direction detection state was implemented. Knowing the direction enables the robot to safely keep turning (to slight extent) even when the sensors reading indicate that the black line is centrally underneath it (this “trick” is referred to as “slight turn technique” in the sections below). That significantly improves the speed and smoothness of movement.



Direction detection state

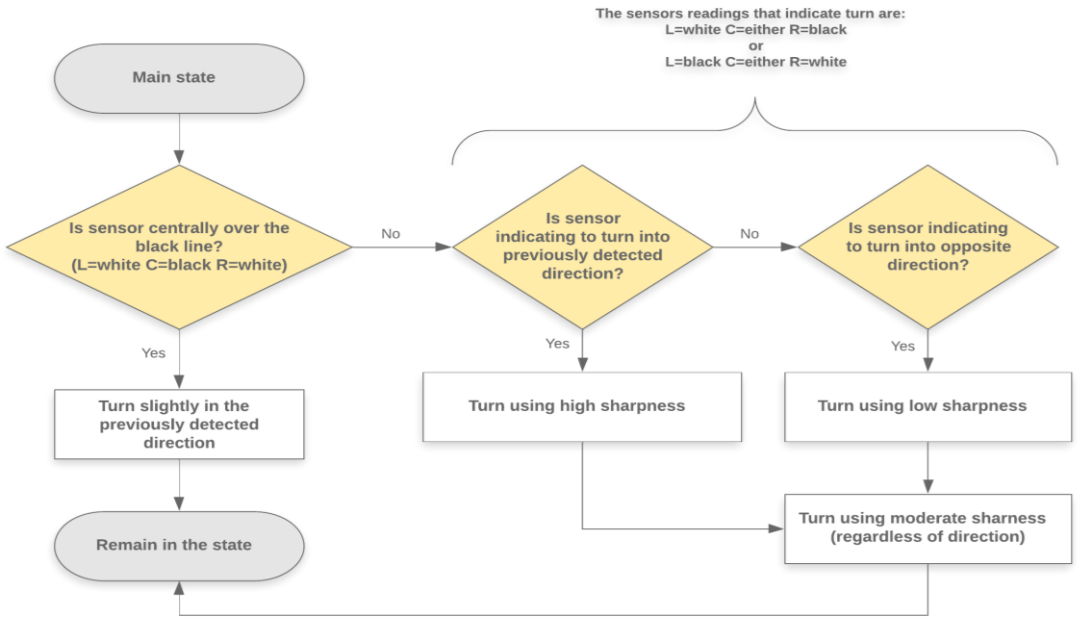
In the initial state (detection direction), if the sensor indicates that the black line is located directly underneath it, the robot is instructed to move forward (the “slight turn technique” is not used yet). Once the central sensor goes off the line, the robot is instructed to turn. This leads to a situation where:

- If robot was positioned **inside** the circle at the beginning of the run, it will turn once to the outside but then it will repetitively turn into the same direction over and over again.
- If robot was positioned **outside** the circle at the beginning of the run, then it will first turn to the inside of the circle, then once outside, but the following inside turn will indicate the direction of the run.

In both possible cases above, counting the turns in each direction and comparing them with number 2 will allow to find out the direction of the run (regardless of the fact if the car was positioned inside or outside of the circle in the beginning).

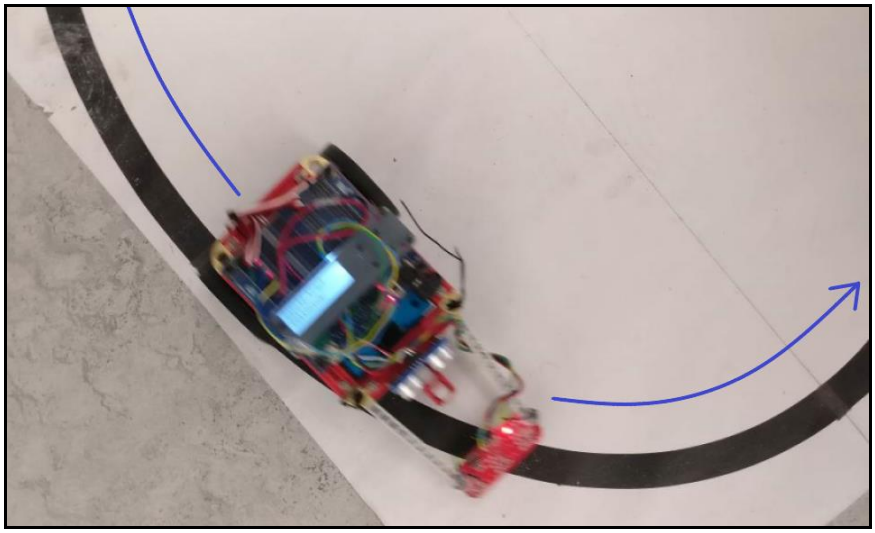
The main state

Once the direction is found out, the speed is set to maximum and the “slight turn technique” is used for a smooth movement. Another difference from the direction detection state is the sharpness of turning which varies depending on direction.



Optimal path following

Thanks to the fact that the line detecting sensors are located at the front, during the circle track run, the robot is located slightly inside of the circle. This allows the robot to cover less total distance during each run while still touching the line by 1 wheel. It decreases the time needed to complete a full round by the robot.



Performance over 5 rounds around the track

The slower result during the first round was caused by the direction detection state which uses slightly decreased speed and has less efficient way of moving forward (it does not make the slight turn when sensor is centrally over the black line).

The test was done using new batteries, the ReadPowerSupply function returned 9.13V just before the test.

Round	Time
1	9.97s
2	9.67s
3	9.65s
4	9.65s
5	9.54s

Rectangle track following

Preface

The method described below was implemented after consultation with the module tutor, and confirming that the method must work with the particular version of the rectangle track provided, and not necessarily with any kind of rectangle track.

Method

The time required to reach the end of the sides of rectangle is estimated and used as a trigger to start turning. The front module is used to keep at the centre of the black line for most of the time. In order to avoid being affected by the perpendicular line, the front module is set to be ignored just before reaching the perpendicular line. This is indicated to the user by a beep sound. Another beep is played when the vehicle reaches the end of the line.

Calculation adjustment

Initially the time required to reach the end of each side was measured using stopwatch hardcoded in the program and refined using trial and error. Then more of the factors were taken into account in the calculation, factors such as:

- Length of the side and how it corresponds to the time required to reach the end.
- Battery level (using ReadPowerSupply function provided by EMoRo library).

Expected travel time and side length relation

It was observed that the decrease in length of the line does not result in direct equivalent decrease of the time required to reach the end.

For example, the optimal time to travel:

a) 70cm took 3.5s (around 50ms per 1 cm)

b) 42cm took 2.3s (around 54ms per 1cm)

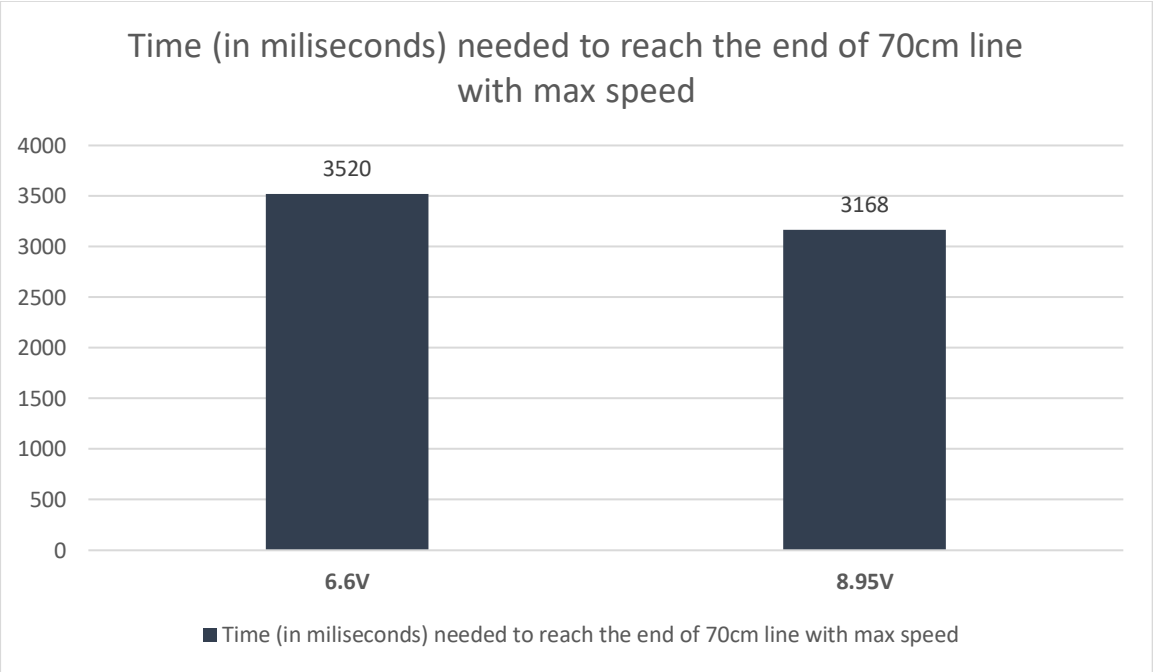
In order to make rough estimate of the time required to travel the distance, the code is using linear interpolation using 2 collected data samples. In particular, the "map" function is used to translate the length of the track into estimated time required to travel to the end of it. The pseudocode below presents how exactly it was used.

```
map( length_of_the_track_to_be_translated,   // (example scenario: the track is
slightly longer, the task is to estimate how much time it will take to reach end
of it using previously collected data)
     shorter_length,                         // used in previous test
     longer_length,                          // used in previous test
     time_required_to_travel_shorter_length, // observed in previous test
     time_required_to_travel_longer_length )  // observed in previous test

map( 75.0
     42.0,
     70.0,
     2300.0,
     3520.0 )

// The return of such map function would be: 3737.
```

Possible improvement:

Even better approach (but more complex to implement) would be to perform more timing tests and use cubic interpolation to estimate the time. It would allow for more precise estimation, which would be especially helpful when the track length is very low or very high.

Expected travel time and battery level relation



It was noticed that the time required to reach the end of the long side of rectangle with 6.6V power supply was 3520ms (batteries almost running out). When 8.95V power supply was used that number decreased to 3168ms (using fresh batteries). Using few more trial and error tests the following formula was created:

```
// mode_FollowRect.cpp file

void FollowRect::Update() {
  if (is_first_turn && !start_time) {
    float voltage = ReadPowerSupply();
    voltage_adjustment_mult = power_supply_significance * (voltage - 6.6);
    AssignEstimatedTravelTime();
  }
  /*  Irrelevant code was removed  */
);

/*  Irrelevant code was removed  */

void FollowRect::AssignEstimatedTravelTime(){
  travel_time = map(side_len, 42, 70, 2300, 3520);
  AdjustTravelTimeByBatteryLevel();
  /*  Irrelevant code was removed  */
);

/*  Irrelevant code was removed  */

/*  Higher voltage results in lower estimated travel time.  */
void FollowRect::AdjustTravelTimeByBatteryLevel() {
  travel_time -= (travel_time * voltage_adjustment_mult);
}
```

First I attempted to get reading from ReadPowerSupply multiple times during a single run. Thanks to debugging through Bluetooth I observed that the return of ReadPowerSupply is inconsistent during

the run (but consistent when robot is standing still) and it is better to call it only once, at the beginning of the run.

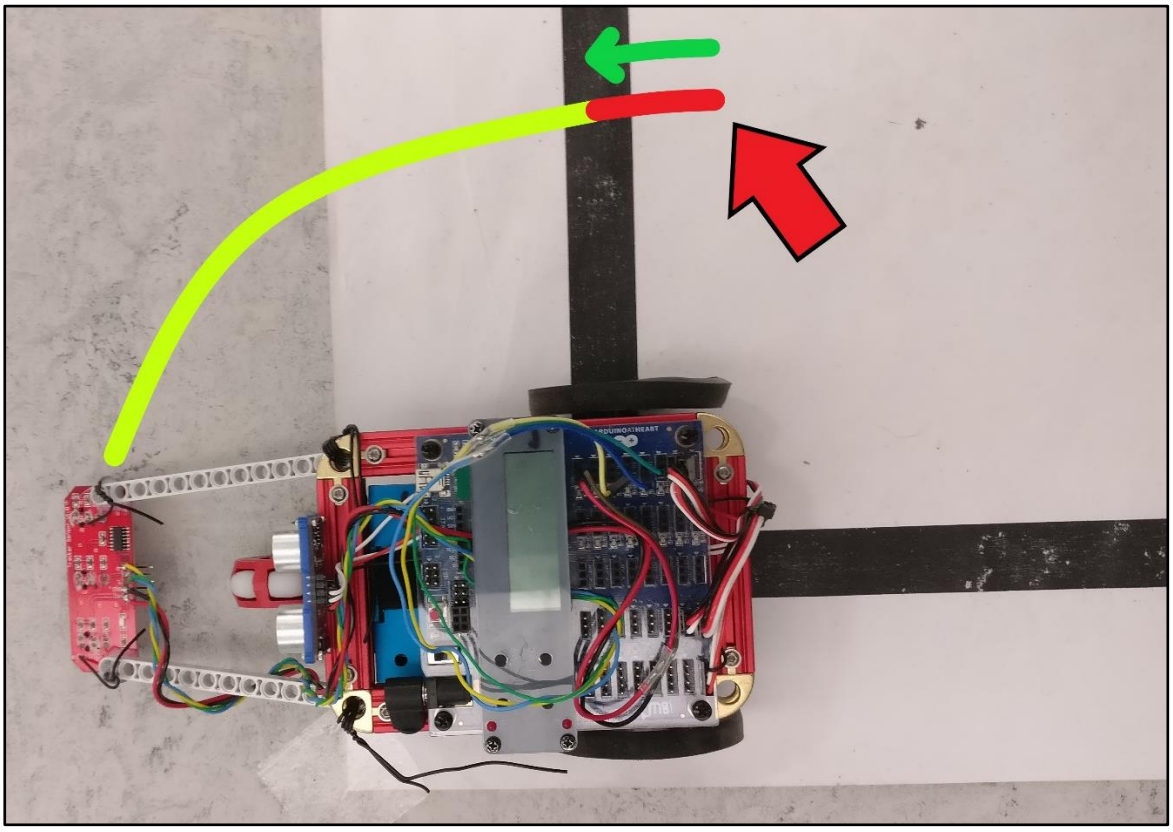
The power_supply_significance variable is configurable through Bluetooth, which allowed for fine adjustment.

```
10:31:31.105 ipss
10:31:31.233 power_supply_significance = 0.18
10:31:31.741 ipss
10:31:31.834 power_supply_significance = 0.19
10:31:32.798 dpss
10:31:32.908 power_supply_significance = 0.18
10:31:33.378 dpss
10:31:33.449 power_supply_significance = 0.17
```

Simple Bluetooth Terminal application (blue - messages sent, green - response received)

The function and purpose of "Rebound"

The turning method involves stopping once the line is recognized under the front module. The stopping does not happen immediately, resulting in position mismatch as shown on the image below.



The red arrow shows the position of the front module after the robot finishes the turn. The green arrow shows the location of the front module during the rebound turn.

Rebound turn is used to fix the position of the car. The extent of rebound can also be adjusted through Bluetooth.

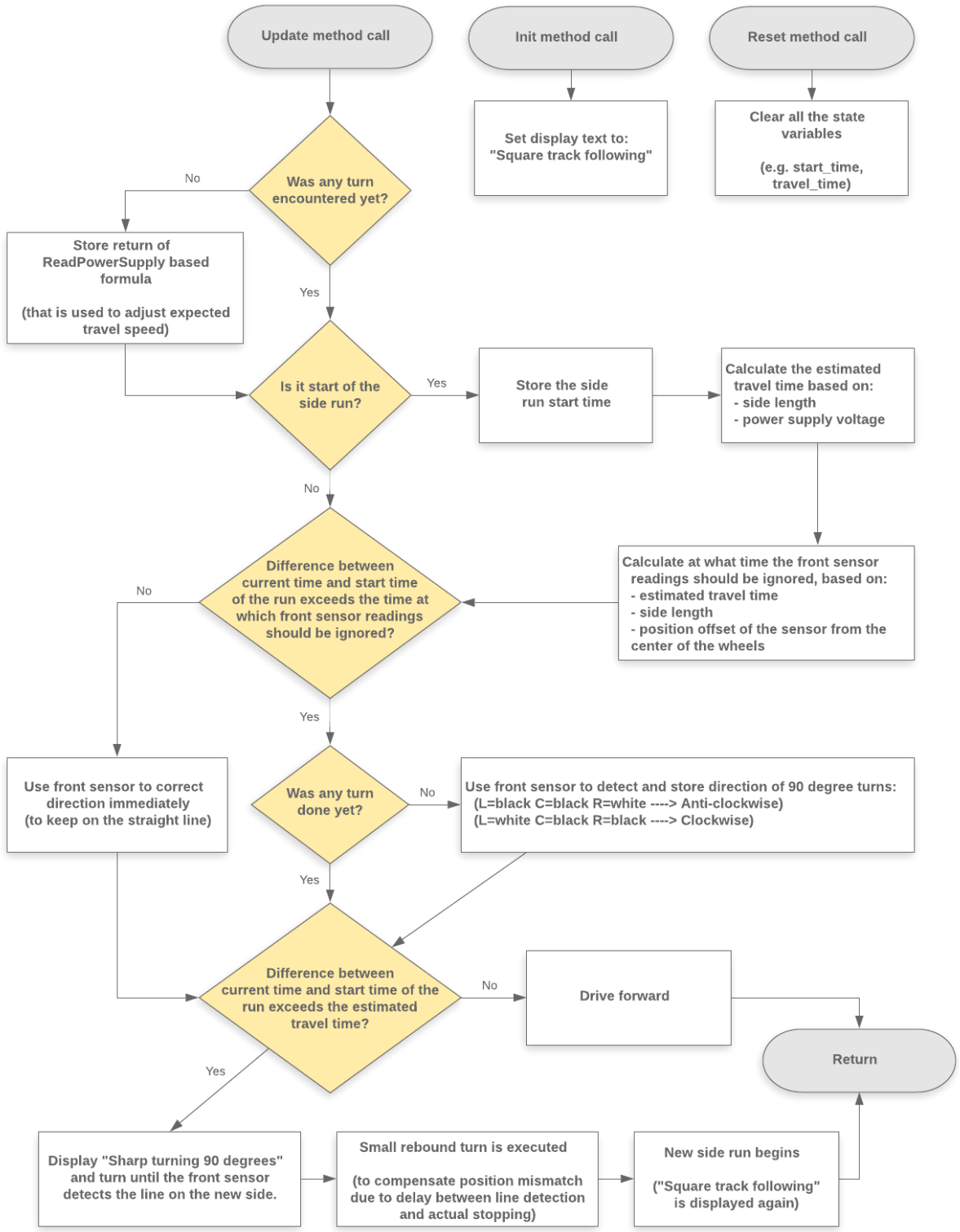
The rebound_size value is the number of degrees that is supplied to Drive::Turn method.

Dynamic configuration through Bluetooth

Several factors that influence performance of the rectangle following mode can be adjusted before and during the run. The following variables can be adjusted:

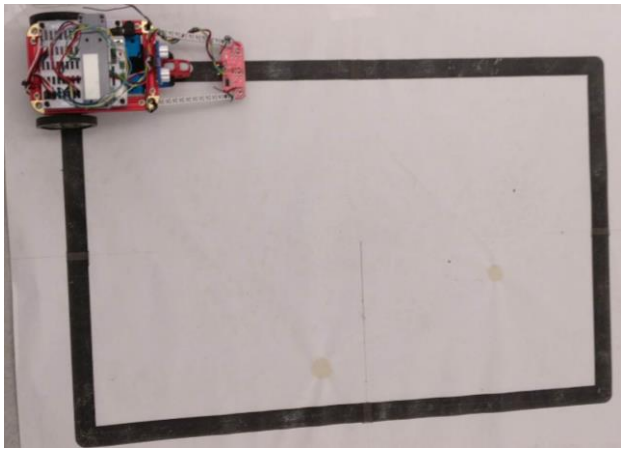
Variable (FollowRect class methods)	**Bluetooth command**	**Description**
long_side_len SetLongSideLen(int len)	lsl=<number>	How long is the longer side of the rectangle, it impacts the estimated travel time.
short_side_len SetShortSideLen(int len)	ssl=<number>	As above but for the shorter side.
following_sharpness IncreaseFollowingSharpness() DecreaseFollowingSharpness()	ifs	How sharply the robot turns to follow the straight line.
	dfs	
rebound_size IncreaseReboundSize() DecreaseReboundSize()	irs	How long is the counter-turn after the line is found under sensor during the 90 degree turn.
	drs	
power_supply_significance IncreasePowerSupplySignificance() DecreasePowerSupplySignificance()	ipss	How much the voltage level will impact the estimated travel time.
	dpss	

Flowchart


Update method call
Init method call
Reset method call
Set display text to:
"Square track following"
Clear all the state variables
(e.g. start_time, travel_time)
Was any turn encountered yet?
No
Yes
Store return of ReadPowerSupply based formula
(that is used to adjust expected travel speed)
Is it start of the side run?
Yes
Store the side run start time
Calculate the estimated travel time based on:
- side length
- power supply voltage
No
Difference between current time and start time of the run exceeds the time at which front sensor readings should be ignored?
Calculate at what time the front sensor readings should be ignored, based on:
- estimated travel time
- side length
- position offset of the sensor from the center of the wheels
No
Yes
Use front sensor to correct direction immediately
(to keep on the straight line)
Was any turn done yet?
No
Use front sensor to detect and store direction of 90 degree turns:
(L=black C=black R=white ----> Anti-clockwise)
(L=white C=black R=black ----> Clockwise)
Yes
Difference between current time and start time of the run exceeds the estimated travel time?
No
Drive forward
Return
Yes
Display "Sharp turning 90 degrees" and turn until the front sensor detects the line on the new side.
Small rebound turn is executed
(to compensate position mismatch due to delay between line detection and actual stopping)
New side run begins
("Square track following" is displayed again)

Limitations

The robot must be placed at the beginning of the line at the start of the run as shown on the image below.



Another limitation which stems from the time estimation method is that the length of both sides of the track must be input through Bluetooth or set as hardcoded initial values in Arduino code.

Performance over 5 rounds around the track

The ReadPowerSupply function returned 8.67V just before the test.

Round	Time
1	13.30s
2	13.98s
3	13.60s
4	13.61s
5	13.57s

The configuration used for the test can be seen on the image below.

```
16:44:00.739 ----------
16:44:00.739 FollowRect SETTINGS:
16:44:00.739 long_side_len = 70
16:44:00.740 short_side_len = 42
16:44:00.741 following_sharpness = 2.00
16:44:00.761 rebound_size = 16
16:44:00.761 power_supply_significance = 0.07
16:44:00.762 ----------
```

Slight differences in achieved times may be the result of human error when pressing the stopwatch button. It can be noticed especially when looking at the average of 1st and 2nd round results.

(13.30 + 13.98) / 2 = 13.6

It is likely that the button was pressed too early at the end of the 1st run (which was also the beginning of the 2nd run.

Potential improvements

1. When turning 90 degrees and trying to recognize the black line under the front module, avoid checking whether the sensor is centrally over the line. Instead, check if any of the 3 sensors from the front module is over the black line. This way the required rebound would be smaller, resulting in improved speed around the corners.
2. To avoid consequences of too high estimated travel time (robot moving past the perpendicular line), make use of 2 sensors underneath the robot to detect white colour and force the robot to turn at that point.

Additional Information

Bluetooth communication details

EMoRo2560 robot is based on ATmega2560 microcontroller which provides 3 auxiliary serial ports (emoro.eu, 2019) (arduino.cc, 2019). One of these ports (Serial1) is connected to the RX/TX pins of the Bluetooth module located on the GLAM module. To improve the readability of the code I decided to use the following alias for “Serial1” which visually seems very much like the “Serial”:

```
// Bluetooth.h file

#define BT_SERIAL Serial1
```

Thanks to it, any file that imports “bluetooth.h” file is allowed to use “BT_SERIAL” instead of “Serial1” which makes debugging lines clear in terms of their purpose. Below I present example of how it is used to send data through the Bluetooth module:

```
// mode_FollowRect.cpp file

void FollowRect::OutputAllSettings() {
  BT_SERIAL.println("\n------------\nFollowRect SETTINGS:");
  BT_SERIAL.print("long_side_len = "); BT_SERIAL.println(long_side_len);
  BT_SERIAL.print("short_side_len = "); BT_SERIAL.println(short_side_len);
  BT_SERIAL.print("following_sharpness = ");
BT_SERIAL.println(following_sharpness_div);
  BT_SERIAL.print("rebound_size = "); BT_SERIAL.println(rebound_size);
  BT_SERIAL.print("power_supply_significance = ");
BT_SERIAL.println(power_supply_significance);
  BT_SERIAL.println("------------");
}
```

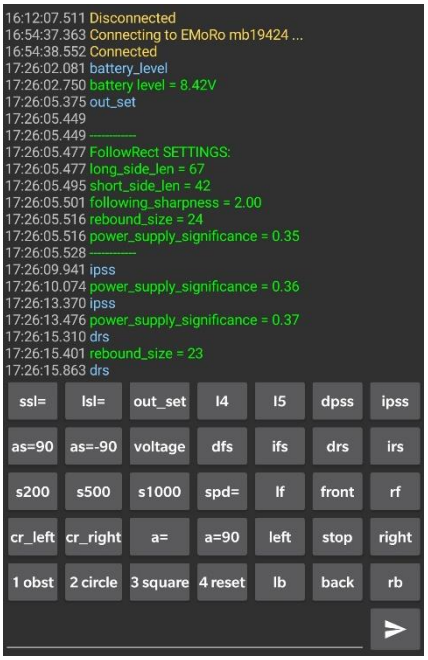
By looking at the “High-level flowchart” in the “Introduction” section, it can be noticed that handling of data received by bluetooth module is performed in a loop along with the “current_mode->Update()” function. This allows to update configuration during the operation of the modes, without the need to stop/reset the robot.

“Serial Bluetooth Terminal” application by Kai Morich was used as user interface on the smartphone to interact with the robot. (Google_Play, 2019)

Aside from the commands already listed in the “Rectangle track following” section, the protocol included the following commands:

Command	Description
s, f, b, l, r, a, d, z, c	Single character commands that allow manual control over robot movement.
1, 2, 3, 4	Single character commands that set the mode that corresponds to the respective button number.
spd=<int>	Sets the speed of the car
a=<int>	Turns the car (positive – right turn, negative – left turn)
a_stop_at_line=<int>	Turns the car as above but stops once the front module detects the black line being located centrally under it.
battery_level	Requests the robot to use ReadPowerSupply() and to respond with the return of it.
out_set	Respond with settings used in FollowRect class.

Remembering the commands or inputting them manually could be difficult. The Simple Bluetooth Terminal app allows to set macro buttons to automatically send or automatically insert values in the input text box. In the image on the right, blue messages are the ones sent from mobile application to the robot. The green messages are responses sent back to the mobile app.



In the bottom-right corner, an array of 3x3 buttons allows for manual movement of the robot (e.g. lf = left front, rb = right back). In the bottom-left corner there are 4 buttons dedicated to simulating 4 buttons of the GLAM module. “s200”, “s500” and “s1000” set the speed of the robot by sending spd=<value> command.

Producing sound effects

EMoRo robot includes a buzzer component that can be used to create sound effects. I used examples from the EMoRo library to get familiar with how to use it (toneMelody_example, 2019). The function shown below was used to produce the effects:

```
tone(BUZ_BUILTIN, frequency, duration);
```

It was used as follows in the obstacle avoidance mode to notify of obstacle being detected and the new optimal direction being found:

```
// mode_AvoidObstacles.cpp file

double AvoidObstacles::FindBestAngle() {
      /*  Irrelevant code removed   */

      // the tone is played if new optimal direction is found
      tone(BUZ_BUILTIN, 131, 50); // NOTE_C3  131
      /*  Irrelevant code removed  */
}

/*  Irrelevant code removed  */

void AvoidObstacles::PlayDetectionSound() {
  /*  NOTE_C1 reference:
      https://github.com/inovatic-ict/emoro-2560-
library/blob/master/examples/02.%20Digital/toneMelody/pitches.h   */
  tone(BUZ_BUILTIN, 33, 50); // NOTE_C1  33
  delay(100);
  tone(BUZ_BUILTIN, 33, 50); // NOTE_C1  33
  delay(100);
  tone(BUZ_BUILTIN, 33, 50); // NOTE_C1  33
  delay(500);
}
```

It was also used in the rectangle track following mode to notify the user of 2 things:
- point in time at which the front module readings start being ignored (just before the front module goes over the perpendicular line)
- point in time where 90 degree turn should occur (when wheels go over the perpendicular line)

```
void FollowRect::Update() {
/*  Irrelevant code removed  */

  /*  Make sound at which point the front module readings should be ignored.  */
  if (!is_near_end && time_since_last_turn > is_near_end_threshold_time) {
    /*  65 is NOTE_C2 in the pitches.h file from toneMelody example provided by
Emoro library  */
    tone(BUZ_BUILTIN, 65, 200);
    is_near_end = true;
  }

  /*  Make another sound using higher pitch when its time to turn.  */
  if (!reached_end && time_since_last_turn > travel_time) {
    /*  131 is NOTE_C3 in the pitches.h file from toneMelody example provided by
```

```
Emoro library  */
    tone(BUZ_BUILTIN, 131, 200);
    reached_end = true;
  }

/*  Irrelevant code removed  */
}
```

Conclusion

Performance

All required objectives from assignment specification were completed. Aside from that, some additional functions were implemented, functions such as:

- Optimal direction finding in obstacle avoiding mode
- Sound effects in obstacle avoiding and square track mode
- Bluetooth communication including remote control, configuration and debugging

Learning experience

This project helped me to understand the importance of structuring code and prioritizing work. Initially, to some extent, I "jumped" into implementing the behaviour of the buttons listed in assignment specification. This was not the best idea considering how inconvenient the debugging, testing and writing the code was without building the solid foundation first. Solid foundation consisting of handy functions separated into their own files, providing single-line calls for:

- Displaying messages on screen (display.cpp / .h)
- Outputting debugging information through Bluetooth (bluetooth.cpp / .h)
- Controlling the movement of the robot (drive.cpp / .h)
- Changing the current mode (mode.cpp / .h and its' sub-classes)
- Getting front module state (tracking_sensor.cpp /.h)

Building that foundation made the work on this assignment easier.

I also realized how important it is to ask questions and share considerations when I am unsure about how to complete the assignment. That helped me to overcome being stuck with the rectangle track following task. That is because prior to asking about it I was sure that the method used in this assignment would not be acceptable.

References

arduino.cc. (2019). *MultiSerialMega*. Retrieved 12 08, 2019, from https://www.arduino.cc/en/Tutorial/MultiSerialMega

emoro.eu. (2019). *EMoRo 2560 controller*. Retrieved 12 08, 2019, from https://www.emoro.eu/shop/index.php?route=product/product&product_id=52

Google_Play. (2019). *Serial Bluetooth Terminal by Kai Morich*. Retrieved 12 08, 2019, from https://play.google.com/store/apps/details?id=de.kai_morich.serial_bluetooth_terminal&hl=en_GB

toneMelody_example. (2019). Retrieved from https://github.com/inovatic-ict/emoro-2560-library/blob/master/examples/02.%20Digital/toneMelody/toneMelody.ino

Source code

Assignment2.ino

```
/*  CE243 C Programming and Embedded Systems - Assignment 2
    Deadline date: 10/12/2019  */

#include "includes.h"

/*  "bt" is a pointer to the bluetooth module object which was useful for
debugging and testing.
    Testing behaviour of the robot with the cable connected is inconvenient.
    Having the cable unplugged limits debugging capabilities. The Bluetooth
    class solves that problem by providing a way to debug and interact with the
program
    using bluetooth terminal application.   */
Bt *bt;

/*  FollowRect parameters are dimensions of the track  */
Mode *modes[] = { new AvoidObstacles(), new FollowCircle(), new FollowRect(70,
42), new ResetMode() };

/*  Setup function is called only once at the begining of the program execution.
*/
void setup() {
  InitEmoro();   // Initializes all available inputs and outputs on EMoRo 2560.

  /*  Initiate servos (used as wheels).  */
  EmoroServo.attach(SERVO_0);
  EmoroServo.attach(SERVO_1);

  /*  Initiate ultrasonic sensor that will be used for obstacle avoidance by
measuring distance to the object in front of the car.  */
  Ultrasonic.attach(GPP_0);

  /*  Initialize the tracker_sensor by supplying pins it is connected to
```

```
(respectively: left, center, right)  */
  tracker_sensor = new TrackerSensor(IO_2, IO_3, IO_4);

  /*  Initialize gyroscope sensor that is used for finding optimal angle when
avoiding obstacles.  */
  Gyr.init();

  /*  Set baud rate of serial communication (allowing to use "Tools->Serial
Monitor" feature of Arduino IDE to debug the code.  */
  Serial.begin(9600);

  /* Output serial messages to diagnose problems in case of faulty sensor or
incorrect wiring */
  if(!(ReadEmoroHardware()& LCD_AVAILABLE))
    Serial.println("LCD is not available");

  if(!(ReadEmoroHardware() & SW_AVAILABLE))
    Serial.println("Switches are not available");

  if(!(ReadEmoroHardware() & GYR_AVAILABLE))
    Serial.println("Gyroscope is not available");

  /* Initialize bluetooth by setting its' name and pin. */
  bt = new Bt("EMoRo mb19424", "3737");

  /* Call 4th button behaviour (reset), which displays the menu options. */
  Mode::Set(modes[3]);
}

void loop() {
  bt->Update();

  /*  Declaring the array as static prevents unnecessary overwriting of memory at
every iteration of the loop.
      The array is initialized only once. It's better than using global variable
because it's close to where it's used.  */
  static int switch_id[] = { SW_1, SW_2, SW_3, SW_4 };
  if(ReadEmoroHardware() & SW_AVAILABLE)    // if switches are available
    for (int i = 0; i < 4; ++i)
      if (ReadSwitch(switch_id[i]))
        /*  static Set method below sets the currently active mode.  */
        Mode::Set(modes[i]);

  /* Each of the modes has its' own implementation of the Update function.
     static Get() method below returns the currently active mode.  */
  Mode::Get()->Update();
}
```

Assignment2.h

```
#ifndef ASSIGNMENT2_H
#define ASSIGNMENT2_H

#include "mode.h"

#define CLOCKWISE     1
#define ANTI_CLOCKWISE 2

extern Mode *modes[];
```

```
#endif
```

bluetooth.cpp

```
#include "includes.h"

Bt::Bt(char name_[16], char pin[16], bool d)
  : debug(d)
{
  if(!(ReadEmoroHardware() & BLUETOOTH_AVAILABLE)) {
    Serial.println("Bluetooth is not available");
  }
  else if (Bluetooth.changeNameAndPasskey(name_, pin)) {
      Serial.println("Failed setting bluetooth name and passkey");
  }
  BT_SERIAL.setTimeout(50); // default timeout is 1000ms so it would make the bt
communication slow
}

void Bt::CheckProtocol(String str) {
  if (str.length() == 1){
    char c = str[0];
    switch(c){
      /* Bunch of functions to remotely and manually control the car. Just for
convenience of use.  */
      case 's': drive.Stop();             break;
      case 'f': drive.Forward();          break;
      case 'b': drive.Backward();         break;
      case 'l': drive.Left();             break;
      case 'r': drive.Right();            break;
      case 'a': drive.Left(true);         break; // sharp
      case 'd': drive.Right(true);        break;
      case 'z': drive.Left(false, true);  break; // back (not sharp)
      case 'c': drive.Right(false, true); break;
    }

    /*  When number between 1 and 4 is received then call corresponding
funcitonality of a switch.  */
    if (c >= '1' && c <= '4')
        Mode::Set(modes[c - '1']);
  }

  /*  Set the speed of the car (max 1000, it's 1000 by default)  */
  if (str.startsWith("spd="))
      drive.SetSpeed(str.substring(4).toInt());

  /*  Turn at the specified number of deegres.
      Positive number supplied = move to the right
      Negative number supplied = move to the left  */
  if (str.startsWith("a=")) {
    int angle=0;
    sscanf(str.c_str(), "a=%d", &angle);
    // TurnAtAngleRelative_GyroBased(angle, false);
    drive.Turn(angle);
  }

  /*  Same as "a=" above but will stop once the black line is spotted under the
tracker_sensor module.  */
```

```
  if (str.startsWith("a_stop_at_line=")) {
    int angle=0;
    sscanf(str.c_str(), "a_stop_at_line=%d", &angle);
    //TurnAtAngleRelative_GyroBased(angle, true);
    drive.Turn(angle, true);
  }

  /*  Requests and sends back battery level (used for adjusting time between turns
in square following function)  */
  if (str.equals("battery_level")) {
    BT_SERIAL.print("battery level = "); BT_SERIAL.print(ReadPowerSupply());
BT_SERIAL.println("V");
  }

  /*  Output direction - used for circle track following.  */
  if (str.startsWith("cr_dir"))
    reinterpret_cast<FollowCircle*>(modes[1])->OutputDirection();

  /*  Circle track direction.  */
  if (str.equals("cr_right"))
    reinterpret_cast<FollowCircle*>(modes[1])->SetClockwise();

  if (str.equals("cr_left"))
    reinterpret_cast<FollowCircle*>(modes[1])->SetAntiClockwise();

  /*  Update line lengths - used for rect track following.  */
  if (str.startsWith("lsl=")) {
    int len=0;
    sscanf(str.c_str(), "lsl=%d", &len);
    FollowRect * fr = reinterpret_cast<FollowRect*>(modes[2]);
    fr->SetLongSideLen(len);
  }

  if (str.startsWith("ssl=")) {
    int len=0;
    sscanf(str.c_str(), "ssl=%d", &len);
    FollowRect * fr = reinterpret_cast<FollowRect*>(modes[2]);
    fr->SetShortSideLen(len);
  }

  /*  Increase following sharpness - used for rect track following. */
  if (str.equals("ifs")) {
    FollowRect * fr = reinterpret_cast<FollowRect*>(modes[2]);
    fr->IncreaseFollowingSharpness();
  }

  if (str.equals("dfs")) {
    FollowRect * fr = reinterpret_cast<FollowRect*>(modes[2]);
    fr->DecreaseFollowingSharpness();
  }

  /*  Increase rebound size - used for rect track following. */
  if (str.equals("irs")) {
    FollowRect * fr = reinterpret_cast<FollowRect*>(modes[2]);
    fr->IncreaseReboundSize();
  }

  /*  Decrease rebound size - used for rect track following. */
  if (str.equals("drs")) {
```

```
    FollowRect * fr = reinterpret_cast<FollowRect*>(modes[2]);
    fr->DecreaseReboundSize();
  }

  /*  Increase power supply significance - used for rect track following. */
  if (str.equals("ipss")) {
    FollowRect * fr = reinterpret_cast<FollowRect*>(modes[2]);
    fr->IncreasePowerSupplySignificance();
  }

  /*  Increase power supply significance - used for rect track following. */
  if (str.equals("dpss")) {
    FollowRect * fr = reinterpret_cast<FollowRect*>(modes[2]);
    fr->DecreasePowerSupplySignificance();
  }

  /*  Outputs all settings - used for rect track following. */
  if (str.equals("out_set")) {
    FollowRect * fr = reinterpret_cast<FollowRect*>(modes[2]);
    fr->OutputAllSettings();
  }


  /*  Allows to examine the exact contents of the string by outputting its' ascii
values through serial.
      This way bytes like '\n' and '\r' can be spotted.  */
  if (debug) {
    Serial.println("Protocol::Check, str consists of:");
    for (int i = 0; i < str.length(); i++)
      Serial.println(str[i],DEC);
  }
}


/*  Reads the string received through bluetooth.
    It also passes input of Serial monitor through bluetooth
    (used for communication testing purposes in the early development stage)  */
void Bt::Update() {
  if (Bluetooth.connection()) {
    String rec;

    while (BT_SERIAL.available())
      rec += BT_SERIAL.readString();

    if (rec.length()) {
      rec.trim(); // removes \r\n and other leading/trailing whitespace, ref:
https://www.arduino.cc/reference/en/language/variables/data-
types/string/functions/trim/

      if (debug) {
        Display::Msg(rec);
        Serial.println("BT rec: " + rec);
      }

      CheckProtocol(rec);
    }

    while (Serial.available())
      BT_SERIAL.write(Serial.read());
```

```
  }
}
```

bluetooth.h

```
#ifndef BLUETOOTH_H
#define BLUETOOTH_H

#define BT_SERIAL Serial1

#include <Arduino.h>

class Bt {
  bool debug;
public:
  Bt(char name_[16], char pin[16], bool debug = false);
  void CheckProtocol(String s);
  void Update();
};

#endif
```

display.cpp

```
#include "display.h"

namespace Display {
  void Msg(String msg) {
    Lcd.clear();
    Lcd.locate(0,0); Lcd.print(msg);
  }

  void Msg(String msg_row_1, String msg_row_2) {
    Lcd.clear();
    Lcd.locate(0,0); Lcd.print(msg_row_1);
    Lcd.locate(1,0); Lcd.print(msg_row_2);
  }

  void Msg(String top_left, String top_right, String bottom_left, String
bottom_right) {
    Lcd.clear();
    Lcd.locate(0,0); Lcd.print(top_left);
    int pos = 16 - top_right.length();
    Lcd.locate(0, pos < 8 ? 8 : pos); Lcd.print(top_right);
    Lcd.locate(1,0); Lcd.print(bottom_left);
    pos = 16 - bottom_right.length();
    Lcd.locate(1, pos < 8 ? 8 : pos); Lcd.print(bottom_right);
  }
}
```

display.h

```
#ifndef DISPLAY_H
#define DISPLAY_H
```

```
#include <Arduino.h>

namespace Display {
  extern void Msg(String msg);
  extern void Msg(String msg_row_1, String msg_row_2);
  extern void Msg(String top_left, String top_right, String bottom_left, String
bottom_right);
}

#endif
```

drive.cpp

```
#include "includes.h"

#define LEFT_WHEEL SERVO_1
#define RIGHT_WHEEL SERVO_0

#define BACKWARD  STOP - speed_
#define STOP      1500
#define FORWARD   STOP + speed_
#define FIX_DIRECTION(x, y) x == LEFT_WHEEL ? y - (y - STOP)*2 : y

#define MAX_SPEED 1000

Drive drive;

Drive::Drive():
  speed_(MAX_SPEED)
{

}

void Drive::SetSpeed(int s) {
  speed_ = s;
  if (speed_ > MAX_SPEED) speed_ = MAX_SPEED;
  Serial.println("Speed set to = " + String(speed_));
}

void Drive::WheelsServosWrite(int left_val, int right_val) {
  Serial.println("WheelsServosWrite left_val = " + String(left_val) + ", right_val
= " + String(right_val));

  EmoroServo.write(LEFT_WHEEL, FIX_DIRECTION(LEFT_WHEEL, left_val));
  EmoroServo.write(RIGHT_WHEEL, FIX_DIRECTION(RIGHT_WHEEL, right_val));
}

void Drive::Stop() { WheelsServosWrite(STOP, STOP); }
void Drive::Forward() { WheelsServosWrite(FORWARD, FORWARD); }
void Drive::Backward() { WheelsServosWrite(BACKWARD, BACKWARD); }

/* The divider determines how much 1 of the wheels slows down during the turn.
   If sharp arg is true then 1 of the wheels turns backwards.
   Back parameters allows for convenient control of the car using bluetooth.  */
void Drive::Left(bool sharp, bool back, float divider) {
  if (sharp) {
    WheelsServosWrite(BACKWARD, FORWARD);
  }
  else {
```

```
    if (!back) WheelsServosWrite(FORWARD - speed_/divider, FORWARD); //
file:///C:/Users/mb19424/Downloads/819-Article%20Text-1846-1-10-20170130.pdf
    else WheelsServosWrite(BACKWARD + speed_/divider, BACKWARD);
  }
}

/* Equivalent to Left function */
void Drive::Right(bool sharp, bool back, float divider) {
  if (sharp) {
    WheelsServosWrite(FORWARD, BACKWARD);
  }
  else {
    if (!back) WheelsServosWrite(FORWARD, FORWARD - speed_/divider); //
file:///C:/Users/mb19424/Downloads/819-Article%20Text-1846-1-10-20170130.pdf
    else WheelsServosWrite(BACKWARD, BACKWARD + speed_/divider);
  }
}

/*  Rebound is required when the Turn function recognizes line underneath the
sensor.
    That is because the following stop of the turn happens too late.  */
void Drive::Rebound(bool right, int size_) {
  unsigned long start_time = millis();
  BT_SERIAL.println("Drive::Rebound start");

  Stop();
  delay(80);
  if (right) Turn(-size_, true);
  else       Turn(size_, true);

  Stop();
  BT_SERIAL.println("Drive::Rebound took " + String(millis() - start_time) + "ms.
right = " + String(right) + ", size_ = " + String(size_));
}


/* Designed mainly for 90deg turns using top speed, other angles may not be very
accurate
   45 deg turn takes around 300ms
   90 deg turn takes around 470ms (but delay of 470 is not enough, 12 turns
resulted in 90 degrees discrepancy, (470 + 470/12) was tested and working
properly, 510ms is used because of that)
   180 deg turn takes around 940ms

   Returns true if the line was found during the turn (assuming that stop_at_line
argument was true, otherwise return is always false). */
bool Drive::Turn(int deg, bool stop_at_line, int rebound_size) {
  bool is_direction_right = deg > 0;
  unsigned long delay_time = (int)((float)abs(deg) * 5.67); // 510ms / 90 degrees

  if (is_direction_right)
    Right(true);
  else
    Left(true);

  if (stop_at_line) {
    /*  If the turn supposed to end when black line is recognized under the sensor
then
        checks must be made during the turn, therefore "delay" function can't be
```

```
used.
        For that reason the while loop below checks the time passed and reads the
state of
        the front infra red sensors.  */
    unsigned long start_time = millis();
    while (millis() - start_time < delay_time) {
      if (tracker_sensor->IsOnTheLine()){
        BT_SERIAL.println("Drive::Turn - LINE FOUND");
        Rebound(is_direction_right, rebound_size);
        return true;
      }
    }
  }
  else {
    /*  If stopping at line is not done then no additional actions have to be done
while waiting. So simple delay is used.  */
    delay(delay_time);
  }
  Stop();
  return false;
}

/*  Inaccurate, probably need to delete it  */
void Drive::Turn_GyroBased(double angle, bool stop_at_line) {
  int cur_angle = (int)Gyr.readDegreesZ();
  int end_angle = (cur_angle + (int)angle) % 360;
  BT_SERIAL.println("Current angle = " + String(cur_angle) + ", angle provided = "
+ String((int)angle) + ", end angle = " + String(end_angle));
  unsigned long start_time = millis();
  TurnAbsolute_GyroBased((double)end_angle, stop_at_line);
  BT_SERIAL.println(String((int)angle) + "deg. angle turn took " + String(millis()
- start_time) + "ms." );
}

/*  Inaccurate, probably need to delete it  */
void Drive::TurnAbsolute_GyroBased(double angle, bool stop_at_line) {
  Serial.println("TurnAtAngle");
  if (GetAnglesDiff(angle, Gyr.readDegreesZ()) > 0) {
    Right(true);
    while (GetAnglesDiff(angle, Gyr.readDegreesZ()) > 0) {
      int cur_ang = (int)Gyr.readDegreesZ();
      BT_SERIAL.println("angle = " + String((int)angle) + ", Gyr.readDegreesZ = "
+ String(cur_ang) + ", GetAnglesDiff = " + String((int)GetAnglesDiff(angle,
Gyr.readDegreesZ())));
      BT_SERIAL.println(tracker_sensor->StrValues());
      if (stop_at_line && tracker_sensor->IsOnTheLine()) {
        Stop();
        Display::Msg("LINE FOUND");
        BT_SERIAL.println("LINE FOUND");
        delay(1000);
        break;
      }
    }
  }
  else {
    Left(true);
    while (GetAnglesDiff(angle, Gyr.readDegreesZ()) < 0) {
      if (stop_at_line && tracker_sensor->IsOnTheLine()) {
        Stop();
```

```
        Display::Msg("LINE FOUND");
        delay(1000);
        break;
      }
    }
  }
  Stop();
  Serial.println("TurnAtAngle (return)");
}
```

drive.h

```
#ifndef DRIVE_H
#define DRIVE_H

#include <Arduino.h>

class Drive {
private:
  int speed_;

  void Rebound(bool right, int size_);

public:
  Drive();
  void SetSpeed(int s);

  void WheelsServosWrite(int left_val, int right_val);

  /* Set of functions that mainly call WheelsServosWrite with correct values. */
  void Stop();
  void Forward();
  void Backward();
  void Left(bool sharp = false, bool back = false, float divider = 2.0);
  void Right(bool sharp = false, bool back = false, float divider = 2.0);

  /* Function that is different from "Left/Right" functions.
     It waits until the turn is done, and allows to stop the turn if the line is
recognized under the 3-sensors module. */
  bool Turn(int deg, bool stop_at_line = false, int rebound_size = 20);

  /* Gyroscope sensor is inaccurate, the Turn function (based on time estimate) is
better for that. */
  void TurnAbsolute_GyroBased(double angle, bool stop_at_line = false);
  void Turn_GyroBased(double angle, bool stop_at_line = false);
};

/*  "Externing" the variable this way will allow to use the object in all the
files that import this file.  */
extern Drive drive;

#endif
```

includes.cpp

```
#include "includes.h"

int GetAnglesDiff(double a1, double a2) {
```

```
  //https://stackoverflow.com/a/36001014
  return 180 - (180 - (int)a1 + (int)a2) % 360;
}
```

includes.h

```
/* This file allows to include all the necessary header files concisely using a
single "#include" statement. */

#ifndef INCLUDES_H
#define INCLUDES_H

#include "Assignment2.h"

/* Bluetooth class allows controlling the car remotely and debugging its' state
conveniently.
   Its' use is straightforward and includes:
   - initialization
        Bt *bt = new Bt("EMoRo mb19424", "3737");
   - 1 instruciton to be called in a loop
        bt->Update();
   - sending messages using BT_SERIAL alias of Serial1
        BT_SERIAL.println("Some var =" + String(some_var));  */
#include "bluetooth.h"

/* Display namespace encapsulates "Msg" functions with 1,2 and 4 parameters. */
#include "display.h"

/* Drive class provides the functions to control the movement of Emoro robot. */
#include "drive.h"

/* Mode is an abstract class. Thanks to it (and the use of polymorphism),
   different modes can be switched in a concise way. */
#include "mode.h"

/*  Set of subclasses implementing the Mode abstract class.  */
#include "mode_AvoidObstacles.h"
#include "mode_FollowCircle.h"
#include "mode_FollowRect.h"
#include "mode_ResetMode.h"

/* TrackerSensor class holds functions to get state of the 3 sensors from the
following module: https://www.ebay.co.uk/itm/4Pcs-3-Way-Infrared-IR-Line-Tracking-
Sensor-Module-For-Raspberry-Pi-U3/193184039763
   It can be done using tracker_sensor.GetAll method.  */
#include "tracker_sensor.h"

extern int GetAnglesDiff(double a1, double a2);

#endif
```

mode.cpp

```
#include "mode.h"

Mode * Mode::current = nullptr;

void Mode::Set(Mode *m) {
```

```
  if (current)
    current->Reset();

  current = m;
  current->Init();
}
```

mode.h

```
#ifndef MODE_H
#define MODE_H

#include <Arduino.h>

class Mode {
  static Mode *current;

protected:
  bool presentation;

public:
  Mode() : presentation(false) {};
  static void Set(Mode *m);
  static Mode * Get() { return current; }

  virtual void Init() = 0;
  virtual void Update() = 0;
  virtual void Reset() = 0;
};

#endif
```

mode_AvoidObstacles.cpp

```
#include "includes.h"

#define OBSTACLE_DISTANCE_THRESHOLD 50

void AvoidObstacles::Init() {
  Display::Msg("moving forward");
}

void AvoidObstacles::Update() {

  /*
  int val = Ultrasonic.read(GPP_0);
  -Result of function:
  (0-399)-Sensor distancein cm
  (400)-Sensor out of range
  (-1) -Error: Argument „port“ out of range: [GPP_0-GPP_7]
  (-2)-Error: Ultrasonic sensor is notinitialized
  */

  static unsigned long last_display_update = 0;
  if (millis() - last_display_update > 100) {
    int dist = Ultrasonic.read(GPP_0);
    bool is_obstacle = dist < OBSTACLE_DISTANCE_THRESHOLD;
```

```
    Display::Msg(is_obstacle ? "obstacle detected" : "moving forward", "GPP_0 = "
+ String(dist));
    last_display_update = millis();

    if (is_obstacle) {
      drive.Stop();

      /*  Make 3 quick low sound effects to notify the usef about detected
obstacle.  */
      PlayDetectionSound();

      /*  Move 360 degrees around in order to find the best angle.  */
      double best_angle = FindBestAngle();
      String cur_angle = String((int)Gyr.readDegreesZ());

      Display::Msg("best = ", String((int)best_angle), "current = ", cur_angle);
      BT_SERIAL.println("Best angle = " + String((int)best_angle) + ", current
angle = " + cur_angle);
      drive.Stop();

      delay(presentation ? 3000 : 100);

      int diff = GetAnglesDiff(best_angle, Gyr.readDegreesZ());
      Display::Msg("Turning " + String(diff), "degrees");
      BT_SERIAL.println("Turning " + String(diff) + " degrees");

      delay(presentation ? 3000 : 100);

      drive.Turn(diff);
      drive.Stop();
    }
  }

  drive.Forward();
}

void AvoidObstacles::Reset() {

}


/*  Turn around 360 degrees while repetitively checking ultrasonic distance
sensor, trying to find optimal direction to follow.
    (optimal direction in terms of avoiding obstacles)

    Gyroscope sensor is used to check the current angle.  */
double AvoidObstacles::FindBestAngle() {
  /*  diagnostic_reading_counter increases each time the Ultrasonic sensor reading
is done during the turn.
      Low number would indicate possibly inaccurate estimation of the best angle.
*/
  int diagnostic_reading_counter = 0;

  double start_angle = Gyr.readDegreesZ();
  double angle_where_max_dist;
  int max_dist = 0;

  BT_SERIAL.println("start_angle = " + String((int)start_angle));
```

```
  bool angle_exceeded_360 = false;
  double new_angle, last_angle;
  int dist;
  drive.Right(true);
  delay(200);
  while ((new_angle = Gyr.readDegreesZ()) < start_angle || !angle_exceeded_360) {
    /*  + 10 to avoid false detection of full circle while standing still due to
fluctuating sensor reading  */
    if ((new_angle + 10.0) < last_angle) {
      if (last_angle < start_angle && angle_exceeded_360) {
        break;
      }
      angle_exceeded_360 = true;
    }

    /* Save new largest distance position (and the distance itself) */
    if ((dist = Ultrasonic.read(GPP_0)) > max_dist) {
      max_dist = dist;
      angle_where_max_dist = new_angle;

      /* Make sound (with higher pitch than the one when obstacle was detected).
         Using sound instead of display/serial message gives better insight in
this case.
         (because we can't look at 2 things at once and time plays big role here)
*/
      tone(BUZ_BUILTIN, 131, 50); // NOTE_C3  131
    }
    last_angle = new_angle;

    static int last_dist = 0;
    if (dist != last_dist || dist == 400)
      diagnostic_reading_counter++;
    last_dist = dist;
  }
  BT_SERIAL.println("angle_where_max_dist = " + String((int)angle_where_max_dist)
+ ", max_dist = " + String(max_dist) + ", number of distinct readings done during
360 turn = " + String(diagnostic_reading_counter));
  return angle_where_max_dist;
}


void AvoidObstacles::PlayDetectionSound() {
  /*  NOTE_C1 reference:
      https://github.com/inovatic-ict/emoro-2560-
library/blob/master/examples/02.%20Digital/toneMelody/pitches.h   */
  tone(BUZ_BUILTIN, 33, 50); // NOTE_C1  33
  delay(100);
  tone(BUZ_BUILTIN, 33, 50); // NOTE_C1  33
  delay(100);
  tone(BUZ_BUILTIN, 33, 50); // NOTE_C1  33
  delay(500);
}
```

mode_AvoidObstacles.h

```
#ifndef AVOIDOBSTACLES_H
#define AVOIDOBSTACLES_H

#include "mode.h"
```

```
class AvoidObstacles : public Mode {
  double FindBestAngle();

  void PlayDetectionSound();
public:
  void Init() override;
  void Update() override;
  void Reset() override;
};

#endif
```

mode_FollowCircle.cpp

```
#include "includes.h"

#define DIR_DETECTION_TURN_THRESHOLD 3

void FollowCircle::Init() {
  Display::Msg("Circle track", "following.");
  BT_SERIAL.print("Battery level at the begining of circle run = ");
BT_SERIAL.print(ReadPowerSupply()); BT_SERIAL.println("V");
  drive.SetSpeed(600);
  drive.Forward();
}

void FollowCircle::Update() {
  bool l_,c,r;
  tracker_sensor->GetAll(l_,c,r);

  if (!l_ && c && !r) {
    if (!direction)
      drive.Forward();
    else
      SlightTurnTechnique();

    last_iteration_had_right_turn = false;
    last_iteration_had_left_turn = false;
  }

  if (l_ && !r) {
    drive.Left(false, false, direction ? GetOptimalSharpness(ANTI_CLOCKWISE) :
1.8);

    if (!last_iteration_had_left_turn){
      number_of_left_turns++;
      OutputTurnsCounts();

      last_iteration_had_left_turn = true;
      last_iteration_had_right_turn = false;
    }
  }


  if (!l_ && r) {
    drive.Right(false, false, direction ? GetOptimalSharpness(CLOCKWISE) : 1.8);

    if (!last_iteration_had_right_turn) {
```

```
      number_of_right_turns++;
      OutputTurnsCounts();

      last_iteration_had_right_turn = true;
      last_iteration_had_left_turn = false;
    }
  }

  if (!direction) {
    if (number_of_left_turns >= DIR_DETECTION_TURN_THRESHOLD)
      SetDirection(ANTI_CLOCKWISE);

    if (number_of_right_turns >= DIR_DETECTION_TURN_THRESHOLD)
      SetDirection(CLOCKWISE);
  }
}

void FollowCircle::Reset() {
  number_of_left_turns = number_of_right_turns = NULL;
  last_iteration_had_left_turn = last_iteration_had_right_turn = false;
  direction = NULL;
}

void FollowCircle::SetDirection(int dir) {
  direction = dir;
  BT_SERIAL.println("The number of " + String((dir == CLOCKWISE ? "right" :
"left")) + " turns exceeded threshold. " + (dir == CLOCKWISE ? "Clockwise" :
"Anti-Clockwise") + " direction was detected.");
  DirectionFoundSoundEffect();
  drive.SetSpeed(1000);
}

/*  Knowing the direction, slightly turn to the inside of the circle even when the
sensor indicates being centrally over the black line.
    This way the movement is smooth.  */
void FollowCircle::SlightTurnTechnique() {
  if (direction == CLOCKWISE)
    drive.Right(false, false, 2.0);
  else
    drive.Left(false, false, 2.0);
}

float FollowCircle::GetOptimalSharpness(int dir) {
  return direction == dir ? 1.5 : 5.0;
}

void FollowCircle::OutputDirection() {
  if (!direction)
    BT_SERIAL.println("direction = NULL");
  else
    BT_SERIAL.println("direction = " + String(direction == CLOCKWISE ? "CLOCKWISE"
: "ANTI-CLOCKWISE"));
}

void FollowCircle::OutputTurnsCounts(){
  if (!direction)
    BT_SERIAL.println("L=" + String(number_of_left_turns) + ", R=" +
String(number_of_right_turns));
}
```

```
/* Adapted from: https://www.hackster.io/jrance/super-mario-theme-song-w-piezo-buzzer-and-arduino-1cc2e4 */
void FollowCircle::DirectionFoundSoundEffect() {
  if (mario_sound) {
    drive.Stop();
    static int melody[] = {
      2637, 2637, 0, 2637,
      0, 2093, 2637, 0,
      3136, 0, 0,  0,
      1568, 0, 0, 0
    };

    static int size = sizeof(melody) / sizeof(int);
    for (int i = 0; i < size; i++) {
      tone(BUZ_BUILTIN, melody[i], 83);
      delay(108);
      tone(BUZ_BUILTIN, 0, 83);
    }
    delay(200);
    SlightTurnTechnique();
    return;
  }

  // C2
  tone(BUZ_BUILTIN, 65, 200);
}
```

mode_FollowCircle.h

```
#ifndef FOLLOWCIRCLE_H
#define FOLLOWCIRCLE_H

#include "mode.h"
#include "Assignment2.h"

class FollowCircle : public Mode {
  bool last_iteration_had_left_turn, last_iteration_had_right_turn, mario_sound;
  int number_of_left_turns, number_of_right_turns, direction;

  float GetOptimalSharpness(int dir);
  void OutputTurnsCounts();
  void DirectionFoundSoundEffect();
  void SetDirection(int dir);
  void SlightTurnTechnique();
public:
  FollowCircle() : direction(NULL), number_of_left_turns(NULL),
number_of_right_turns(NULL), last_iteration_had_left_turn(false),
last_iteration_had_right_turn(false), mario_sound(true) {}

  void Init() override;
  void Update() override;
  void Reset() override;

  void SetClockwise() { direction = CLOCKWISE; }
  void SetAntiClockwise() { direction = ANTI_CLOCKWISE; }
  void OutputDirection();
};
```

```
#endif
```

mode_FollowRect.cpp

```
#include "includes.h"


void FollowRect::Init() {
  Display::Msg("Square track", "following.");
  BT_SERIAL.print("Battery level at the begining of rect run = ");
BT_SERIAL.print(ReadPowerSupply()); BT_SERIAL.println("V");
  Reset();
}

void FollowRect::Update() {
  if (is_first_turn && !start_time) {
    /*  ReadPowerSupply has to be read once at the begining,
        reading it during operation sometimes results in lower readings,
        it's unstable.  */
    float voltage = ReadPowerSupply();
    voltage_adjustment_mult = power_supply_significance * (voltage - 6.6);
    BT_SERIAL.print("time mult = "); BT_SERIAL.print(voltage_adjustment_mult);
BT_SERIAL.print(", voltage = "); BT_SERIAL.print(voltage); BT_SERIAL.println("V");
    AssignEstimatedTravelTime();
  }

  if (!start_time)
    start_time = millis();

  bool l_,c,r;
  tracker_sensor->GetAll(l_,c,r);

  int time_since_last_turn = millis() - start_time;

  /*  Make sound at which point the front sensor readings should be ignored.  */
  if (!is_near_end && time_since_last_turn > is_near_end_threshold_time) {
    /*  262 is NOTE_C4 in the pitches.h file from toneMelody example provided by
Emoro library
        65 is NOTE_C2 */
    tone(BUZ_BUILTIN, 65, 200);
    is_near_end = true;
  }

  /*  Make another sound using higher pitch when its time to turn.  */
  if (!reached_end && time_since_last_turn > travel_time) {
    /*  523 is NOTE_C5 in the pitches.h file from toneMelody example provided by
Emoro library
        131 is NOTE_C3  */
    tone(BUZ_BUILTIN, 131, 200);
    reached_end = true;
  }

  /*  Drive forward if the sensor shows correct position over the line.
      If the car is close to the turning spot, then blindly go forward until that
spot is reached.  */
  if (!l_ && c && !r || is_near_end)
```

```
    drive.Forward();

  /*  Make slight corrections (left/right) but only if the car is not close to the
turning spot.
      The sharpness of the turns can be adjusted through bluetooth.
      Using this method it was determined that 2.3 is around optimal value for it.
*/
  if (!is_near_end) {
    if (l_ && !r)
      drive.Left(false, false, following_sharpness_div); // params: is_sharp
(should 1 of the wheels move backward), back (should the car reverse while
turning), speed_decrease_divider (how sharp the turn is)

    if (!l_ && r)
      drive.Right(false, false, following_sharpness_div);
  }

  /*  Recognize which what to turn but only before the first turn.
      Rarely (1 in around 20 turns) this is misrecognized so that's why it's done
only once. */
  if (is_near_end && is_first_turn) {
    if (l_ && c && !r)
      turn_direction = ANTI_CLOCKWISE;

    if (!l_ && c && r)
      turn_direction = CLOCKWISE;
  }

  /*  If the car reached the turning spot then turn and update the information
about next turning spot.
      It also sends diagnostic information through bluetooth, allowing the method
to be adjusted.
      This debugging method allowed to adjust time calculation due to battery
voltage level.
      (which results in higher speed of the car, that is a factor that is
currently taken into consideration by the code).  */
  if (reached_end) {
    drive.Stop();
    Display::Msg("Sharp turning", "90 degrees.");
    bool was_line_encountered = drive.Turn(135 * (turn_direction == CLOCKWISE ? 1
: -1), true, rebound_size); // true = stop_at_line
    Display::Msg("Square track", "following.");

    side_len = (side_len == long_side_len ? short_side_len : long_side_len);
    BT_SERIAL.println("side_len = " + String(side_len));

    AssignEstimatedTravelTime();
    is_first_turn = is_near_end = reached_end = false;
    start_time = NULL;


  }
}


void FollowRect::Reset() {
  start_time = NULL;
  travel_time = NULL;
  side_len = long_side_len;
```

```
  is_near_end_threshold_time = NULL;
  voltage_adjustment_mult = NULL;
  turn_direction = NULL;
  is_first_turn = true;

  is_near_end = false;
  reached_end = false;

  following_sharpness_div = 2.0;
  following_sharpness_div += voltage_adjustment_mult;
}


/*  It was observed that the decrease in lenght of the line does not result in
direct equivalent decrease of the time required to reach the end.
    For example, the optimal time to travel:
      - 70cm took 3.5s (around 50ms per 1 cm)
      - 42cm took 2.3s (around 54ms per 1cm)

    In order to make rough estimate of the time required to travel the distance,
the code is using linear interpolation using 2 collected data points above.
    In particular, the "map" function is used to translate the length of the track
into estimated time required to travel to the end of it.

    e.g.
    map(lenght_of_the_track_to_be_translated,   // (example scenario: the track is
slightly longer, the task is to estimate how much time it will take to reach end
of it using previously collected data)
        shorter_length,                         // used in previous test
        longer_length,                          // used in previous test
        time_required_to_travel_shorter_length, // observed in previous test
        time_required_to_travel_longer_length)  // observed in previous test

    map(75.0
        42.0,
        70.0,
        2300.0,
        3520.0)

    The return of such map funtion would be: 3737.

    Even better approach (but more complex to implement) would be to perform more
timing tests and use cubic interpolation to estimate the time.
    It would allow for more precise estimation, which would be especially helpful
when the track length is very low.
    */
void FollowRect::AssignEstimatedTravelTime(){
  travel_time = map(side_len, 42, 70, 2300, 3520);
  AdjustTravelTimeByBatteryLevel();

  /*  Update at which point in time the car should ignore the front sensors.  */
  is_near_end_threshold_time = (int)((float)travel_time / (float)side_len *
((float)side_len - SENSORS_OFFSET));

  BT_SERIAL.println("travel_time = " + String(travel_time) + ", side_len = " +
String(side_len) + ", is_near_end_threshold_time = " +
String(is_near_end_threshold_time));
}
```

```
/*  Higher voltage results in lower estimated travel time.  */
void FollowRect::AdjustTravelTimeByBatteryLevel() {
  travel_time -= (travel_time * voltage_adjustment_mult);
}


void FollowRect::SetLongSideLen(int len) {
  long_side_len = len;
  BT_SERIAL.print("long_side_len = "); BT_SERIAL.println(long_side_len);
}

void FollowRect::SetShortSideLen(int len) {
  short_side_len = len;
  BT_SERIAL.print("short_side_len = "); BT_SERIAL.println(short_side_len);
}

void FollowRect::IncreaseFollowingSharpness() {
  following_sharpness_div -= 0.1;
  BT_SERIAL.print("following_sharpness = ");
BT_SERIAL.println(following_sharpness_div);
}

void FollowRect::DecreaseFollowingSharpness() {
  following_sharpness_div += 0.1;
  BT_SERIAL.print("following_sharpness = ");
BT_SERIAL.println(following_sharpness_div);
}

void FollowRect::IncreaseReboundSize() {
  rebound_size += 1;
  BT_SERIAL.print("rebound_size = "); BT_SERIAL.println(rebound_size);
}
void FollowRect::DecreaseReboundSize() {
  rebound_size -= 1;
  BT_SERIAL.print("rebound_size = "); BT_SERIAL.println(rebound_size);
}

void FollowRect::IncreasePowerSupplySignificance() {
  power_supply_significance += 0.01;
  BT_SERIAL.print("power_supply_significance = ");
BT_SERIAL.println(power_supply_significance);
}

void FollowRect::DecreasePowerSupplySignificance() {
  power_supply_significance -= 0.01;
  BT_SERIAL.print("power_supply_significance = ");
BT_SERIAL.println(power_supply_significance);
}


void FollowRect::OutputAllSettings() {
  BT_SERIAL.println("\n------------\nFollowRect SETTINGS:");
  BT_SERIAL.print("long_side_len = "); BT_SERIAL.println(long_side_len);
  BT_SERIAL.print("short_side_len = "); BT_SERIAL.println(short_side_len);
  BT_SERIAL.print("following_sharpness = ");
BT_SERIAL.println(following_sharpness_div);
  BT_SERIAL.print("rebound_size = "); BT_SERIAL.println(rebound_size);
  BT_SERIAL.print("power_supply_significance = ");
BT_SERIAL.println(power_supply_significance);
```

```
  BT_SERIAL.println("------------");
}
```

mode_FollowRect.h

```
#ifndef FOLLOWRECT_H
#define FOLLOWRECT_H

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

#define SENSORS_OFFSET         25.0

/*  new batteries provide around 8.95V (read using float-ReadPowerSupply function)
    old batteries provide around 6.6V
    The car drives faster with new batteries.
    It was observed that a car with batteries almost running out (6.6V when
checked with ReadPowerSupply)
    drives around 10% slower than a car with new batteries (8.95V when checked
with ReadPowerSupply).
    The exact optimal timings that were observed are supplied to the "map"
function in the AssignEstimatedTravelTime method. */

/*   Setting these worked well with low battery level (6.6V) (before compensation
method was implemented)
     #define LONG_SIDE_TRAVEL_TIME  3520
     #define SHORT_SIDE_TRAVEL_TIME 2300

   Setting that worked well with high battery level (8.95V) (before compensation
method was implemented)
     #define LONG_SIDE_TRAVEL_TIME  3168 // decreased by %10
     #define SHORT_SIDE_TRAVEL_TIME 2070      */

class FollowRect : public Mode {
  unsigned long start_time, travel_time, is_near_end_threshold_time;
  float voltage_adjustment_mult, following_sharpness_div,
power_supply_significance;
  bool is_near_end, reached_end, is_first_turn;
  int side_len, long_side_len, short_side_len, turn_direction, rebound_size;

  void AssignEstimatedTravelTime();
  void AdjustTravelTimeByBatteryLevel();

public:

  FollowRect(int ldl, int sdl) : long_side_len(ldl), short_side_len(sdl),
rebound_size(24), power_supply_significance(0.12) {}
  void Init() override;
  void Update() override;
  void Reset() override;

  void SetLongSideLen(int len);
  void SetShortSideLen(int len);
  void IncreaseFollowingSharpness();
  void DecreaseFollowingSharpness();
  void IncreaseReboundSize();
  void DecreaseReboundSize();
  void IncreasePowerSupplySignificance();
  void DecreasePowerSupplySignificance();
```

```
  void OutputAllSettings();
};



#endif
```

mode_ResetMode.cpp

```
#include "includes.h"

void ResetMode::Init() {
  drive.Stop();
  Display::Msg("1.Obst", "2.Circle", "3.Square", "4.Reset");
}

void ResetMode::Update() {}
void ResetMode::Reset() {}
```

mode_ResetMode.h

```
#ifndef RESETMODE_H
#define RESETMODE_H

#include "mode.h"

class ResetMode : public Mode {
public:
  void Init() override;
  void Update() override;
  void Reset() override;
};

#endif
```

tracker_sensor.cpp

```
#include "tracker_sensor.h"

TrackerSensor *tracker_sensor;

TrackerSensor::TrackerSensor(int l, int c, int r)
  : left_pin(l), center_pin(c), right_pin(r)
{
  pinMode(l, INPUT_PULLUP);  // left
  pinMode(c, INPUT_PULLUP);  // center
  pinMode(r, INPUT_PULLUP);  // right
}

/*  Read all 3 sensors from the board. Arguments passed by reference work similar
to pointers in plain C.
    It allows setting 3 "return values" at once. */
void TrackerSensor::GetAll(bool &l, bool &c, bool &r) {
  l = !digitalRead(left_pin);
  c = !digitalRead(center_pin);
  r = !digitalRead(right_pin);
}
```

```
/*  Function that allows convenient printing of sensor state. Useful for
debugging.  */
String TrackerSensor::StrValues() {
  bool l_,c,r;   GetAll(l_,c,r);
  return "tracker_sensor = " + String(l_) + " " + String(c) + " " + String(r);
}

/*  Check if the front sensor is directly over the line.  */
bool TrackerSensor::IsOnTheLine() {
  bool l_,c,r;    GetAll(l_,c,r);
  return (!l_ && c && !r);
}
```

tracker_sensor.h

```
#ifndef TRACKERSENSOR_H
#define TRACKERSENSOR_H

#include <Arduino.h>

class TrackerSensor {
  int left_pin;
  int center_pin;
  int right_pin;

public:
  TrackerSensor(int l_pin, int c_pin, int r_pin);

  void GetAll(bool &l_state, bool &c_state, bool &r_state);
  bool IsOnTheLine();
  String StrValues();
};

/*  "Externing" the variable this way will allow to use the object in all the
files that import this file.  */
extern TrackerSensor *tracker_sensor;

#endif
```