Robot Lab – Final Report

Andra Alazaroaie – student number: 5518318

1. Description of the used timer settings on the STM32.

I used the Arduino library, so no timer was used. To start the robot, accelerate and decelerate, I used the *analogWrite()* function. I set the desired duty cycles and added some delays to fulfil the 3-4 seconds requirements.

2. Description of the obstacle detection algorithm.

For obstacle detection, I used the Arduino library with interrupts to detect the changes in the states of the ECHO pin of the ultrasound sensor. Every 100 milliseconds, I send a short ultrasonic pulse from the TRIG pin of the sensor. This is done by first clearing the TRIG pin by setting it to LOW for 2 microseconds, using *digitalWrite()* to HIGH from the Arduino library, then a delay of 10 microseconds until we set it to LOW again. At the moment after the pulse being transmitted, the ECHO pin becomes HIGH. The pulse will travel with the speed of sound until reaching the obstacle (or until a timeout after 38ms), then the reflected wave will travel back to the sensor and the ECHO pin will receive it, becoming LOW again. These changes of state on the ECHO pin are detected by our interrupt and the interrupt service routine is executed when they take place. We keep the time in microseconds of when ECHO became HIGH as start time and when it becomes LOW again we keep an end time and update a flag telling us that we have a new distance available. In the loop of the program, we check this flag and if it said we have a new distance available, we calculate it by using the basic formula of *distance = speed * time* and the *speed of light (340m/s).* The time is the difference between the end time and start time that we remembered. We also have to divide the distance by 2 at the end, as during the time we recorded, the pulse travelled to the object, but also back. Doing the calculations as we want to get the distance in centimetres, we come to the formula of *distance = (end time – start time) / 58*. Then, we check if this distance is less than 15 centimetres, in which case we stop the motors. Otherwise, we keep going in front.

3. Description of the line detection algorithm.

My line detection algorithm also used the Arduino library and is quite simple. The infrared reflective sensors emits infrared light, which is reflected on the surface, back to the sensor. Then, the sensor gives a value dependent on the reflectance of the surface. In the loop of the program, I read both infrared sensors using the *analogRead()* function of the library, receiving 2 values, the reflectance values detected by the left, respectively right sensor. The next step is determining when the sensors actually detect a dark line. Experimentally, I came up with a threshold value of 80 on the grading tracks, using some sort of binary search, knowing the values are between 0 and 1023. When the sensors give a value higher than my 80 threshold, I consider that the respective sensor detected the line. Hence, if both values are below the threshold, I should keep going in front, as the line must be between the two sensors. If only the value detected by the left sensor is above the threshold, it means the line is going to the left, so I should slowly turn left. Similarly, I should turn right when only the right sensor reads a value over the threshold. It shouldn't happen that both values are above the threshold, yet in this case I also go front. For turning left or right, I make one motor go forward slightly faster than the other motor, which goes backwards. To make sure I can do small, detailed tracks, I decided to go very slow and stop for a very small amount of time every time I change direction. This improved the line follower on sharp or consecutive turns, as I give the sensors more time to detect what's necessary. One more thing important to mention would be that on top of the obstacle detection algorithm explained before, in the function getting the distance detected by the ultrasound sensors on an echo pin change, to avoid jitter, I keep the previously detected distance and use it in the cases of a very large distance being detected, but also when calculating the new distance in a weighted mean, to avoid spikes in the distance calculation. This is meant to make the sensors a bit more accurate.

4. Description of the obstacle avoidance algorithm.

The obstacle avoidance algorithm is the most complex one and the one that also required the most tuning. It reuses a lot of code and ideas from the obstacle detection algorithm, but also from the line detection algorithm. Yet,

this time, the obstacle detection algorithm is implemented for 2 ultrasound sensors, one at the front of the robot in order to stop when first detecting the obstacle and one on the right side in order to help with the avoidance. There are two interrupts attached, one for each sensor, each with its own interrupt service routine, as well as two triggering functions, one for triggering each ultrasound. The moments in which we trigger each sensor depends on the state we are in. I implemented a finite state machine with 5 states. The first one is ON_LINE, which is the beginning, when we only have to do normal line following, but also apply obstacle detection on the front ultrasound sensor. When we detect a distance smaller than 15cm, we stop the robot and enter a change state. We turn the robot 90 degrees left (the degrees have been measured experimentally by applying a delay of 500 milliseconds when turning left) , then go forward for a small time before we go to the next step: BEFORE_BOX. In this state, we are in front of the obstacle, parallel with it and we are going forward until our obstacle detection algorithm no longer detects anything closer than 25cm. It means we are now past the obstacle from its front side and we can stop, turn right and go a bit in front in order to be able to detect the obstacle with the side sensor again, this time from its left side. We now move to the LEFT_OF_BOX state, in which we proceed the same: we go forward until our side sensor no longer detects the obstacle, then we turn right 90 degrees and move to the next state: AFTER_BOX. In this state, we are past the obstacle and we need to trace back the line. Thus, we go forward until detecting the line. Detecting the line is when at least one of the infrared sensors finds a value greater than the threshold described in the line detection algorithm. Now we stop, turn left 90 degrees and change state again. This is the final state: ON_LINE_AFTER, in which we assume we are correctly oriented and just apply the line follower. With these 5 states, we went through 3 sides of the obstacle in the shape of 3 straight lines. The first and last states were the line followers. The algorithm is general, being able to avoid rectangular objects, going parallel to its sides, while with other shapes (such as cylinders) it still works, as we are using the side sensor to detect where the obstacle starts and ends.

5. Description of the algorithm on calculating the traveling distance.

For fixed distance travelling, the essentials were the 2 speed encoders. Most of my code of the travelling on fixed distance algorithm is on the line following

part and is identical to the line detection algorithm. The change is in my function for going in front, as I have to measure only the distance recorded by the speed encoders when going in front. Any direction change that happens to be able to perfectly follow the line does not increase the distance travelled, but it would influence the speed encoders, since they record what happens with the wheels moving when the motors are on. These small wheels have 20 gaps (directly measured) all over their length, which the rotary encoders detect in order to allow the calculation of the number of rotations. We call these gaps steps that the encoders make. The diameter of the wheel is 7cm, so its radius is 3.5cm. One length of the wheel is *2*PI*radius*, which we divide by the number of steps to obtain the distance between 2 of the gaps. Then, by multiplying with the number of gaps counter by the encoders, we can obtain the total distance travelled by the robot. When this reaches the fixed distance that we set, we stop the robot. Now, the tricky part: how do the encoders count the number of gaps? Well, at every loop, when the direction is going forward, we read both encoders with *digitalRead()* of the Arduino library and if they had a change of state, going from LOW to HIGH or from HIGH to LOW, we increase the counter of the respective encoder. We then make an average of the left and right one's counters in order to have a more precise approximation.