1 Hardware Änderungen am Asuro

1.1 Um die Genauigkeit der Sensoren zu erhöhen, wurde an den Liniensensoren jeweils ein Lichtschutz angebracht. Wie auf Übungsblatt 8 beschrieben, werden dadurch die Messungen weniger vom Umgebungslicht beeinflusst.

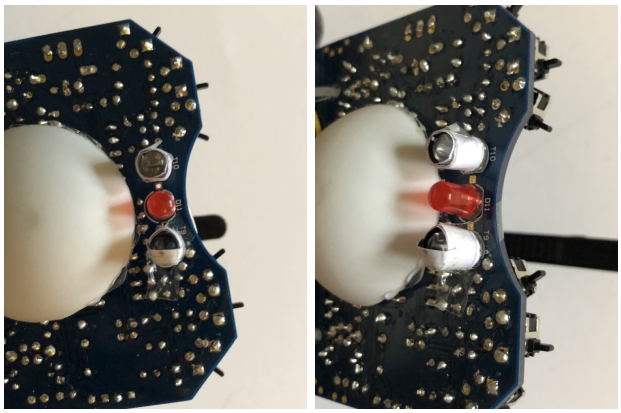


Abbildung 1: Umgebungslichtschutz für die Liniensensoren

1.2 Die Odometriesensoren messen die Schwarz-Weiß Übergänge der Encoderscheiben. Mir ist aufgefallen, dass die Encoderscheiben ziemlich viel Spielraum haben und so beim Fahren auf der Achse vor und zurückrutschen. Je weiter die Encoderscheiben nach außen rutschen, desto weniger Schwarz-Weiß Übergänge erkennen die Odometriesensoren. Um dieses Problem zu lösen, habe ich zwei Büroklammern auseinandergebogen und sie auf der Unterseite des Asuros so angebracht, das die Encoderscheiben nicht mehr so weit nach außen rutschen können.

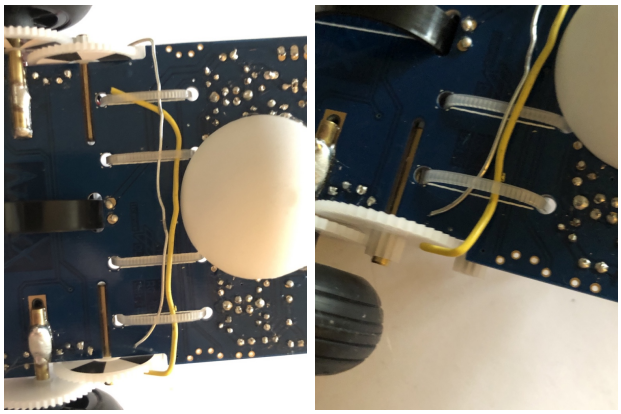
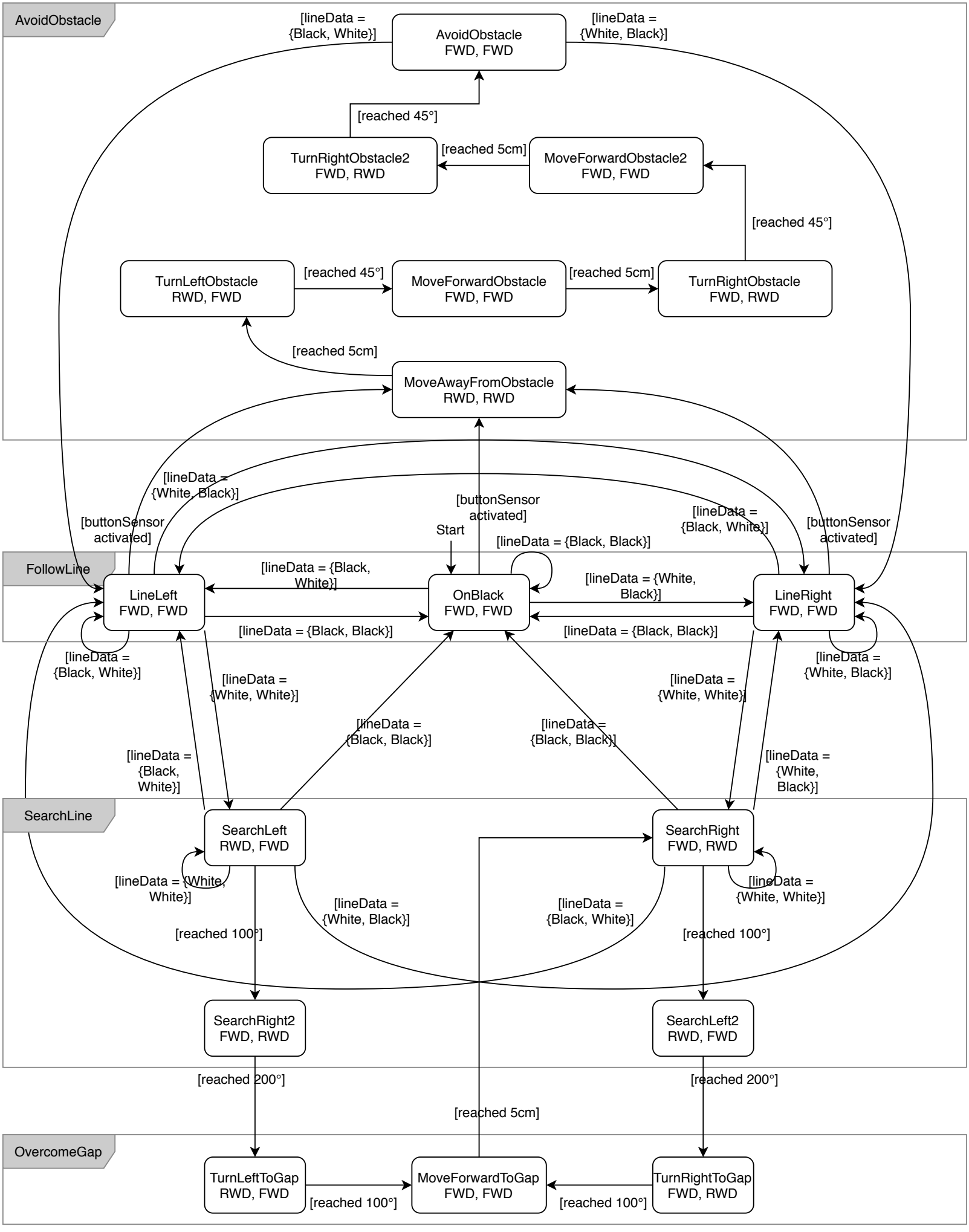


Abbildung 2: Büroklammerhalterung für die Encoderscheiben

2 Zustandsautomat

Der grundlegende Zustandsautomat besteht wie in Übungsblatt 8 aus den vier Zuständen AvoidObstacle, FollowLine, SearchLine und OvercomeGap. Um diesen vereinfachten Zustandsautomat implementieren zu können, habe ich einen detailierteren Zustandsautomat aus insgesammt 17 Zuständen erstellt. Jeder Zustand besitzt dabei seine eigenen Motorrichtungen und Geschwindigkeiten. Im Diagramm habe ich unter dem Namen des Zustandes angedeutet, in welche Richtungen sich die Motoren drehen. So dreht sich beispielsweise im Zustand SearchLeft der linke Motor rückwärts und der rechte vorwärts. Die Übergänge von einem Zustand zum nächsten werden immer von Sensormesswerten eingeleitet. Das können die Tastsensoren, Odometriesensoren, oder Liniensensoren sein.

buttonSensor activated	Tastsensoren wurden ausgelöst; Egal welcher
reached 100°/cm	Odometriesensoren haben einen gewissen Tickschwellwert überschritten
lineData={Black, White}	linker Liniensensor misst Schwarz und rechter Weiß


AvoidObstacle
[lineData = {Black, White}]
AvoidObstacle
FWD, FWD
[lineData = {White, Black}]
[reached 45°]
TurnRightObstacle2
FWD, RWD
[reached 5cm]
MoveForwardObstacle2
FWD, FWD
[reached 45°]
TurnLeftObstacle
RWD, FWD
[reached 45°]
MoveForwardObstacle
FWD, FWD
[reached 5cm]
TurnRightObstacle
FWD, RWD
[reached 5cm]
MoveAwayFromObstacle
RWD, RWD
FollowLine
[lineData = {White, Black}]
[buttonSensor activated]
[buttonSensor activated]
Start
[buttonSensor activated]
[lineData = {Black, Black}]
[lineData = {Black, White}]
[buttonSensor activated]
[lineData = {Black, White}]
LineLeft
FWD, FWD
[lineData = {Black, White}]
OnBlack
FWD, FWD
[lineData = {White, Black}]
LineRight
FWD, FWD
[lineData = {Black, Black}]
[lineData = {Black, Black}]
[lineData = {Black, White}]
[lineData = {White, White}]
[lineData = {White, White}]
[lineData = {White, Black}]
[lineData = {Black, Black}]
[lineData = {Black, Black}]
[lineData = {Black, White}]
[lineData = {White, Black}]
SearchLine
SearchLeft
RWD, FWD
SearchRight
FWD, RWD
[lineData = {White, White}]
[lineData = {White, White}]
[lineData = {White, Black}]
[lineData = {Black, White}]
[reached 100°]
[reached 100°]
SearchRight2
FWD, RWD
SearchLeft2
RWD, FWD
[reached 200°]
[reached 200°]
[reached 5cm]
OvercomeGap
TurnLeftToGap
RWD, FWD
[reached 100°]
MoveForwardToGap
FWD, FWD
[reached 100°]
TurnRightToGap
FWD, RWD

3 Code Documentation

Im Code habe ich den Zustandsautomat implementiert. Die wichtigste Komponente ist hierbei das struct State. Dieses repräsentiert einen Zustand im Zustandsdiagramm. Als Attribute enthält ein State Transitions, also Zustandsübergänge. Diese werden ausgeführt, wenn das entsprechende Sensorereignis eintritt. Also zum Beispiel wird die Transition onWhiteWhite ausgeführt, wenn der linke und der rechte Liniensensor weiß messen. Die Transition onTicks wird ausgeführt, falls die Odometriesensorticks den tickThreshold übersteigt. tickThreshold ist auch ein Attribut eines jeden Zustands. Außerdem hat ein Zustand noch die Attribute direction und speed. In der main()-Methode werden die Motoren immer auf die Richtung und Geschwindigkeit des aktuellen Zustands gesetzt.

void initializeStates(struct State *states); In dieser Methode wird der Zustandsautomat nach dem Vorbild des Diagramms initialisiert. Es werden also für jeden der 17 Zustände die entsprechenden Zustandsübergänge, einen tickThreshold, die Motorrichtungen und die Motorgeschwidigkeiten angelegt. Die fertigen Zustände werden in dem Parameter Array states gespeichert, das in der main()-Methode deklariert wurde.

void checkTrigger(struct State *state, struct Transition *transition); Eingabeparameter sind der aktuelle Zustand state und eine leere Transition. Die Methode überprüft nun, ob ein Tastsensor aktiviert wurde. Falls Ja, muss die onTrigger Transition von state ausgeführt werden. Dazu wird die bisher leere Parametertransition auf die onTrigger Transition gesetzt.

void checkTicks(struct State *state, struct Transition *transition, int *ticks); Das Prinzip ist dasselbe, wie bei checkTrigger. der Eingabeparameter state entspricht dem aktuellen Zustand. Falls einer der Tastsensoren aktiviert wurde, ist transition nicht mehr leer, da checkTrigger diese schon auf einen gültigen Wert gesetzt hat. In diesem Fall tut diese Methode nichts. Falls transition noch leer ist, wird überprüft, ob der tickThreshold von state schon überstiegen wurde. Dazu dient der Eingabeparameter ticks. Hier sind alle Schwarz-Weiß Übergänge seit dem letzten Reset gespeichert. Falls also gilt state->tickThreshold <= ticks[LEFT]+ticks[RIGHT], dann wird transition auf die onTicks Transition von state gesetzt.

void checkLineData(struct State *state, struct Transition *transition); Die Liniensensoren haben die niedrigste Priorität. Wenn transition bis hierhin noch leer ist, werden die Liniensensoren überprüft und transition auf die entsprechende onBlackBlack, onBlackWhite, onWhiteBlack, oder onWhiteWhite Transition gesetzt.

void updateOdometry(int *ticks); In dieser Methode werden die Odometriesensoren ausgelesen und bei einem gefundenen Schwarz-Weiß, oder Weiß-

Schwarz Übergang die ticks um eins erhöht. Dazu wird wie in Übungsblatt 5 das Encodersignal mit einem FIR Filter gefiltert und anschließend mit einem Schmitt-Trigger die gemessene Farbe bestimmt. Unterscheidet sich die Farbe von der letzten Messung, dann wird ticks um eins hochgezählt.

void setSpeed(unsigned char *speed, unsigned char left, unsigned char right); Aus empirischen Studien ging hervor, dass der linke Motor leicht schwächer ist, als der rechte. Darum wird bei allen Geschwindigkeitsänderungen die Geschwindigkeit am rechten Motor ein wenig verringert. So können die unterschiedlichen Motorstärken ausgeglichen werden.

void frictionBoost(enum DrivingState *drivingState, unsigned char *direction, unsigned char *speed); Die Grundgeschwindigkeit von 80 ist zu niedrig, als dass der Asuro damit aus dem Stand anfahren könnte. Darum bietet diese Methode eine Anfahrhilfe, wann immer der Asuro aus dem Stand anfährt. drivingState zeigt hierbei an, ob der Asuro steht oder fährt. direction und speed sind die Attribute des aktuellen Zustands. Direkt nach der Anfahrhilfe werden die Motorrichtungen und Geschwindigkeiten auf diese Werte gesetzt.

int main(void); Diese Methode bildet den zentralen Ablaufplan. Zunächst wird mit dem Aufruf der initializeStates() Methode der Zustandsautomat initialisiert. Weitere Starteinstellungen, wie Anschalten der FrontLED, Initialisierung des Startzustandes, oder Initialisierung der Taskperioden erfolgen als nächstes. Mit dem Eintritt in die while(1) Schleife beginnt die eigentliche Fahrt des Asuros. Die RadEncoder werden mit einem Aufruf der Methode updateOdometry() in einer recht hohen Frequenz ausgelesen, um keine Schwarz-Weiß Übergänge zu verpassen. In einer niedrigeren Frequenz werden die Motorrichtungen und geschwindigkeiten angepasst. Um dieses Ziel zu erreichen wird zunächst eine leere Transition erstellt. leer heißt in diesem Fall, dass das nextState Attribut auf 255 und somit einen ungültigen Wert gesetzt wird. Anschließend werden die Sensormethoden in folgender Reihenfolge aufgerufen: checkTrigger(), checkTicks(), checkLineData(). Die zuvor erstellte Transition hat nun einen gültigen nextState Eintrag. Außerdem gibt die Transition an, ob die EncoderTicks auf 0 gesetzt werden müssen, und ob der Roboter eine kurze Pause machen muss. Die Pause ist immer dann nötig, falls ein Motor seine Richtung von vorwärts zu rückwärts oder andersherum ändert. Ohne die Pause kann es sein, dass die Motoren verhaken und der Asuro nicht mehr weiterfährt. Zum Schluss wird der aktuelle Zustand auf den nextState der Transition gesetzt und die MotorRichtungen und Geschwindigkeiten aktualisiert.