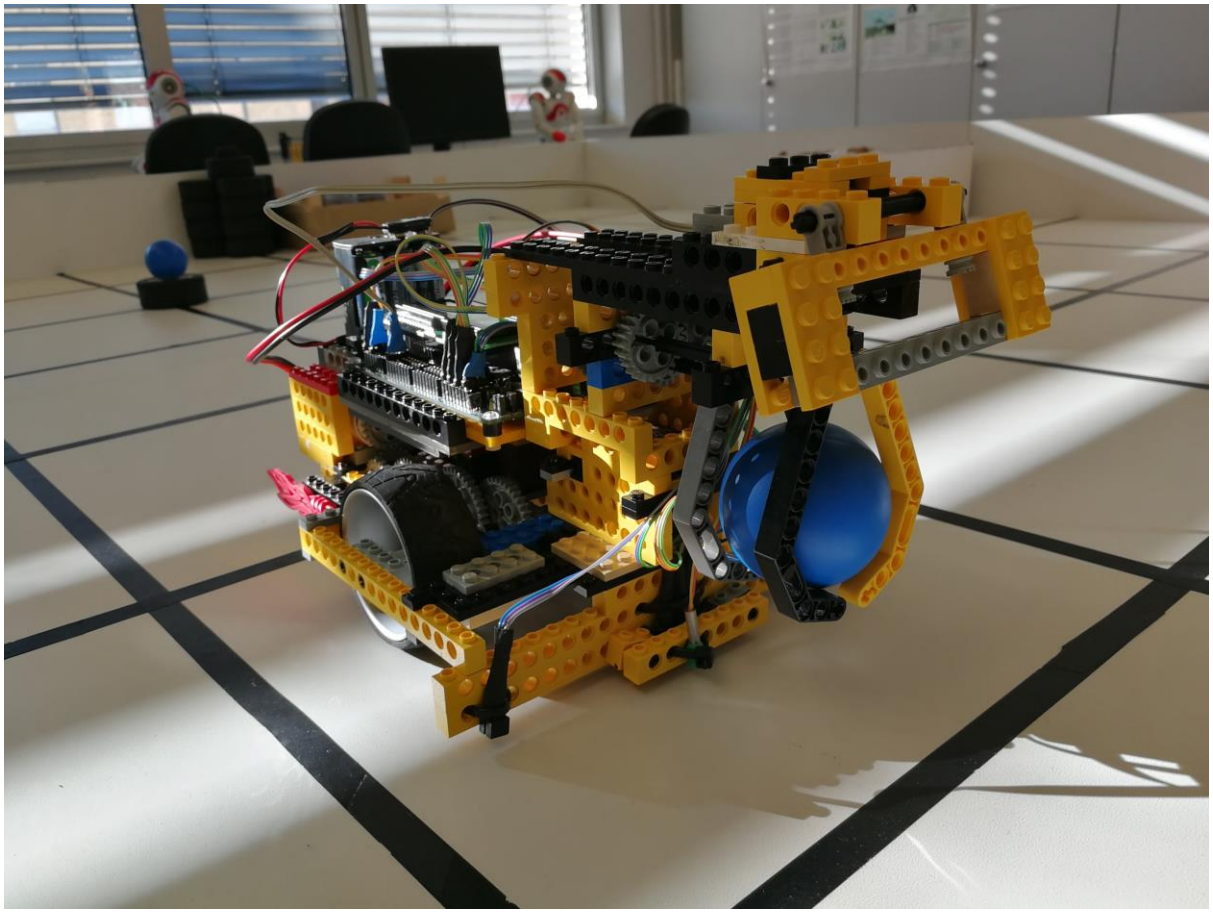
Technische Hochschule Brandenburg
University of Applied Sciences

Projekt - Autonome Mobile Systeme

Rüdiger, der Pizzabote

Bau und Programmierung eines Lego-Roboters mit einem AkSen-Board und ANSI C

Tobias Trompell, Peter Neuhoff



Inhalt

Projektaufbau und Funktionalität

Im Verlauf des Projektes soll ein ”Pizzabote” entwickelt werden. Dabei handelt es sich um einen Legoroboter, der mit Hilfe von Motoren, Sensoren, Legomechanik und dem AkSen-Mikrocomputer autonom einen Fahrauftrag ausführen kann. Hierbei erhält er, abhängig vom Fahrplan eine oder mehrere “Pizzen” - zu Modellzwecken sind es hier kleine Plastikbälle - welche an ihre jeweiligen Endstationen befördert werden sollen. Der Roboter besitzt hierbei das globale Wissen über die statische Karte, sodass Wege im Voraus geplant werden können.

Das Streckennetz, welches als eine 71-stellige Zeichenkette übergeben wird, besteht dabei aus frei befahrbaren Kreuzungen (.), unpassierbaren Kreuzungssperren (*x*) und Endstationen (*F*). Die Strecke selbst besteht dabei immer aus geraden Linien, die in gleichen Abständen parallel laufen oder zueinander im 90°-Winkel liegen können (s. Abb. 1).

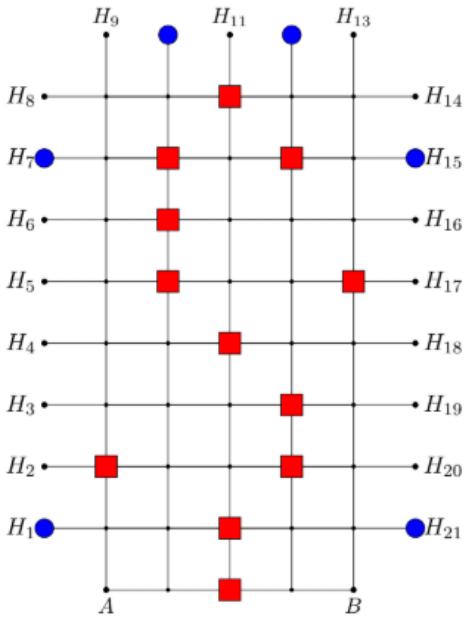


Abb. 1: Modell der Strecke *xxFxFxxx..x..xF.x.x.Fx.x...xx.x..xxx..x..xx...x.xxx..x.xF..x..Fx..x..x*
Rote Rechtecke repräsentieren unpassierbare Kreuzungen, Blaue Kreise repräsentieren Endstationen

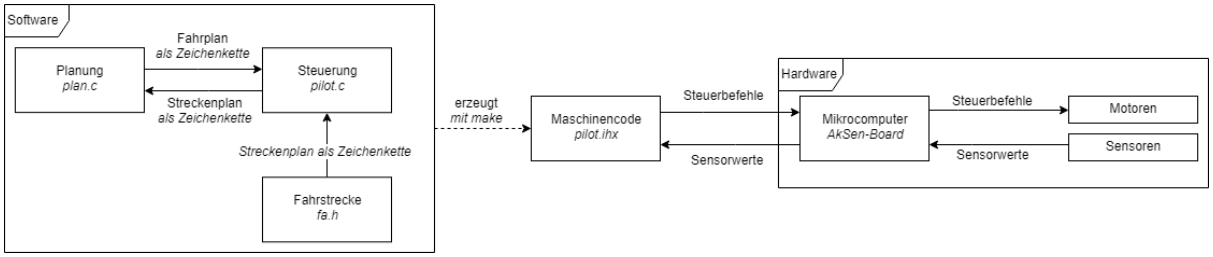


Abb. 2: Aufbau des Projektes

Der Aufbau des Roboters kann in drei große Teile geteilt werden:

Die Mechanik des Roboters sollte ausbalanciert und robust sein. Das Getriebe sollte Antriebsstark sein, ohne die Motoren zu überfordern. Der Roboter sollte Platz für den Mikrocomputer besitzen, um Ausgaben zu ermöglichen, und die Batterie sollte schnell und einfach zugänglich sein.

Der Pilot des Roboters übernimmt die Steuerfunktionen des Roboters. Dazu zählen die Steuerung der Motoren, die Verarbeitung erhaltener Sensordaten und die Verarbeitung von Steuerbefehlen.

Der Planer des Roboters übernimmt die Verarbeitung der Karte und findet auf ihr einen Weg zu den Zielknoten. Er ist so konzipiert, dass er unabhängig vom Roboter einen Weg planen kann und der Roboter dann je nach Bauweise den Plan interpretiert.

Mechanik

Getriebe

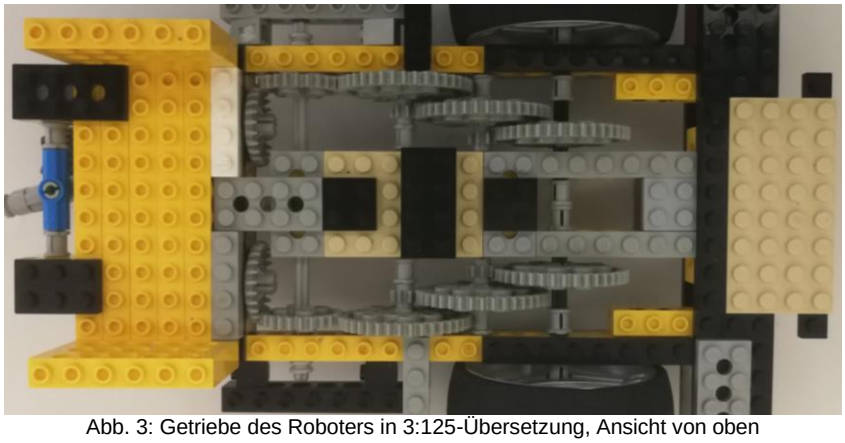


Abb. 3: Getriebe des Roboters in 3:125-Übersetzung, Ansicht von oben

Das Getriebe stellte eine der ersten großen Herausforderungen dar. Es musste mehrere Iterationen durchlaufen, bis wir mit dem jetzigen zufrieden waren. Das Getriebe existiert zweimal, so dass jedes Rad einzeln angesprochen werden kann. Es besteht aus 3 40-zähnigen Rädern, 2 8-zähnigen Rädern, 2 normalen 24-zähnigen Rädern und einem angewinkelten 24-zähnigen Rad. Das angewinkelte Zahnrad sorgt für eine Drehung der Übersetzung um 90°, was es ermöglichte die Räder näher aneinander zu positionieren, da die Motoren parallel zur Fahrtrichtung gesetzt werden konnten. Die Zahnräder sind nach folgender Reihenfolge angeordnet: 24, 24, 24, 40 - 8, 40 - 8, 40; was sich dann letztendlich zu einer 3:125 Übersetzung auswirkt. Die Zahnräder wurden so ausgewählt und verbunden, dass ein möglichst kleiner Widerstand durch sie entsteht. Dadurch ist unser Getriebe zwar schwächer als andere, aber schneller und der geringe Widerstand durch gute Zahnräder hilft die mangelnde Kraft auszugleichen.

Die Motoren werden unterhalb des Batteriefachs auf jeweils eines der 24-zähnigen Zahnräder aufgesetzt. Im Piloten werden die einzelnen Motoren anschließend mit den Direktiven `leftMotor` für 0 und `rightMotor` für 1 angesprochen.

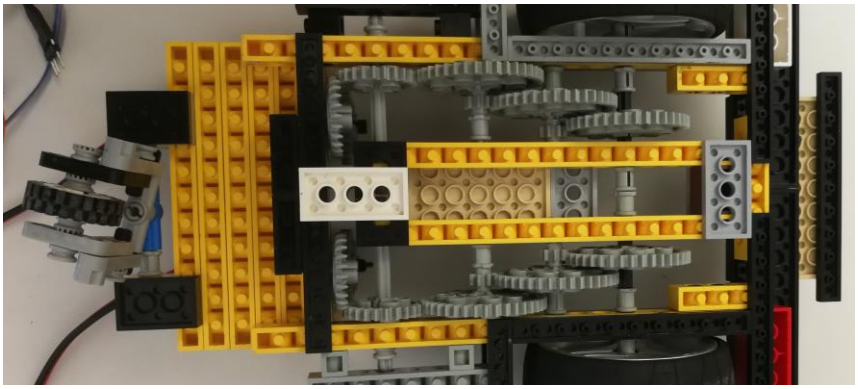


Abb. 4: Getriebe des Roboters mit Nachstellrad, Ansicht von unten

Das Nachstellrad ist dafür konzipiert, sich immer in Fahrtrichtung zu drehen und damit nur eine Unterstützung des Gewichts zu sein. Problem bei unserem Nachstellrad ist, dass die Achse, an der es sich Drehen kann, nicht sehr lang ist und dadurch das Rad leicht verbiegt, was den Widerstand erhöht.

Sensoren und Taster

Zur Erkennung von Linien werden fünf Optokoppler verwendet, welche in einer Reihe im vorderen Teil des Roboters angebracht und fixiert sind. Die Sensoren sind knapp über dem Boden angebracht, sodass sie die schwarzen Streifen, welche sie Verfolgen sollen, gerade so nicht berühren.

Zur Lichterkennung wird ein einfacher Lichtsensor verwendet, der leicht abseits der Frontsensoren angebracht ist (s. Abb. 5), um beim Startsignal nicht direkt auf einer Linie zu stehen.

Abb. 5: Frontsensoren des Roboters, von links nach rechts: Linker Kreuzungssensor, Linker Liniensensor, Mittlerer Liniensensor, Rechter Liniensensor, Lichtsensor (grün), Rechter Kreuzungssensor

Bei der Verteilung der Sensoren im AkSen-Board gelten Konventionen, die im Piloten mit `#define`-Direktiven festgelegt werden (s. Abb. 6), um das Auftreten von Magic-Nummern zu vermeiden.

Port#	Sensor	Direktive im Piloten
0	Linker Liniensensor	`leftSensor`

2	Mittlerer Liniensensor	`middleSensor`
4	Rechter Liniensensor	`rightSensor`
6	Linker Kreuzungssensor	`leftSideSensor`
8	Rechter Kreuzungssensor	`rightSideSensor`
10	Lichtsensor für die Starterkennung	`lightSensor`

Abb. 6: Konvention der Zuweisung des analogen Ports im Pilotprogramm

Der Schalter befindet sich im Greifer, und ist im Piloten der Direktive `button` zugewiesen. Der digitale Port dieses Schalters ist der Port 0. Der Schalter ist dabei hochkant eingebaut und

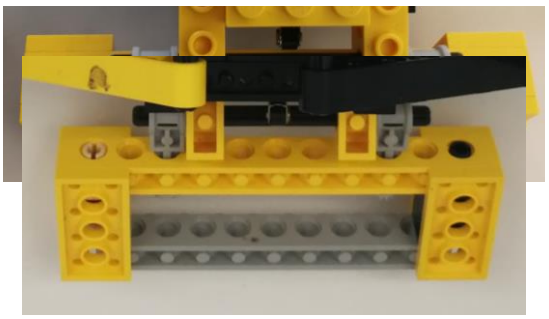


Abb. 7: Taster und Schalter, Ansicht von oben (links) und von unten (rechts)

Greifer

Der Servomotor des Greifers befindet sich im unteren Frontteil des Roboters, verbunden mit der darunterliegenden Sensorleiste und gestützt durch die Controllerplatte (s. Abb. 8). Direkt darauf befindet sich der Greifer (s. Abb. 9), welcher die Servoeinstellungen mit einer Übersetzung von 1:1 übersetzt. Der Servomotor befindet sich im Servo-Port 0, welches im Piloten mit der Direktive `claw` versehen wird.

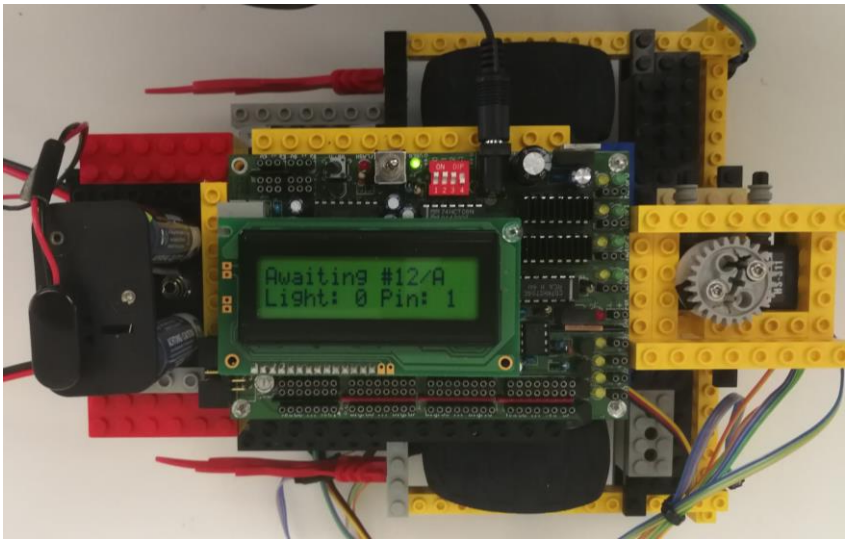


Abb. 8: Gehäuse des Roboters mit sichtbarem Servomotor

Der Greifer selbst ist eine einfache Klaue, welche hoch genug ist, um nicht mit einem der Zielstationen zu kollidieren, aber dennoch niedrig genug ist, um einen Ball von einer dieser Stationen aufheben zu können. Er besitzt zwei Zustände, die im Piloten als `claw_open` und `claw_closed` definiert sind; Eine Schließung der Klaue lässt den Servomotor um 10° drehen, während eine Öffnung der Klaue eine 40°-Drehung voraussetzt.

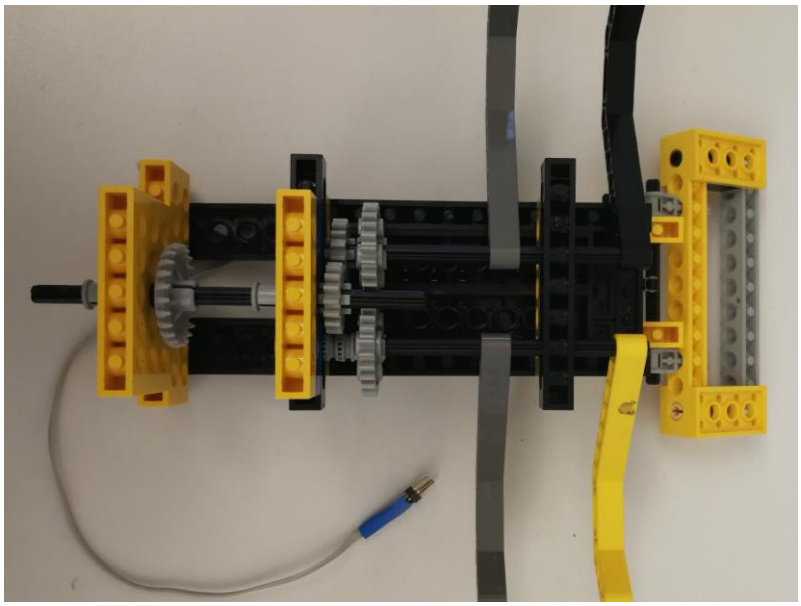


Abb. 9: Greifer und Taster des Roboters, Ansicht von unten

Da mit jedem Start des AkSen-Boards der Greifer geschlossen wird, sollte bei erstmaligem Start darauf geachtet werden, dass das Getriebe des Greifers nicht mit dem Servomotor

verbunden ist, um ein mögliches Überspannen des Motors zu verhindern. Anschließend jedoch kann der Greifer frei eingestellt werden.

Pilot

Das Pilotprogramm ist das Herzstück des Roboters. Die Funktion `AksenMain` dient dabei als zentraler Einstiegspunkt des Programms.

Konventionen der Fahrsteuerung

Um Magic-Nummern zu verhindern, werden, wie im vorherigen Abschnitt auch, `#define`-Direktiven festgelegt. Da viele der Richtungen lediglich mit `char` beschrieben werden, wird dabei in der Fahrtrichtung festgelegt, dass die Vorwärtsrichtung `forward` mit 0 beschrieben und die Rückwärtsrichtung `backward` mit 1 beschrieben wird. Ähnliches gilt für die Drehrichtung: Die Linksdrehung `left` wird mit 0 und die Rechtsdrehung `right` mit 1 beschrieben.

Im Verlauf des weiteren Abschnittes wird des Öfteren der Datentyp `bool` verwendet. Da im reinen ANSI C allerdings kein boolscher Datentyp existiert, wird dieser vorher mittels einer Typdefinition festgelegt (s. Abb. 10). Dies dient der Lesbarkeit des Quellcodes.

```
typedef enum { false = 0, true = 1} bool;
```

Abb. 10: Typdefinition des booleschen Datentyps `bool`

Fahrsteuerung

Die vier Grundfunktionen der Fahrsteuerung (s. Abb. 11) bilden das Grundgerüst des reaktiven Roboters. Die Definition der Funktionen dient der Redundanzvermeidung.

```
void   stop();
```

Stoppt alle Motoren.

```
void   drive(char direction);
```

Lässt den Roboter in die Richtung `direction` mit der Geschwindigkeit `motorSpeed` fahren.

```
void   turn(char turnDirecton, char wheelDirection);
```

Lässt den Roboter in die Richtung `turnDirection` mit der Geschwindigkeit `motorSpeed` drehen. `wheelDirection` gibt dabei an, ob dabei vorwärts oder rückwärts gedreht wird. Bei dieser Art der Drehung bewegt sich das Rad in Drehrichtung nicht.

```
void   crossTurn(char turnDirecton, char wheelDirection);
```

Lässt den Roboter in die Richtung `turnDirection` mit der Geschwindigkeit `motorSpeed` drehen. `wheelDirection` gibt dabei an, ob dabei vorwärts oder rückwärts gedreht wird. Bei dieser Art der Drehung bewegt sich das Rad in Drehrichtung mit der Geschwindigkeit `supportMotorSpeed`.

Abb. 11: Funktionen der einfachen Motorsteuerung

Linienfindung

Für die Funktionen der Linienfindung und -verfolgung werden die drei Frontsensoren genutzt.

Die Erkennung einer dunklen Fläche - also des schwarzen, zu verfolgenden Streifens - wird über die Abfrage der Optokoppler-Werte in der Funktion `onLine` getätigt (s. Abb. 12). Überschreitet der abgefragte Wert den in der Präprozessordirektive `black` definierten Wert 140, dann befindet sich der Optokoppler definitiv auf einer schwarzen Fläche.

```
bool   onLine(char sensor);
```

Gibt an, ob sich der Sensor im analogen Port `sensor` eine schwarze Linie erkennt

```
void   followLine();
```

Steuert den Roboter vorwärts in Abhängigkeit zu den Linien, die die Sensoren `leftSensor`, `middleSensor` und `rightSensor` erkennen. Wiederholtes Anwenden der Funktion ermöglicht die Linienverfolgung.

```
void   followLineBackwards();
```

Steuert den Roboter rückwärts in Abhängigkeit zu den Linien, die die Sensoren `leftSensor`, `middleSensor` und `rightSensor` erkennen. Wiederholtes Anwenden der Funktion ermöglicht die Linienverfolgung.

Abb. 12: Funktionen der Liniensteuerung

Die Linienverfolgung mit `followLine` und `followLineBackwards` steuert den Roboter mit Hilfe der im vorherigen Abschnitt erläuterten Grundbewegungsfunktionen (s. Abb. 13) in Abhängigkeit zu den aktuell erkannten Frontsensorwerten. Ein tatsächliches Verfolgen der Linie entsteht somit erst über den wiederholten Aufruf der Funktion.

Die Linienverfolgung reagiert darauf, ob der mittlere Liniensensor eine Linie erkennt. Ist dies der Fall, dann fährt der Roboter geradeaus, auch wenn einer der seitlichen Liniensensoren eine Linie erkennt (s. Abb. X). Befindet sich der mittlere Liniensensor nicht auf einer Linie, dann steuert der Roboter dementsprechend entgegen.

```
void followLine(){
      if (onLine(leftSensor))
            turn(left, forward);
      if (onLine(rightSensor))
            turn(right, forward);
      if (onLine(middleSensor))
            drive(forward);
}
```

Abb. 13: Implementation der `followLine`-Funktion

Die Linienverfolgung mit `followLineBackwards` reagiert ähnlich der Vorwärtslinienverfolgung; Es wird lediglich rückwärtsgefahren und in die Driftrichtung gefahren. Da das Nachstellrad des Roboters das Rückwärtsfahren des Roboters nicht blockiert, kann der Roboter ohne großen Widerstand rückwärtsfahren.

Kreuzungssteuerung

Zur Erkennung der Kreuzung werden die beiden äußeren Kreuzungssensoren verwendet. Ähnlich der Erkennung von Linien erkennt die Funktion `isCrossing` (s. Abb. 14), ob sich einer der Kreuzungssensoren auf einer Linie befindet. Ist dies der Fall, dann befindet sich der Roboter auf einer Kreuzung, und es kann dementsprechend reagiert werden.

```
bool   isCrossing();
```

Gibt an, ob der linke oder der rechte Kreuzungssensor eine Linie erkennt.

```
void   turnCrossing(char direction);
```

Dreht den Roboter solange in die Richtung `direction`, bis die Liniensensoren eine Linie erkennen.

Abb. 14: Funktionen der Kreuzungsteuerung

Soll ein Roboter eine Kreuzung überfahren, darf der Roboter dieselbe Kreuzung nicht zweimal erkennen. Besonders anfällig dafür sind die `turn`-Funktionen, welche eine Kreuzung bis zu drei Mal erkennen können. Um dies zu vermeiden, wird die `sleep`-Funktion benutzt. Damit sorgt die `turnCrossing`-Funktion, dass der Roboter sich für eine halbe Sekunde in die gewünschte Richtung dreht (s. Abb. 15). Anschließend dreht er sich solange weiter in die gewünschte Richtung, bis der mittlere Liniensensor eine Linie entdeckt.

```
void turnCrossing(char direction){
      turn(direction, forward);
      sleep(500);
      while (!onLine(middleSensor))
            turn(direction, forward);
}
```

Abb. 15: Implementation der `turnCrossing`-Funktion

Plansteuerung

Zur Ausführung eines gegebenen Planes ist die Funktion `driveByPlan` verantwortlich (s. Abb. 16). Der Plan muss dabei ein Char-Array sein. Die Funktion erwartet dabei einen von fünf Befehlen, die ausgeführt werden können (s. Abb. 17).

```
bool   isButtonPressed();
```

Gibt an, ob der Taster gedrückt wurde.

```
void   changeClawStatus();
```

Ändert die Position des Greifers.

```
void   driveByPlan(char plan[]);
```

Fährt einen gegebenen Plan ab.

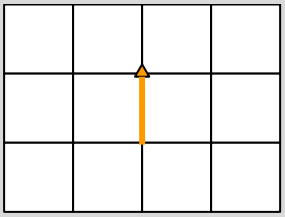
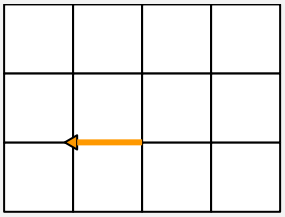
```
void   printPlan(char *plan, int i);
```

Gibt den gegebenen Plan beginnend an der Stelle `i` auf dem AkSen-Board aus.

Abb. 16: Funktionen der Plansteuerung

Ein einzelner Befehl im Plan wird bis zur nächsten erkannten Kreuzung ausgeführt. Danach wird der Zähler der internen Schleife inkrementiert und der nächste Befehl ausgeführt. Eine weitere Abbruchbedingung stellt das Betätigen des Schalters während der Linienverfolgung dar. Im Normalfall sollte der Befehl, der dadurch aufgerufen wird, der Greiferbefehl `G` im Planer sein, welcher die Motoren für eine Sekunde stoppt und mit der `changeClawStatus`-Funktion die global definierte Position des Greifers verändert.

Erhält der Planer einen Befehl, den er nicht kennt, so werden die Motoren gestoppt und die Ausführung des Planes beendet.

Kommando	Position nach Planausführung
`F` Fahre vorwärts bis zum nächsten Kreuzungspunkt	
`L` Biege links ab und fahre bis zum nächsten Kreuzungspunkt	

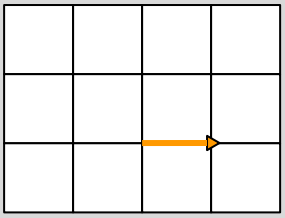
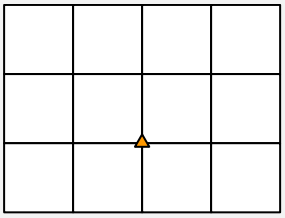
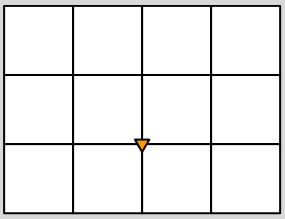
`R`	Biege rechts ab und fahre bis zum nächsten Kreuzungspunkt	
`G`	Ändere die aktuelle Greiferposition.	
`W`	Drehe dich auf der Stelle und bewege dich bis zum nächsten Kreuzungspunkt	

Abb. 17: Grafische Darstellung der planlogischen Befehle

Eine Besonderheit in den Befehlen stellt der Wendebefehl `W` dar. Die Logik des Wendebefehls setzt voraus, dass der Roboter eine perfekte Drehung auf der Stelle vollbringt. Da dies jedoch nicht gewährleistet werden kann, ist die Alternative eine Wende, die vom Folgebefehl abhängig ist. Dies ist kein Problem, wenn dieser entweder `L` oder `R` ist, da der Roboter, nachdem er rückwärts auf die Kreuzung gefahren ist, eine Kreuzungsdrehung in die entgegengesetzte Richtung des Befehls ausführen kann. Schwieriger ist es jedoch, einen auf eine Wende folgenden Vorwärtsbefehl auszuführen; Für dieses Manöver ist eine Dreipunktwende notwendig, die den Roboter zunächst nach links auf die Linie einbiegen, dann auf die Kreuzung zurückfahren und anschließend auf den korrekten Weg links einbiegen lässt (s. Abb. 18). Nach dem Ausführen des Befehls wird der Folgebefehl, der zur Ausführung der Wende in Betracht gezogen wurde, übersprungen.

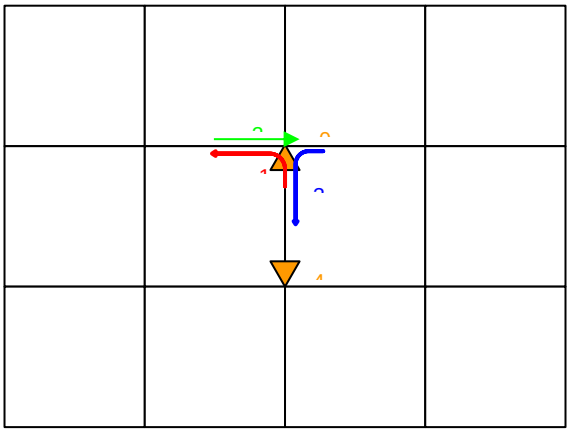


Abb. 18: Grafische Darstellung der Ausführung eines Vorwärtsfahrtbefehls nach einem Wendebefehl. 0: Startposition des Roboters an der Kreuzung, 1: Linkswende bis zum Erkennen einer Linie, 2: Rückwärtsfahren bis zur Kreuzung, 3: Linksabbiegen bis zum Erkennen einer Linie, 4: Linienverfolgung bis zum Erkennen der Kreuzung.

Aufruf des Pilotprogrammes

Mit dem Start des AkSen-Boards wird die Funktion `AksenMain` ausgeführt. Diese wiederum berechnet zunächst einen Plan mit der durch den Planer bereitgestellten Funktion `getPlan` einen Fahrplan, der von zwei Dingen abhängig ist.

Zum einen muss der Fahrauftrag in die Funktion mitgegeben werden was über die Variable `_fa` der im Piloten inkludierten Header-Datei `fa.h` geschieht. Die Definition des Fahrauftrages erfolgt über die vor der Inklusion der Header-Datei erfolgten Präprozessordirektive `#define FAn`, wobei `n` mit der Nummer des Fahrauftrags ersetzt wird.

Zum anderen muss die Startposition des Roboters auf dem Feld mitgegeben werden. Da die Funktion den Index der Startposition auf dem Brett benötigt, werden diese beim Start durch die Position des 0. Dip-Schalters übergeben (s. Abb. 19).

Position	Index	Präprozessordirektive	Dip-Schaltung
A	64	`AStart`	0
B	68	`BStart`	1

Abb. 19: Schaltung des 0. Dip-Schalters und Übergabeparameter an die `getPlan`-Funktion

Da der Startschuss über eine Lampe erfolgt, wird nach der Berechnung eines Planes über eine Endlosschleife die Lichterkennung über die `isLightOnSensor`-Funktion abgefragt. Der Schwellwert für das Vorhandensein von Licht auf dem Lichtsensor ist hierbei 30. Damit der Roboter bei schlechten Lichtverhältnissen nicht einfach losfährt, dient der 3. Dip-Schalter als Kontrollschalter. Erst, wenn dieser betätigt worden ist und Licht erkannt wurde, startet der Roboter. Alternativ dazu startet der Roboter, wenn der 2. Dip-Schalter betätigt worden ist, auch, wenn kein Licht vorhanden ist.

Abb. 20: Ausgabe des AkSen-Boards nach erfolgreicher Streckenplanung für die Strecke FA23 auf der Startposition B

Nach erfolgreicher Berechnung des Fahrplans wird auf dem AkSen-Board zur Kontrolle der korrekten Angaben die Strecke, die Startposition auf der Strecke, das Ergebnis der Lichterkennung und die Position des Kontrollschalters angegeben (s. Abb. 20). Der 1. Dip-Schalter kann verwendet werden, um sich den Fahrplan der aktuellen Strecke anzusehen.

Wegplanung

Die Wegplanung dient dazu aus einer vorgegebenen Welt eine Liste aus erreichbaren Zielen zu ermitteln und für diese dann einen Plan aus Aktionen zu erstellen, den der Roboter bekommt und abarbeitet. Aufgrund der eingeschränkten Funktionalität des Aksen-Boards konnte keine dynamische Speicherverwaltung verwendet werden und stattdessen wurden statische Arrays verwendet. Weiterhin mussten die Arrays, wegen der geringen Stackgröße von 128 Byte global im Quellcode definiert werden (s. Abb. 21).

```
char costs[71] = { 0 };
char indices[150];
char customers[10] = { 0 };
char currentPath[30];
char currentPathBack[60];
```

Abb. 21: Global definierte Arrays am Kopf des Quellcodes

Die wichtigste und einzige vom Piloten aufgerufene Funktion ist `getPlan()`.

```
void getPlan(char* map, char* actions, char start) { … }
```

Diese Funktion bekommt durch den Parameter `map` eine Zeichenkette, die die aktuelle Karte repräsentiert. Der Parameter `actions` ist ein leeres char-Array mit der Länge 200 für die Ausgabe und `start` ist der Index des Startknotens. Innerhalb der Funktion werden dann folgende Unterfunktionen aufgerufen.

getCostMap

```
void getCostMap(unsigned char* input, char* output, int length,
int startIndex);
```

Diese Funktion füllt das Array `output` mit einer Karte, aller von `startIndex` erreichbaren Knoten. Dazu wird zuerst das `input`-Array kopiert und in `output` geschrieben und die Symbole zu Zahlenwerten umgeformt.

```
for (i = 0; i < length; i++)
{
      if (input[i] == 'x')
```

```
            output[i] = -1;
      else if (input[i] == '.')
            output[i] = -2;
      else if (input[i] == 'F')
            output[i] = 30;
      else
            output[i] = input[i];
}
```

Abb. 22: Schleife zum Kopieren des input-Arrays

Dann ermitteln wir mittels einer Kostenflutung innerhalb einer Schleife für jede Position mit einer -2 die nötigen Schritte, um vom Startpunkt dorthin zu kommen. Alle Positionen die gar nicht erreichbar sind bleiben mit -2 belegt.

```
while (!finished)
{
      int somethingChanged = 0;
      for (i = 0; i < length; i++)
      {
            if (output[i] == cost)
            {
                  if (i + 1 >= 0 && i + 1 <= length)
                  {
                        if (output[i + 1] == -2)
                        {
                              output[i + 1] = cost + 1;
                              somethingChanged = 1;
                        }
                  }
                  if (i - 1 >= 0 && i - 1 <= length)
                  {
                        if (output[i - 1] == -2)
                        {
                              output[i - 1] = cost + 1;
                              somethingChanged = 1;
                        }
                  }
                  if (i - 7 >= 0 && i - 7 <= length)
                  {
                        if (output[i - 7] == -2)
                        {
                              output[i - 7] = cost + 1;
                              somethingChanged = 1;
                        }
                  }
                  if (i + 7 >= 0 && i + 7 <= length)
                  {
```

```
                        if (output[i + 7] == -2)
                        {
                              output[i + 7] = cost + 1;
                              somethingChanged = 1;
                        }
                  }
            }
      }
      if (!somethingChanged)
      {
            finished = 1;
            output[length - 1] = 0;
      }
      else
      {
            cost++;
      }
}
```

Abb. 23: Algorithmus zur Flutung der Karte mit Kosten zum Startpunkt

getIndexComplete

```
void getIndexComplete(char* costs, char* indices, int length);
```

Um aus der Kostenkarte einen Pfad aus Knoten zu ermitteln die wir abfahren müssen, rufen wir diese Funktion auf. Sie benötigt als Parameter eine Kostenkarte `costs`, ein leeres char-Array `indices` zum Befüllen mit der Ausgabe und der Länge der Kostenkarte `length` für Schleifen. Zu Beginn wird aus der Kostenkarte eine Liste aus Zielknoten ermittelt, die tatsächlich erreichbar sind und diese werden dann noch gezählt.

```
getFinishList(costs, customers, length);
for (i = 0; i < 10; i++)
{
      if (customers[i] != 0)
            customerCount++;
      else
            break;
}
```

Abb. 24: Funktion zur Ermittlung der Ziele und Schleife zum Zählen der Ziele

Danach wird für jeden Index in der `customers`-Liste ein optimaler Pfad ermittelt und dann zu einem Hin- und Rückweg zusammengesetzt.

```
for (i = 0; i < customerCount; i++)
{
	getIndexList(costs, currentPath, customers[i], length);
	getIndex2Way(currentPath, currentPathBack, 60);
	for (j = 0; j < 60; j++)
	{
		indices[fullLength + j] = currentPathBack[j];
	}
	currentLength = getPathLength(currentPathBack);
	fullLength += currentLength;
}
```

Abb. 25: Schleife zur Ermittlung und Konkatenation der einzelnen Pfade

getActionPlan

```
void getActionPlan(char* indices, char start, char* actions, int
length);
```

Abschließend brauchen wir eine Funktion, die aus einer Liste aus Knoten einen Aktionsplan erstellt. Diese Funktion nimmt als Parameter eine Liste aus Indizes `indices` die abgefahren werden sollen, den Index des Startpunktes `start`, ein leeres char-Array `actions` für die Ausgabe und die Länge des `actions`-Arrays `length`. In dieser Funktion wird in einer Schleife für jeden Punkt die nötige Aktion berechnet, um zum nächsten Punkt zu kommen (s. Abb. X). 26

```
for (i = 1; i < length; i++)
{
	actions[i - 1] = calculateDirection(positionP, directionP, indices[i]);
}
```

Abb. 26: Schleife für die Ermittlung der Action für jeden Knoten

Hilfsfunktionen

```
int getValidNeighbour(char* map, char target, int length);
```
Überprüfen, ob ein Zielknoten `target` in einer Kostenkarte `map` einen erreichbaren Nachbarn hat. Gibt 1 zurück, wenn `target` einen erreichbaren Nachbarn hat.

`void getFinishList(char* input, char* output, int length);`
Gibt eine Liste aus allen erreichbaren Zielknoten zurück.

`int getIndex(char* map, char value, char previous, int length);`
Durchsucht eine Kostenkarte nach einem Knoten mit dem Wert `value` und gibt den Index dieses Knoten zurück.

`void getIndexList(char* costs, char* indices, char target, int length);`
Sucht für einen Zielknoten `target` einen Pfad in der Kostenkarte `costs` und schreibt dies Pfad in das Array `indices`.

`void getIndex2Way(char* indices, char* out, int length);`
Nimmt einen Indexpfad `indices` und berechnet daraus den einen Pfad für Hin- und Rückweg. Dieser wird dann in `out` geschrieben.

`void getIndexComplete(char* costs, char* indices, int length);`
Erstellt einen gesamten Indexpfad für alle Ziele in einer Karte und schreibt ihn in `indices`.

`char calculateDirection(char* position, int* direction, char target);`
Berechnet für eine gegebene Position `position` und Ausrichtung `direction` die nötige Aktion, um zum Knoten `target` zu kommen. Gibt die Aktion als character zurück.

Tests

Fahrtests ergaben, dass der gebaute Legoroboter durch sein 3:125-Getriebe eine Menge Kraft erzeugen kann, sodass er deutlich schneller auf geraden Stecken als andere Roboter fährt. Der Roboter besitzt, vor allem durch den Greifer und den Servomotor, eine starke Gewichtsverteilung nach vorn, was durch die Batterie, welche sich im hinteren Teil des Roboters befand, ausbalanciert wurde, ohne jedoch das Stützrad zu sehr zu belasten. Dies sorgt dafür, dass das Rückwärtsfahren ohne großen Widerstand erfolgen kann. Der Greifer ist belastbar und kann Bälle aus den Stationen aufheben und diese selbst unter Belastung von außen festhalten.

Schnell ergab sich jedoch ein Problem; Das starke Getriebe sorgte schnell dafür, dass der Roboter zu groß geworden ist. Teilweise wird dadurch im viel zu großen Bogen in Kreuzungen eingebogen, teilweise werden dadurch Ablieferungen von Bällen gestoppt, weil der Roboter zu früh eine Wand anfährt und die Linie verliert.

Fazit

Im Verlauf des Projektes wurde der Roboter *Rüdiger* entwickelt: Ein schneller, aber recht ungenauer und vor allem großer Legoroboter. Die 3:125-Übersetzung des Getriebes erzeugt zwar viel Kraft, braucht aber dementsprechend auch viel Platz, sodass Drehungen und Wenden schwerfällig und platzaufwändig werden. Das Planprogramm ist schnell und durch seine iterative Natur Stack schonend. Der Roboter selbst ist dabei zwar robust, aber unzuverlässig, sodass meist nur ein Objekt ausgeliefert werden kann, bis die Strecke verloren wird.