Technische Hochschule
Brandenburg
University of
Applied Sciences

Fachbereich Informatik und Medien

Dokumentation zum KI - Projekt

Pizzabote

Studiengang Medizininformatik

Wintersemester 2017/2018

vorgelegt am 20.01.2018

von Sandra Bieseke & Iryna Omelchuk

Inhaltsverzeichnis

1 Motivation und Aufgabenstellung (Sandra Bieseke)

Im Rahmen des AMS-Projektes hatten wir die Aufgabenstellung, unter Verwendung des Aksen-Boards, Lego-Steinen, verschiedenen Sensoren und natürlich selbst in der Programmiersprache C programmierter Software einen Roboter zu entwickeln. Dieser sollte der in der Lage sein, in einem vorgegebenen Streckennetz (einfaches Gitter), in welchem aber verschiedene Kreuzungen gesperrt sein können an vorgegebene Ziele „Pizza“ auszuliefern. Das Spielfeld setzt sich aus 70 Straßenkreuzungen, einigen gesperrten Kreuzungen, 2 Start- bzw. Zielpunkten und insgesamt bis zu 6 Zielobjekten zusammen. Der aktuelle Fahrauftrag mit Angabe der Ziele und der jeweiligen gesperrten Kreuzungen wird durch Einbinden eines Planungsmoduls (_fa.h), in welchem verschiedene Fahraufträge implementiert sind, vorgegeben. Die Fahraufträge werden als Zeichenkette mit 70 Zeichen übergeben, wobei ein x ein gesperrtes Feld und ein F einen Fahrauftrag anzeigt. Die Planung der Routen sollte mit Hilfe der Breitensuche realisiert werden. Dies sollte am Ende einem autonomen System mit SAE-Level 5 (Die Stufen der Automatisierung wurden von der internationalen Ingenieurs- und Automobilindustrie-Vereinigung SAE definiert, wobei Stufe 0 dem Fahren ohne jegliche Assistenz entspricht) entsprechen, dem autonomen Fahren. Der Roboter fährt komplett selbständig, anstatt Fahrer und Passagiere sitzen nur noch Passagiere (in unserem Fall die Pizza) im Fahrzeug. Pedale oder ein Lenkrad sind nicht notwendig. [1] In der nachstehenden Tabelle sind die verschiedenen Level noch einmal übersichtlich dargestellt. Zu Beginn des Semesters bekamen wir eine umfassende Einweisung zur Funktionsweise des verwendeten Boards sowie der unterschiedlichen zur Verfügung stehenden Sensoren sowie eine kleine Einführung zur Planung von Getrieben. Bevor wir überhaupt angefangen haben, stand der Name unseres zukünftigen Roboters auch schon fest. Da wir das einzige reine Frauen-Team waren sollte sie also „Roberta“ heißen. So konnten wir uns auch gleich an den Aufbau von Roberta machen.

SAE level	Name	Narrative Definition	Execution of Steering and Acceleration/ Deceleration	*Monitoring* of Driving Environment	Fallback Performance of *Dynamic Driving Task*	System Capability *(Driving Modes)*
***Human driver* monitors the driving environment**						
0	No Automation	the full-time performance by the *human driver* of all aspects of the *dynamic driving task*, even when enhanced by warning or intervention systems	Human driver	Human driver	Human driver	n/a
1	Driver Assistance	the *driving mode*-specific execution by a driver assistance system of either steering or acceleration/deceleration using information about the driving environment and with the expectation that the *human driver* perform all remaining aspects of the *dynamic driving task*	Human driver and system	Human driver	Human driver	Some driving modes
2	Partial Automation	the *driving mode*-specific execution by one or more driver assistance systems of both steering and acceleration/ deceleration using information about the driving environment and with the expectation that the *human driver* perform all remaining aspects of the *dynamic driving task*	**System**	Human driver	Human driver	Some driving modes
***Automated driving system* ("system") monitors the driving environment**						
3	Conditional Automation	the *driving mode*-specific performance by an *automated driving system* of all aspects of the dynamic driving task with the expectation that the *human driver* will respond appropriately to a *request to intervene*	System	**System**	Human driver	Some driving modes
4	High Automation	the *driving mode*-specific performance by an automated driving system of all aspects of the *dynamic driving task*, even if a *human driver* does not respond appropriately to a *request to intervene*	System	System	**System**	Some driving modes
5	Full Automation	the full-time performance by an *automated driving system* of all aspects of the *dynamic driving task* under all roadway and environmental conditions that can be managed by a *human driver*	System	System	System	**All driving modes**

Abbildung 1.1: SAE-Level 0-5 / Copyright © 2014 SAE International

2 Aufbau von Roberta (Iryna Omelchuk)

2.1 Technische Ausstattung

Auf Abbildung 2.1 ist ein Aksenboard und die Verkabelung zu sehen:

- 2x Motoren für den Antrieb (Motor: 0; 3)
- 1x Akku für Strom
- 5x Optokoppler für die Streckenerkennung und das Linienfolgen (An.: links 0, rechts 1, Mitte 3, außen: rechts 6; links 5)
- 1x Servomotor für das Abladen/Beladen der Pizza (S:0)
- 1x Lichtsensor für das Startlichtsignal (An.: 08) 1x Drucksensor (Dig: 00)

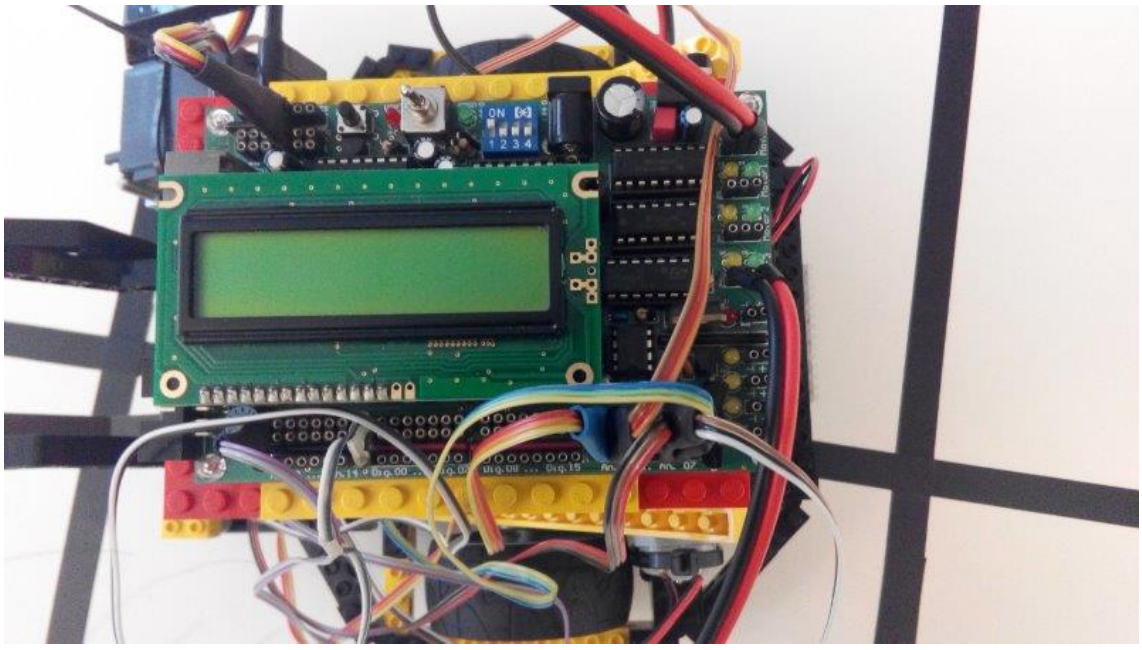

Abbildung 2.1: Aksenboard und Verkabelung

2.2 Motoren und Getriebe

Das Grundgerüst des Roboters bilden zwei Räder, die dazugehörigen Motoren, der Akku (Abb. 2) und dem darüber liegenden Aksenboard.

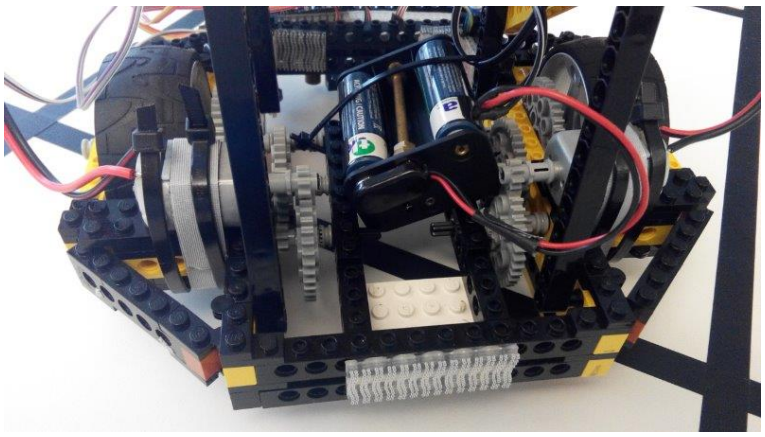

Abbildung 2.2: Motoren und Getriebe

Das Getriebe eines Rades besteht aus sechs Zahnrädern: drei davon sind die größte, weil sie 40 Zähne haben und anderen drei sind die kleinste mit nur acht Zähnen. (Abb.2.3 und 2.4)

Abbildung 2.3: Zahnräder (von oben)

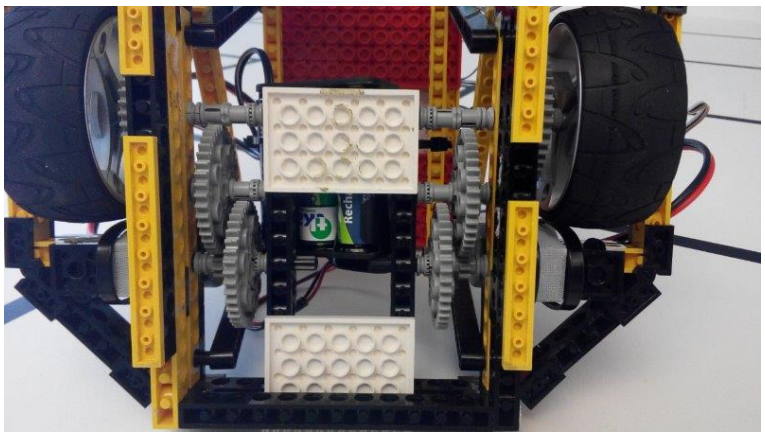

Abbildung 2.4: Zahnräder (von unten)

Damit die Motoren ausreichende Kraftübertragung liefern können, wurde mithilfe von jeweils sechs Zahnrädern eine Untersetzung von 1:125 gemacht. Das bedeutet, dass wir eines der kleinen an dem Motor befestigt haben. Dieses ist wiederum mit einem großen Zahnrad verbunden (40:8). Also, im Getriebe haben wir insgesamt dreimal die gleiche Konstruktion verwendet ((40:8)*(40:8)*(40:8)). Daraus ergibt sich, dass der Motor sich 125 Mal drehen muss, damit sich das Rad einmal komplett gedreht hat. Diese Untersetzung ist geeignet für unseren Roboter, weil er dann die Linie folgt und überhaupt zuverlässig fahren kann. Bei sehr hoher Geschwindigkeit wäre es so, dass der Roboter nicht schnell genug die Linie findet, besonders bei den Kurven. Noch ein Servomotor (Abb.2.5) befindet sich neben dem Greifer und im Unterschied zur anderen Motoren, dreht sich nur bis zu einem bestimmten Winkel (für Abladen/ Beladen der Pizza)

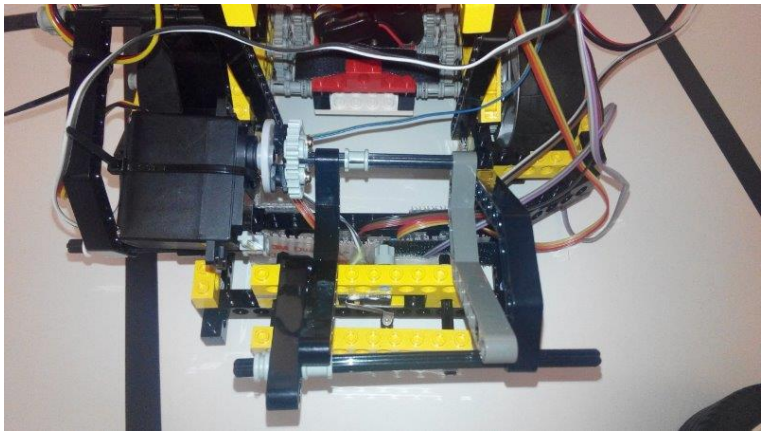

Abbildung 2.5: Servomotor mit den Getriebe und dem Greifer

2.3 Sensoren

Die Abbildund 2.6 zeigt, wie die Sensoren (Optokoppler) für das Linienführung und die Kreuzungerkennung mithilfe von Klebeband befestigt wurden. Jeder von ihnen beinhaltet eine Infrarot-LED sowie einen passenden Infrarot-Empfänger. Dabei drei Sensoren sind ganz vorne und der Roboter kann damit

die Linie folgen, denn diese schwarz ist, während der restliche Boden weiß ist. Sie können zwischen hell und dunkel unterscheiden und geben Werte zwischen 0 und 255 aus. Die beiden hinteren, die etwas weiter außen sind, dienen zur Erkennung von Kreuzungen, damit ROBERTA abbiegen kann. In der Mitte hat ROBERTA ein Lichtsensor oder Photodetektor, der das Lichtsignal empfangen kann und damit es als Startsignal ist. Erst wenn der Roboter das Licht sieht, darf er anfangen zu fahren.

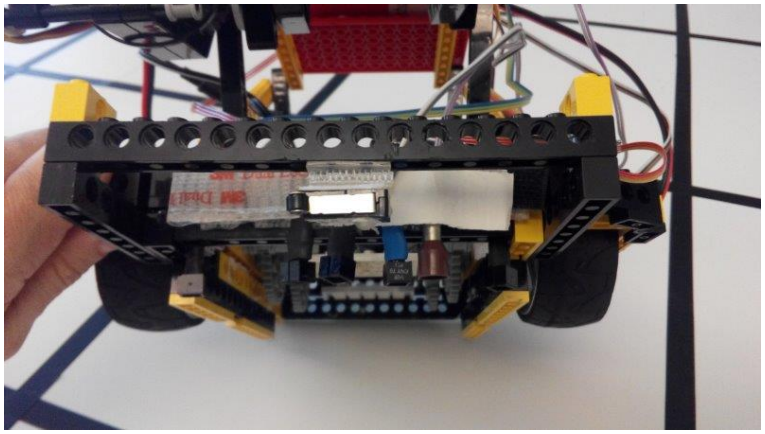

Abbildung 2.6: Sensoren und Fotodetektor

Damit der Greifer weiß, wann er abladen soll, wird noch ein weiterer Optokoppler oder sogenannter Drucksensor benötigt. Dieser befindet sich unter den Greiferachsen in der Mitte Sobald die Pizza (der Ball) zu nah kommt, nimmt er eine Veränderung wahr und der Greifer ablädt. Zuerst wurde der Drucksensor (Taster) einfach vorne gebaut. (Abb. 2.7). Dann haben wir uns entschieden, ihn zwischen den zwei Legobauteilen einzubauen, um die größere Angriffsfläche zu machen. (Abb.2.8)

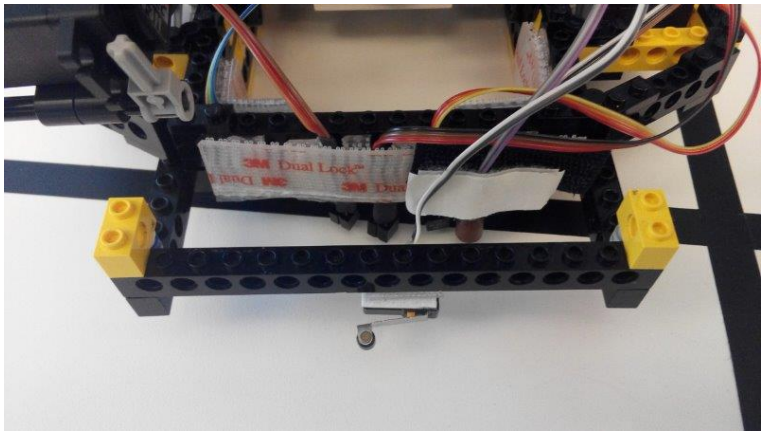

Abbildung 2.7: Der Drucksensor vorher

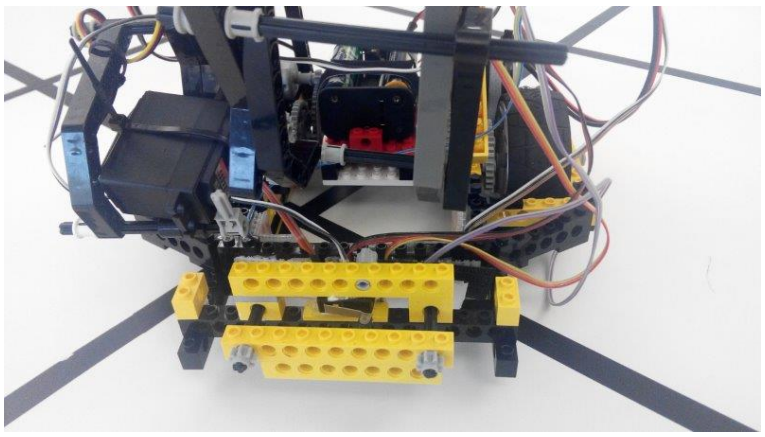

Abbildung 2.8: Der Drucksensor nachher

2.4 Greifer

Zuletzt verfugt unsere ROBERTA noch über einen Greifer, mit welchen Pizza (den Ball) abgeladen/beladen wird. (Abb.2.9) Der Greifer ist ebenfalls aus Lego-Bausteinen aufgebaut und ist als ein separates Werkzeug. Er kann abmontiert werden, um es durch die anderen Legoteile zu ersetzen und arbeitet mit einem Servomotor. Wenn diesem das Signal zum Abladen/Beladen in Form eines Winkels gegeben wird, bewegt sich der Greifer nach unten oder nach oben. Das Signal zum Auslösen des Greifers gibt ein Drucksensor (Taste).

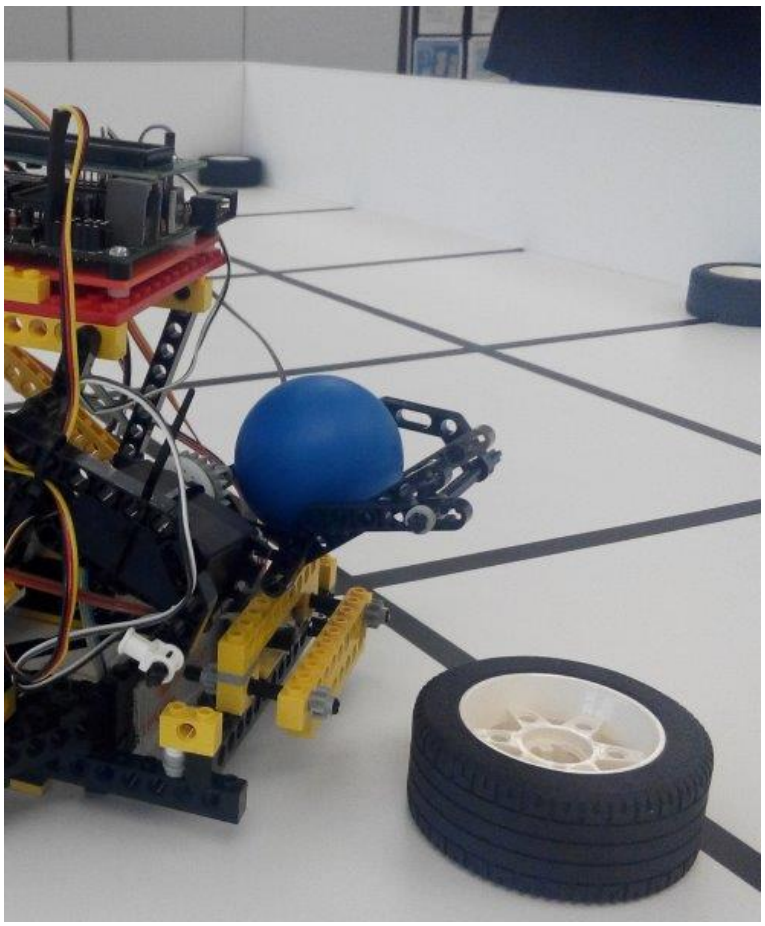

Abbildung 2.9: Greifer

2.5 Probleme

Bei der Konstruktion des Antriebes hatten wir einige Schwierigkeiten: wegen des Antriebs mit den Legoteilen bauen, auch wegen der Stabilität des Rahmens. Dieser musste zum späteren Zeitpunkt erneut zusammen gebaut werden, aber deutlich stabiler, als vorher. Die nächste Schwierigkeit war bei dem Wettbewerb, dass der Servomotor wahrscheinlich viel Strom verbraucht hat, deswegen ROBERTA wurde schnell abgestürzt.

3 Software und Problemlösungsstrategien (Sandra Bieseke)

Bevor wir uns mit dem Problem der Routenplanung befassen konnten, musste Roberta lernen, sich entlang der auf dem Spielfeld schwarz markierten Linien zu bewegen, sowie auf Kommando nach rechts und links abzubiegen. Erst als das funktionierte, befassten wir uns mit der Streckenplanung. Dafür haben wir eine Zeichenkette mit den zu testenden Fahraufträgen in der Form „GRLGGLLRR ...“, wobei 'G' für Geradeaus, 'R' ... naja für Rechts steht. Roberta musste ausserdem lernen, die Kreuzungen zu erkennen, was sich wie viele der folgenden Steuerungsprobleme über die optischen Sensoren regeln ließ: Für die Kreuzungserkennung haben wir den rechten äußeren Optokoppler genutzt: immer wenn dieser schwarz erkannt hat, musste es eine Kreuzung sein. Zum Linienhalten und abbigen haben wir die folgenden 3 Funktionen implementiert:

3.1 Linien halten

Die Methode zum Linien halten regelt, dass, wenn der mittlere Optokoppler die schwarze Linie erkennt, beide Motoren mit voller Kraft fahren. Wenn sich der vordere linke oder rechte Optokoppler auf der schwarzen Linie befindet und der mittlere Optokoppler auf der weißen Fläche fährt, steuern die Motoren in die jeweils entgegengesetzte Richtung. Diese Funktion wird immer dann ausgeführt, wenn der rechte äussere Optokoppler keine Kreuzung erkennt (siehe Hauptprogrammroutine).

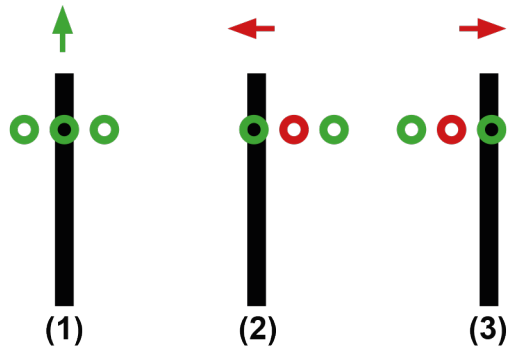

Abbildung 3.1: Umsetzung der Linienführung

Algorithmus 3.1 Methode zum Linienfolgen

```
void fahreGeradeaus()
{
  if (black(analog(IR_MITTE)))                                    //(1)
  {
      motor_pwm(MOTOR_RECHTS, 10);
      motor_pwm(MOTOR_LINKS, 10);
  }

  else if ((black(analog(IR_RECHTS))) && (weiss(analog(IR_LINKS)))) //(2)
  {
      motor_pwm(MOTOR_RECHTS, 0);
      motor_pwm(MOTOR_LINKS, 10);
  }

  else if ((weiss(analog(IR_RECHTS))) && (black(analog(IR_LINKS)))) //(3)
  {
      motor_pwm(MOTOR_RECHTS, 10);
      motor_pwm(MOTOR_LINKS, 0);
  }
}
```

3.2 Das Abbiegen und Geradeaus über eine Kreuzung fahren

Wenn nach Erkennen einer Kreuzung (1),ein Befehl zum Abbiegen erfolgt, wird die passende Methode zum Rechts- oder Linksabbiegen aufgerufen. Da zum Linien halten nur nur nötig war, dass einer der vorderen Optokoppler eine schwarze Linie erkennt, haben wir zum Abbiegen diesen Fakt ausgenutzt und die Funktionen zum Abbiegen so gestaltet, dass je nach Richtung, der der Fahrtrichtung entgegengesetzte Motor zunächst mit voller Kraft solange fährt und der Motor in Fahrtrichtung gestoppt wird, bis der Optokoppler in Abbiegerichtung auch garantiert keine schwarze Linie resultierend aus der Linienverfolgung mehr erkennt (2). Dafür haben wir einen Sleep-Timer von 0.2 Sekunden genutzt. Danach wird mit der selben Motoreinstellung weiter in Abbiegerichtung gefahren, bis der vordere Optokoppler (rechts für rechtsabbiegen und links für linksabbiegen) wieder eine scharze Linie erkennt (3) und damit ohne Probleme nach dem Abbiegeprozess weiter geradeaus gefahren werden kann. Die Erfahrung in der Praxis zeigte aber, das Roberta häüfig, aber leider nicht immer den Weg zurück zur Linie fand.

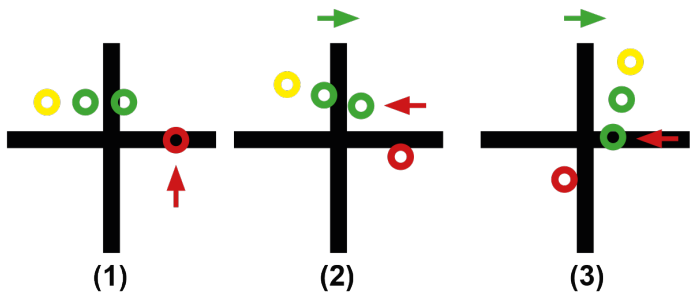

Abbildung 3.2: Beispiel für das Abbiegen nach rechts: (1) der rechte äußere Optokoppler erkennt eine Kreuzung (2) der rechte vordere Optokoppler ist weg von der schwarzen Linie, die nach Geradeaus weitergehen würde (3) Roberta hat sich soweit nach rechts gedreht bis der rechte vordere Optokoppler wieder eine schwarze Linie erkennt

Algorithmus 3.2 Methoden zum Abbiegen

```
void biegeLinksAb()
{
   motor_pwm(MOTOR_RECHTS, 10);
   motor_pwm(MOTOR_LINKS, 0);
   sleep(200);
   motor_pwm(MOTOR_RECHTS, 10);
   motor_pwm(MOTOR_LINKS, 0);
   while (weiss(analog(IR_LINKS)));
}

void biegeRechtsAb()
{
   motor_pwm(MOTOR_RECHTS, 0);
   motor_pwm(MOTOR_LINKS, 10);
   sleep(200);
   motor_pwm(MOTOR_RECHTS, 0);
   motor_pwm(MOTOR_LINKS, 10);
   while (weiss(analog(IR_RECHTS)));
}
```

Nun war also Roberta in der Lage, die Linien zu halten, und nach rechts und links zu fahren. Was aber ist, wenn sie als Befehl ein 'G' bekommt und geradeaus über eine Kreuzung fahren muss? An dieser Stelle fährt Roberta einfach mit voller Motorleistung ein Stück weiter, bis die Kreuzung nicht mehr vom rechten äußeren Optokoppler erkannt wird. Dafür nutzten wir einen Sleeptimer von 0.4 Sekunden. Für das Kreuzung überspringen haben wir nicht extra noch eine Methode geschrieben, sondern dies in die Hauptprogrammroutine integriert.

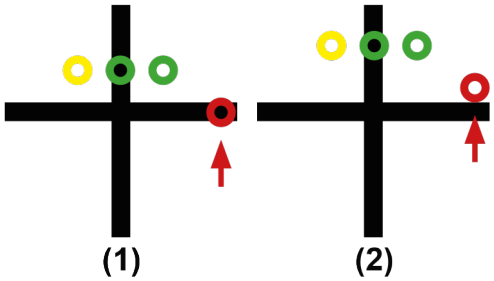

Abbildung 3.3: Roberta muss eine Kreuzung „überspringen“ (1) Roberta erkennt die Kreuzung (2) nach 0,4 Sekunden ist der Optokoppler für die Kreuzungserkennung garantiert nicht mehr auf der Kreuzung und wartet darauf, die nächste Kreuzung zu erkennen

3.3 Richtungswechsel (Wenden)

Diese Funktion haben wir erst kurz vor dem Wettbewerb implementiert und dabei einige graue Haare (mehr) bekommen. Unsere Problematik war, dass Roberta, nachdem sie eine Pizza abgeliefert hat bzw. den Startpunkt wieder erreicht hat sich so drehen muss, dass sie am Ende der Drehung einerseits mit einem der vorderen Sensoren auf der richtigen Linie nach geradeaus stehen muss und mit dem rechten äußerem Sensor vor der Kreuzung, an der der nächste Befehl erscheint. Das hieß für uns drehen und zurücksetzen. Die Motoren werden entgegengesetzt geschaltet, so dass Roberta in der Lage ist, sich auf der Stelle zu drehen, bis der mittlere vordere Optokoppler auf der richtigen Linie zum Linienfolgen ist und der Optokoppler zum Kreuzung erkennen hinter der Kreuzung ist. Die Methode,

die wir implementiert haben, löste in der Theorie ... und meistens auch in der Praxis unser Problem (ist aber eigentlich viel zu kompliziert... und im Nachhinein betrachtet, haben wir uns das Leben unnütz schwer gemacht).

Algorithmus 3.3 Funktion zum Drehen

```
void dreheUm()
{
   motor_richtung(MOTOR_LINKS, 0);
   motor_richtung(MOTOR_RECHTS, 0);
   motor_pwm(MOTOR_RECHTS, 10);
   motor_pwm(MOTOR_LINKS, 10);
   sleep(800);
   motor_pwm(MOTOR_RECHTS, 9);
   motor_pwm(MOTOR_LINKS, 10);
   while (weiss(analog(IR_MITTE)));
   motor_pwm(MOTOR_RECHTS, 9);
   motor_pwm(MOTOR_LINKS, 10);
   while (black(analog(IR_MITTE)));
   motor_pwm(MOTOR_RECHTS, 9);
   motor_pwm(MOTOR_LINKS, 10);
   while (weiss(analog(IR_MITTE)));
   motor_pwm(MOTOR_RECHTS, 9);
   motor_pwm(MOTOR_LINKS, 10);
   while (weiss(analog(IR_MITTE)) && weiss(analog(IR_RECHTS)));
   motor_richtung(MOTOR_LINKS, 1);
   motor_richtung(MOTOR_RECHTS, 0);
   motor_pwm(MOTOR_RECHTS, 10);
   motor_pwm(MOTOR_LINKS, 9);
   while (weiss(analog(IR_RECHTSAUSSEN)));
   motor_pwm(MOTOR_RECHTS, 10);
   motor_pwm(MOTOR_LINKS, 9);
   while (black(analog(IR_RECHTSAUSSEN)));
   motor_richtung(MOTOR_LINKS, 0);
   motor_richtung(MOTOR_RECHTS, 1);
}
```

3.4 Das Ablegen des Balls

Das Ablegen des Balles wird in der Hauptprogrammroutine durch Betätigen des Tasters ausgelöst, also immer wenn Roberta gegen ein Hindernis fährt. Sie stoppt beide Motoren und ändert den Winkel des Servomotors von 60 auf 20 Grad. Nach dem Ablegen des Balles fährt Roberta ein Stück zurück, damit sie genug Platz zum Wenden hat.

Algorithmus 3.4 Methode zum Ablegen des Balls

```
void ablegen()
{
   // Servo nach unten
   servo_arc(0, 20);
   sleep(500);
   // Servo nach oben
   servo_arc(0, 60);
   //Stück Rückwärts
   motor_richtung(MOTOR_LINKS, 1);
   motor_richtung(MOTOR_RECHTS, 0);
   motor_pwm(MOTOR_LINKS, 6);
   motor_pwm(MOTOR_RECHTS, 5);
   sleep(800);
   // Motoren zurückstellen
   motor_richtung(MOTOR_LINKS, 0);
   motor_richtung(MOTOR_RECHTS, 1);
}
```

3.5 Die Routenplanung

Wir haben die Routenplanung mit Hilfe der Breitensuche realisiert. Unsere Lösung dafür werde ich an einem konkreten Fahrauftrag erklären. Der Fahrauftrag wird uns als Zeichenkette aus der Datei „_fa.h“ in der Form:
unsigned char _fa [] = "xFxxxFxx.x...xx..x.xxx..x..xx..x..xF..x..Fx..x.xxxx.x..xF...x.Fx..x..x"; übergeben. Dieser Fahrauftrag entspricht übertragen auf 7 Spalten und 10 Zeilen dem in der folgenden Abbildung dargestelltem Spielfeld. Dabei ergibt sich bei Auftauchen eines 'x' eine gesperrte Kreuzung, ein '.' beteutet, dass die Kreuzung frei ist und ein 'F' steht für einen Fahrauftrag.

x	F	x	x	x	F	x
x	.	x	.	.	.	x
x	.	.	x	.	x	x
x	.	.	x	.	.	x
x	.	.	x	.	.	x
F	.	.	x	.	.	F
x	.	.	x	.	x	x
x	x	.	x	.	.	x
F	.	.	.	x	.	F
x	.	.	x	.	.	x

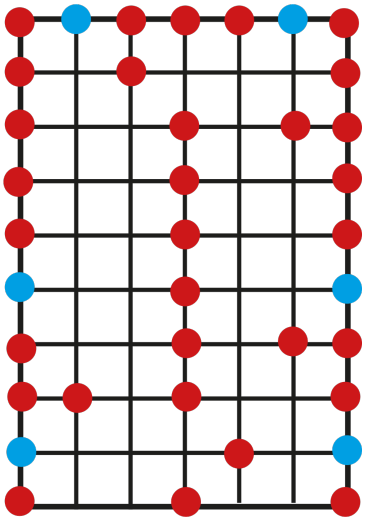

Abbildung 3.4: Spielfeld zum übertragenem Fahrauftrag

3.5.1 Fahrauftrag in die Kostenmatrix übertragen

Der vorgegebene Fahrauftrag wird zuerst in unsere erste Kostenmatrix übertragen, die ebenfalls als Zeichenkette definiert ist.

Algorithmus 3.5 Übertragen des Fahrauftrages in die Kostenmatrix

```
void leseFahrauftrag()
{
    int i;
    for (i = 0; i < sizeof(kosten); i++)
    {
        kosten[i] = _fa[i];
    }
}
```

3.5.2 Startpunkt festlegen und Kosten für den Startpunkt eintragen

Der Startpunkt wird erst kurz vor dem Rennen ausgelost, deshalb nutzten wir die Dip-Schalter am Aksenboard, um Roberta mitzuteilen, von welchem Punkt aus sie starten muss. Der Startpunkt A befindet sich in dem char-Array an Position 64, Startpunkt B an Position 68. Wenn der erste Dip-Schalter an Position 0 auf eien 1 zurückgibt, starten wir vom Startpunkt A, sonst von B. Anschließend

werden an der Position für den Startpunkt in der Kostenmatrix die Kosten von 0 eingetragen. in unserem Berechnungsbeispiel arbeiten wir mit Startpunkt B.

Algorithmus 3.6 Startpunkt festlegen und Kosten 0 für Startpunkt eintragen

```
void startPunktEintragen()
{
   if (dip_pin(0))
   {
       startpunkt = 64;
   }
   else startpunkt = 68;
   kosten[startpunkt] = '0';
}
```

3.5.3 Agenda Anlegen

Die für die Breitensuche notwendige Agenda haben wir als int-Array der Größe 40 angelegt und alle Felder mit der Zahl 99 gefüllt. Die Zahl 99 ist willkürlich gewählt, wir brauchten lediglich eine Zahl, die größer als 70 ist. In die Agenda werden bei der Breitensuche die jeweils erreichbaren Felder hinten einsortiert (an die erste Stelle, die kleiner als 99 ist), und das jeweils an erster Stelle liegende Feld expandiert. Nachdem das erste Feld expandiert wurde, rutschen die folgenden Felder um eine Stelle nach vorne. Es werden nur Zahlen expandiert, die kleiner als 99 sind. (siehe Algorithmus zur Breitensuche)

Algorithmus 3.7 Methode zum Füllen der Agenda

```
void fuelleAgenda()
{
   int i;
   for (i = 0; i < MAXAGENDA; i++)
   {
       agenda[i] = 99;
   }
}
```

3.5.4 Kosten füllen (Breitensuche)

Um die Breitensuche zu starten, wird der Startpunkt als erster für uns erreichbarer Punkt in die Agenda eingetragen. In unserem Fall die 68, und damit als erstes expandiert. Expandieren bedeutet nun, dass nun geschaut wird, welche Punkte nördlich (-7 Stellen von der aktuellen Position), östlich (+1 Stelle von der aktuellen Position) und westlich (-1 Stelle von der aktuellen Position) erreichbar (und überhaupt auf unserem Spielfeld vorhanden) sind. Ist ein Punkt erreichbar, wird die Position hinten in die Agenda eingetragen und an die erreichbare Stelle in der Kostenmatrix Kosten+1 des gerade expandierten Punktes eingetragen. Das heißt also für den Startpunkt, der ja die Kosten 0 hat, dass alle Punkte, die von dort aus zu erreichen sind, die Kosten 1 erhalten, alle Punkte, die von einem Punkt mit den Kosten 1 erreichbar sind erhalten die Kosten 2 u.s.w.. Anschließend haben wir die Kosten in eine "int-Matrix" übertragen (ascii-Zahlen starten bei 48) x wird hier jetzt zur 72 und ein F zur 22, ein nicht erreichbarer Punkt wird zur -2 .

Daraus ergeben sich dann die gefüllte Kostenmatrix, die zweite Kostenmatrix, welche nur noch Zahlen enthält und zur Veranschaulichung das Spielfeld mit den erreichbaren Kreuzungen und deren Kosten.

x	F	x	x	x	F	x
x	.	x	10	9	10	x
x	.	.	x	8	x	x
x	.	.	x	7	8	x
x	.	.	x	6	7	x
F	.	.	x	5	6	F
x	.	.	x	4	x	x
x	x	.	x	3	2	x
F	.	.	.	x	1	F
x	.	.	x	1	0	x

72	22	72	72	72	22	72
72	-2	72	10	9	10	72
72	-2	-2	72	8	72	72
72	-2	-2	72	7	8	72
72	-2	-2	72	6	7	72
22	-2	-2	72	5	6	22
72	-2	-2	72	4	72	72
72	72	-2	72	3	2	72
22	-2	-2	-2	72	1	22
72	-2	-2	72	1	0	72

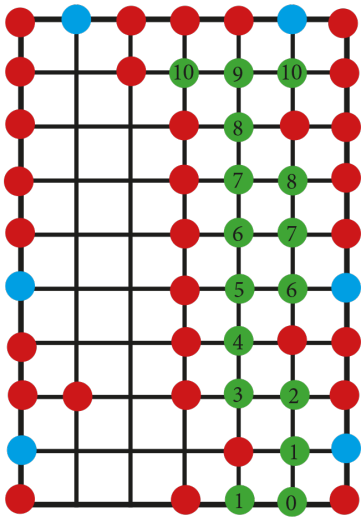

Abbildung 3.5: gefüllte Kostenmatrix (char[] und int[]) und Spielfeld mit Kosten

Algorithmus 3.8 Breitensuche zum Füllen der Kostenmatrix

```
void kostenfuellen()
{
    int i;
    int index_agenda = 0;
    agenda[0] = startpunkt;

    while (agenda[0] < 99)
    {
        //Es gibt einen nördlichen Punkt
        if ((agenda[0] - 7) >= 0)
        {
            //nördlicher Punkt ist erreichbar
            if (kosten[agenda[0] - 7] == '.')
            {
                //nördlichen Punkt in die Agenda packen (
                //erste Stelle, die keine Zahl < 99 ist)
                while (agenda[index_agenda] < 99)
                {
                    index_agenda++;
                }
                agenda[index_agenda] = agenda[0] - 7;
                index_agenda = 0;
                // in der Kostenmatrix die Kosten (+1) eintragen
                kosten[agenda[0] - 7] = kosten[agenda[0]] + 1;
            }
        }
        //Es gibt einen westlichen Punkt
        if ((agenda[0] - 1) >= 0 && (agenda[0] - 1 % 7 != 0))
        {
            if (kosten[agenda[0] - 1] == '.')
            {
                while (agenda[index_agenda] < 99)
                {
                    index_agenda++;
                }
                agenda[index_agenda] = agenda[0] - 1;
                index_agenda = 0;
                kosten[agenda[0] - 1] = kosten[agenda[0]] + 1;
            }
        }
        //Es gibt einen östlichen Punkt
        if ((agenda[0] + 1) >= 0 && (agenda[0] + 1 % 7 != 6))
        {
            if (kosten[agenda[0] + 1] == '.')
            {
                while (agenda[index_agenda] < 99)
                {
                    index_agenda++;
                }
                agenda[index_agenda] = agenda[0] + 1;
                index_agenda = 0;
                kosten[agenda[0] + 1] = kosten[agenda[0]] + 1;
            }
        }
        // expandierten Punkt aus der Agenda entfernen
        //(die anderen Einträge rutschen um 1 nach vorn)
        for (i = 0; i < MAXAGENDA; i++)
        {
            agenda[i] = agenda[i + 1];
        }
        agenda[MAXAGENDA - 1] = 99;
    }
    //Kosten in die "int-Matrix" übertragen
    for (i = 0; i < 70; i++)
    {
        kosten2[i] = (int)kosten[i] - 48;
    }

}
```

3.5.5 Erreichbare Fahraufträge feststellen

Fahraufträge haben jetzt in unsere Kostenmatrix die Zahl 22. Erreichbar können sie nur sein, wenn wenn oben/unten/rechts/oder links eine Zahl in der Kostenmatrix > 0 und < 22 steht. Je nach dem, wo sich der gefundene Fahrauftrag befindet, muss immer nur eine Kreuzungsrichtung abgefragt werden. Die für uns erreichbaren Fahraufträge werden in ein int[]-Array geschrieben. Gleichzeitig merken wir uns die Kosten der nächsten erreichbaren Kreuzung der gefundenen Fahraufträge in einem weiteren Array (kosten_erreichbar). Dies ist notwendig für die anschließende Priorisierung der Fahraufträge.

Algorithmus 3.9 Methode zum Eintragen der erreichbaren Fahraufträge und registrieren der Kosten dervom Fahrauftrag nächsten erreichbaren Kreuzung

```
void erreichbar(){
    int i;
    int zaehler = 0;
    for (i = 0; i < 70; i++)
    {
        if (kosten2[i] == 22)
        {   //Ziel ist rechts?
            if (i % 7 == 6)
            {   // Punkt westlich vom Ziel ist erreichbar!
                if (kosten2[i - 1] >= 0 && kosten2[i - 1] < 22)
                {
                    erreichbar_M[zaehler] = i;
                    kosten_erreichbar[zaehler] = kosten2[i - 1];
                    zaehler++;
                }
            }
            //Ziel ist links
            else if (i % 7 == 0)
            {   // Punkt östlich vom Ziel ist erreichbar!
                if (kosten2[i + 1] >= 0 && kosten2[i + 1] < 22)
                {
                    erreichbar_M[zaehler] = i;
                    kosten_erreichbar[zaehler] = kosten2[i + 1];
                    zaehler++;
                }
            }
            //Ziel ist unten
            else if (i >= 63 && i < 70)
            {   //Punkt nördlich vom Ziel ist erreichbar!
                if (kosten2[i - 7] >= 0 && kosten2[i - 7] < 22)
                {
                    erreichbar_M[zaehler] = i;
                    kosten_erreichbar[zaehler] = kosten2[i - 7];
                    zaehler++;
                }
            }
            //Ziel ist oben
            else if (i >= 0 && i < 7)
            {   //unten erreichbar!
                if (kosten2[i + 7] >= 0 && kosten2[i + 7] < 22)
                {
                    erreichbar_M[zaehler] = i;
                    kosten_erreichbar[zaehler] = kosten2[i + 7];
                    zaehler++;
                }
            }
        }
    }
}
```

3.5.6 Priorisieren der Fahraufträge

Da Roberta sich beim Abarbeiten längerer Fahraufträge einige Male wie eine „Prinzessin“ verhalten hat, dachten wir, es wäre eine gute Idee, zu überlegen, die Fahraufträge nach der zurückzulegenden Distanz zu sortieren. So kann sich Roberta langsam an die Herausforderungen des Alltags gewöhnen. Die Fahraufträge werden nun aufsteigend nach den Kosten der jeweils nächsten erreichbaren Kreuzung sortiert. So erhalten Fahraufträge mit geringeren Kosten eine höhere Priorität als jene mit höheren Kosten. Als Sortierverfahren haben wir den Bubblesort angewendet.

Algorithmus 3.10 Priorisieren der Fahrauftäge mit Hilfe des Bubblesort

```
void prioFahrauftraege()
{
    // in Abhängigkeit von den Kosten des nächsten erreichbaren
    // Punktes wird die Priorität der Fahraufträge festgelegt
    // Bubblesort
    int i, j, tmp;

    for (i = 0; i < 3; i++)
    {
        for (j = i; j < 3; j++)
        {
            if (kosten_erreichbar[i] > kosten_erreichbar[j])
            {
                tmp = erreichbar_M[i];
                erreichbar_M[i] = erreichbar_M[j];
                erreichbar_M[j] = tmp;
            }
        }
    }
}
```

3.5.7 Berechnen der Rückwege

Für die Berechnung der Rückwege haben wir ein Array mit 3 Zeilen und 30 Spalten angelegt, je eine Zeile für einen Fahrauftrag und wieder alle Stellen mit der Zahl 99 aufgefüllt. In erreichbar_M stehen nun die Adressen der sortierten Fahraufträge. Falls in erreichbar_M an den vorderen Stellen jetzt noch eine 0 steht, wurden weniger als 3 Fahraufträge gefunden. Die tatsächlichen Fahraufträge werden jeweils dann auch schon in dem Array für die Rückwege an die erste Stelle der Zeile (je nach Priorität) eingetragen. Ausgehend vom jeweiligen Zielpunkt werden die nächsten erreichbaren Punkte mit den geringsten Kosten an die jeweils nächste Stelle in der Richtigen Zeile des Arrays für die Rückwege eingetragent,bis kein nächster erreichbarer Punkt mehr zu finden ist.

Algorithmus 3.11 Berechnen der Rückwege

```
void berechneRueckwege()
{
    int i;
    int j = 0;
    int guenstiger_knoten = 99;
    int akt_knoten = 1;
    for (i = 0; i < 3; i++)
    {
        if (erreichbar_M[i] != 0)
        {
            rueckwege[i][0] = erreichbar_M[i];
        }
    }
    for (i = 0; i <3; i++)
    {
        for (j = 0; j < 30; ++j)
        {
            if (rueckwege[i][j] != 99)
            {
                if ((kosten2[rueckwege[i][j] - 7] < kosten2[rueckwege[i][j]]) && kosten2[rueckwege[i][j] - 7] < 22
                    && kosten2[rueckwege[i][j] - 7] > -1 && rueckwege[i][j] - 7 >-1 && rueckwege[i][j] - 7 < 70)
                {
                    guenstiger_knoten = rueckwege[i][j] - 7; //aktuell guenstigster naechster Knoten
                }
                if ((kosten2[rueckwege[i][j] + 7] < kosten2[rueckwege[i][j]]) && kosten2[rueckwege[i][j] + 7] < 22
                    && kosten2[rueckwege[i][j] + 7] > -1 && kosten2[rueckwege[i][j] + 7] < guenstiger_knoten
                    && rueckwege[i][j] + 7 >-1 && rueckwege[i][j] + 7 < 70)
                {
                    guenstiger_knoten = rueckwege[i][j] + 7; //aktuell guenstigster naechster Knoten
                }
                if ((kosten2[rueckwege[i][j] - 1] < kosten2[rueckwege[i][j]]) && kosten2[rueckwege[i][j] - 1] < 22
                    && kosten2[rueckwege[i][j] - 1] > -1 && kosten2[rueckwege[i][j] - 1] < guenstiger_knoten
                    && rueckwege[i][j] - 1 >-1 && rueckwege[i][j] - 1 < 70)
                {
                    guenstiger_knoten = rueckwege[i][j] - 1; //aktuell guenstigster naechster Knoten
                }
                if ((kosten2[rueckwege[i][j] + 1] < kosten2[rueckwege[i][j]]) && kosten2[rueckwege[i][j] + 1] < 22
                    && kosten2[rueckwege[i][j] + 1] > -1 && kosten2[rueckwege[i][j] + 1] < guenstiger_knoten
                    && rueckwege[i][j] + 1 >-1 && rueckwege[i][j] + 1 < 70)
                {
                    guenstiger_knoten = rueckwege[i][j] + 1; //aktuell guenstigster naechster Knoten
                }
                if (guenstiger_knoten < 99)
                {
                    rueckwege[i][akt_knoten] = guenstiger_knoten;
                }
                akt_knoten++;
                guenstiger_knoten = 99;
            }
        }   akt_knoten = 1;
    }
}
```

Ausgehend vom jeweiligen Zielpunkt werden die nächsten erreichbaren Punkte mit den geringsten Kosten an die jeweils nächste Stelle in der Richtigen Zeile des Arrays für die Rückwege eingetragen, bis kein nächster erreichbarer Punkt mehr zu finden ist. Als Ergebnis erhalten wir für unseren Fahrauftrag:

```
62  61  68  99  99  99  99  99  99  99  99  99  99  99  99  99  99  99  99  99  99  99  99  99  99  99  99  99  99  99
41  40  39  46  53  54  61  68  99  99  99  99  99  99  99  99  99  99  99  99  99  99  99  99  99  99  99  99  99  99
 5  12  11  18  25  32  39  46  53  54  61  68  99  99  99  99  99  99  99  99  99  99  99  99  99  99  99  99  99  99
```

Abbildung 3.6: Ausgabe der Rückwege auf der Console

Rückweg1: 62 61 68

Rückweg2: 41 40 39 46 53 54 61 68

Rückweg3: 5 12 11 18 25 32 39 46 53 54 61 68

Spätestens hier können wir an der Länge der in dem Array einsortierten Rückwege sehen, dass die Priorisierung unserer Fahraufträge geklappt hat.

Um noch einmal zu verdeutlichen, wie der Suchalgorithmus funktioniert, habe ich die folgende Abbildung zusammengestellt.

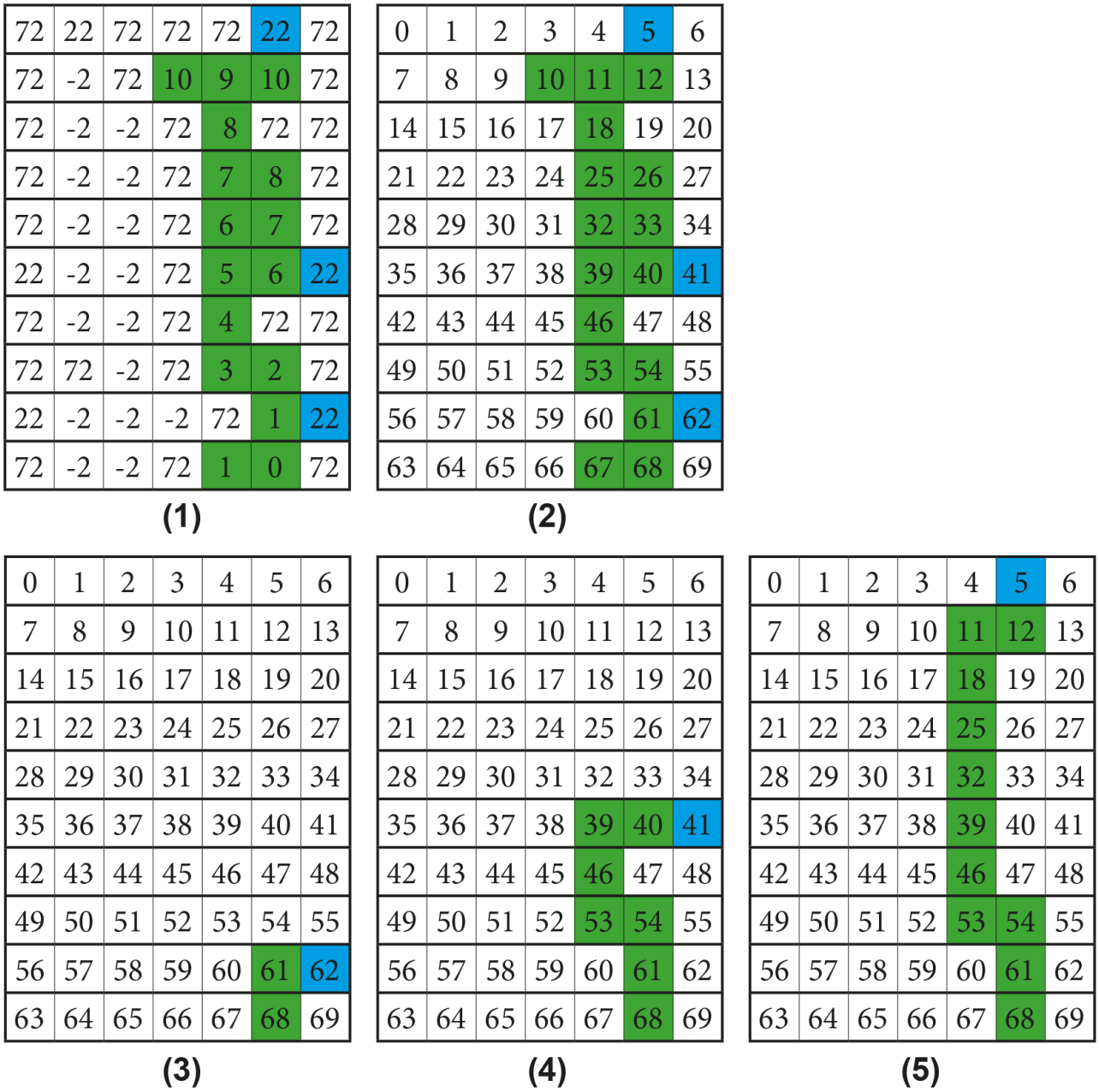

Abbildung 3.7: Berechnung der kürzesten Rückwege: (1) unsere Kostenmatrix mit den errechneten Kosten für alle erreichbaren Wege; (2) die selbe Matrix, nur das hier nur die Indizes eingetragen sind (3) der berechnete kürzeste Rückweg mit höchster Priorität; (2), (3) Die Rückwege mit geringerer Priorität

3.5.8 Hinwege (Spiegeln der Rückwege)

Um die Hinwege zu erhalten, spiegeln wir die Rückwege in ein eigens für die Hinwege angelegtes Array, welches die gleichen Eigenschaften hat, wie das Array für die Rückwege.

Algorithmus 3.12 Spiegeln der Rückwege

```
void berechneHinwege()
{
    int k = 0;
    int i, j;
    {
        for (i = 0; i < 3; i++)
        {
            for (j = 29; j >= 0; j--)
            {
                hinwege[i][k] = rueckwege[i][j];
                k++;
            }
            k = 0;
        }
    }
}
```

3.5.9 Zusammenführen der Wege

Nun werden alle Wege zu einer Sequenz in ein eindimensionales Array zusammengefügt. Als Ergebniss erhalten wir eine Sequenz, mit der wir uns im folgenden Abschnitt weiter auseinander setzen werden.

Algorithmus 3.13 Zusammenfügen aller Wege zu einer Sequenz

```
void mergeWege()
{
    int index = 0;
    int i;

    for (i = 0; i < 30; i++) {
        if (hinwege[0][i] < 99) {
            allewege[index] = hinwege[0][i];
            index++;
        }
    }
    for (i = 0; i < 30; i++) {
        if (rueckwege[0][i] < 99) {
            allewege[index] = rueckwege[0][i];
            index++;
        }
    }
    for (i = 0; i < 30; i++) {
        if (hinwege[1][i] < 99) {
            allewege[index] = hinwege[1][i];
            index++;
        }
    }
    for (i = 0; i < 30; i++) {
        if (rueckwege[1][i] < 99) {
            allewege[index] = rueckwege[1][i];
            index++;
        }
    }
    for (i = 0; i < 30; i++) {
        if (hinwege[2][i] < 99) {
            allewege[index] = hinwege[2][i];
            index++;
        }
    }
    for (i = 0; i < 30; i++) {
        if (rueckwege[2][i] < 99) {
            allewege[index] = rueckwege[2][i];
            index++;
        }
    }
}
```

3.5.10 Bestimmen der Fahrtrichtungen

Als Ergebnis des Zusammenführens aller Hin- und Rückwege ergibt sich für unseren aktuellen Fahrauftrag folgende Sequenz: 68 61 62 62 61 68 68 61 54 53 46 39 40 41 41 40 39 46 53 54 61 68 68 61 54 53 46 39 32 25 18 11 12 5 5 12 11 18 25 32 39 46 53 54 61 68.
Die erste Stelle ist definitiv der Startpunkt mit 68 und bekommt in unserer neu erstellten Sequenz für die Richtungsanweisungen ein 'S' eingetragen. Und auch sonst wissen wir, dass wenn eine 68 in der Sequenz auftaucht, Roberta sich wieder am Startpunkt befindet. Wir wissen außerdem, dass Roberta zu Beginn am Start in Richtung Norden guckt. Dieses merken wir uns als aktuellen Status für die Roberta-Richtung. Weiterhin wissen wir, dass, wenn Zahlen doppelt in der Sequenz auftauchen, das dies entweder Start-oder Zielpunkte sein müssen. Weiterhin kennen wir schon aus der Breitensuche für die Kosten und aus der Berechnung der Rückwege die Richtungen, welche sich auf dem Spielfeld aus den Differenzen (-1, +1, -7 und + 7) ergeben. Wenn also Roberta am Anfang in Richtung Norden guckt und die nächste erreichbare Kreuzung den Index 61 hat, ergibt sich eine Differenz von -7. In diesem Fall fährt Roberta geradeaus, und ändert Ihre Roberta-Richtung nicht. Jetzt sollte Roberta in der Theorie die Kreuzung 61 betreten und weiß, dass die darauf folgende Kreuzung den Index 62 hat. Sie schaut immer noch nach Norden, die Differenz von 62 zu 61 beträgt +1. So weiß Roberta, dass sie jetzt nach rechts abbiegen muss. Also wird an dieser Stelle ein 'R' eingetragen Wenn Roberta vorher in Richtung Norden geschaut hat und nun nach rechts abbiegt, ändert sich ihre Roberta-Richtung in Richtung Osten. Da nach der 62 wieder eine 62 folgt, prüfen wir nun in unserem Algorithmus ob es sich um einen Start- oder Zielpunkt handelt. Bei einem Startpunkt, wird ein 'B' für „Beladen“ eingetragen, bei einem Zielpunkt ein 'A' für „Abladen“ und muss wenden, ändert an diesen Punkten also in die entgegengesetzte Fahrtrichtung. So wird die Sequenz Stelle für Stelle analysiert und es ergibt sich für den aktuellen Fahrauftrag endlich die finale Zusammenstellung der von uns gewünschten Anweisungen.

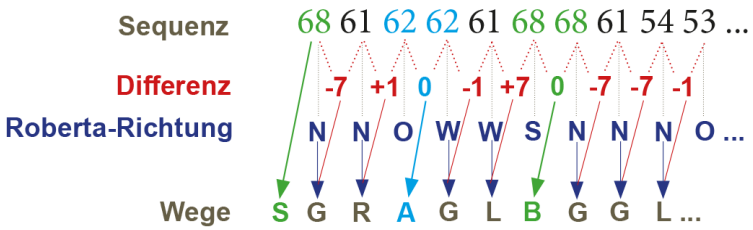

Abbildung 3.8: Bestimmen der Fahrtrichtungen

3.6 Hauptprogrammroutine

In der Hauptprogrammroutine werden zuerst einmal alle implementierten und zuvor beschriebenen Methoden zur Routenplanung ausgeführt. Weiterhin schalten wir hier die Sensoren ein und bringen die Antriebsmotoren sowie auch den Servomotor in Ausgangsstellung.

Roberta erhält im Hauptprogramm die Anweisung, dass sie, solange, wie sie nicht an ein Hindernis stößt (der Taster also nicht ausgelöst wird) geradeaus fährt und an jeder Kreuzung, die der rechte äußere Optokoppler wahrnimmt, entscheidet, welche Aktion als nächstes folgt. Die Anweisung erhält sie natürlich aus der von uns zuvor mühsam erstellten Sequenz für die endgültigen Fahrtrichtungen.

Literaturverzeichnis

[1] SAE INTERNATIONAL. Automated driving - levels of driving automation are defined in new sae international standard j3016.

Abbildungsverzeichnis

```c
//Autorin: Sandra Bieseke
//Standard-Include-Files
#include <stdio.h>
#include <regc515c.h>
#include <stub.h>
#include <time.h>
#define FA18 // Hier Fahrauftrag eingeben!!!
#include "fa.h"

#define IR_LINKS 0  // Ports für die Infrarotsensoren
#define IR_RECHTS 1
#define IR_MITTE 3
#define IR_RECHTSAUSSEN 6
#define IR_LINKSAUSSEN 5
#define PHOTO 8
#define MOTOR_LINKS 3 // Ports für die Motoren
#define MOTOR_RECHTS 0
#define black(x) (x > 150) // def black
#define weiss(x) (x < 150) // def white
#define START_A 64
#define START_B 68




unsigned int startpunkt;
const unsigned char MAXAGENDA = 40;
unsigned char kosten[] = "..................................
..................................";
int agenda[40];
int kosten2[71];
int erreichbar_M[3];
int kosten_erreichbar[3];
int fahrauftraege[3];
int rueckwege[3][30] = {
        { 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99,
          99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99 }, //Rueckweg1
        { 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99,
          99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99 }, //Rueckweg2
        { 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99,
          99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99 }  //Rueckweg3
};
int hinwege[3][30] = {
        { 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99,
          99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99 }, //Hinweg1
        { 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99,
          99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99 }, //Hinweg2
        { 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99,
          99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99 }  //Hinweg3
};

int allewege[90];
char wege[100] = { '*', '*', '*', '*', '*', '*', '*', '*', '*', '*',
                   '*', '*', '*', '*', '*', '*', '*', '*', '*', '*',
                   '*', '*', '*', '*', '*', '*', '*', '*', '*', '*',
                   '*', '*', '*', '*', '*', '*', '*', '*', '*', '*',
                   '*', '*', '*', '*', '*', '*', '*', '*', '*', '*',
                   '*', '*', '*', '*', '*', '*', '*', '*', '*', '*',
                   '*', '*', '*', '*', '*', '*', '*', '*', '*', '*',
                   '*', '*', '*', '*', '*', '*', '*', '*', '*', '*',
                   '*', '*', '*', '*', '*', '*', '*', '*', '*', 0 };
char roboter_richtung = 'N';


/**********Funktionsdeklarationen*****************/
void leseFahrauftrag();
void startPunktEintragen();
void fuelleAgenda();
void kostenfuellen();
void erreichbar();
void prioFahrauftraege();
```

```
72  void berechneRueckwege();
73  void berechneHinwege();
74  void mergeWege();
75  void berechneWege();
76  void fahreGeradeaus();
77  void biegeLinksAb();
78  void biegeRechtsAb();
79  void dreheUm();
80  void ablegen();
81
82  /******aktuellen Fahrauftrag einlesen und in die Kostenmap übertragen********/
83  void leseFahrauftrag()
84  {
85      int i;
86      for (i = 0; i < sizeof(kosten); i++){
87          kosten[i] = _fa[i];
88      }
89  }
90
91  /*****Startpunkt wird anhand der Position des Dipschalters bestimmt 1000 für Start_A
92  sonst Start_B an Position des ausgewählten Startpunktes werden Kosten 0
93  in der Kostenmatrix festgelegt*******/
94  void startPunktEintragen()
95  {
96      //64 ist Start_A; 68 ist Start_B
97
98      if (dip_pin(0))
99      {
100         startpunkt = 64;
101     }
102     else startpunkt = 68;
103     kosten[startpunkt] = '0';
104 }
105
106 /************99 an alle Positionen***************/
107 void fuelleAgenda()
108 {
109     int i;
110     for (i = 0; i < MAXAGENDA; i++)
111     {
112         agenda[i] = 99;
113     }
114 }
115
116 /********Füllen der Kostenmatrix mit Hilfe der Breitensuche*********/
117
118 void kostenfuellen()
119 {
120     int i;
121     int index_agenda = 0;
122     agenda[0] = startpunkt; //Startpunkt wird als erstes expandiert
123
124     //Es wird immer der index[0] der Agenda expandiert und geschaut, ob bei -7, -1
    oder + 1 ein '.' steht
125
126     while (agenda[0] < 99) // solange noch etwas zu expandieren ist
127     {
128         if ((agenda[0] - 7) >= 0) //Es gibt einen nördlichen Punkt (Bsp. 57)
129         {
130             if (kosten[agenda[0] - 7] == '.') //nördlicher Punkt ist erreichbar
131             {
132                 //nördlichen Punkt in die Agenda packen (erste Stelle, die keine
                    Zahl < 99 ist)
133                 while (agenda[index_agenda] < 99)
134                 {
135                     index_agenda++;
136                 }
137                 agenda[index_agenda] = agenda[0] - 7;
138                 index_agenda = 0;
139                 kosten[agenda[0] - 7] = kosten[agenda[0]] + 1;// in der Kostenmatrix
                    die Kosten (+1) eintragen
```

```
140             }
141
142         }
143
144         if ((agenda[0] - 1) >= 0 && (agenda[0] - 1 % 7 != 0)) //Es gibt einen
            westlichen Punkt
145         {
146             if (kosten[agenda[0] - 1] == '.') //westlicher Punkt ist erreichbar
147             {
148                 //westlichen Punkt in die Agenda packen (erste Stelle, die keine
                    Zahl < 99 ist)
149                 while (agenda[index_agenda] < 99)
150                 {
151                     index_agenda++;
152                 }
153                 agenda[index_agenda] = agenda[0] - 1;
154                 index_agenda = 0;
155                 kosten[agenda[0] - 1] = kosten[agenda[0]] + 1;// in der Kostenmatrix
                    die Kosten (+1) eintragen
156             }
157         }
158
159         if ((agenda[0] + 1) >= 0 && (agenda[0] + 1 % 7 != 6)) //Es gibt einen
            östlichen Punkt
160         {
161             if (kosten[agenda[0] + 1] == '.') //westlicher Punkt ist erreichbar
162             {
163                 //westlichen Punkt in die Agenda packen (erste Stelle, die keine
                    Zahl < 99 ist)
164                 while (agenda[index_agenda] < 99)
165                 {
166                     index_agenda++;
167                 }
168                 agenda[index_agenda] = agenda[0] + 1;
169                 index_agenda = 0;
170                 kosten[agenda[0] + 1] = kosten[agenda[0]] + 1;// in der Kostenmatrix
                    die Kosten (+1) eintragen
171             }
172         }
173
174
175         //expandierten Punkt aus der Agenda entfernen (die anderen Einträge rutschen
            um 1 nach vorn)
176         for (i = 0; i < MAXAGENDA; i++)
177         {
178             agenda[i] = agenda[i + 1];
179         }
180         agenda[MAXAGENDA - 1] = 99;
181     }
182     /***Kosten in die "int-Matrix" übertragen (ascii-Zahlen starten bei 48) x wird
        hier jetzt zur 72 und ein F zur 22,
183     ein nicht erreichbarer Punkt wird zur -2 *****/
184     for (i = 0; i < 70; i++)
185     {
186         kosten2[i] = (int)kosten[i] - 48;
187     }
188 }
189
190 /********* Erreichbare Fahraufträge **********/
191 //(Felder mit der Zahl 22) in eine Matrix (erreichbar_M) packen (erreichbar nur
    dann, wenn oben/unten/rechts/oder links
192 //eine Zahl in der Kostenmatrix > 0 und < 22, Ziele sind immer nur am Rand, d.h. es
    ist von da aus immer nur ein index erreichbar
193 //gleichzeitig merken wir uns die Kosten des nächsten erreichbaren Punktes der
    gefundenen Fahraufträge (notwendig für die Priorisierung)
194
195 void erreichbar(){
196     int i;
197     int zaehler = 0;
198     for (i = 0; i < 70; i++)
199     {
```

```
200         if (kosten2[i] == 22)
201         {
202             //Ziel ist rechts?
203             if (i % 7 == 6)
204             {
205                 if (kosten2[i - 1] >= 0 && kosten2[i - 1] < 22) // Punkt westlich
                    vom Ziel ist erreichbar!
206                 {
207                     erreichbar_M[zaehler] = i;
208                     kosten_erreichbar[zaehler] = kosten2[i - 1];
209                     zaehler++;
210                 }
211             }
212
213             //Ziel ist links
214             else if (i % 7 == 0)
215             {
216                 if (kosten2[i + 1] >= 0 && kosten2[i + 1] < 22) // Punkt östlich vom
                    Ziel ist erreichbar!
217                 {
218                     erreichbar_M[zaehler] = i;
219                     kosten_erreichbar[zaehler] = kosten2[i + 1];
220                     zaehler++;
221                 }
222             }
223
224             //Ziel ist unten
225             else if (i >= 63 && i < 70)
226             {
227                 if (kosten2[i - 7] >= 0 && kosten2[i - 7] < 22) //Punkt nördlich vom
                    Ziel ist erreichbar!
228                 {
229                     erreichbar_M[zaehler] = i;
230                     kosten_erreichbar[zaehler] = kosten2[i - 7];
231                     zaehler++;
232                 }
233             }
234
235             //Ziel ist oben
236             else if (i >= 0 && i < 7)
237             {
238                 if (kosten2[i + 7] >= 0 && kosten2[i + 7] < 22) //unten erreichbar!
239                 {
240                     erreichbar_M[zaehler] = i;
241                     kosten_erreichbar[zaehler] = kosten2[i + 7];
242                     zaehler++;
243                 }
244             }
245         }
246     }
247 }
248
249 /*****Priorisieren der Reihenfolge der Fahraufträge *****/
250 void prioFahrauftraege()
251 {
252     // in Abhängigkeit von den Kosten des nächsten erreichbaren Punktes wird die
        Priorität der Fahraufträge festgelegt
253     // Bubblesort
254     int i, j, tmp;
255
256     for (i = 0; i < 3; i++)
257     {
258         for (j = i; j < 3; j++)
259         {
260             if (kosten_erreichbar[i] > kosten_erreichbar[j])
261             {
262                 tmp = erreichbar_M[i];
263                 erreichbar_M[i] = erreichbar_M[j];
264                 erreichbar_M[j] = tmp;
265             }
266         }
```

```
267         }
268     }
269
270     /******Berechnen der Wege*******/
271     //
272     void berechneRueckwege()
273     {
274         int i;
275         int j = 0;
276         int guenstiger_knoten = 99;
277         int akt_knoten = 1; //Stelle, an die der berechnete günstigste nächste Knoten
            eingetragen wird
278
279         // in erreichbar_M stehen nun die Adressen der sortierten Fahraufträge Falls in
            erreichbar_M an den vorderen Stellen
280         // jetzt noch eine 0 steht, wurden weniger als 3 Fahrauftraege gefunden.
281         // Die tatsächlichen Fahraufträge werden jeweils dann auch schon in dem Array
            für die Rückwege an die erste Stelle der
282         // Zeile (je nach Priorität) gesetzt.
283         for (i = 0; i < 3; i++)
284         {
285             if (erreichbar_M[i] != 0)
286             {
287                 rueckwege[i][0] = erreichbar_M[i];
288             }
289         }
290
291         // Ausgehend vom jeweiligen Zielpunkt werden die nächsten erreichbaren Punkte
            mit den geringsten Kosten an die nächste Stelle gesetzt,
292         // bis kein nächster Punkt mehr zu finden ist
293         for (i = 0; i <3; i++)
294         {
295             for (j = 0; j < 30; ++j)
296             {
297                 if (rueckwege[i][j] != 99)
298                 {
299                     // Nördlicher Punkt ist erreichbar, hat geringere Kosten als der
                        aktuelle Punkt und hat
300                     // geringere Kosten als ein Zielpunkt
301                     if ((kosten2[rueckwege[i][j] - 7] < kosten2[rueckwege[i][j]]) &&
                        kosten2[rueckwege[i][j] - 7] < 22
302                         && kosten2[rueckwege[i][j] - 7] > -1 && rueckwege[i][j] - 7 >-1
                            && rueckwege[i][j] - 7 < 70)
303                     {
304                         guenstiger_knoten = rueckwege[i][j] - 7; //aktuell guenstigster
                            naechster Knoten
305                     }
306
307                     // südlicher Punkt ist erreichbar, hat geringere Kosten als der
                        aktuelle Punkt, hat
308                     // geringere Kosten als ein Zielpunkt
309                     // und geringere Kosten als der bisher in guenstiger_knoten
                        gespeicherte Wert
310
311                     if ((kosten2[rueckwege[i][j] + 7] < kosten2[rueckwege[i][j]]) &&
                        kosten2[rueckwege[i][j] + 7] < 22
312                         && kosten2[rueckwege[i][j] + 7] > -1 && kosten2[rueckwege[i][j]
                            + 7] < guenstiger_knoten
313                         && rueckwege[i][j] + 7 >-1 && rueckwege[i][j] + 7 < 70)
314                     {
315                         guenstiger_knoten = rueckwege[i][j] + 7; //aktuell guenstigster
                            naechster Knoten
316                     }
317
318                     // westlicher Punkt ist erreichbar, hat geringere Kosten als der
                        aktuelle Punkt, hat
319                     // geringere Kosten als ein Zielpunkt
320                     // und geringere Kosten als der bisher in guenstiger_knoten
                        gespeicherte Wert
321
322                     if ((kosten2[rueckwege[i][j] - 1] < kosten2[rueckwege[i][j]]) &&
```

```
                    kosten2[rueckwege[i][j] - 1] < 22
323                     && kosten2[rueckwege[i][j] - 1] > -1 && kosten2[rueckwege[i][j]
                        - 1] < guenstiger_knoten
324                     && rueckwege[i][j] - 1 >-1 && rueckwege[i][j] - 1 < 70)
325                 {
326                     guenstiger_knoten = rueckwege[i][j] - 1; //aktuell guenstigster
                        naechster Knoten
327                 }
328
329                 // östlicher Punkt ist erreichbar, hat geringere Kosten als der
                    aktuelle Punkt, hat
330                 // geringere Kosten als ein Zielpunkt
331                 // und geringere Kosten als der bisher in guenstiger_knoten
                    gespeicherte Wert
332
333                 if ((kosten2[rueckwege[i][j] + 1] < kosten2[rueckwege[i][j]]) &&
                    kosten2[rueckwege[i][j] + 1] < 22
334                     && kosten2[rueckwege[i][j] + 1] > -1 && kosten2[rueckwege[i][j]
                        + 1] < guenstiger_knoten
335                     && rueckwege[i][j] + 1 >-1 && rueckwege[i][j] + 1 < 70)
336                 {
337                     guenstiger_knoten = rueckwege[i][j] + 1; //aktuell guenstigster
                        naechster Knoten
338                 }
339
340                 if (guenstiger_knoten < 99)
341                 {
342                     rueckwege[i][akt_knoten] = guenstiger_knoten;
343                 }
344                 akt_knoten++;
345                 guenstiger_knoten = 99; //Rücksetzen
346             }
347         }   akt_knoten = 1; //bei neuem Rückweg, Rücksetzen des Wertes für den
            aktuellen Knotens auf 1
348     }
349 }
350
351 // Die berechneten Rückwege werden nun gespiegelt und in das Array für die Hinwege
    eingefügt
352
353 void berechneHinwege()
354 {
355     int k = 0;
356     int i, j;
357     {
358         for (i = 0; i < 3; i++)
359         {
360             for (j = 29; j >= 0; j--)
361             {
362                 hinwege[i][k] = rueckwege[i][j];
363                 k++;
364             }
365             k = 0;
366         }
367     }
368 }
369
370 // Hin- und Rückwege werden nun zu einer Sequenz zusammengefügt
371 void mergeWege()
372 {
373     int index = 0;
374     int i;
375
376     for (i = 0; i < 30; i++)
377     {
378         if (hinwege[0][i] < 99)
379         {
380             allewege[index] = hinwege[0][i];
381             index++;
382         }
383     }
```

```
384
385         for (i = 0; i < 30; i++)
386         {
387             if (rueckwege[0][i] < 99)
388             {
389                 allewege[index] = rueckwege[0][i];
390                 index++;
391             }
392         }
393
394         for (i = 0; i < 30; i++)
395         {
396             if (hinwege[1][i] < 99)
397             {
398                 allewege[index] = hinwege[1][i];
399                 index++;
400             }
401         }
402
403         for (i = 0; i < 30; i++)
404         {
405             if (rueckwege[1][i] < 99)
406             {
407                 allewege[index] = rueckwege[1][i];
408                 index++;
409             }
410         }
411
412         for (i = 0; i < 30; i++)
413         {
414             if (hinwege[2][i] < 99)
415             {
416                 allewege[index] = hinwege[2][i];
417                 index++;
418             }
419         }
420
421         for (i = 0; i < 30; i++)
422         {
423             if (rueckwege[2][i] < 99)
424             {
425                 allewege[index] = rueckwege[2][i];
426                 index++;
427             }
428         }
429     }
430
431     // Entspricht der
432     void berechneWege()
433     {
434         int i;
435         //char roboter_richtung = 'N'; global
436         // Die erste Stelle in wege bekommt ein "S" zum Beladen der ersten Pizza und für
        die Erkennung der Startlampe
437
438         wege[0] = 'S';
439
440         for (i = 0; i < 90; i++)
441         {
442             if (allewege[i] > 0)
443             {
444                 // Roberta schaut Ri Norden und nächster Schritt liegt im Osten
445                 // Roberta fährt also nach rechts und sie schaut jetzt Ri Osten
446                 if (allewege[i + 1] - allewege[i] == 1 && roboter_richtung == 'N')
447                 {
448                     wege[i + 1] = 'R';
449                     roboter_richtung = 'O';
450                 }
451                 // Roberta schaut Ri Norden und nächster Schritt liegt im Westen
452                 // Roberta fährt also nach links und sie schaut jetzt Ri Westen
453                 else if (allewege[i + 1] - allewege[i] == -1 && roboter_richtung == 'N')
```

```
454             {
455                 wege[i + 1] = 'L';
456                 roboter_richtung = 'W';
457             }
458             // Roberta schaut Ri Norden und nächster Schritt liegt im Norden
459             // Roberta fährt also geradeaus und sie schaut weiterhin Ri Norden
460             else if (allewege[i + 1] - allewege[i] == -7 && roboter_richtung == 'N')
461             {
462                 wege[i + 1] = 'G';
463             }
464             // Roberta schaut Ri Norden und nächster Schritt liegt an der selben
                Stelle
465             // Kann nur ein Zielpunkt sein
466             // Roberta wendet also (und lädt ab) und ändert die Richtung in 'S'
467             else if (allewege[i + 1] - allewege[i] == 0 && roboter_richtung == 'N')
468             {
469                 wege[i + 1] = 'A';
470                 roboter_richtung = 'S';
471             }
472
473             //Süden
474             // Roberta schaut Ri Süden und nächster Schritt liegt im Osten
475             // Roberta fährt also nach links und sie schaut jetzt Ri Osten
476             else if (allewege[i + 1] - allewege[i] == 1 && roboter_richtung == 'S')
477             {
478                 wege[i + 1] = 'L';
479                 roboter_richtung = 'O';
480             }
481             // Roberta schaut Ri Süden und nächster Schritt liegt im Westen
482             // Roberta fährt also nach rechts und sie schaut jetzt Ri Westen
483             else if (allewege[i + 1] - allewege[i] == -1 && roboter_richtung == 'S')
484             {
485                 wege[i + 1] = 'R';
486                 roboter_richtung = 'W';
487             }
488             // Roberta schaut Ri Süden und nächster Schritt liegt im Süden
489             // Roberta fährt also geradeaus und sie schaut weiterhin Ri Süden
490             else if (allewege[i + 1] - allewege[i] == +7 && roboter_richtung == 'S')
491             {
492                 wege[i + 1] = 'G';
493             }
494             // Roberta schaut Ri Süden und nächster Schritt liegt an der selben
                Stelle
495             // Ist entweder ein Ziel ('A' für Abladen der Pizza) und Wendemanöver //
496             // oder ein Start ('B') für Wendemanöver und Beladen
497             // Roberta wendet also und ändert die Richtung in 'S'
498             else if (allewege[i + 1] - allewege[i] == 0 && roboter_richtung == 'S')
499             {
500                 if (allewege[i] == startpunkt)
501                 {
502                     wege[i + 1] = 'B';
503                 }
504                 else
505                 {
506                     wege[i + 1] = 'A';
507                 }
508                 roboter_richtung = 'N';
509             }
510
511             //Osten
512             // Roberta schaut Ri Osten und nächster Schritt liegt im Osten
513             // Roberta fährt also geradeaus
514             else if (allewege[i + 1] - allewege[i] == 1 && roboter_richtung == 'O')
515             {
516                 wege[i + 1] = 'G';
517             }
518             // Roberta schaut Ri Osten und nächster Schritt liegt im Norden
519             // Roberta fährt also nach links und sie schaut jetzt Ri Norden
520             else if (allewege[i + 1] - allewege[i] == -7 && roboter_richtung == 'O')
521             {
522                 wege[i + 1] = 'L';
```

```
523                 roboter_richtung = 'N';
524             }
525             // Roberta schaut Ri Osten und nächster Schritt liegt im Süden
526             // Roberta fährt also nach rechts und sie schaut richtung Süden
527             else if (allewege[i + 1] - allewege[i] == +7 && roboter_richtung == 'O')
528             {
529                 wege[i + 1] = 'R';
530                 roboter_richtung = 'S';
531             }
532             // Roberta schaut Ri Osten und nächster Schritt liegt an der selben
                Stelle
533             // Ist entweder ein Ziel ('A' für Abladen der Pizza) und Wendemanöver //
534             // oder ein Start ('B') für Wendemanöver und Beladen
535             // Roberta wendet also und ändert die Richtung in 'W'
536             else if (allewege[i + 1] - allewege[i] == 0 && roboter_richtung == 'O')
537             {
538                 if (allewege[i] == startpunkt)
539                 {
540                     wege[i + 1] = 'B';
541                 }
542                 else
543                 {
544                     wege[i + 1] = 'A';
545                 }
546                 roboter_richtung = 'W';
547             }
548
549             //Westen
550             // Roberta schaut Ri Westen und nächster Schritt liegt im Westen
551             // Roberta fährt also geradeaus
552             else if (allewege[i + 1] - allewege[i] == -1 && roboter_richtung == 'W')
553             {
554                 wege[i + 1] = 'G';
555             }
556             // Roberta schaut Ri Westen und nächster Schritt liegt im Norden
557             // Roberta fährt also nach rechts und sie schaut jetzt Ri Norden
558             else if (allewege[i + 1] - allewege[i] == -7 && roboter_richtung == 'W')
559             {
560                 wege[i + 1] = 'R';
561                 roboter_richtung = 'N';
562             }
563             // Roberta schaut Ri Westen und nächster Schritt liegt im Süden
564             // Roberta fährt also nach links und sie schaut richtung Süden
565             else if (allewege[i + 1] - allewege[i] == +7 && roboter_richtung == 'W')
566             {
567                 wege[i + 1] = 'L';
568                 roboter_richtung = 'S';
569             }
570             // Roberta schaut Ri Westen und nächster Schritt liegt an der selben
                Stelle
571             // Ist entweder ein Ziel ('A' für Abladen der Pizza) und Wendemanöver //
572             // oder ein Start ('B') für Wendemanöver und Beladen
573             // Roberta wendet also und ändert die Richtung in 'O'
574             else if (allewege[i + 1] - allewege[i] == 0 && roboter_richtung == 'W')
575             {
576                 if (allewege[i] == startpunkt)
577                 {
578                     wege[i + 1] = 'B';
579                 }
580                 else
581                 {
582                     wege[i + 1] = 'A';
583                 }
584                 roboter_richtung = 'O';
585             }
586
587         }
588     }
589 }
590
591 void fahreGeradeaus(){
```

```
592
593         if (black(analog(IR_MITTE))) // allet jut
594         {
595             motor_pwm(MOTOR_RECHTS, 10);
596             motor_pwm(MOTOR_LINKS, 10);
597
598         }
599
600
601         else if ((black(analog(IR_RECHTS))) && (weiss(analog(IR_LINKS))))
602         {
603             motor_pwm(MOTOR_RECHTS, 0);
604             motor_pwm(MOTOR_LINKS, 10);
605
606         }
607
608         else if ((weiss(analog(IR_RECHTS))) && (black(analog(IR_LINKS))))
609         {
610             motor_pwm(MOTOR_RECHTS, 10);
611             motor_pwm(MOTOR_LINKS, 0);
612
613         }
614     }
615
616     void biegeLinksAb()
617     {
618
619         motor_pwm(MOTOR_RECHTS, 10);
620         motor_pwm(MOTOR_LINKS, 0);
621         sleep(200);
622         motor_pwm(MOTOR_RECHTS, 10);
623         motor_pwm(MOTOR_LINKS, 0);
624         while (weiss(analog(IR_LINKS)));
625     }
626
627     void biegeRechtsAb()
628     {
629         motor_pwm(MOTOR_RECHTS, 0);
630         motor_pwm(MOTOR_LINKS, 10);
631         sleep(200);
632
633         motor_pwm(MOTOR_RECHTS, 0);
634         motor_pwm(MOTOR_LINKS, 10);
635         while (weiss(analog(IR_RECHTS)));
636     }
637
638     void dreheUm()
639     {
640         //Räder entgegengesetzt
641         motor_richtung(MOTOR_LINKS, 0);
642         motor_richtung(MOTOR_RECHTS, 0);
643
644         // drehen über die schwarze Linie
645         motor_pwm(MOTOR_RECHTS, 10);
646         motor_pwm(MOTOR_LINKS, 10);
647
648         sleep(800);
649
650         motor_pwm(MOTOR_RECHTS, 9);
651         motor_pwm(MOTOR_LINKS, 10);
652         while (weiss(analog(IR_MITTE)));
653
654
655         motor_pwm(MOTOR_RECHTS, 9);
656         motor_pwm(MOTOR_LINKS, 10);
657         while (black(analog(IR_MITTE)));
658
659         motor_pwm(MOTOR_RECHTS, 9);
660         motor_pwm(MOTOR_LINKS, 10);
661         while (weiss(analog(IR_MITTE)));
662
```

```
663
664         motor_pwm(MOTOR_RECHTS, 9);
665         motor_pwm(MOTOR_LINKS, 10);
666         while (weiss(analog(IR_MITTE)) && weiss(analog(IR_RECHTS)));
667
668         // Rückwärts bis hinter die Kreuzung
669         motor_richtung(MOTOR_LINKS, 1);
670         motor_richtung(MOTOR_RECHTS, 0);
671
672         motor_pwm(MOTOR_RECHTS, 10);
673         motor_pwm(MOTOR_LINKS, 9);
674         while (weiss(analog(IR_RECHTSAUSSEN)));
675
676         motor_pwm(MOTOR_RECHTS, 10);
677         motor_pwm(MOTOR_LINKS, 9);
678         while (black(analog(IR_RECHTSAUSSEN)));
679
680         //Motoren zurückstellen
681         motor_richtung(MOTOR_LINKS, 0);
682         motor_richtung(MOTOR_RECHTS, 1);
683     }
684
685     void ablegen()
686     {
687         servo_arc(0, 20);
688         sleep(500);
689         motor_richtung(MOTOR_LINKS, 1);
690         motor_richtung(MOTOR_RECHTS, 0);
691         servo_arc(0, 60);
692
693         motor_pwm(MOTOR_LINKS, 6);
694         motor_pwm(MOTOR_RECHTS, 5);
695         sleep(800);
696
697         motor_richtung(MOTOR_LINKS, 0);
698         motor_richtung(MOTOR_RECHTS, 1);
699
700     }
701
702
703
704
705     //Hauptprogrammroutine
706     void AksenMain(void)
707     {
708         int i = 1;
709
710         leseFahrauftrag();
711         startPunktEintragen();
712         lcd_uint(startpunkt);
713         sleep(1000);
714         fuelleAgenda();
715         kostenfuellen();
716         erreichbar();
717         prioFahrauftraege();
718         berechneRueckwege();
719         berechneHinwege();
720         mergeWege();
721         berechneWege();
722
723         lcd_cls();
724
725         for (i = 0; i < 16; i++)
726             lcd_putchar(wege[i]);
727         sleep(1500);
728         lcd_cls();
729
730         i = 1; // S wird ignoriert
731
732         //Infrarot einschalten
733         led(IR_LINKS, 1);
```

```
734         led(IR_RECHTS, 1);
735         led(IR_MITTE, 1);
736         led(IR_RECHTSAUSSEN, 1);
737         led(IR_LINKSAUSSEN, 1);
738         led(PHOTO, 1);
739 
740 
741         motor_richtung(MOTOR_LINKS, 0);
742         motor_richtung(MOTOR_RECHTS, 1);
743 
744         servo_arc(0, 60); // Servo zum Ball nehmen einstellen
745 
746         while (analog(8) > 30);
747 
748 
749 
750         while (1)
751         {
752 
753             if (digital_in(0) == 0){
754                 ablegen();
755                 dreheUm();
756             }
757             else if (black(analog(IR_RECHTSAUSSEN)))
758             {
759 
760                 if (wege[i] == 'R')
761                 {
762                     biegeRechtsAb();
763                 }
764                 else if (wege[i] == 'L')
765                 {
766                     biegeLinksAb();
767                 }
768 
769                 else if (wege[i] == 'G' && wege[i - 1] == 'A')
770                 {
771                 }
772 
773                 else if (wege[i] == 'G')
774                 {
775                     motor_pwm(MOTOR_LINKS, 10);
776                     motor_pwm(MOTOR_RECHTS, 10);
777                     sleep(300);
778                 }
779 
780                 else if (wege[i] == 'B' || wege[i] == 'A')
781                 {
782                 }
783 
784                 else
785                 {
786                     motor_pwm(MOTOR_LINKS, 0);
787                     motor_pwm(MOTOR_RECHTS, 0);
788                     lcd_putchar(wege[i]);
789                     while (1);
790 
791                 }
792 
793                 i++;
794                 lcd_uint(i);
795                 lcd_puts(" ");
796                 lcd_putchar(wege[i]);
797 
798             }
799             fahreGeradeaus();
800         }
801     }
802 
803 
804 
```