Projektseminar Echtzeitsysteme

Team Nullpointer
Projektseminar eingereicht von
Alina Weber, Robert Wiesner, Marcel Mann, Ralf Kundel
am 7. April 2015

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachgebiet Echtzeitsysteme

Elektrotechnik und Informationstechnik (FB18)

Zweitmitglied Informatik (FB20)

Prof. Dr. rer. nat. A. Schürr
Merckstraße 25
64283 Darmstadt

www.es.tu-darmstadt.de

Gutachter: Géza Kulcsár
Betreuer: Géza Kulcsár

Erklärung zum Projektseminar

Hiermit versichern wir, das vorliegende Projektseminar selbstständig und ohne Hilfe Dritter angefertigt zu haben. Gedanken und Zitate, die wir aus fremden Quellen direkt oder indirekt übernommen haben, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen und wurde bisher nicht veröffentlicht.

Wir erklären uns damit einverstanden, dass die Arbeit auch durch das Fachgebiet Echtzeitsysteme der Öffentlichkeit zugänglich gemacht werden kann.

Darmstadt, den 7. April 2015

(Alina Weber, Robert Wiesner, Marcel Mann, Ralf Kundel)

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einführung

1.1 Aufgabenstellung

Die Aufgabenstellung des Projektseminars Echtzeitsysteme an der TU Darmstadt im Wintersemester 2014/15 ist die Programmierung von autonom fahrenden Modellautos, ausschließlich durch die Verwendung von am Auto installierter Sensorik, gewesen. Die Aufgabenstellung lässt sich in vier Bereiche unterteilen, die in dieser Arbeit in den folgenden Kapiteln genauer erläutert werden. Diese vier Bereiche sind

- Geradeaus Fahren und Kurve Nehmen
- Einparken
- RC-Modus
- Car-2-Car Kommunikation

Bei all diesen Aufgaben war ein gewisses Maß an Spielraum für eigene Ideen und Erweiterungen vorhanden. Des Weiteren gab es in der Aufgabenstellung die Möglichkeit, einige über die Pflichtaufgaben hinausgehende Funktionalitäten zu implementieren, beispielsweise das Einparken im rechten Winkel zur Fahrtrichtung.

1.2 Überblick der bereitgestellten Hardware und Software

Die zur Verfügung gestellte Hardware sind handelsübliche Modellautos, die mit einem Mikrocontroller, Sensorik und einem Funkmodul ausgestattet sind. Der Mikrocontroller ist ein 16 Bit Mikrocontroller aus der MB96300-Serie von Fujitsu Microelectronics. Als Betriebssystem wurde das Open-Source Echtzeitbetriebssystem FreeRTOS[7] verwendet. Als Antrieb kam ein normaler Gleichstrommotor zum Einsatz, der durch einen Fahrtregler nahezu stufenlos angesteuert werden kann. Die Lenkung wird, wie es bei normalen Modellautos üblich ist, durch ein Servo angesteuert. Sowohl der Fahrtregler als auch das Lenkservo sind an den Mikrocontroller angeschlossen, der über ein pulsweitenmoduliertes Signal den aktuellen Motorwert bzw. Lenkeinschlag übermittelt. Die drei Ultraschallsensoren, der Drehzahlsensor, das 868 MHz Funkmodul sowie der Bluetooth-Chip sind über serielle Busse mit dem Controller verbunden. Die softwareseitige Integration der Hardware wurde bereits im Rahmen vorhergehender Projektseminare integriert und als fertige API zur Verfügung gestellt. Ein Kerntask, der Bestandteil der API ist, hat regelmäßig alle Sensoren abgefragt und die aktuellen Werte in Datenstrukturen abgelegt. Dadurch war es möglich, direkt auf die Sensorwerte zuzugreifen sowie den Fahrzeugaktoren Steuerbefehle zu senden, ohne zeitaufwändiges Implementieren von Schnittstellen und Treibern.

2 Geradeaus Fahren und Kurven Nehmen

Erste Aufgabe im Rahmen des Projektseminars war es, eine Regelung zu implementieren, durch die das Auto in einem konstanten Abstand zur rechten Wand fährt und um Kurven steuern kann.

2.1 Grundlagen der Abstandsregelung

Die Eingangsgröße für die Regelung ist der gewünschte Abstand zur Wand (Führungsgröße), die durch die Regelung möglichst exakt und schnell eingehalten werden soll. Ausgang des Reglers ist der Lenkeinschlag (Stellgröße), durch den das Auto (Regelstrecke) beeinflusst wird. Als Grundlage für die Regelung wird die Differenz zwischen Führungsgröße und Regelgröße gebildet, welche in der Regelungstechnik als Regelabweichung bezeichnet wird. Diese Regelabweichung ist in unserem Fall die Differenz zwischen der gewünschten und der tatsächlichen Distanz zur Wand. Daraus ergibt sich das folgende Blockschaltbild des geschlossenen Regelkreises:

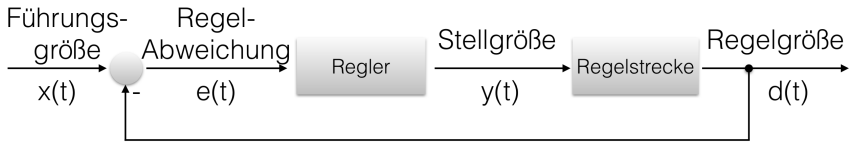


Abbildung 2.1: geschlossener Regelkreis des gesamten Systems

Aufgrund von Vorkenntnissen aus einer anderen Veranstaltung[1] war bekannt, dass ein PID-Regler für diese Art von Regelung vermutlich am geeignetsten sein würde. Bei einem PID-Regler setzt sich das Ausgangssignal y(t), das in unserem Fall den Lenkeinschlag darstellt, aus drei Komponenten zusammen. Diese sind alle von der Regelabweichung e(t) abhängig. Die erste Komponente ist proportional zu e(t). Wenn das Auto weiter als gewünscht von der Wand entfernt ist, wird dadurch ein Lenkeinschlag in Richtung Wand gesteuert, der proportional zur Regelabweichung ist. Die zweite Komponente ist der Integralanteil. Hier wird der Fehler e(t) über die Zeit integriert. Dadurch entsteht eine stationäre Genauigkeit, wodurch Störeinflüsse, wie z. B. eine schräg eingestellte Lenkung, kompensiert werden können. Die dritte Komponente ist der Differentialanteil. Hier wird die Veränderung der Regelabweichung über die Zeit betrachtet. Wenn das Auto schräg auf die Wand zufährt, ist dieser Anteil negativ, wenn es von der Wand wegfährt positiv. Dadurch wird das Gesamtsystem stabilisiert, da das Auto ansonsten ab einer gewissen Geschwindigkeit Schlangenlinien fahren würde. Diese drei Komponenten werden jeweils mit einem konstanten Faktor multipliziert und anschließend addiert. In Abbildung 2.2 ist dies grafisch dargestellt.

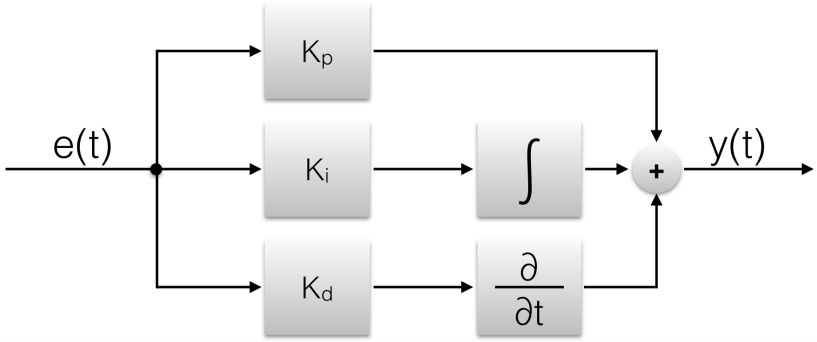


Abbildung 2.2: prinzipieller Aufbau eines PID-Reglers

Diese mathematischen Zusammenhänge lassen sich wie folgt beschreiben:

$$e(t) = x(t) - d(t)$$
$$y(t) = K_p e(t) + K_d \frac{\partial e(t)}{\partial t} + K_i \int_0^t e(t)\,\mathrm{d}t$$

Bei dieser Betrachtung wird davon ausgegangen, dass der gemessene Abstand der tatsächliche Abstand zur Wand ist. Sobald das Auto jedoch schräg zur Wand steht, ist die gemessene Distanz höher als die tatsächliche Distanz, siehe Abbildung 2.3.

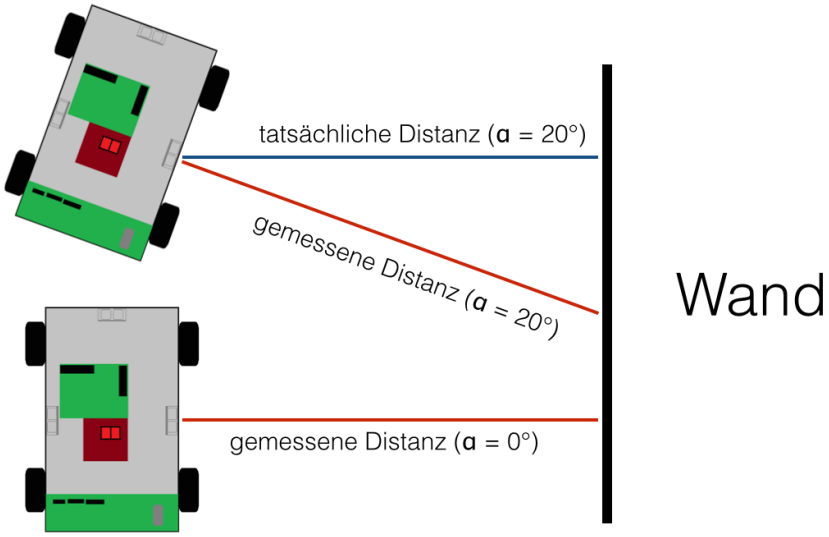


Abbildung 2.3: Wanddistanz in Abhängigkeit von α

Da sich der Winkel in dem das Auto zur Wand steht nicht zuverlässig messen und berechnen lässt, haben wir uns dazu entschieden, die Regelung mit dem gemessenen Wert aufzubauen. Für kleine Winkel ist dieser Fehler jedoch so gering, dass dadurch keine negativen Effekte auftreten. Beispielsweise bei einem Winkel von 30° ist die tatsächliche

Distanz um ca. 13% kleiner als die gemessene Distanz. In unserer Implementierung des Reglers haben wir verhindert, dass das Auto in einem zu großen Winkel auf die Wand zufährt, weswegen unsere Lösung durch diese Falschmessung nicht negativ beeinflusst wird.

2.1.1 Implementierung des PD-Reglers

Um den oben beschriebenen Regler auf den Einplatinencomputern der Autos zu implementieren, waren einige Anpassungen nötig. Hauptunterschied zwischen der Theorie und der Praxis ist, dass keine kontinuierliche, sondern nur eine diskrete Signalverarbeitung möglich ist. Ursprünglich hatten wir eine Funktion implementiert, die im Intervall von 300ms durch das Betriebssystem aufgerufen wird, und dabei jedes mal die aktuelle Ableitung und das Integral berechnet. Diese Werte wurden in globalen Variablen gespeichert, sodass andere Teile des Programms darauf zugreifen und damit dann beispielsweise den Lenkwinkel berechnen können.

Die Implementierung der diskreten Ableitung gestaltete sich relativ einfach, da man lediglich die Differenz zwischen dem aktuellen Wert und dem vorherigen Wert bilden musste.

$$\frac{\Delta e(t)}{\Delta t} = e(t) - e(t - 300ms)$$

Die Implementierung der Integration ist deutlich komplizierter, da der Wert der Summe aller vorhergehenden Werte entspricht, was in der Praxis zu extrem großen Fehlern und damit zu Problemen führen kann. Da wir bei Tests durch den Integralanteil keine nennenswerten Vorteile feststellen konnten, haben wir uns dazu entschieden, nur einen PD-Regler zu nutzen. Die stationäre Genauigkeit haben wir aber trotzdem erreicht, indem wir für jedes Auto einen Servo-Offset ermittelt haben, der in der API automatisch über die Car-ID ermittelt und zum Signal addiert wird. Dadurch verhält sich jedes Auto annähernd identisch.

Da im Laufe des Projektseminars die Rechenleistung des Mikrocontrollers an seine Grenzen kam, haben wir die oben beschriebene Methode, die alle 300ms die aktuellen Werte berechnet, in die API ausgelagert. Da es in der API ohnehin schon einen Task gab, der regelmäßig die Sensorwerte in die Struktur Us-Data schreibt, haben wir diese Funktion so erweitert, dass bei jeder Aktualisierung der Sensorwerte auch die Ableitung berechnet und ebenfalls in die, leicht modifizierte Struktur Us-Data geschrieben wird. Die Berechnung der Ableitung aufgrund von zwei Messungen, die direkt hinter einander ausgeführt wurden, ist nicht sonderlich exakt, weswegen wir die Ableitung immer über die letzten vier Messwerte berechnet haben. Außerdem haben wir die Messung der Ultraschallsensoren von einer Genauigkeit von 1cm auf die Genauigkeit von 1mm umgebaut. Die alten Datenstrukturen, in denen die Messwerte in Zentimetern gespeichert wurden, sind weiterhin vorhanden. Dieser Wert wird durch eine Kombination von Shift- und Additions-Operationen berechnet, sodass keine teure Division durch zehn nötig wird. Dieser Aufwand war nötig, da nur

so eine vollständige Abwärtskompatibilität der API garantiert werden konnte. Durch die Umstellung auf Millimeter und die Auslagerung der Berechnung der Ableitung war eine noch exaktere Regelung des Autos möglich.

2.1.2 Türrahmen-Filter

Eine Herausforderung stellten Türrahmen dar, da beim Vorbeifahren durch die kurzzeitige Änderung der gemessenen Distanz die Regelung starke Lenkausschläge steuerte. Ursache hierfür ist der Differentialanteil des PD-Reglers, da sich in einem sehr kurzen Zeitraum der Abstand zur Wand sehr stark verändert. Um diesen Effekt zu kompensieren, muss man den Differentialanteil des PD-Reglers für die Zeit der Türrahmenvorbeifahrt deaktivieren, was durch die Veränderung der Konstante K_d auf 0 problemlos möglich ist. Diese Problematik wird in Abbildung 2.4 dargestellt.

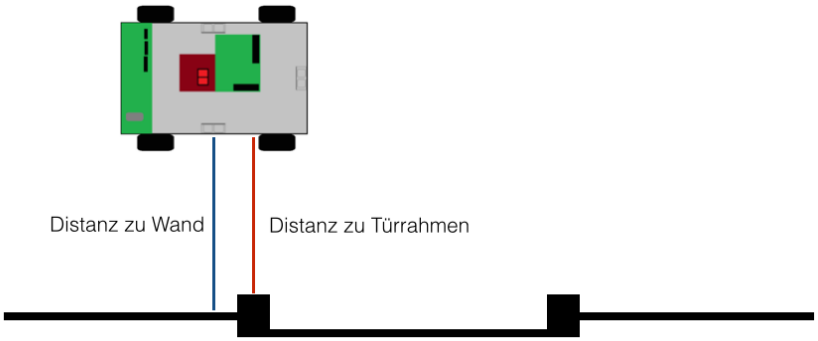


Abbildung 2.4: Abstände bei Türrahmenvorbeifahrt

Die Erkennung von Türrahmen ist etwas problematischer, da dadurch ausschließlich Türrahmen und keine anderen Veränderungen der Wanddistanz erkannt werden sollen. Wir haben hierbei mehrere Verfahren getestet, wobei wir die besten Ergebnisse mit der zweiten Ableitung der Wanddistanz erzielen konnten, welche normalerweise immer ungefähr 0 ist. Als Bedingung für einen Türrahmen haben wir die folgende Bedingung gewählt:

$$\left| \frac{\Delta e(t)}{\Delta t} - \frac{\Delta e(t-1)}{\Delta t} \right| \geq 10$$

Der PD-Regler nutzt dann für die nächsten fünf Berechnungen des Lenkwinkels nur den P-Anteil. Dadurch kann das Auto ungestört an Türrahmen vorbeifahren.

2.1.3 Universelle Verwendung des Reglers

Der von uns implementiere Regler ist ein eigenständiges Codemodul, das beliebig wiederverwendbar ist. So ist es möglich, dass verschiedene Konfigurationen des Autos, beispielsweise Szenario „Einparken“, immer denselben Regler nutzen. Dieses Codemodul liefert immer den aktuellen Lenkausschlag zurück, so dass das Auto geradeaus fährt. Durch

Setter-Methoden ist es möglich, dem Modul die gewünschte Wanddistanz sowie die aktuelle Geschwindigkeit mitzuteilen. Die Parameter K_d und K_p sind beide von der Geschwindigkeit abhängig. In Tabelle 2.1 sieht man die Zuordnung der Faktoren zu den verschiedenen Motorwerten. Dabei ist zu beachten, dass ab einem Tempo von acht keine Geschwindigkeitszunahme mehr vorliegt, wodurch sich ab hier die Faktoren auch nicht mehr ändern.

Tempo	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7	8	9	10
K_p	2	3	3	3	4	5	7	6	6	5	3	2	1	1	1	1
K_d	4	5	5	5	5	0	20	17	15	11	10	7	4	4	4	4

Tabelle 2.1: Parameter K_p und K_d in Abhängigkeit der Geschwindigkeit

2.2 Kurven

Zweiter Teil dieser Aufgabe war es, die Implementierung so zu erweitern, dass das Auto um Rechtskurven fahren kann. Zusätzliche Bonusaufgabe war es, dass dies nicht nur für Kurven von 90°, sondern auch für beliebige Winkel möglich ist, was mit unserer Lösung ebenfalls funktioniert. Außerdem haben wir die Logik um Linkskurven erweitert.

2.2.1 Kurvenerkennung

Herausforderung bei dieser Aufgabe war es, zu erkennen, dass das Auto eine Kurve fahren muss und wann diese zu Ende ist. Dies haben wir ebenfalls über den rechten Abstandssensor gelöst. Als Beginn für eine Rechtskurve haben wir die folgende Bedingung definiert:

$$Distanz(t) - Distanz(t-1) \geq 40cm$$

Zum Zeitpunkt $t-1$ war der Abstand zur Wand noch der gewünschte Sollwert, zum Zeitpunkt t ist der Abstand nun aber deutlich größer, da zwischenzeitlich das Ende der Wand erreicht wurde. Nun soll die Rechtskurve eingeleitet werden. Das Kurvenende ist erreicht, sobald sich der gemessene Abstand zur Wand wieder vergrößert. Das klingt zuerst unlogisch, ist aber korrekt, da der gemessene Abstand orthogonal zur Fahrtrichtung gemessen wird, wie man in Abbildung 2.3 sehen kann.

2.2.2 Fahrlogik

Die Fahrlogik wurde in Form eines Zustandsautomaten implementiert. Startzustand ist „Forward“, sodass das Auto standardmäßig erst ein mal geradeaus fährt, bis es eine Kurve erkennt. Sobald eine Kurve erkannt wird, was im Kapitel 2.2.1 genauer erklärt wird, wird in den Zustand „Right“ gegangen. Dieser wird verlassen, sobald die Kurve beendet ist. Als zusätzliche Erweiterung haben wir eine Linkskurvenregelung implementiert. Hierbei wird, sobald eine gewisse Frontdistanz unterschritten wird, die Linkskurve eingeleitet. Dabei fährt das Auto zuerst ein Stück mit vollem Lenkausschlag nach links, setzt dann mit vollem Lenkausschlag nach rechts zurück und beendet anschließend die Kurve. Dies ist in

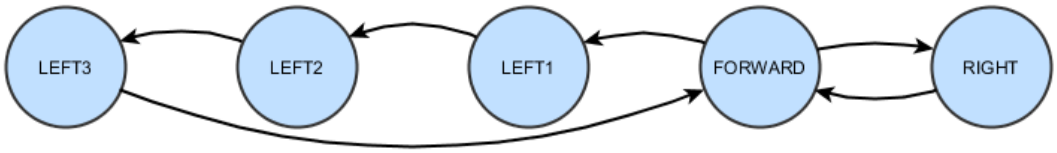


Abbildung 2.5: Zustandsgraph der Fahrlogik

Abbildung 2.5 durch die Zustände „Left1“ bis „Left3“ zu erkennen. Die ursprüngliche Lösung, die Kurve in einem Zug, also ohne Zurücksetzen zu fahren, wurde von uns aufgrund anhaltender Probleme wieder verworfen. Problem war hierbei, dass die Kurve schon sehr früh begonnen werden musste, was als Startbedingung einen relativ hohen Grenzwert für die Frontdistanz zur Folge hatte. Durch den großen Öffnungswinkel des Ultraschallsensors wurden nun aber auch reflektierende Türrahmen fälschlicherweise als Linkskurven erkannt.

3 Einparken

Eine weitere Aufgabenstellung umfasste das Einparken, das von unserer Gruppe sowohl parallel als auch orthogonal implementiert wurde. Da sich diese beiden Arten im Vorgehen des tatsächlichen Einparkens unterscheiden, werden sie in den jeweiligen Unterkapiteln genauer beschrieben. Die Abläufe der beiden Parkvorgänge lassen sich auf einfache Zustandsautomaten zurückführen, deren Übergänge durch Veränderungen der Distanz nach rechts oder über die gefahrene Strecke auslösen. Wir entschieden uns, das Einparken in weiten Teilen statisch zu implementieren, da die Messwerte der Ultraschallsensoren bei kleinen Distanzen unter 15 cm starke Schwankungen aufwiesen und ein rückwärtiger Sensor für das orthogonale Parken fehlte.

3.1 Erkennen einer Parklücke

Das Erkennen einer Parklücke erfolgt über die Veränderung des Abstandes zur rechten Wand. Hierbei wählten wir die Messung in Zentimetern und verglichen aus Gründen der Zuverlässigkeit nicht mit dem letzten sondern dem vorletzten Wert. Ein parkendes Auto wird hierbei durch eine Abfolge von einer Distanzverminderung um je nach Modus ca. 10 bis 20 cm und einer kurz darauf folgenden Distanzsteigerung dargestellt. Dieser Vorgang wird in unserm Code über die beiden Zustände CAR_DETECT und CAR_END_DETECT abgebildet. Wurde das Ende eines ersten parkenden Autos erkannt, wird der Zustand CAR2_DETECT ausgelöst und über den Hallsensor die Strecke bis zu einer erneuten Distanzminderung, also einem zweiten Auto, gemessen. Ist die gemessene Strecke kleiner als die entsprechende Schwelldistanz für den jeweiligen Einparkmodus, wird der Parkvorgang gestartet. Sollte die Parklücke zu klein sein, wird die Suche durch einen Sprung zurück in den Zustand CAR_DETECT wiederholt.

3.2 Berechnung der Wandabstände

Durch das Nutzen des PD-Reglers während der Parklückensuche bekamen wir das Problem, dass das Auto weiterhin versuchte, sich am Wandabstand zu orientieren und wir in Schlangenlinien an den parkenden Autos vorbeifuhren. Um dies zu verhindern, mussten wir bei jedem Zustandsübergang einen neuen Sollabstand nach rechts berechnen. Wir starteten mit einem konstanten Abstand zur Wand, der DISTANCE_TO_WALL. Sobald wir das erste Auto erreicht hatten, fuhren wir mit gerader Lenkung ohne Regelung und setzten die Variable distancetocar mit unserem aktuellen Sensormesswert. Mit diesen beiden Werten und der Konstante CAR_WIDTH ließ sich die Zieldistanz nach rechts, mit der wir später beim parallelen Einparken in der Parklücke stehen wollten, wie folgt berechnen:

$$final\_distance = DISTANCE\_TO\_WALL - distancetocar - CAR\_WIDTH \quad (1)$$

Bei der orthogonalen Variante entschieden wir uns, einen Zielabstand von 5 cm zum direkt benachbarten parkenden Auto vorzugeben. Bessere Werte waren durch den begrenzten Lenkeinschlag der Autos nicht möglich. Die Formel für den Abstand mit dem wir bei der parallelen Version zwischen den beiden Autos fuhren, berechnete sich mit den

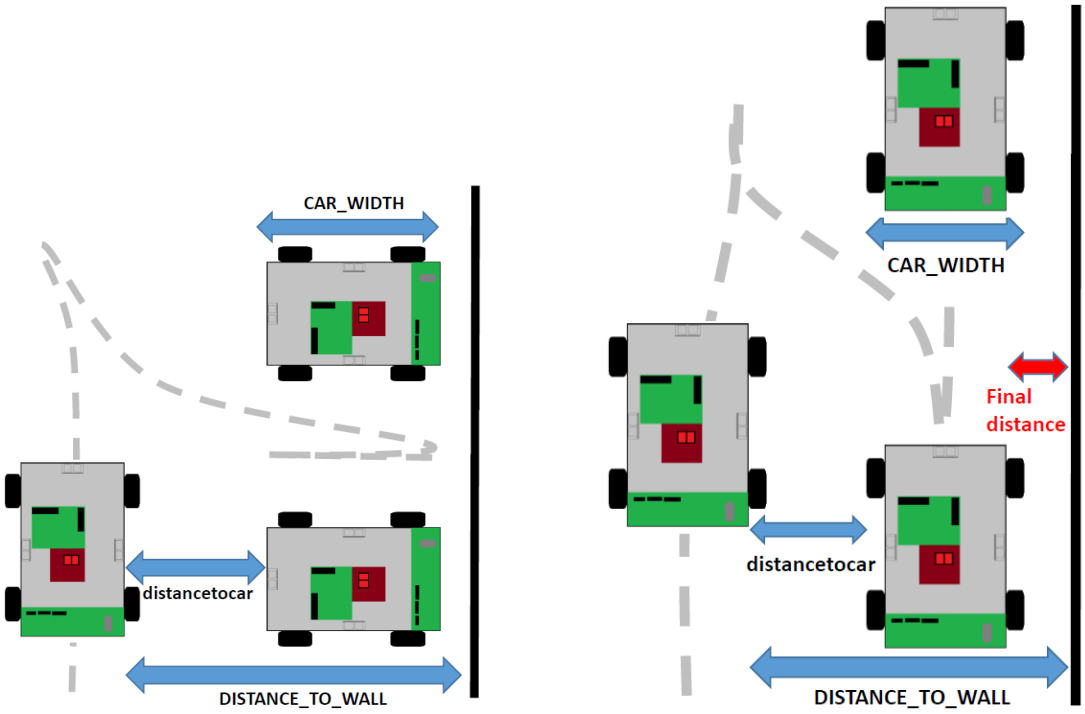
gegebenen Werten zu:

$$final\_distance + CAR\_WIDTH + 5 \tag{2}$$

Bei der orthogonalen Version fuhren wir entsprechend mit dem Abstand:

$$DISTANCE\_TO\_WALL - distancetocar - DISTANCE\_TO\_CAR \tag{3}$$

Hierbei stellt DISTANCE_TO_CAR einen Referenzwert dar, mit dem wir eine kleine Korrektur unseres bisherigen Fahrtweges vornehmen konnten. Hierdurch erreichten wir einen möglichst geringen Abstand zum zweiten Auto, an dem wir nun wieder mit gerader Lenkung vorbeifuhren, sodass wir eng die Kurve nehmen und die minimale Größe der Parklücke verkleinern konnten.



(a) orthogonales Einparken (b) paralleles Einparken

Abbildung 3.1: Distanzen der Parkvorgänge

3.3 Paralleles Einparken

Bei dieser Version des Einparkens starteten wir nach der Erkennung der Parklücke direkt mit dem Einlenken. Dieser Vorgang und das anschließende Gegenlenken sind aus bereits oben beschriebenen Gründen statisch implementiert. Hierbei setzen wir im vorangegangenen Zustand unsere aktuelle Fahrtdistanz, mit der wir jeweils die aktuelle Distanz vergleichen, und verwenden für beide Zustände je einen festen Einschlagswinkel, der die Servo-Offset Werte der jeweiligen Autos mit einbezieht. Sind wir entweder eine fixe Weite

von 100 Einheiten des Hallsensors während des Gegenlenkens zurückgefahren oder haben wir die Gesamtweite der Parklücke bereits ausgenutzt, ziehen wir unter Verwendung unseres PD-Reglers noch einmal nach vorne gerade, bis wir einen Abstand von 20cm zum vor uns parkenden Auto unterschritten haben.

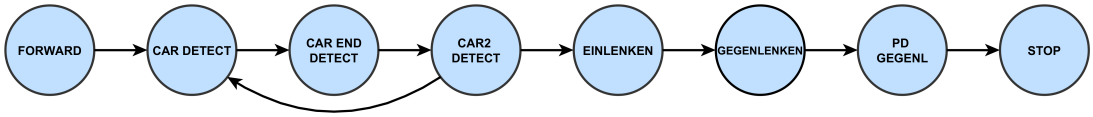


Abbildung 3.2: Zustandsautomat paralleles Einparken

3.4 Orthogonales Einparken

Um auch schmalere Parklücken nutzen zu können, entschieden wir uns bei diesem Modus, zunächst noch ein Stück nach vorne zu fahren und scharf nach links einzuschlagen, damit wir enger am zweiten bereits parkenden Auto um die Kurve kommen konnten. Hierdurch war es uns möglich, Parklücken von einer Breite ab etwa 45 cm zu nutzen. Dafür führten wir den neuen Zustand DETECTED ein, bei dem wir den Sonderfall beachten mussten, dass wir auch das Ende des zweiten Autos erreicht hatten, um einen korrekten Wandabstand einhalten zu können. Auch dieser Zustand wird wieder durch das Überschreiten einer festen Fahrtdistanz verlassen, worauf rückwärts in die Lücke gefahren wird. In diesem Fall musste eine statische Länge genutzt werden, da wir ohne rückwärtigen Sensor "blind"fahren und Kollisionen mit der Wand verhindern mussten. Die unterschiedlichen Lenkeinschläge bekamen bei dieser Aufgabe nun allerdings eine neue Tragweite, da wir je nach verwendetem Auto unterschiedlich schräg standen und sich diese minimalen Abweichungen des Einschlags auch nicht mehr mit einem Servo-Offset abfingen ließen. Dies führte dazu, dass wir unserem Automaten zwei weitere Zustände für das Vorwärts- und Rückwärtsfahren hinzufügten. Zunächst fahren wir mindestens einmal unter Verwendung unseres PD-Reglers eine kurze Strecke geradeaus und anschließend wieder rückwärts. Nach dem Zurückstoßen überprüfen wir die Ableitung unserer Wanddistanz. Sollte sich unser Abstand zum rechten Auto um mehr als 10mm geändert haben, starten wir einen neuen Korrekturzug, ansonsten beendeten wir den Einparkvorgang. Dieses Vorgehen erwies sich trotz der niedrigen Schwellwerte für die Ableitung der Distanz bei den meisten Autos als erstaunlich zuverlässig, sodass wir recht gute Ergebnisse erreichen konnten.

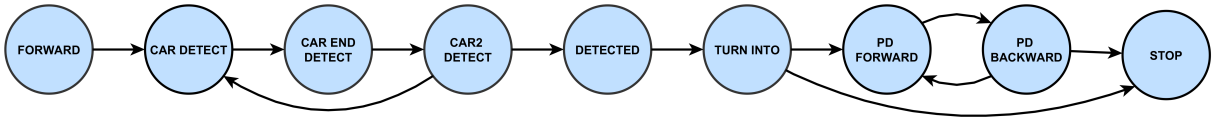


Abbildung 3.3: Zustandsautomat orthogonales Einparken

4 Remote-Control

Der Remote-Control-Modus sollte unter Verwendung einer der verfügbaren Funkverbindungen zum Auto implementiert werden. Zur Verfügung standen ein Bluetooth-Modul sowie das AMB8425 868MHz-Funkmodul [2], auf welches über den AMB8465 USB-Stick [3] Zugriff besteht. Die naheliegendste Möglichkeit war die Verwendung der Bluetoothverbindung, da schon ein einfaches Android-Smartphone weitreichende Gestaltungsmöglichkeiten bei der Entwicklung einer eigenen App bietet. Im späteren Verlauf des Projektes entstand die Idee, das Auto unter Verwendung eines Gamepads zu steuern. Teils um das Testen zu vereinfachen, teils um Zusatzfeatures zu implementieren, wurde der Funktionsumfang der App immer wieder erweitert und verändert.

Am Ende des Projekts können wir mit unserer entwickelten App nicht nur das Auto fernsteuern, sondern auch in die verschiedene Modi (Einparken, Wallfollow etc.) setzen. Zusätzlich ist die im Handling sehr ansprechende Steuerung über ein XBox-Gamepad von Microsoft möglich. Die einzelnen Entwicklungsschritte und Features unserer Ergebnisse sollen nun ausführlicher erklärt werden.

4.1 Die App

Um den zusätzlichen Aufwand, den das direkte Programmieren einer Android-App bedeutet hätte, zu sparen, griffen wir auf den kostenlos verfügbaren Online-Baukasten „MIT APP Inventor 2“ zurück [4]. Schon im privaten Bereich hatten wir bei der Kombination einer Inventor-App und der Bluetooth-Kommunikation des Smartphones positive Erfahrungen sammeln können, was uns zusätzlich in unserem Vorhaben bestärkte. Das Tool bietet eine grafische Browser-Programmieroberfläche, die es ermöglicht ohne großes Vorwissen eine eigene Android App zu implementieren. Dabei werden Code-Blöcke und User-Inteface Bausteine per Drag-and-Drop miteinander kombiniert.

Die ersten Schritte in der Entwicklung der App waren das Verstehen des Bluetooth-Protokolls und des vorgegebenen Software-Frameworks des Autos. Das Smartphone stellt im Bluetooth-Protokoll einen Bluetooth-Clienten, das Auto einen Bluetooth-Server dar. Folglich kann immer nur ein Smartphone als Client an einem Auto als Server angemeldet sein. Das Framework basiert auf Message-Konstrukten (einer Sequenz aus Bytes), die neben den eigentlich zu übermittelnden Daten, der Payload, auch noch eine Menge an zusätzlichen Informationen enthalten. Der App-Inventor stellt dazu eine passende Funktion bereit, die das einfache Versenden einer Liste von Bytes ermöglicht.

Aufbau der Messages:

1. Start-Byte: 0x02, STX nach ASCII-Kodierung
2. Lengthbyte: Die Anzahl der Bytes ohne Start-, Stop- und Lengthbyte
3. Source-ID-Byte: ID des Senders (bei C2C wichtig)
4. Destination-ID-Byte: ID des Empfängers (bei 868MHz-Verbindung zum Empfangen erforderlich)

5. Topic-ID-Byte: ID des Topics, auf das Methoden im Auto abbonnieren müssen um die Nachricht zu erhalten
6. Priority-Byte: je kleiner der Wert, desto höher die Priorität
7. Datalength-Byte: Anzahl der Bytes der Payload
8. Payload-Bytes: im Auto realisiert als Array
9. Stop-Byte: 0x03, ETX nach ASCII-Kodierung

4.2 User-Interface

Die Benutzeroberfläche der App ist in zwei Bereiche geteilt. Da das Smartphone im Breitformat gehalten wird, stehen zur Bedienung beide Daumen zur Verfügung. Im linken Bildschirmteil wird über verschiedene Checkboxes das gewünschte Feature der App ausgewählt. Auf der rechten Seite befinden sich dann die Feature-spezifischen Bedienelemente wie Listpicker und Send-Buttons. So wurde die App mit einfachen Mitteln mit einer intuitiven Bedienbarkeit ausgestattet.

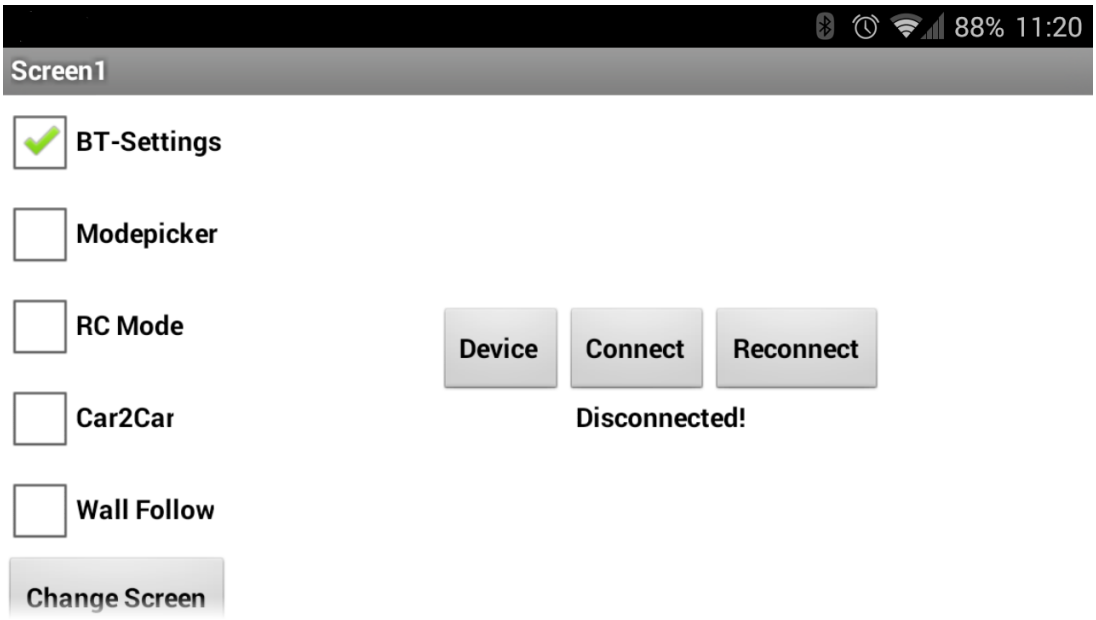


Abbildung 4.1: Screenshot der App im Bluetooth-Menü. Siehe auch Abbildung B.1 im Anhang

4.3 Die Features im Einzelnen

4.3.1 Verbindungsverwaltung

Um die App mit dem Auto koppeln zu können, müssen die Geräte zuerst im Bluetooth-Menü des Telefons mit dem Code '1234' gekoppelt werden. In der App ist ein Drop-Down Menü implementiert, welches das gewünschte Auto auswählen lässt. Über einen Connect/Disconnect-Button wird die Verbindung aufgebaut bzw. beendet. Wird das Auto neugestartet, kann ein Verbindungsneuaufbau über den Reconnect-Button ausgelöst werden. Außerdem zeigt ein Label ständig den aktuellen Verbindungsstatus an. Codeblöcke: Siehe Abbildung B.2 im Anhang.

4.3.2 Fernsteuerungsfunktionalität

Zur Fernsteuerung besteht die Payload aus zwei Bytes, jeweils ein Byte für Geschwindigkeits- und Lenkwert. Zur Fernsteuerung wird immer 'Topic 0' verwendet. Der erste und einfachste Ansatz zur Fernsteuerung war die Verwendung von zwei separaten Slidern, über die sich die Steuerungsparameter einstellen lassen und die bei jeder Änderung eine Message senden. Da die Steuerung mit Slidern jedoch nur schwierig zu handhaben, sehr einfach und wenig originell war, genügte sie unseren eigenen Ansprüchen nicht und wir entschieden uns die Steuerung über das Auslesen der Daten des Orientierungssensors zu lösen. Während das Smartphone horizontal gehalten wird, werden mit einer Abtastrate von 20Hz der Kipp- und Neigungswinkel ermittelt, um daraus Lenkeinschlag und Geschwindigkeitsstufe zu bestimmen. Die Übersetzung von Neigungswinkel zu Lenkeinschlag ist linear, wohingegen bei der Übersetzung von Kippwinkel zu Geschwindigkeit einem größeren Bereich der Geschwindigkeitswert 0 zugewiesen wird, um das Stoppen des Autos leichter zu gestalten. Zusätzlich existieren ein Notknopf, zum sofortigen Anhalten des Autos, und ein Button zum Stoppen der Übertragung, ohne die Verbindung zu unterbrechen. Nach einigen Optimierungen wurde diese sensorgestütze Lösung als gut genug befunden, sodass die Sliderunterstützung aus der App entfernt wurde. Codeblöcke: Siehe Abbildung B.3 im Anhang.

4.3.3 Modepicker

Der Modepicker dient zum Auswählen der verschiedenen Modi des Autos (Idle, RC, Wallfollow, zwei verschiedene Parkmodi, C2C). Mit einem Listpicker wird der entsprechende Modus gewählt und mit einem Klick auf „Senden“ dessen Listenindex als Payload einer Nachricht auf dem Topic 6 gesendet. Das Auto startet standardmäßig im Idle-Modus, der einen Listener initialisiert, welcher auf einen Input der App wartet. Nach Empfang einer Nachricht wird ein eventuell noch laufender Task gestoppt und der gewünschte neue Task gestartet. Codeblöcke: Siehe Abbildung B.4 im Anhang.

4.3.4 Car2Car

Eine weitere Funktion der App ist das Setzen der C2C-Parameter. So ist es möglich jedem Auto dieselbe Software aufzuspielen, und die Richtungsinformationen (Start/Ziel)

im laufenden Betrieb zu setzen. Dies funktoiniert ähnlich wie der Modepicker. Über zwei Listpicker werden die Richtungen gewählt und mit einem Send-Button auf das Topic 8 gesendet. Analog zum Modepicker fängt ein Listener die Nachricht ab und setzt die Parameter im Auto. Dazu mehr im Abschnitt über Car2Car. Codeblöcke: Siehe Abbildung B.5 im Anhang.

4.3.5 WallFollow

Über einen einfachen Schieberegler kann im Wallfollower-Modus die Geschwindigkeit angepasst werden. Die Implementierung ist ähnlich der des Modepickers; das verwendete Topic ist 4. Codeblöcke: Siehe Abbildung B.6 im Anhang.

4.3.6 Testen

Zum empirischen Testen von verschiedenen Parametern für den PD-Regler wurde die Möglichkeit geschaffen, diese im laufenden Betrieb zu ändern. Das Funktionsprinzip ist bekannt, Topic ist ebenfalls 4.

4.4 Gamepad Remote

Die Gamepad-Steuerung wird hardwareseitig mit einem Windows-Rechner, einem XBox-Gamepad von Microsoft und dem bereits erwähnten Funkstick AMB8465 realisiert. Ursprünglich war die Verwendung der Bluetooth-Schnittstelle eines Notebooks vorgesehen, was nach einigen Problemen bei der Verwendung des Bluetooth-SPP (Serial Port Profile) unter Java jedoch verworfen wurde. Wir entschieden uns stattdessen die Verbindung mithilfe des 868MHz-Funkmoduls herzustellen. Softwareseitig kommt eine kleine Java-Bibliothek von Aplu.ch [5] zum Einsatz, welche den Zugriff auf das Gamepad regelt. Das Programm implementiert Listener für die einzelnen Knöpfe des Gamepads, welche bei einer Änderung ausgelöst werden. Aus den aktuellen Werten errechnet das Programm die jeweils zu sendenden Werte für Beschleunigung und Lenkeinschlag. Die Verbindungsfunktionalität wird durch Teile des GPS-Config Projekts realisiert. Diese errechneten Werte werden an ein Message Objekt übergeben, welches anschließend über die vom Funkstick emulierte serielle Schnittstelle gesendet wird. Das Programm bietet die Möglichkeit den jeweils zu verwendenden COM-Port auszuwählen sowie das Unterbrechen und Neustarten der Übertragung, wobei ein Thread gestartet bzw. gestoppt wird. Falls der Verbindungsaufbau zur seriellen Schnittstelle nicht erfolgreich war oder kein Controller angeschlossen ist, zeigt das Programm eine Fehlermeldung an, ohne zu terminieren.

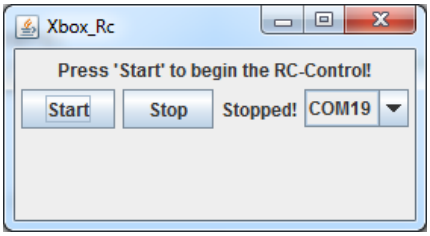


Abbildung 4.2: Screenshot des Java Programms

5 Car2Car Kreuzung

Ziel der Aufgabe war es, zwei oder mehr Autos an einer Kreuzung anhalten zu lassen, worauf sie sich autonom absprechen sollten, wer die Kreuzung zuerst überqueren darf. Wir entschieden uns für einen modularen Aufbau aus drei Komponenten, die miteinander interagieren. Dieser Aufbau spiegelt sich auch in unserer Implementierung durch die Sourcedateien Car2Car, C2C_comm und C2C_log wieder, wobei es sich bei C2C_comm um die Kommunikation zwischen den Autos und mit der App handelt und C2C_log die Logik der Kreuzung implementiert.

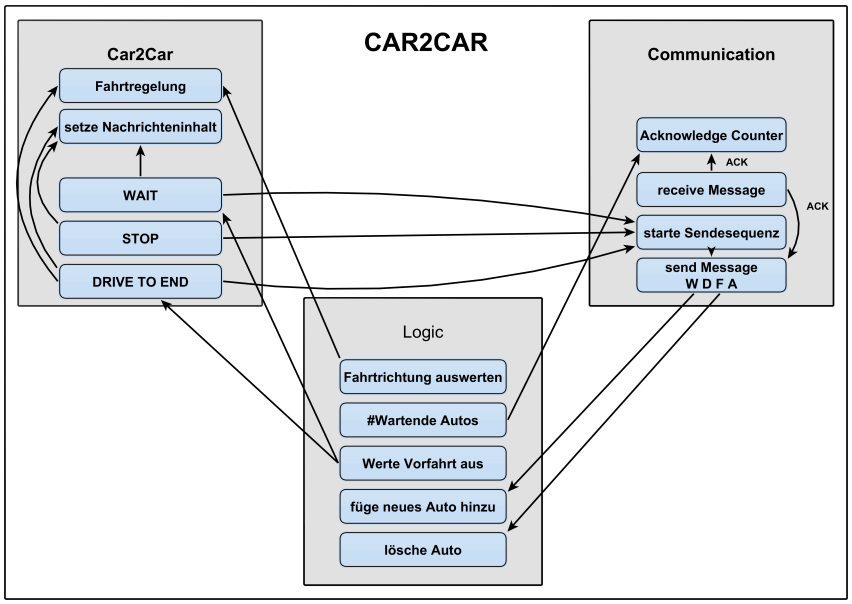


Abbildung 5.1: Modularer Aufbau der Kreuzungssimulation

5.1 Fahrtregelung

Die Fahrtregelung wird im wesentlichen über zwei Zustandsautomaten gesteuert. Ein Automat regelt den Ablauf der eigentlichen Kreuzungssimulation, der andere ist für das Überqueren der Kreuzung zuständig.

5.1.1 Kreuzungssimulation

Die Kreuzungssimulation startet im Regelfall damit, dass das Auto auf eine Eingabe über die App wartet, mit der Start- und Zielpunkt festgelegt werden. Ist diese Eingabe erfolgt, beginnt das Auto, gesteuert über den PD-Regler, auf die Kreuzung zuzufahren bis eine Linie auf dem Boden erkannt wird. Danach startet es eine Kommunikation mit allen anderen erreichbaren Autos und wartet 6 Sekunden auf Antworten. Die Kreuzungssituation wird ausgewertet und das Auto mit Vorfahrt kann beginnen die Kreuzung zu überqueren. Wurde das Ende der Kreuzung erkannt, sendet das Auto erneut einen Broadcast, um den anderen Autos mitzuteilen, dass der Nächste fahren kann. Zu Debuggingzwecken hielten wir uns die Möglichkeit offen, für jedes Auto auch manuell im Quellcode

die Fahrtrichtungen setzen zu können. Im Normalfall erwies sich allerdings die Nutzung der App als praktischer, da jedes Auto mit dem gleichen Code geflasht werden konnte. Innerhalb dieses Moduls wird auch bereits die Vorarbeit für die Kommunikation geleistet, da je nach Kreuzungszustand (Warten, Kreuzung überqueren, Kreuzung verlassen) unterschiedliche Nachrichtenarten für das Kommunikationsmodul gesetzt werden und der Zustandsautomat für das Versenden von Paketen neu angestoßen wird.

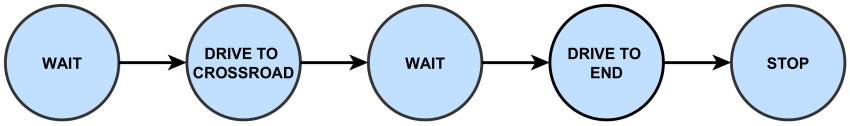


Abbildung 5.2: Wesentlicher Ablauf der Kreuzungssimulation

5.1.2 Linienerkennung

Die Linienerkennung, so wie sie in der API vorgeben war, hat mit den meisten Fahrzeugen nicht zuverlässig funktioniert, weswegen wir sie neu schreiben mussten. Erschwert wurde das Ganze durch die großen Unterschiede der Sensoren der Fahrzeuge. Wir verwendeten die Funktion LS_getSensorValue(i) um die Werte der einzelnen Funktionen auszulesen. Nach ersten Tests gingen wir davon aus, dass die Sensoren im Falle einer dunklen Linie Werte nahe 0 messen. Die auf dieser Annahme aufbauende Methode hat allerdings nur bei einem Auto zufriedenstellend funktioniert. Wir stellten fest, dass manche Sensoren invertiert sind. Das heißt, dass sie bei hellen Böden sehr kleine Werte zurückgeben und bei dunklen hohe. In der finalen Version speichern wir deswegen für jeden Sensor einzeln den Betrag des Messwertes in einem Array. Von diesen gespeicherten Werten werden bei der nächsten Messung die neuen Beträge subtrahiert. Die Summe davon wird mit der Summe der letzten Messung verglichen.

5.1.3 Überqueren der Kreuzung

Hier war es uns wichtig, eine komplette Kreuzungssimulation fertig zu stellen, in der sämtliche Abläufe bis zum endgültigen Verlassen der Kreuzung im Unfang enthalten und alle Fahrtmöglichkeiten abgedeckt sind. Da wir, sobald wir die Kreuzung betreten, keine Orientierung mehr durch eine seitliche Wand herstellen können, wird für diese Zeit der PD-Regler abgeschaltet. Bei einigen der Autos kam es hier zu Problemen, da die Lenkung trotz Servo-Offset so ungenau war, dass die Autos seitlich immer weiter abdrifteten, sodass wir automatisch wieder den PD-Regler aktivieren, wenn die Wand zu nahe kommt. Da die uns zur Verfügung stehende Kreuzung nicht symmetrisch war, mussten jedoch einige Sonderfälle beachtet werden. Je nach Startrichtung des Autos muss beispielsweise bei Linkskurven zunächst eine unterschiedlich lange Strecke geradeaus gefahren werden, um nicht in den Gegenverkehr zu gelangen. Zusätzlich muss an zwei Kreuzungsendpunkten nach der Kurve noch eine feste Strecke geradeaus gefahren werden, um den Kontakt zur rechten Wand wieder herzustellen. Unsere Implementierung beachtet daher, aus welcher Richtung gestartet wird und wo das Ziel liegt, um daraufhin bei Linkskurven eine

feste Route abfahren zu können, mit der selbständig wieder ein Kontakt zur Wand hergestellt werden kann. Dieser Wandkontakt ist wichtig, da er bei unserer Simulation für den Moment steht, in dem die Kreuzung als verlassen gilt und dieses auch den anderen an der Kreuzung wartenden Autos per Broadcast mitgeteilt wird. Außerdem ermöglicht es uns auch, ein Linksabbiegen hintereinander realisieren zu können, ohne dass die Autos miteinander kollidieren.

5.2 Kommunikation

Da wir zwei verschiedene Arten von Kommunikation für die Kreuzung benötigen, legten wir zwei neue Topics für Funknachrichten an, die bei der Initialisierung des Kommunikationsmoduls subscribed werden.

5.2.1 Kommunikation mit der App

Empfängt das Auto eine Nachricht des entsprechenden Topics, fängt der Listener die Nachricht ab. Daraufhin wird das Logikmodul mit den empfangenen Werten für Start- und Zielrichtung initialisiert und dem Simulationsmodul über die Variable inputIsSet mitgeteilt, dass der Wagen eine korrekte Initialisierung erhalten hat und die Kreuzungssimulation starten kann. Das Auto wird sich nun auf die Kreuzung zubewegen.

5.2.2 Kommunikation der Kreuzungsteilnehmer

Da wir während der Bearbeitungsphase feststellen mussten, dass die Funkmodule ihren Dienst nur sehr unzuverlässig verrichteten und Pakete nur selten ankamen, entschlossen wir uns, ein einfaches Protokoll mit Bestätigungsnachrichten zu implementieren. Generell benutzen wir vier verschiedene Stati in den Nachrichteninhalten:

- w = Wir sind an der Kreuzung angekommen
- d = Wir beginnen die Kreuzung zu überqueren
- f = Wir haben das Kreuzungsende erreicht
- a = Acknowledge (wird nur an Absender eines Broadcasts geschickt)

Der restliche Inhalt einer Nachricht besteht bei den ersten drei Arten zusätzlich noch aus Start- und Zielrichtung. Der Beginn einer Sendesequenz wird dabei vom Simulationsmodul ausgelöst, wenn entsprechend die Zustände WAIT, DRIVE_TO_END oder STOP erreicht werden. Eine Sequenz selbst wird wiederum über einen Zustandsautomaten dargestellt, der den Ablauf des Protokolls visualisiert. Zunächst wird der jeweilige Broadcast gesendet und der Counter für die Anzahl empfangener Acknowledgepakete zurückgesetzt. Werden innerhalb einer Sekunde nicht so viele Antworten erhalten, wie wir wissen, dass andere Teilnehmer an der Kreuzung stehen, muss erneut ein Broadcast gesendet werden. Haben wir genug Antworten bekommen, können wir die Sendesequenz beenden. Dieses Vorgehen hat natürlich eine Schwachstelle für das erste Auto das die Kreuzung erreicht. Dieses Auto muss im Zweifel nicht neu senden, da niemand anderes bereits aktiv an der

Kommunikation teilgenommen hat, obwohl andere Autos die Nachricht möglicherweise nicht erhalten haben. Jedoch hätte eine eine weitere Verbesserung des Protokolls deutlich mehr Overhead bedeutet, sodass wir unsere Lösung so beließen.

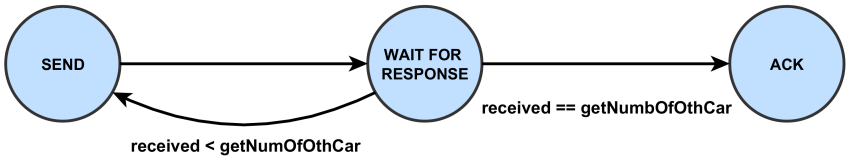


Abbildung 5.3: Protokollablauf

5.3 Logik

Unser Ziel war es mehrere Autos koordiniert über eine Kreuzung zu manövrieren. Wir haben uns dazu entschlossen, ein Verfahren zu entwickeln, das die gängigen Rechts vor Links Verkehrsregeln verwendet und dabei alle möglichen Varianten, außer Wenden abdeckt. Das bedeutet Fahrzeuge können sowohl nach rechts oder links abbiegen als auch geradeaus fahren. Es können auch mehrere Fahrzeuge gleichzeitig über die Kreuzung fahren, wenn keine Kollision droht. Zum Beispiel können zwei Fahrzeuge, die sich entgegen kommen gleichzeitig geradeaus fahren. Das funktioniert theoretisch mit bis zu 4 Autos, wir haben es allerdings nur mit 3 getestet.

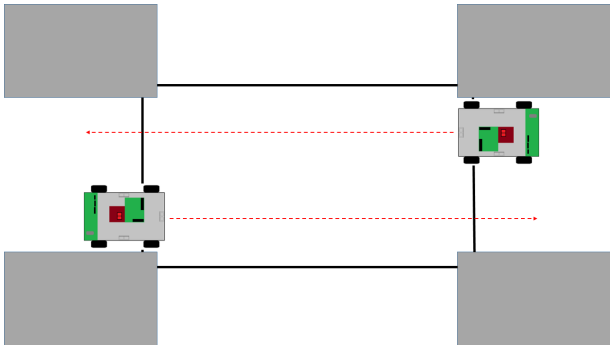


Abbildung 5.4: Hier können beide Fahrzeuge gleichzeitig fahren, was auch von der Software ermöglicht wird.

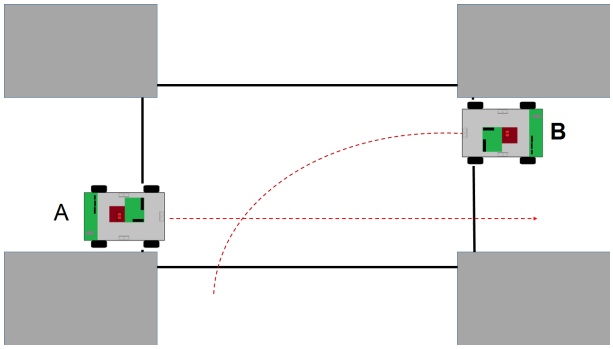


Abbildung 5.5: Auto A hat hier Vorfahrt.

Nach etwas Überlegung kamen wir darauf, dass es am sinnvollsten ist, wenn wir uns an der Realität orientieren. Das bedeutet eine dezentrale Auswertung der Situation in jedem Fahrzeug, dadurch wird die Komplexität der eigentlichen Auswertung reduziert. Dies haben wir für mehrere Fahrzeuge, die jeweils nach rechts oder links abbiegen oder gerade aus fahren dürfen, realisiert. Im folgenden Abschnitt wird im Besonderen auf die Implementierung der zur Umsetzung nötigen Datenstrukturen und Funktionen eingegangen.

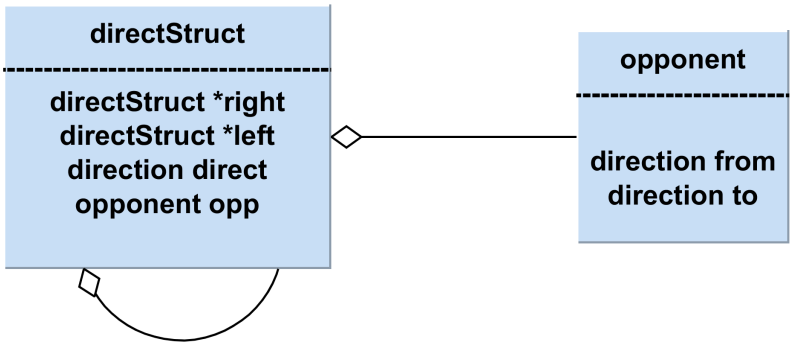
5.3.1 Funktion

Wie schon erwähnt sendet jedes Fahrzeug beim Überfahren der Linie ein Signal, aus dem hervorgeht aus welcher Richtung es auf die Kreuzung zufährt und in welche Richtung es sie verlassen will. Die anderen Fahrzeuge merken sich dies und speichern die Information in einer Datenstruktur ab. Nach einer kurzen Zeitfrist startet bei allen Fahrzeugen die dezentrale Auswertung der Situation. Die Fahrzeuge arbeiten auf einer im Idealfall identischen Datenstruktur, haben aber jeweils eine andere Perspektive. Dies wird im Abschnitt Implementierung weiter erläutert. Da die Prozedur auf jedem Fahrzeug ausgeführt wird, müssen relativ wenige Fälle beachtet werden, um die Vorfahrt zu bestimmen.

- Ein Fahrzeug, das rechts abbiegt, hat automatisch Vorfahrt
- Wenn das Fahrzeug nicht nach rechts abbiegt und von rechts ein anderes Auto kommt, wird keine Vorfahrt genommen.
- Wenn das Fahrzeug nach links abbiegt und ein anderes Fahrzeug aus der Gegenrichtung kommt, das nicht auch aus seiner Perspektive links abbiegt, wird Vorfahrt genommen.
- bei allen anderen Fällen wird Vorfahrt genommen.

5.3.2 Implementierung

Der dafür verwendete Programmcode findet sich in der Datei "C2C_log.c". Zur Repräsentation der Situation verwenden wir eine verzeigerte Datenstruktur. Wir verwenden vier Instanzen eines Struktes, die jeweils eine Einfahrt der Kreuzung repräsentieren. Das Strukt enthält neben der ihm zugeordneten Himmelsrichtung Zeiger auf die benachbarten Instanzen. Zum Beispiel Norden enthält Zeiger auf Westen und Osten, Westen auf Norden und Süden und so weiter. Die Himmelsrichtungen selber werden mit einer Enumeration dargestellt. Ein weiterer Struct Type modelliert die Fahrzeuge. Jedes Fahrzeug enthält die Himmelsrichtung aus der es kommt, und die des Ziels. Die Einfahrten können jeweils eine Referenz auf ein Fahrzeug enthalten. Dazu kommt noch eine Zeigervariable, welche auf die eigene Einfahrt verweist, sie wird für die Auswertungsfunktionen benötigt. Durch diese Datenstrukturen wird die Situation genau genug modelliert, um zu entscheiden, ob gefahren werden kann.



Direction ist eine Enumeration bestehend aus den Himmelsrichtungen

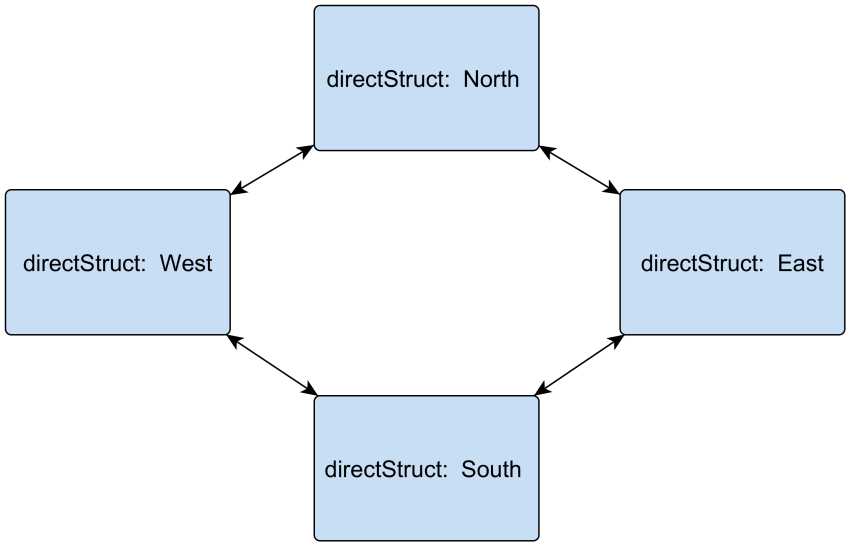


Abbildung 5.6: Verzeigerte Struktur

Es gibt mehrere wichtige Funktionen, die wir hierfür benötigt haben:

- void C2C_log_newOpponent(char cFrom, char cTo) Fügt einen neuen Verkehrsteilnehmer in die Datenstruktur ein.
- void C2C_log_deleteOpponent(char cFrom) Entfernt das Fahrzeug an der übergebenen Position aus dem System.
- int C2C_log_checkForCollision(void) Übeprüft, ob das Fahrzeug Vorfahrt hat und gibt gegebenenfalls 1 zurück, ansonsten 0.

NewOpponent erzeugt eine neue Instanz des Struktes, mit dem die Fahrzeuge repräsentiert werden. Dieses wird mit den übergebenen Werten initialisiert. Eine Referenz auf diese Instanz wird in der Datenstruktur, die dem übergebenen "cFrom“ entspricht abgespeichert.

CheckForCollision überprüft anhand der oben beschriebenen Regeln, ob das Fahrzeug Vorfahrt hat. Die Richtungen werden relativ zu der eigenen Perspektive bestimmt. Das heißt,

wenn zum Beispiel Perspektive auf die nördliche Einfahrt zeigt, verweist perspective->right auf die westliche Einfahrt. Dazu kommen noch Funktionen mit denen anhand der in From und To gespeicherten Richtungen, bestimmt wird, ob eine Links- bzw. Rechtskurve oder geradeaus gefahren werden soll.

Zu Beginn dieses Projektseminars haben wir eine grobe zeitliche Planung des gesamten Bearbeitungszeitraums erstellt. Als Grundlage dienten lediglich die Informationen über die Aufgabenstellung, die durch den Veranstalter im Eröffnungsmeeting bekannt gegeben wurden. In Tabelle 6.1 sieht man in der linken Spalte den damals geschätzten Zeitbedarf.

Ziel war es, in den ersten zwei Monaten der Bearbeitungsdauer die ersten drei Aufgaben, also „Geradeaus Fahren und Kurve Nehmen“, „Einparken“ und „RC-Modus“, zu implementieren. Dies klappte auch weitgehend, jedoch haben wir auch noch danach erhebliche Mengen Zeit in die Optimierung dieser Lösung gesteckt. Für die darauf folgenden zwei Monate, also die ersten zwei Monate des neuen Jahres, hatten wir nur noch die letzte Pflichtaufgabe, die Implementierung des Car-2-Car Szenarios, sowie einige eigene Erweiterungen geplant, welche nur teilweise umgesetzt werden konnten.

Zum Zeitpunkt der ursprünglichen Zeitplanung sind wir davon ausgegangen, dass wir im Bearbeitungszeitraum deutlich mehr als die gegebenen Aufgaben bearbeiten könnten. In der Tabelle sieht man, dass der tatsächliche Zeitbedarf in der Summe ca. doppelt so hoch ist wie der geplante Zeitbedarf. Beim genaueren betrachten merkt man, dass die Abweichungen nicht gleichmäßig verteilt sind. Ursache hierfür sind unter anderem einige prinzipbedingte Probleme bei der Implementierung der Abstandsregelung gewesen, die uns zuvor nicht bekannt waren. Bei der Implementierung des Car-2-Car Szenarios bestand die Herausforderung darin, die extrem störanfälligen Funkmodule zuverlässig zu betreiben, was hierbei sehr viel Zeit in Anspruch nahm. Andere Aufgaben, wie beispielsweise das Einparken, stellten sich sogar als einfacher als geplant heraus, so dass bereits nach sehr kurzer Zeit eine funktionierende Lösung zur Verfügung stand. Diese wurde zwar noch weiter optimiert, was allerdings nicht zwingend nötig gewesen wäre.

Tätigkeit	geplanter Zeitbedarf	tatsächlicher Zeitbedarf
PD-Regler	20	150
Kurven	20	150
RC-Modus	70	60
Einparken	70	30
orth. Einparken	50	20
Car-2-Car	80	150
Optimierungen	0	300
Ausarbeitung	60	60
Summe	400	810

Tabelle 6.1: Zeitbedarf des Projektseminars nach Kategorien in Menschstunden

7 Fazit

Die durch uns erarbeitete Lösung entspricht allen Anforderungen der Aufgabenstellung, außerdem haben wir einige zusätzliche Funktionalitäten implementiert. Allerdings hatten wir zu Beginn des Projekts einige Ideen entwickelt, die wir aufgrund von Zeitmangel und eingeschränkter Hardware nicht umsetzen konnten. Alles in allem sind wir mit der von uns erreichten Lösung jedoch sehr zufrieden.

Durch den modularisierten Aufbau unserer Lösung, zu der wir uns während der Bearbeitungszeit hin entwickelten, war es uns möglich, weitere Erweiterungen ohne große Anpassungen zu integrieren. Diese Erkenntnis hat uns nicht nur während des Projektes geholfen effizienter und paralleler zu arbeiten, sondern wird uns sicherlich auch in zukünftigen Projekten eine Hilfe sein.

Außerdem haben wir während des Projekts einen GIT-Server zur Versionsverwaltung unseres Codes verwendet, wodurch alle Gruppenmitglieder theoretisch zeitgleich am Code arbeiten konnten. Die Praxis zeigte jedoch, dass die genutzten Tools unter Umständen eine manuelle Unterstützung benötigen. Hierbei und bei einigen anderen Dingen haben wir die Erfahrung gemacht, dass der Zeitbedarf für organisatorische Tätigkeiten nicht unterschätzt werden darf.

7.1 Mögliche Erweiterungen für zukünftige Projektseminare

Während der Bearbeitung dieses Projektseminars sind einige Ideen für zukünftige Projektseminare entstanden, die nachfolgend genauer erläutert werden. Diese Ideen sind teilweise aufgrund von Problemen, beispielsweise schlechter Sensorik, andererseits aber auch ausschließich durch kreative Ideen entstanden.

7.1.1 Laser Entfernungsmessung

Die momentan in den Autos verbauten Ultraschall Entfernungssensoren haben einen Öffnungswinkel von ca. 30°, was bedeutet, dass als Entfernung die kürzeste Distanz zu einem Objekt, dass sich innerhalb dieser 30°befindet, zuückgegeben wird. Das führt dazu, dass zum Beispiel bei der parallelen Fahrt an der Wand die Distanz zu einem Türrahmen gemessen werden kann, was nicht der echten Distanz nach vorne entspricht.

Daher kam uns die Idee, die Entfernungsmessung durch einen Laser zu ergänzen. Laser Entfernungsmesssysteme werden beispielsweise häufig in der Baubranche zur Vermessung verwendet. Daher sind sie auch relativ kostengünstig, wodurch sie eine interessante Alternative darstellen. Durch die Montage dieses Entfernungsmessers auf einem Servo würde sich dieser Sensor sogar zum Abtasten der Umgebung eignen, wodurch z.B. eine Karte errechnet werden könnte.

7.1.2 Kommunikation und Rechenleistung

Die zwei zur Kommunikation zur Verfügung gestellten Technologien haben sich beide als nicht sehr leistungsstark aber dafür als störanfällig herausgestellt. Im Rahmen einer anderen Arbeit haben wir mit dem MQTT Protokoll [6] sehr gute Erfahrungen gemacht.

Eine Idee wäre, alle Autos mit neuer Hardware, beispielsweise BeagleBone oder RaspberryPi, auszustatten und eine Verbindung zu einem WLAN Netz herzustellen. Über diese WLAN-Verbindung wäre nicht nur die Kommunikation der Autos untereinander möglich, sondern auch die Kommunikation mit externen Sensorknoten. Außerdem wäre eine Programmierung und Überwachung der Autos hierüber möglich, beispielsweise durch eine SSH-Verbindung.

7.1.3 Navigation Szenario

Mit entsprechender Sensorik ist es möglich, ein gewisses Gebiet zu kartographieren, zum Beispiel durch eine Laserabtastung wie oben beschrieben. Eine mögliche Aufgabe wäre, dass aufgrund dieser Karte das Auto selbstständig den kürzesten Weg zum Ziel berechnet und fährt. Ungeplante Hindernisse, wie beispielsweise verschlossene Türen, werden erkannt und eine neue Route wird berechnet. Kommunikation unter den Autos über aktuelle Verkehrsinformationen sind ebenfalls denkbar.

Literatur

[1] Lehrinhalte der Vorlesung: Systemdynamik und Regelungstechnik 1, Prof. Konigorski, TU-Darmstadt

[2] http://amber-wireless.de/61-0-AMB8425.html

[3] http://amber-wireless.de/162-0-AMB8465.html

[4] http://appinventor.mit.edu/explore/

[5] http://www.aplu.ch/home/apluhomex.jsp?site=36

[6] http://mqtt.org

[7] http://www.freertos.org

A Anhang: Abbildungen RC

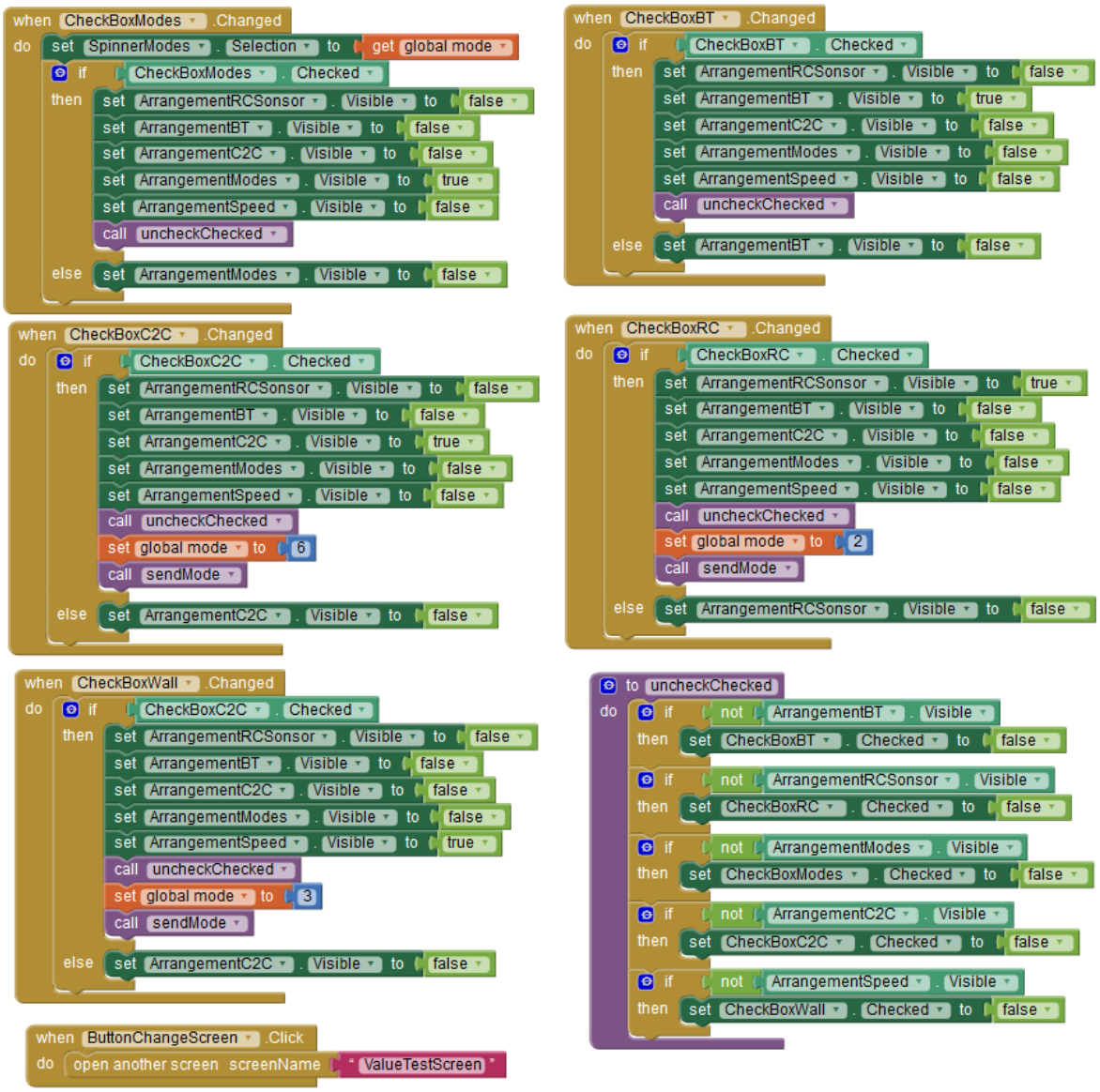


Abbildung A.1: Codeblöcke zum Ein-/Ausblenden der einzelnen UI-Elemente

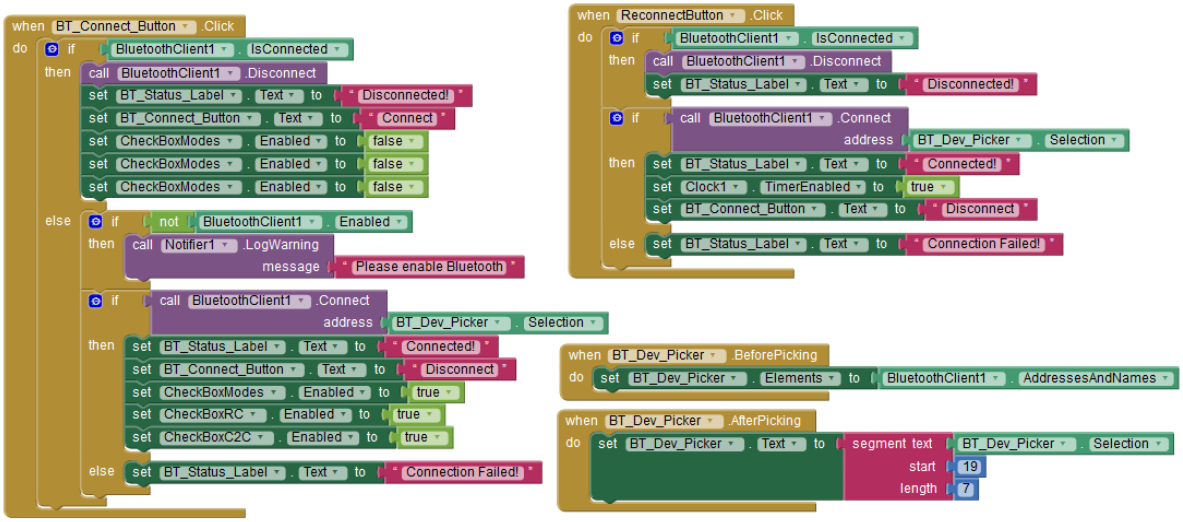


Abbildung A.2: Codeblöcke zur Bluetooth-Verbindungsverwaltung

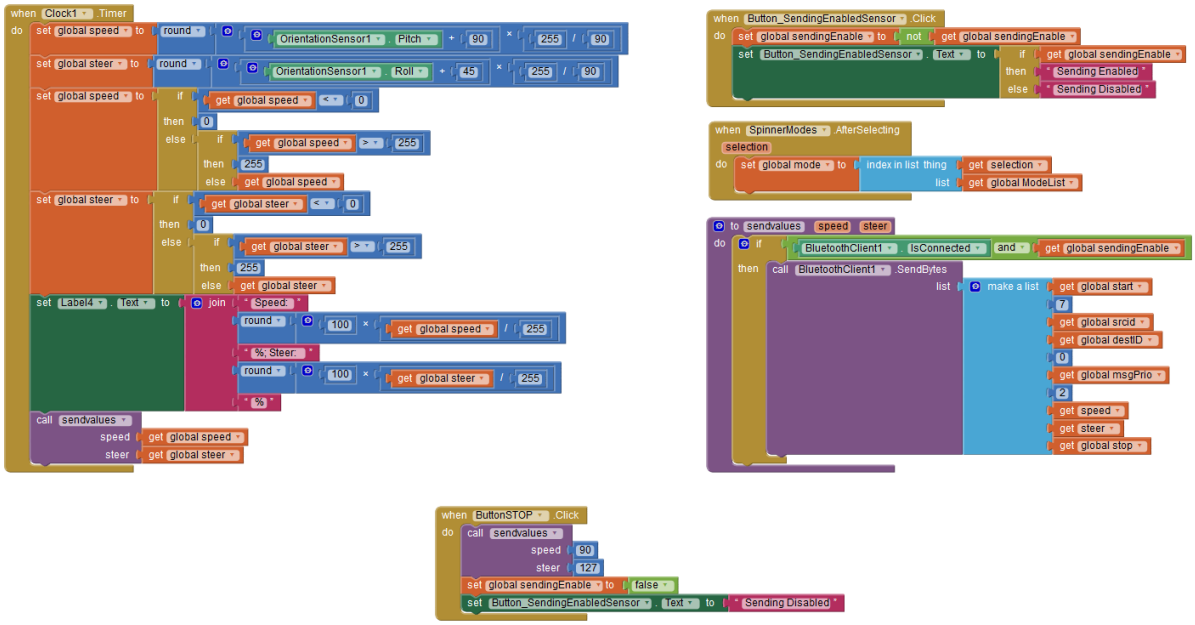


Abbildung A.3: Codeblöcke zur Fernsteuerung über Sensorwerte

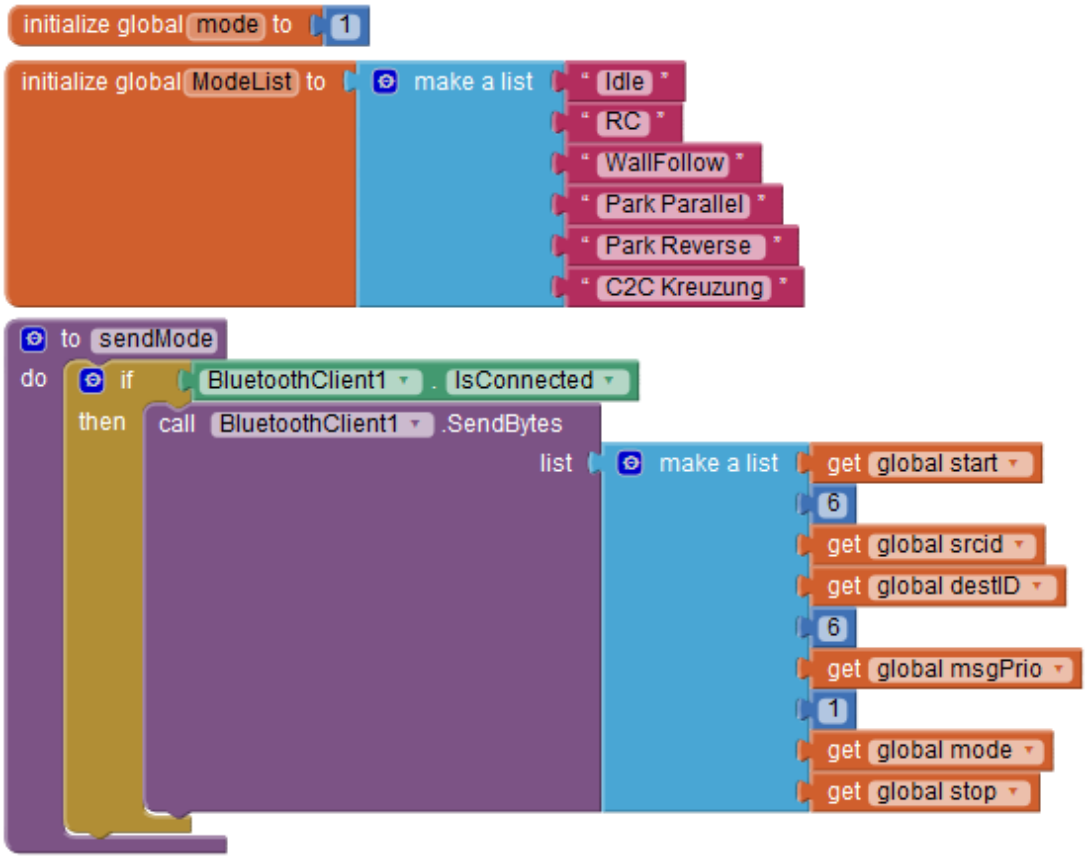


Abbildung A.4: Codeblöcke zum Modepicker

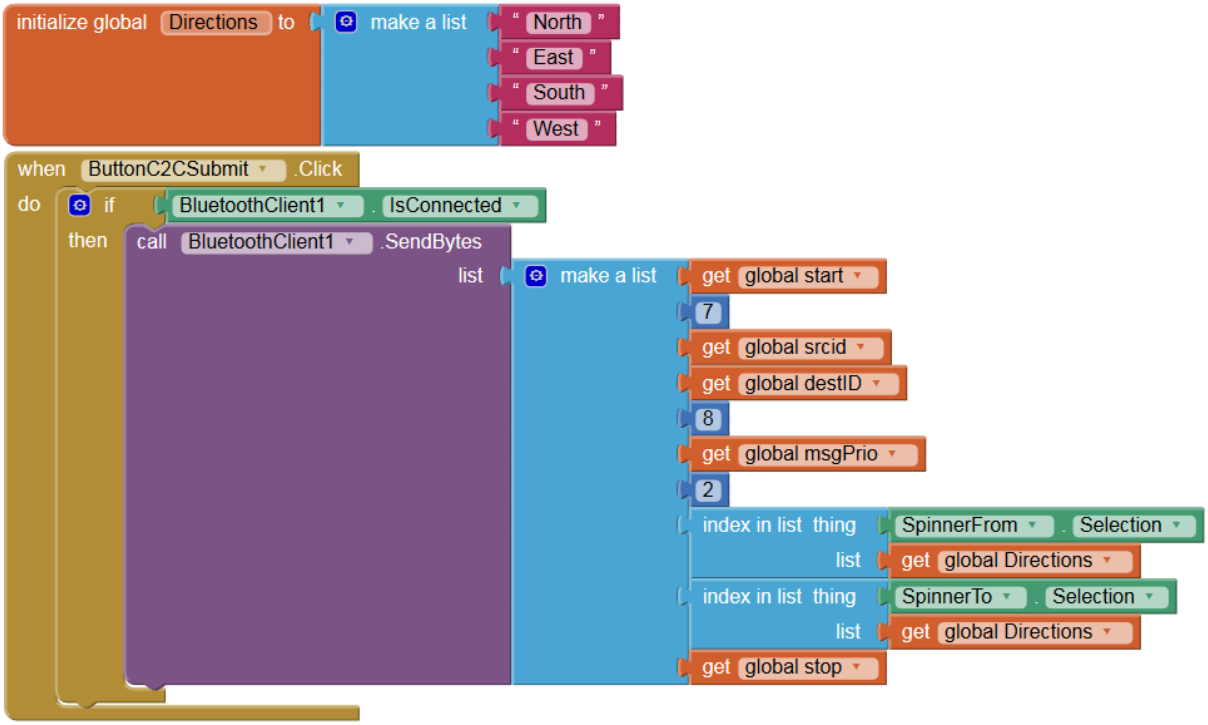


Abbildung A.5: Codeblöcke zum Car2Car Setter

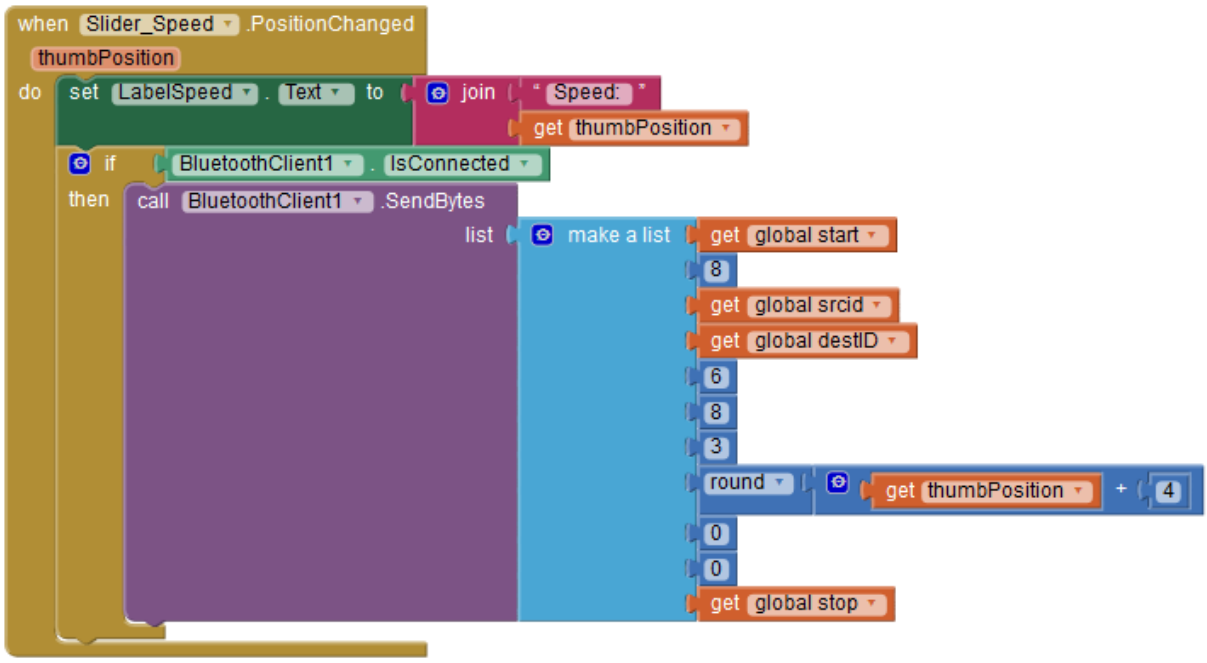


Abbildung A.6: Codeblöcke zum Wallfollower