Technischen Universität Kaiserslautern
Fachbereich Informatik
AG Software Engineering
Prof. Dr. Dieter Rombach

Fraunhofer Institut für Experimentelles Software Engineering
Department for Component Engineering
Fraunhofer-Platz 1, 67663 Kaiserslautern

Projektarbeit

Entwicklung eines Antikollisionssystems für Kraftfahrzeuge mit MARMOT

Sommersemester 2006

Bearbeiter:
Marcel Zimmer *(marcel@zimmerit.de)*

Betreuer:
Dr. rer. nat. Christian Bunse *(christian.bunse@iese.fraunhofer.de)*
Prof. Dr. Dieter Rombach *(rombach@informatik.uni-kl.de)*

Erklärung

Hiermit erkläre ich, Marcel Zimmer, dass ich die vorliegende Projektarbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Kaiserslautern, den 29.09.2006

Zusammenfassung

Die am Fraunhofer-Institut für Experimentelles Software Engineering entwickelte MARMOT-Methode beschreibt einen Ansatz für die komponentenbasierte Entwicklung eingebetteter Systeme. Sie baut auf der ebenfalls am IESE entwickelten KobrA-Methode auf und erweitert diese um spezielle Anforderungen für eingebettete Systeme. Die Idee dahinter ist es, einzelne Komponenten zu modellieren, implementieren und zu testen um später auf vorhandene qualitätsgesicherte Komponenten zurückgreifen zu können, und zu Applikationen zu komponieren ohne diese immer wieder neu entwickeln und testen zu müssen.

Im Rahmen dieser Projektarbeit sollte mit Hilfe der MARMOT-Methode ein Antikollisionssystem für ein Modellauto entwickelt werden. Nach Auswahl der hierfür geeigneten Hardware wurde zunächst ein Grundkonzept für die Sensorik entwickelt. Die vom verwendeten RADAR-Sensor gelieferten Signale müssen für die weitere Verwendung durch einen Mikrocontroller aufbereitet werden. Vor der eigentlichen Systemmodellierung musste deshalb zu diesem Zweck eine Sensorplatine entwickelt werden. Anschließend folgte die Modellierung des Antikollisionssystems in UML 2.0 und die Implementierung in C. Zum Abschluss wurde das Zusammenspiel der Hard- und Software getestet.

Inhaltsverzeichnis

1 Einleitung

Im Jahr 2005 gab es in Deutschland über 2,2 Millionen Verkehrsunfälle. Dabei verunglückten etwa 440.000 Menschen, 5300 tödlich [1]. Ein Großteil dieser Unfälle entstand durch Kollision mit anderen Fahrzeugen. Viele davon hätten sehr wahrscheinlich durch ein integriertes semiautomatisches Antikollisionssystem verhindert werden können, denn neben Unaufmerksamkeit des Fahrers gehören vor allen Dingen auch schlechte Sichtbedingungen zu den häufigsten Unfallursachen. Die Automobilindustrie entwickelt deshalb sogenannte Fahrassistenzsysteme, die den Fahrer durch audiovisuelle Hinweise auf mögliche Probleme hinweisen. Einfache Beispiele sind Einparkhilfen, die durch schnelles bzw. langsames piepsen beim Rückwärtsfahren den Abstand zu möglichen Hindernissen zeigen. Komplexere Systeme bieten in einem sogenannten "Head-Up-Display" kameragestützte Nachtsichtfunktionen. Im Sinne dieser Fahrassistenzsysteme würde ein Antikollisionssystem in der sogenannten Pre-Crash-Phase, also dem Zeitraum vor einer möglicherweise auftretenden Kollision in der der Fahrer noch genug Zeit zum Reagieren hat, arbeiten. Eine Sensorik scannt die Umgebung ununterbrochen nach sich annähernden Hindernissen ab. Unterschreitet ein solches einen festgelegten Mindestabstand, wird der Benutzer durch ein akkustisches bzw. ein visuelles Signal gewarnt um entsprechend rechtzeitig reagieren zu können. Als weitere Funktion könnte das Auto mit Hilfe des ESP (Elektronisches Stabilitätsprogramm) gezielt gebremst werden. Eine vollautomatische Lösung, die ohne Eingriffe des Fahrers das Auto im Gefahrenfall bremst oder ein Ausweichmanöver einleitet ist in Deutschland bei der momentanen Gesetzeslage nicht erlaubt. Die Automobilhersteller schützen sich durch die Semiautomatik auch vor eventuellen Schadensersatzansprüchen im Falle einer Fehlfunktion der Automatik und dabei auftretenden Unfällen.
KFZ-Hersteller und Zulieferer haben solche und ähnliche Systeme bereits im Test. Die "Adaptive Cruise Control" (ACC) von Bosch[2] hält beispielsweise einen Mindestabstand zu vorausfahrenden Fahrzeugen durch Motor- und Bremseingriffe. Das ACC ist bei Oberklassefahrzeugen von BMW bereits im Einsatz. Herzstück des ACC ist die aus einem Radarsensor und einem Steuergerät bestehende "Sensor-Regler-Einheit". Sie überwacht die Umgebung und greift bei Bedarf ins Fahrgeschehen ein.

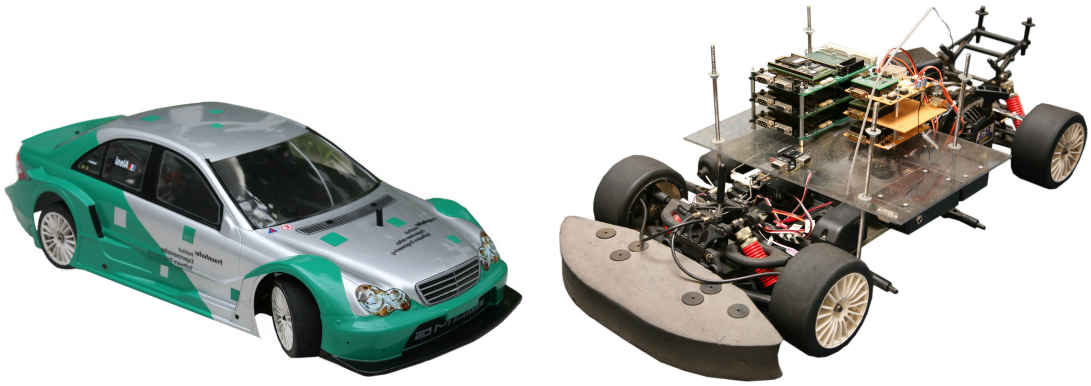
Ziel dieser Projektarbeit war es ein vollautomatisches Antikollisionssystem (ACS) für ein Modellauto zu entwickeln. Das einem Formel1-Wagen sehr ähnliche (bezogen auf die Fahrwerte) Fahrzeug ist ferngesteuert, erreicht mit Hilfe eines Verbrennungsmotors Geschwindigkeiten von bis zu 80km/h und kann durch Scheibenbremsen gezielt gebremst werden. Im Rahmen weiterer Arbeiten wurde bereits eine mikrocontrollerbasierte Steuerung entwickelt. Diese übersetzt Signale einer Fernbedienung in Motor- und Bremssteuersignale und greift, z.B. in Form einer Antischlupfregelung (ASR), fahrerun-

terstützend ein. Frank Böhr beschreibt die Fahrzeugsteuerung und die damit verbundene Inter-Modul-Kommunikation über den CAN-Bus in [13] genauer. Das ACS klinkt sich als weiteres Modul in diesen Bus ein und meldet Kollisionswarnungen an die zentrale Fahrsteuerung. Um die Reaktion auf ein vom ACS gemeldetes Hindernis kümmert sich die Fahrsteuerung.

Die Modellierung der entwickelten Komponenten erfolgte nach der am Fraunhofer IESE entwickelten MARMOT-Methode. Sie beschreibt die komponentenbasierte Entwicklung eingebetteter Systeme auf Basis der ebenfalls am IESE konzipierten KobrA-Methode. KobrA beschreibt wie einzelne Anwendungsbausteine als in sich abgeschlossene Komponenten entwickelt und für weitere Anwendungen wiederverwendet werden können. Hierdurch wird wertvolle Entwicklungs- und Testzeit eingespart. Dank klar vordefinierter Schnittstellen, können die Einzelkomponenten der Zielanwendung durch neue Komponenten ohne großen Aufwand ausgetauscht werden. Die Modellierung der Komponenten erfolgt wie in [12] beschrieben mit UML. Um die speziellen Anforderungen eingebetteter Systeme bestmöglich zu spezifizieren, wurden die Komponenten dieser Arbeit in UML 2.0 [3] modelliert.

2 Auswahl der Komponenten

2.1 Zielplattform



(a) Das ConceptCar

(b) Das Innere des ConceptCars

Abbildung 2.1: Die Zielplattform

Abbildung 2.1(a) zeigt ein Bild der Zielplattform ("ConceptCar"). Es ist ein ferngesteuertes Modellauto im Maßstab 1:5 mit einem 3,2 PS Verbrennungsmotor und erreicht eine Spitzengeschwindigkeit von 80 km/h. In Abbildung 2.1(b) sieht man das Innere des Fahrzeugs. Die darauf befindlichen Mikrocontroller-Module steuern und regeln das Fahrverhalten. Zur Inter-Modul-Kommunikation sind diese über einen CAN-Bus (siehe Glossar) miteinander vernetzt. Ein Modul liest z.B. Werte vom Empfangsmodul der Fernsteuerung, wertet diese aus und sendet sie an die Fahrsteuerung. Die anderen Module bieten Assistenzsysteme wie eine Antischlupfregelung (ASR) oder stellen eine Debugging-Schnittstelle per WLAN bereit. Als weiteres Assistenzmodul tastet das Antikollisionssystem die Umwelt mit Hilfe einer Sensorik nach sich auf das Fahrzeug zubewegenden Hindernissen ab und meldet diese über den CAN-Bus der zentralen Fahrsteuerung.

2.2 Sensorik

Bei der Suche nach der für diese Arbeit am besten geeigneten Sensorik fiel die Wahl auf einen RADAR-Sensor von Infineon. Günstige Sensoren die mit Infrarotlicht oder

Abbildung 2.2: RADAR-Sensor KMY-24

Ultraschall arbeiten schieden aufgrund der zu niedrigen Reichweite aus. Da moderne mit Laser arbeitende LIDAR-Sensoren auf dem freien Markt nur schwer erhältlich sind, konnten diese nicht getestet werden. In der Automobilindustrie werden auf einer Trägerfrequenz von 77 GHz arbeitende RADAR-Sensoren mit einer Reichweite von bis zu 150 Metern [2], verbaut. Aufgrund des zu hohen Preises für Einzelexemplare (weit über 1000 €) schied ein solcher Sensor aus. Der einzige auf dem Markt befindliche RADAR-Sensor der ins Budget passte, ist der mit ca. 50 € sehr günstige auf einer Trägerfrequenz von 2,45 GHz arbeitende RADAR-Sensor KMY-24 (siehe Abbildung 2.2) von Infineon. Laut Datenblatt hat dieser eine Reichweite von bis zu 8 Metern. Für ein funktionierendes Antikollisionssystem, das auch noch bei maximaler Geschwindigkeit fehlerfrei Hindernisse erkennen und rechtzeitig melden muss, ist dies zu wenig. Um das System dennoch zu testen, wurde das Auto deshalb auf eine Maximalgeschwindigkeit von ungefähr 15 km/h beschränkt. Diese Arbeit beschreibt demnach eine Machbarkeitsstudie.

2.3 Mikrocontroller

Die bereits vorhandenen Module bauen auf einem AT90CAN128-Mikrocontroller von Atmel® auf. Der große Vorteil dieses Controllers ist der bereits integrierte CAN-Bus-Controller und die dafür vorhandene Referenz-Software-Implementierung, die Atmel® CAN-Lib [4]. Dies und der ebenfalls integrierte 10 Bit AD-Wandler waren die Gründe, warum ich mich für diesen Controller entschieden habe. Weiterhin besitzt er 32 konfigurierbare Digital-Ein-/Ausgänge, die ich für die Sensorkalibrierung nutzen kann. Für den AT90CAN128 existiert mit dem avr-gcc ein Open Source C-Compiler und mit dem avrdude eine freie Software zum Programmieren des Controller-Flashspeichers. Im Gegensatz zu PIC-Mikrocontrollern von Microchip für die nur kommerzielle Compiler existieren, konnte ich die Software für den AVR unter Linux mit freier Software entwickeln.

3 Hardwarerealisierung

3.1 Überblick

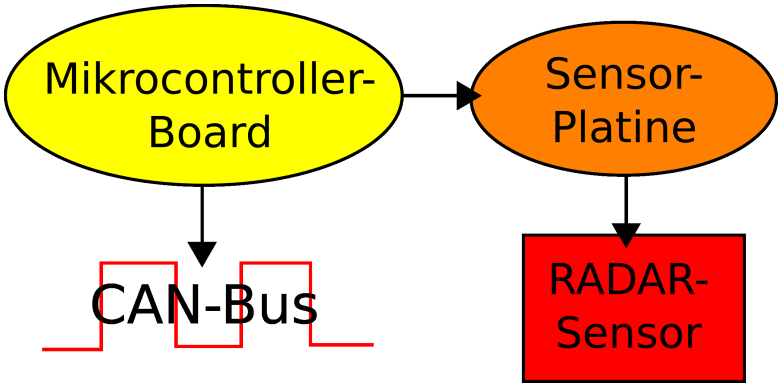


Abbildung 3.1: Hardwaremodule

Das Antikollisionssystem besteht wie in Abbildung 3.1 gezeigt aus drei Hardware-Modulen: Auf dem *Mikrocontroller-Board* um den Atmel AT90CAN128-Mikrocontroller läuft die Software. Diese kalibriert die Sensorik, liest von dieser während der Laufzeit Abstandsdaten und sendet Kollisionswarnungen auf den CAN-Bus. Das zweite Modul ist die *Sensorplatine*. Sie bereitet das vom RADAR-Sensor gelieferte Analog-Signal für die weitere Verarbeitung durch das Mikrocontroller-Board auf.

3.2 Mikrocontroller-Board

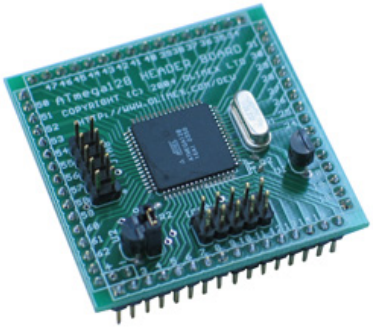


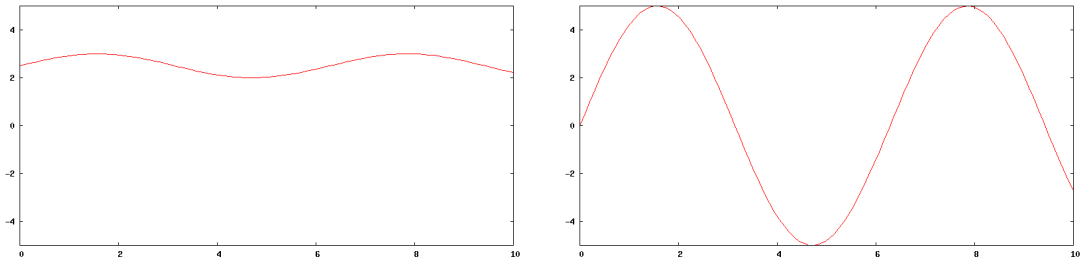
Abbildung 3.2: Mikrocontrollerplatine von Olimex

Das Mikrocontrollerboard basiert auf einer Entwicklungsplatine von Olimex[5] (siehe Abbildung 3.2). Sie wurde um den zum Erzeugen der CAN-Signale benötigten CAN-Treiber und eine serielle Schnittstelle zum Debuggen erweitert. Den zugehörigen Schaltplan zeigt Abbildung 3.3.
Am Analog-Eingang des AT90CAN128 ist die Sensorik angeschlossen. Sie liefert eine Spannung zwischen 0 und 5V, welche anschließend vom Mikrocontroller mit einer Auflösung von 10 Bit digitalisiert wird. Der mit dem CAN-RX- bzw. CAN-TX-Pin verbundene CAN-Treiber wandelt den Pegel der CAN-Bus-Signale in für den Mikrocontroller verarbeitbare Pegel und umgekehrt um. Zusätzliche Digitalausgänge und -eingänge werden für die Kalibrierung der Sensorik genutzt.

3.3 Sensorplatine

3.3.1 Funktionsweise

Die Sensorplatine ist das Herzstück des Antikollisionssystems. Sie liest Daten vom RADAR-Sensor und bereitet diese für das Mikrocontroller-Board auf.
Der Radarsensor liefert ein moduliertes, um etwa 2,5V verschobenes Signal. Er kann nur sich zum Sensor relativ bewegende Objekte erfassen. Die Relativgeschwindigkeit wird aus der Signalfrequenz bestimmt, die Amplitude entspricht dem Abstand Sensor↔Objekt. Da die maximale Amplitude im Millivoltbereich liegt, benötigt man zur Weiterverarbeitung eine Verstärkerschaltung. Die von mir entwickelte Platine verstärkt das Signal in mehreren Stufen. Im ersten Schritt wird mit Hilfe eines Instrumentierungsverstärkers das Signal auf eine Null-Basis gebracht und verzehnfacht. Anschließend wird es noch einmal verhundertfacht und mit Hilfe einer Zenerdiode auf 5V begrenzt.



(a) Ursprungssignal (b) Signal nach der Verstärkung

Abbildung 3.4: Signal am Instrumentierungsverstärker

In Abbildung 3.4 ist dieser Vorgang exemplarisch an einer Sinusschwingung gezeigt. Abbildung 3.4(a) zeigt das Signal bevor und 3.4(b) nachdem es den Instrumentierungsverstärker durchlaufen hat. Die zum Verschieben des Signals im Instrumentierungs-

verstärker benötigte Referenzspannung wird durch einen kalibrierbaren Effektivwertbildner erzeugt. Um das Signal auf eine Null-Basis zu bekomme, verschiebe ich es um seinen Effektivwert nach unten. Durch Toleranzen der verwendeten Widerstände, Kondensatoren und der Versorgungsspannung ist eine Kalibrierung des Effektivwertbildners nötig. Hierfür verwende ich das Digitalpotentiometer MAX5451 von MAXIM, das über die Digitalausgänge des Mikrocontrollers eingestellt wird. Eine zusätzliche Kalibrierschaltung überprüft ob der Effektivwertbildner die richtige Referenzspannung liefert. Diese Schaltung besteht aus zwei Schmitt-Triggern und einem Inverter. Der erste Schmitt-Trigger ist direkt an den Ausgang der Verstärkerstufe angeschlossen und schlägt an sobald das Signal zu hoch ist. Für den zweiten Schmitt-Trigger wird das Ausgangssignal zunächst invertiert. Er zeigt somit an, dass das Signal zu niedrig ist. Die Kalibrierung kann nur durchgeführt werden, wenn sich im Erfassungsbereich des RADAR-Sensors keine bewegenden Objekte befinden, da diese das Signal zusätzlich beeinflussen würden.
Die Operationsverstärker benötigen neben einer positiven auch eine negative Versorgungsspannung. Diese erzeuge ich mit einer einfachen Spannungsinverterschaltung, bestehend aus einem NE555-Timer und Elektrolytkondensatoren. Die Funktionsweise eines solchen Inverters beschreibt beispielsweise [6].

Abbildung 3.5 zeigt den Schaltplan der fertigen Sensorplatine, die Abbildungen 3.7 und 3.6 die Unter- bzw. die Oberseite des fertigen Platinenlayouts. Eine mit Eagle3D[7] und POVRay[8] gerenderte Version der Platine zeigt Abbildung 3.8. Auf ihr kann man sehen, wie die fertig bestückte Platine am Ende aussieht.

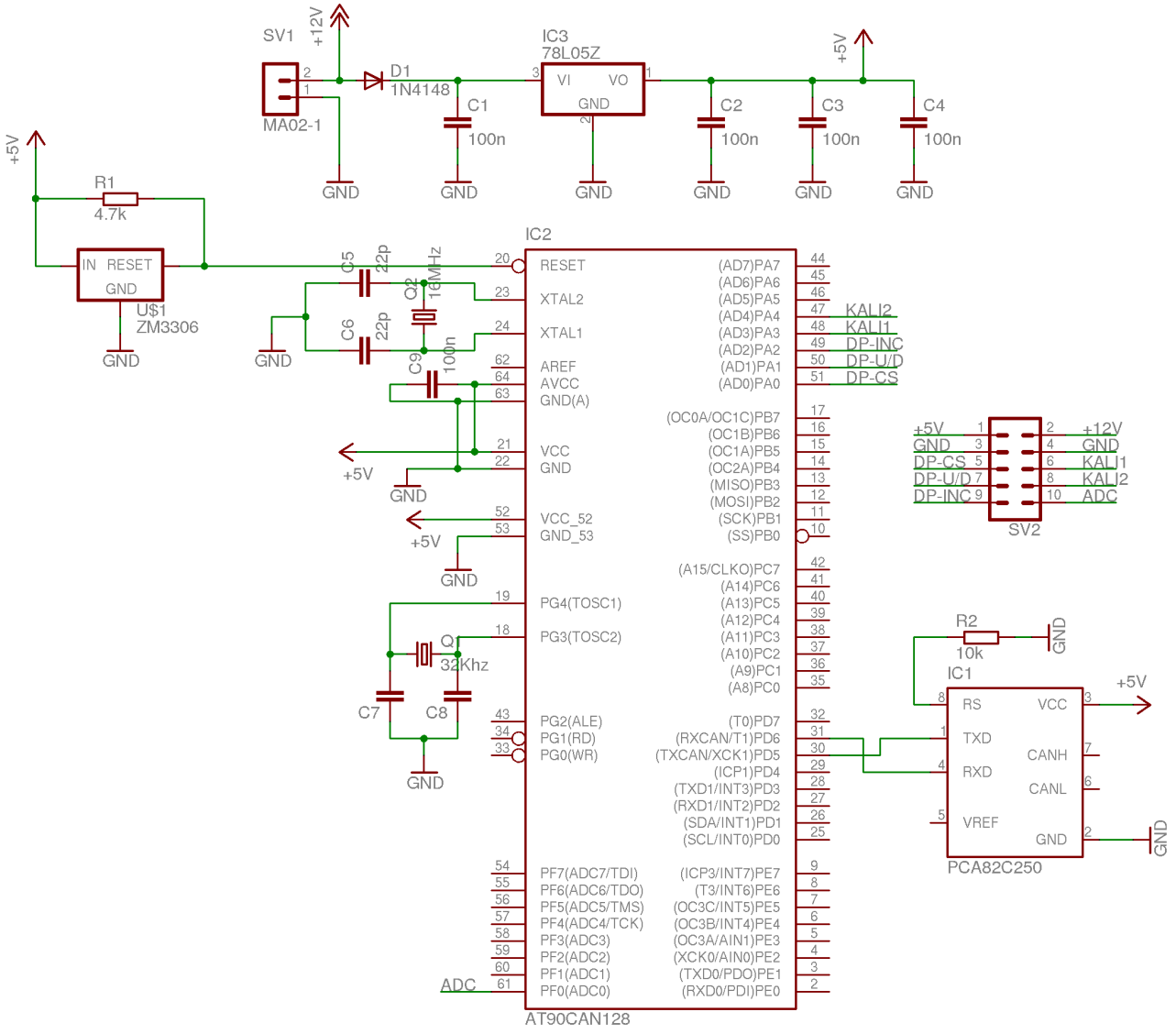


Abbildung 3.3: Schaltplan der vollständigen Mikrocontrollerplatine

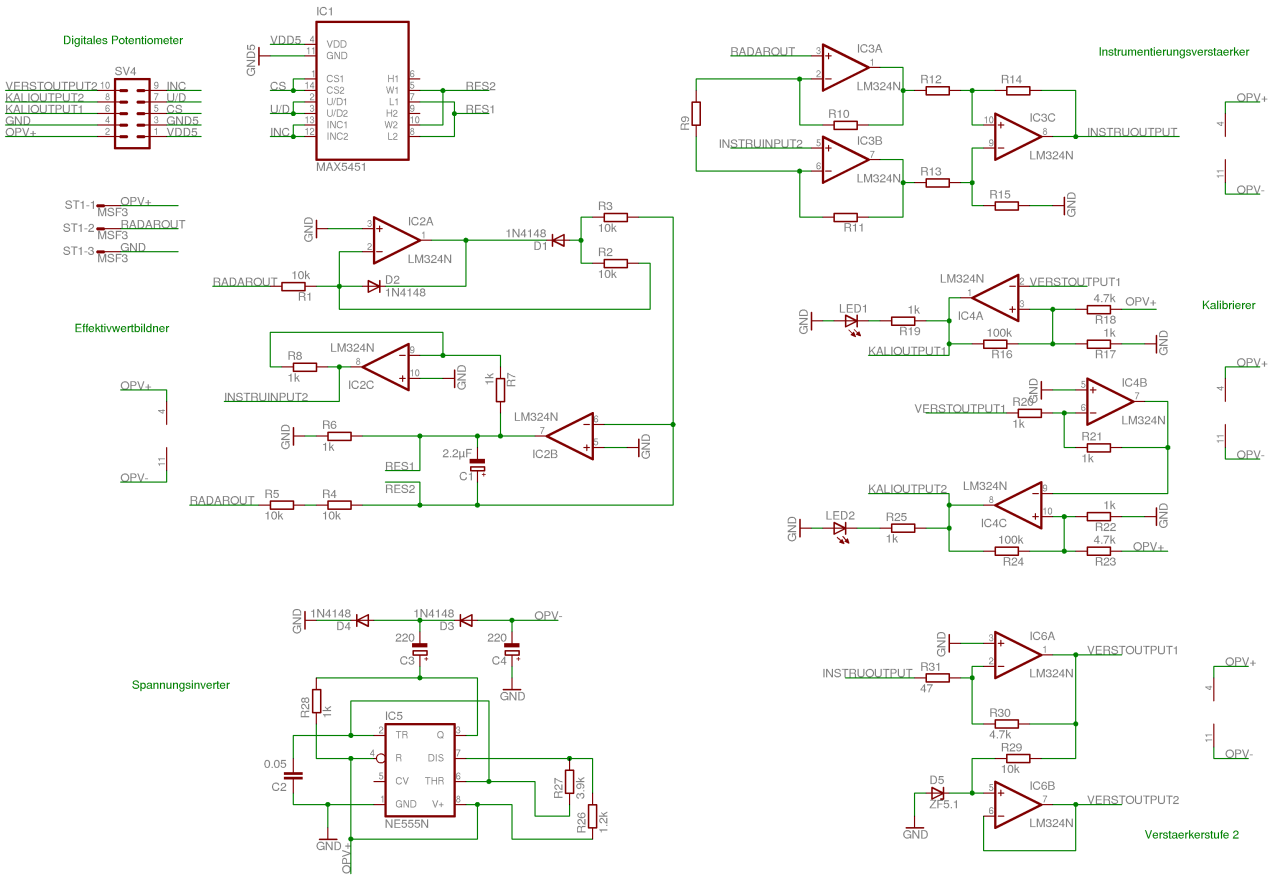


Abbildung 3.5: Schaltplan der Sensorplatine

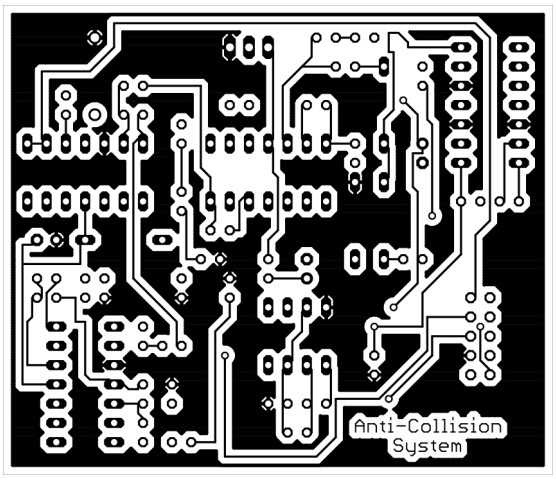


Abbildung 3.6: Platinenlayout der Sensorplatine (Oberseite)

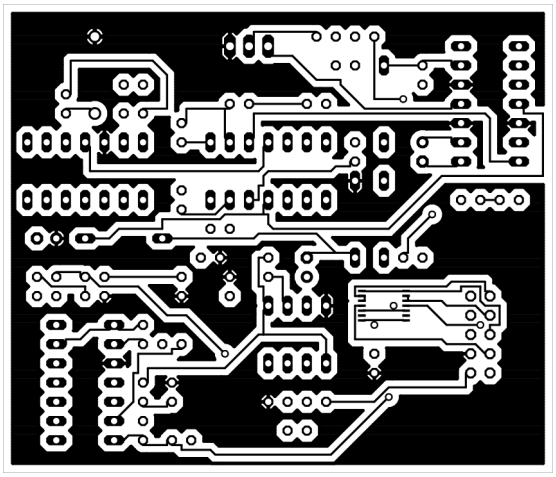


Abbildung 3.7: Platinenlayout der Sensorplatine (Unterseite)

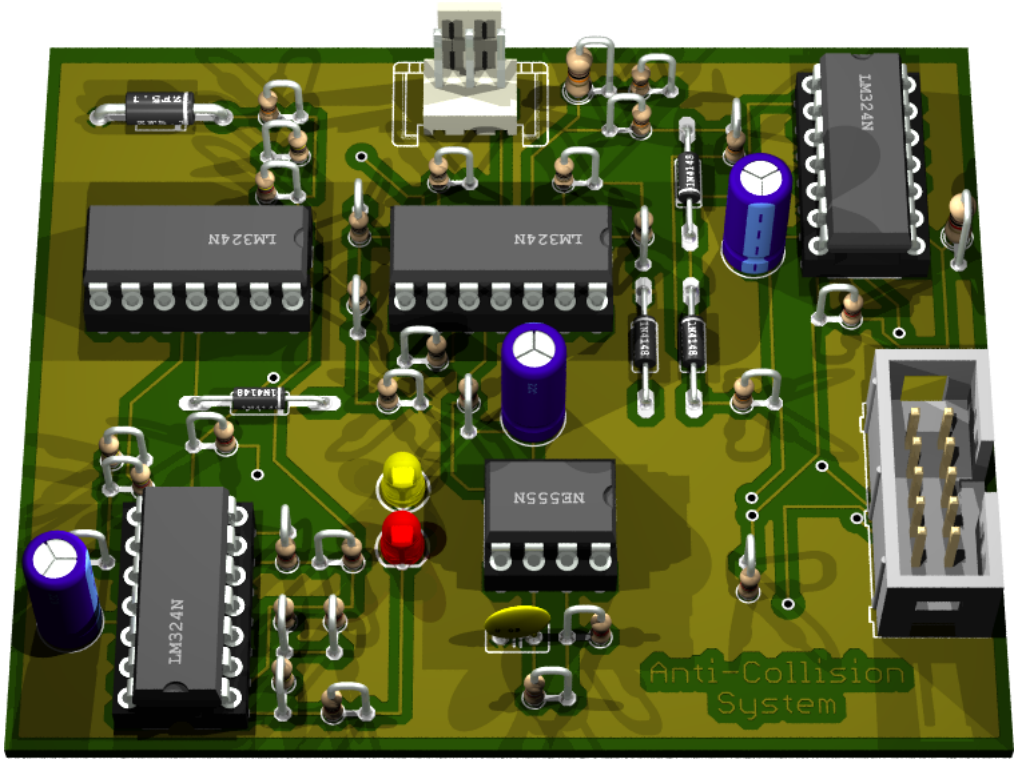


Abbildung 3.8: Rendering der fertigen Sensorplatine (Oberseite)

3.3.2 Entwicklungsschritte

Die Entwicklung der Sensorplatine verlief in mehreren Schritten:
Als erstes begann ich mit Messungen des RADAR-Sensors am Oszilloskop. Das Ergebnis einer solchen Messung zeigt Abbildung 3.9.

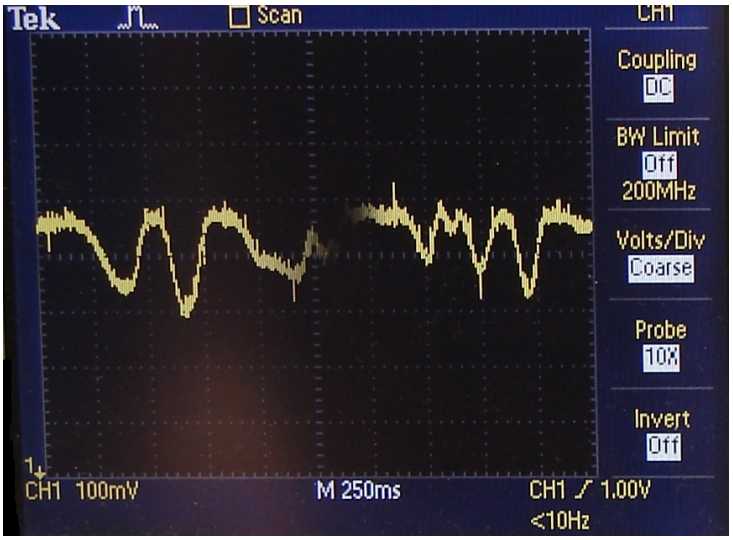


Abbildung 3.9: RADAR-Signal auf dem Oszilloskop

Im nächsten Schritt habe ich das Signal durch den im vorherigen Abschnitt beschriebenen Instrumentierungsverstärker aufbereitet und erste Tests mit dem Mikrocontroller durchgeführt. Als Testumgebung habe ich ein Linux-Programm geschrieben, das Werte von der seriellen Schnittstelle liest und diese in Echtzeit graphisch auf dem Bildschirm darstellt. Die Werte kamen vom AD-Wandler des Mikrocontrollers, der am Ausgang des Instrumentierungsverstärkers angeschlossen war.

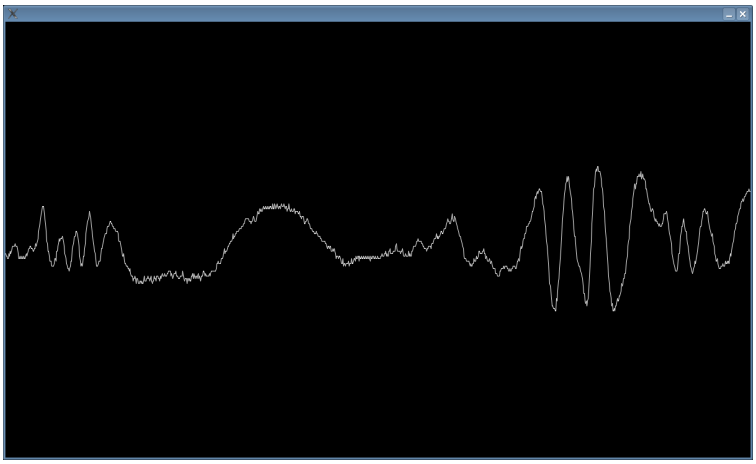


Abbildung 3.10: Screenshot des Oszilloskop-Programms

Abbildung 3.10 zeigt einen Screenshot meines Programms im laufenden Betrieb.

Um das Signal noch besser nutzbar zu machen, habe ich anschließend eine weitere Verstärkerstufe entwickelt, die das Signal noch einmal verstärkt und mit Hilfe von Zener-Dioden auf für den A/D-Wandler verträgliche 5V beschränkt. Zeitgleich kamen die Kalibrierschaltung und der Effektivwertbildner hinzu um ein genaueres Einstellen des Instrumentierungsverstärkers zu ermöglichen. Nachdem nun die elektrotechnische Seite funktionsbereit war, begann ich mit der Modellierung und Implementierung der Software.

4 Systembeschreibung

4.1 Gesamtsystem

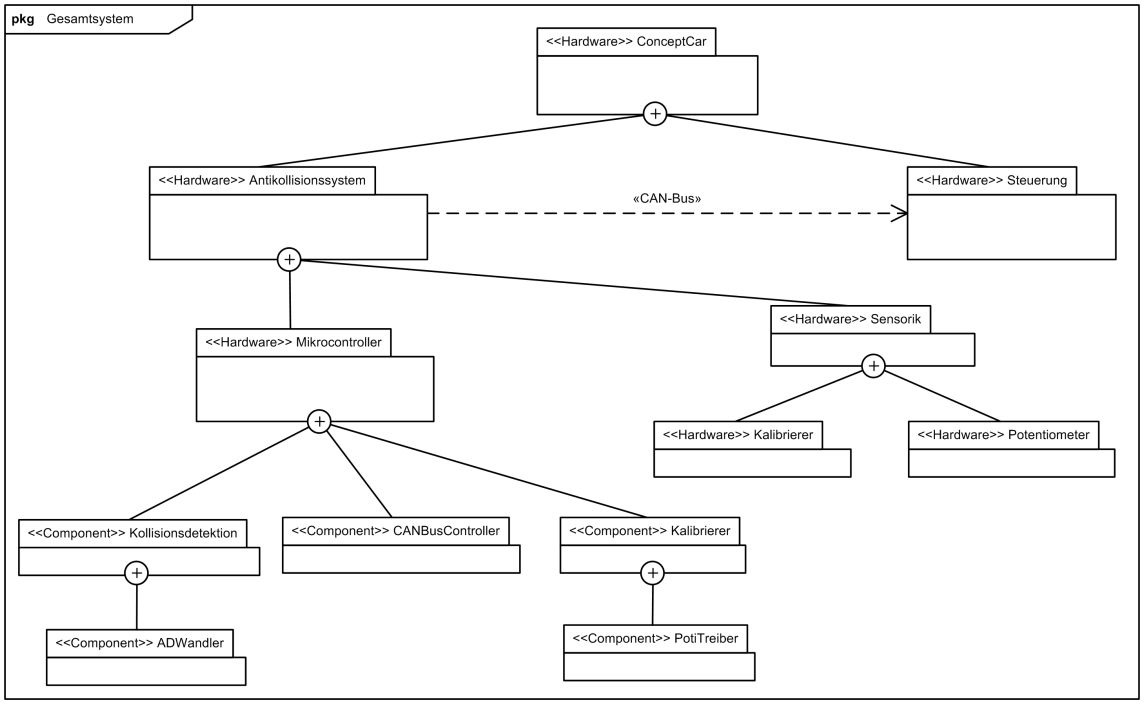


Abbildung 4.1: Paketdiagramm Gesamtsystem

Abbildung 4.1 zeigt einen groben Aufbau des Antikollisionssystems und wie es sich in das Gesamtsystem einfügt. An oberster Stelle steht das *ConceptCar*. Es besteht aus vielen einzelnen Komponenten, von denen hier die für diese Arbeit beiden wichtigen, das *Antikollisionssystem* und die *Steuerung*, gezeigt sind. Die *Steuerung* kümmert sich um die Ansteuerung des Motors und der Bremsen. Durch den Stereotyp *<<Hardware>>* wird verdeutlicht, dass es sich hierbei um Hardwarekomponenten handelt. Das Antikollisionssystem besteht aus weiteren Subkomponenten, die wiederum Subkomponenten enthalten (können). In MARMOT wird eine Komponente soweit in Subkomponenten unterteilt und jeweils rekursiv als eigenständiges MARMOT-Problem gelöst, bis man auf atomare Probleme trifft.
Im Folgenden wird die Entwicklung des Antikollisionssystems betrachtet.

4.2 Antikollisionssystem

Die groben Anforderungen an das System werden in MARMOT durch Use-Cases beschrieben.

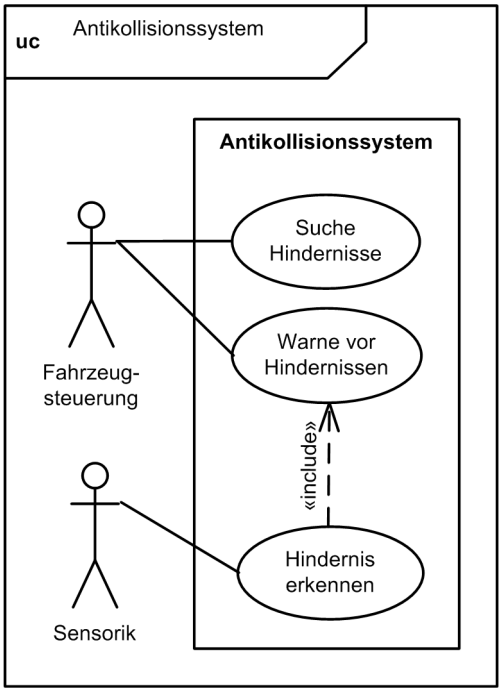


Abbildung 4.2: Use-Case-Diagramm Antikollisionssystem

Abbildung 4.2 zeigt das Use-Case-Diagramm für das Antikollisionssystem.
Das Antikollisionssystem sucht für den Akteur *Fahrzeugsteuerung* nach Hindernissen. Der zweite Akteur, die *Sensorik*, führt diese Suche durch und wird vom Antikollisionssystem regelmäßig abgefragt. Findet dieses mögliche Hindernisse, meldet es sie an die *Fahrzeugsteuerung*. Die Tabellen 4.1, 4.2 und 4.3 beschreiben die einzelnen Use-Cases genauer.

Name	Suche Hindernisse
Akteur	Fahrzeugsteuerung
Ziel	Hindernisse vor dem ConceptCar sollen gefunden werden.
Beschreibung	Während des gesamten Betriebs soll mit Hilfe einer Sensorik aktiv nach vor dem ConceptCar befindlichen Hindernissen gesucht werden.
Ausnahmen	Hindernisse können aufgrund der verwendeten Sensorik nur erkannt werden, wenn sie sich relativ zum Sensor bewegen.
Regeln	N/A
Qualitätsanforderungen	N/A
I/O	N/A
Vorbedingung	Sytem ist initialisiert
Nachbedingung	N/A

Tabelle 4.1: Use-Case-Beschreibung - Suche Hindernisse

Name	Warne vor Hindernissen
Akteur	Fahrzeugsteuerung
Ziel	Vor gefundenen Hindernissen soll gewarnt werden.
Beschreibung	Nachdem das Antikollisionssystem ein Hindernis gefunden hat, soll die Fahrzeugsteuerung vor diesem gewarnt werden.
Ausnahmen	N/A
Regeln	N/A
Qualitätsanforderungen	Die Warnung muss bei der Fahrzeugsteuerung ankommen.
I/O	CAN-Bus
Vorbedingung	Hindernis gefunden
Nachbedingung	Fahrzeugsteuerung wurde vor Hindernis gewarnt.

Tabelle 4.2: Use-Case-Beschreibung - Warne vor Hindernissen

Name	Hindernisse erkennen
Akteur	Sensorik
Ziel	Hindernisse sollen gefunden werden.
Beschreibung	Mit Hilfe der Sensorik sollen vor dem ConceptCar befindliche Hindernisse gefunden werden.
Ausnahmen	Hindernisse außerhalb der Reichweite der Sensorik werden nicht erkannt.
Regeln	N/A
Qualitätsanforderungen	Hindernisse sollen sicher, mit möglichst niedriger Fehlerquote erkannt werden.
I/O	Sensorik
Vorbedingung	System ist initialisiert
Nachbedingung	Hindernis gefunden

Tabelle 4.3: Use-Case-Beschreibung - Hindernisse erkennen

4.2.1 Mikrocontroller

In diesem Abschnitt wird die Komponente *Mikrocontroller* vorgestellt. Im Gegensatz zu den übrigen Komponenten ist sie eine Hybrid-Komponente, da sie sowohl die Hardware "Mikrocontroller" als auch die darauf laufende Software repräsentiert. Dies wird durch den Stereotyp *<<Hardware>>* noch einmal gesondert gekennzeichnet. Die übrigen MARMOT-Komponenten sind sonst durch *<<Component>>* markiert.

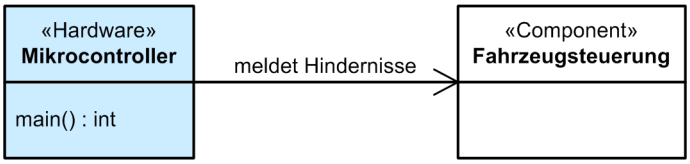


Abbildung 4.3: Spezifikation Mikrocontroller

Abbildung 4.3 zeigt die Spezifikation der Komponente. Die einzige Funktion die sie enthält ist die *main*-Funktion, die beim Anlegen der Stromversorgung, bzw. bei einem Hardware-Reset des Mikrocontrollers ausgeführt wird.

Die funktionale Beschreibung enthält Tabelle 4.4.

Name	Mikrocontroller
Beschreibung	*main:* Hauptfunktion der Software, enthält die Hauptschleife und wird beim Anschalten des Systems aufgerufen.
Einschränkungen	N/A
Empfängt	N/A
Rückgabe	N/A
Sendet	CAN-Telegramme mit Hilfe des CANBusControllers
Liest	Aktuellen Kollisionsstatus von der Kollisionsdetektion
Verändert	N/A
Regeln	Erkannte Hindernisse müssen schnellstmöglich, d.h. so früh, dass das Auto noch vor dem Aufprall zum Stehen kommt, gemeldet werden und es muss sichergestellt sein, dass die Warn-Meldungen alle ankommen.
Vorraussetzungen	System ist mit Strom versorgt
Ergebnis	Mit Hilfe der Treiber-Komponenten wurde die Hardware initialisiert, das Zusammenspiel der Software-Komponenten wurde koordiniert und somit die Software-Funktionalität des Gesamtsystems implementiert.

Tabelle 4.4: Funktionsspezifikation Mikrocontroller

Das Zustandsdiagramm in Abbildung 4.4 zeigt wie das System nach der Initialisierung arbeitet. Es befindet sich so lange im Zustand *Suche nach Hindernissen* bis es ein Hindernis gefunden hat. In einem solchen Fall schaltet es in den Zustand *Warne vor Hindernissen* in dem es eine Warnmeldung versendet, und anschließend wieder zurück in den Hauptzustand.

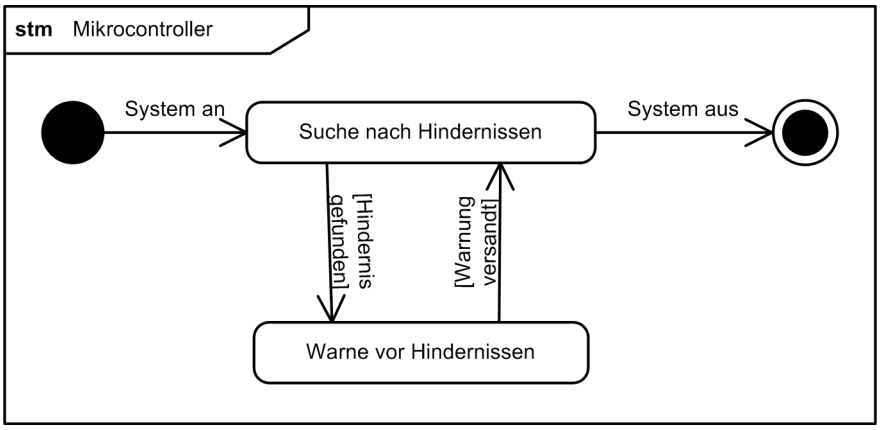


Abbildung 4.4: Zustandsdiagramm Mikrocontroller

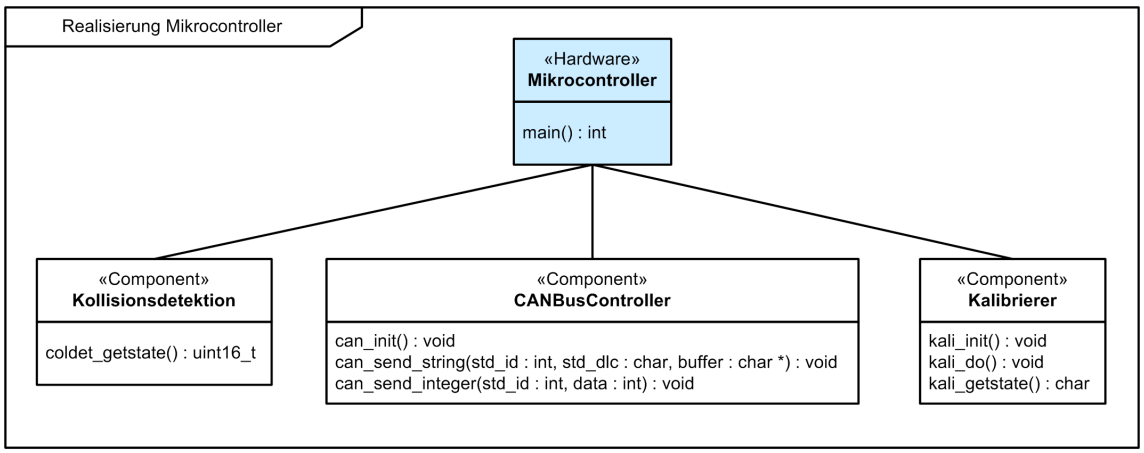


Abbildung 4.5: Realisierung Mikrocontroller

Das Klassendiagramm in Abbildung 4.5 zeigt die Realisierung der Komponente. Sie greift auf Funktionen der Subkomponenten *Kollisionsdetektion*, *CANBusContoller* und *Kalibrierer* zu. Das Sequenzdiagramm in Abbildung 4.6 zeigt die Zugriffe im zeitlichen Ablauf. Zunächst wird der Mikrocontroller initialisiert, anschließend wird in einer Endlosschleife nach Hindernissen gesucht. Der genaue Ablauf des Initialisierungsvorgungs und der Kollisionsdetektion wird in den Subdiagrammen in Abbildung 4.7 und Abbildung 4.8 dargestellt. In der Initialisierungsphase werden nacheinander Kollisionsdetektion, CANBusController, PotiTreiber und Kalibrierer initialisiert.
In der darauf folgenden Kollisionsdetektionsphase wird die Kollisionsdetektion nach dem aktuellen Kollisionsstatus gefragt. Wurde ein Hindernis gefunden, wird dieses über ein CAN-Telegramm der zentralen Fahrsteuerung des ConceptCar gemeldet.

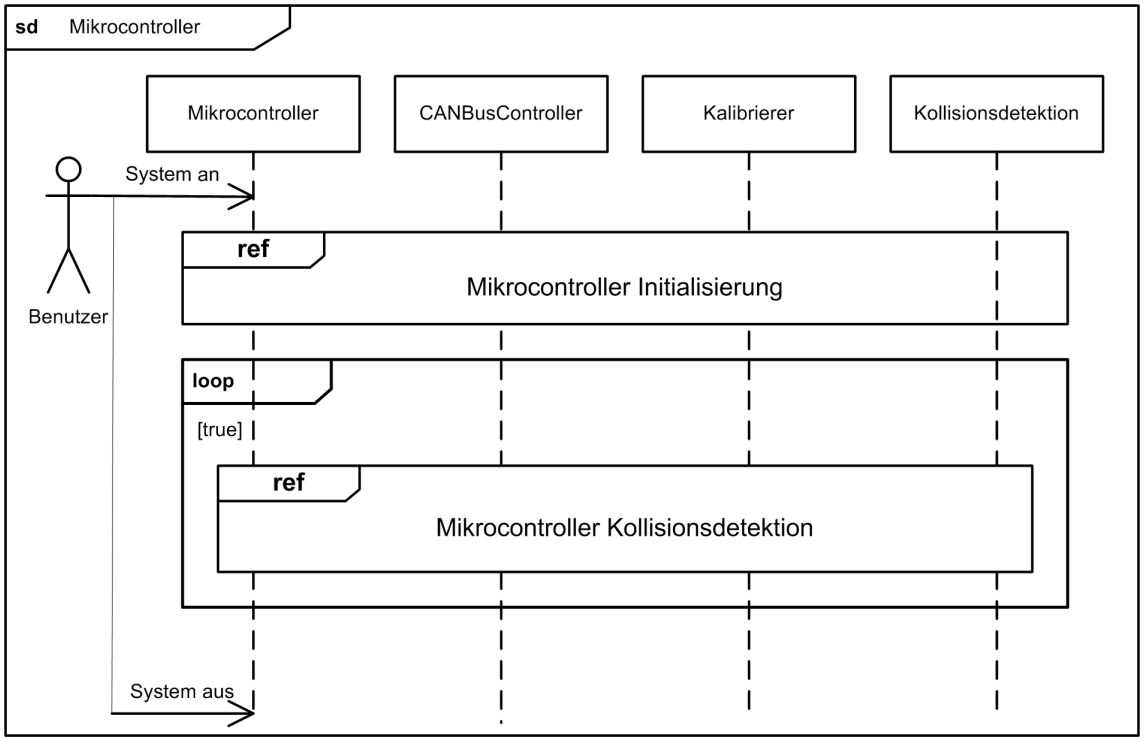


Abbildung 4.6: Sequenzdiagramm Mikrocontroller

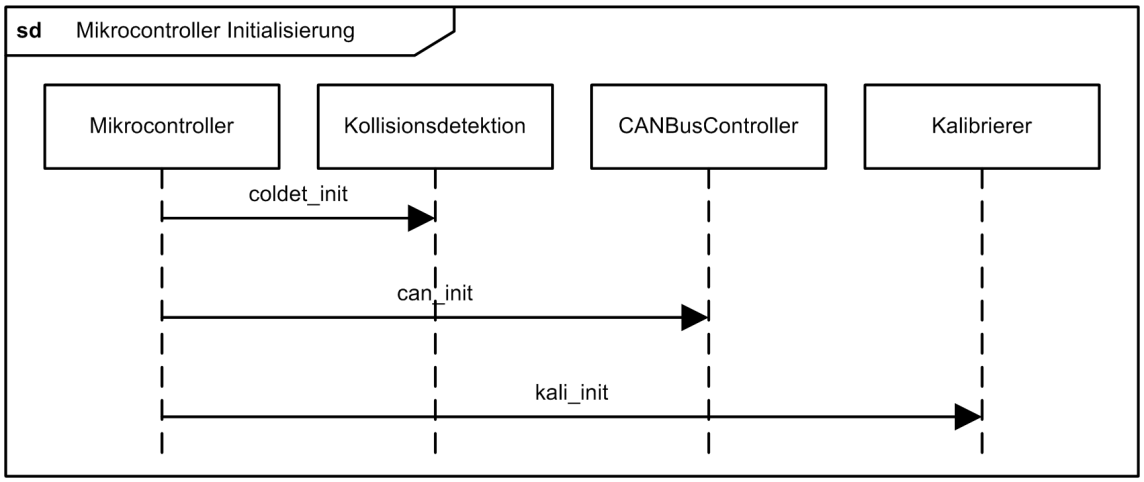


Abbildung 4.7: Sequenzdiagramm Mikrocontroller Initialisierung

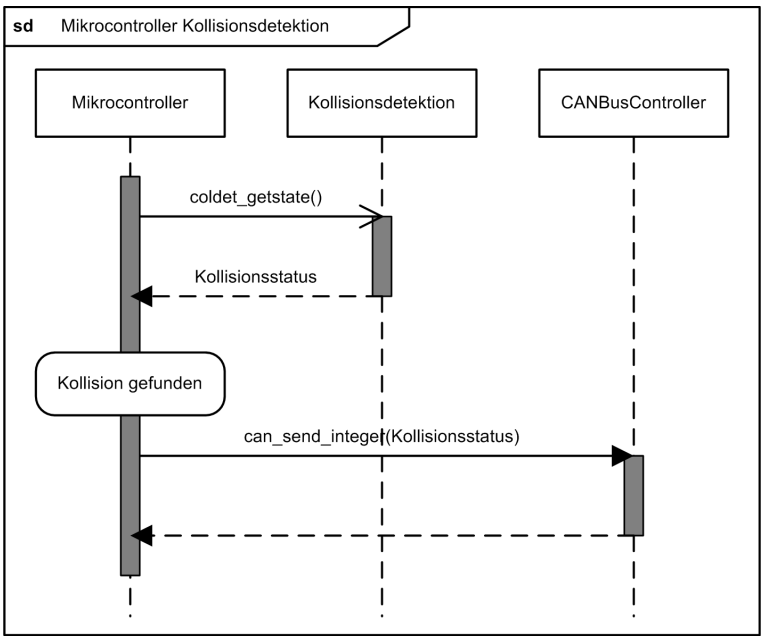


Abbildung 4.8: Sequenzdiagramm Mikrocontroller Kollisionsdetektion

Den algorithmischen Ablauf im Mikrocontroller beschreibt das Aktivitätsdiagramm in Abbildung 4.9. Es kombiniert sozusagen die Informationen aus dem Zustandsdiagramm und den Sequenzdiagrammen und gibt diese noch einmal etwas grober wieder.

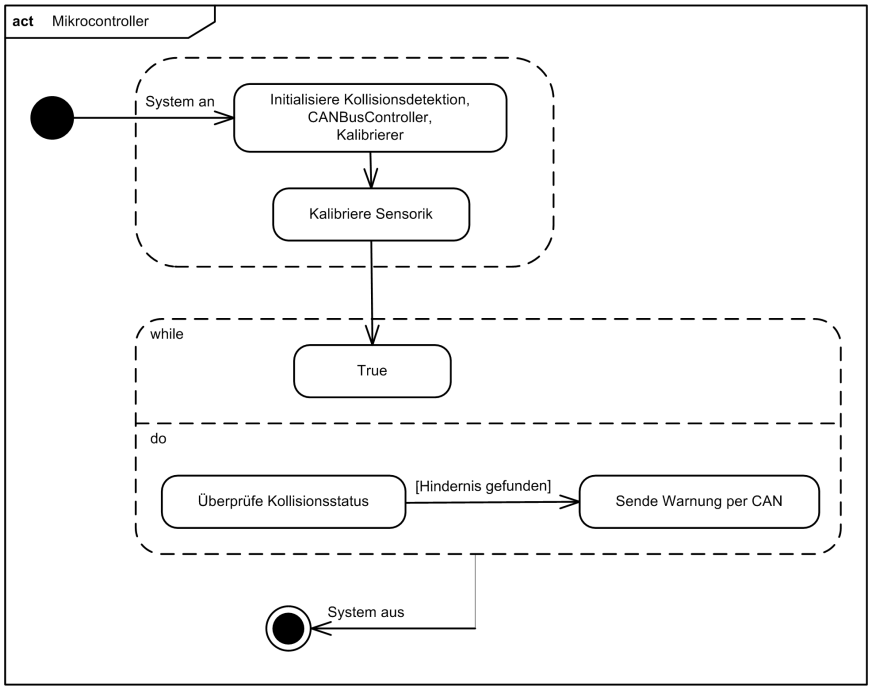


Abbildung 4.9: Aktivitätsdiagramm Mikrocontroller

4.2.2 Kollisionsdetektion

Die erste Subkomponente des Mikrocontrollers ist die *Kollisionsdetektion*.

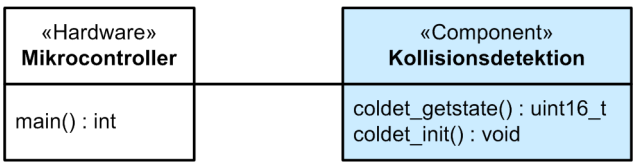


Abbildung 4.10: Spezifikation Kollisionsdetektion

Das Klassendiagramm in Abbildung 4.10 zeigt deren Spezifikation. Die funktionale Beschreibung befindet sich in Tabelle 4.5.

Name	Kollisionsdetektion
Beschreibung	• *coldet_getstate:* Gibt den aktuellen Kollisionstatus zurück (je höher desto kritischer). • *coldet_init:* Initialisiert die von der Kollisionsdetektion benötigten Komponenten.
Einschränkungen	Nur Hindernisse innerhalb eines beschränkten Gebietes können erkannt werden.
Empfängt	N/A
Rückgabe	Aktuellen Status, ob Hindernisse gefunden wurden
Sendet	N/A
Liest	Werte vom ADWandler
Verändert	N/A
Regeln	N/A
Vorraussetzungen	N/A
Ergebnis	Die Kollisiondetektion hat mit Hilfe der Sensorik Hindernisse gesucht und auf Anfrage den aktuellen Kollisionsstatus zurückgegeben.

Tabelle 4.5: Funktionsspezifikation Kollisionsdetektion

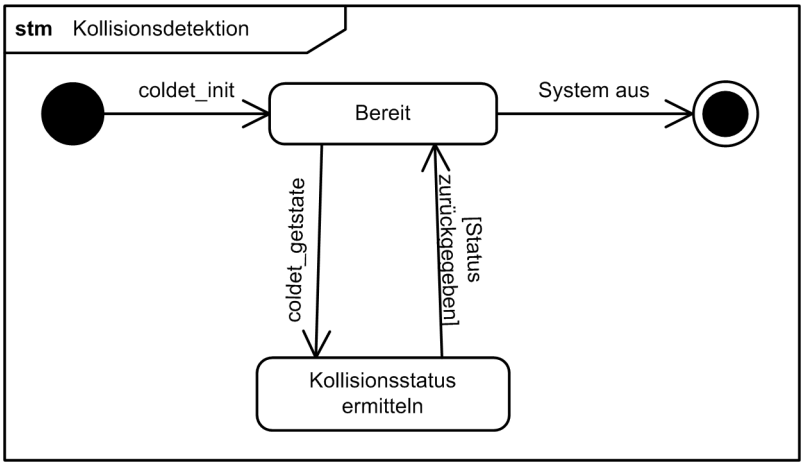


Abbildung 4.11: Zustandsdiagramm Kollisionsdetektion

Abbildung 4.11 zeigt in Form eines Zustandsdiagramms den Ablauf der Komponente. Die Kollisionsdetektion ist solange in einem Wartezustand bis vom Mikrocontroller die Funktion *coldet_getstate* aufgerufen wird. Anschließend ermittelt sie den aktuellen Kollisionsstatus und gibt diesen an den Mikrocontroller zurück.

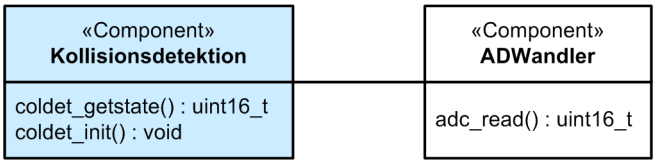


Abbildung 4.12: Realisierung Kollisionsdetektion

Die Realisierung der *Kollisionsdetektion* zeigt Abbildung 4.12.

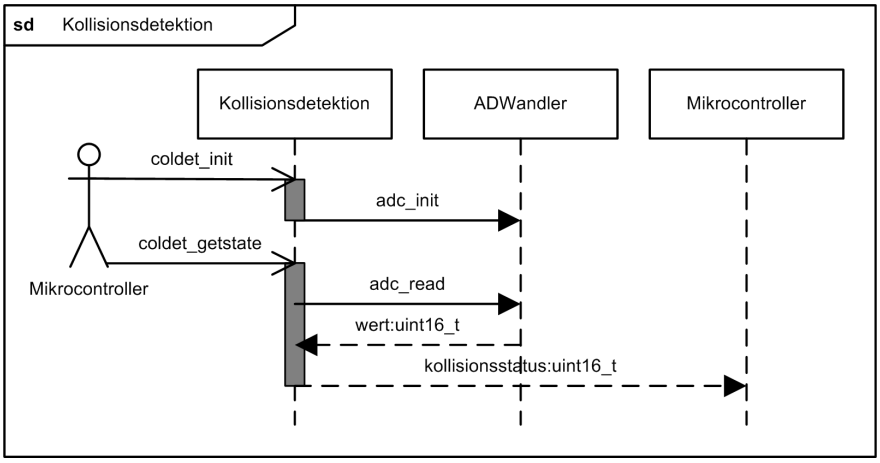


Abbildung 4.13: Sequenzdiagramm Kollisionsdetektion

Abbildung 4.13 zeigt wieder in Form eines Sequenzdiagramms den zeitlichen Ablauf der Kollisionsdetektion. Zunächst liest sie den aktuellen Wert von ADWandler, verarbeitet diesen und gibt ihn in Form eines Kollisionsstatus an den Mikrocontroller zurück.

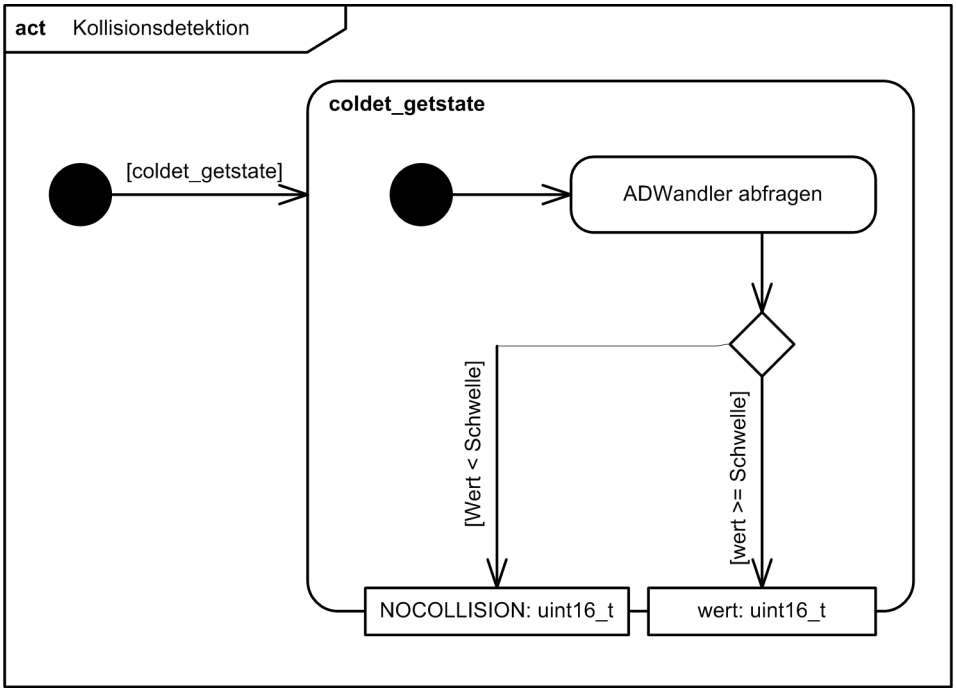


Abbildung 4.14: Aktivitätsdiagramm Kollisionsdetektion

Den genauen algorithmischen Ablauf sieht man in Abbildung 4.14. Das Aktivitätsdiagramm zeigt, dass der ADWandler-Wert über einen einfachen Schwellwert-Schalter ausgewertet wird. Überschreitet dieser eine festgelegte Schwelle, wird der Wert als Kollisionsstatus, andernfalls "keine Kollisionsgefahr" (Null) zurückgegeben.

4.2.3 ADWandler

Die *Kollisionsdetektion* arbeitet mit Hilfe eines ADWandlers.

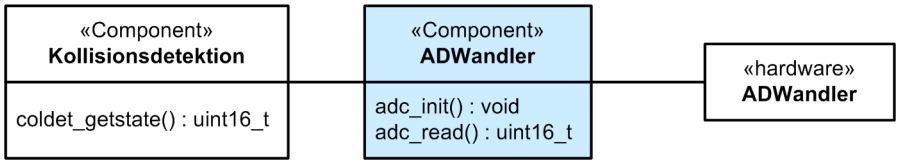


Abbildung 4.15: Spezifikation ADWandler

Die Spezifikation in Abbildung 4.15 zeigt dessen Struktur und dass er von den Komponenten *Mikrocontroller* und *Kollisionsdetektion* verwendet wird. Er arbeitet als Software-Treiberkomponente für den auf dem Mikrocontroller integrierten Hardware-AD-Wandler dessen Zweck es ist, analoge Spannungen einzulesen und dem System als digitale Ganzzahl bereit zu stellen.

Name	ADWandler
Beschreibung	• *adc_init:* Initialisiert den ADWandler • *adc_read:* Konvertiert einen Analogwert und gibt diesen zurück
Einschränkungen	Spannungen die größer sind als die Referenzspannung bzw. kleiner als 0V können nicht gelesen werden, da sie den Mikrocontroller beschädigen können. Werte unter 0V werden als 0 und Werte größer als die Referenzspannung als 1024 zurückgegeben.
Empfängt	N/A
Rückgabe	Verhältnis Spannung/Referenz als positive Ganzzahl
Sendet	N/A
Liest	Analoge Spannung von der Sensorik
Verändert	N/A
Regeln	N/A
Vorraussetzungen	A/D-Wandler ist initialisiert
Ergebnis	Der ADWandler wurde initialisiert, Analogwerte wurden gelesen und in Ganzzahlen umgewandelt.

Tabelle 4.6: Funktionsspezifikation ADWandler

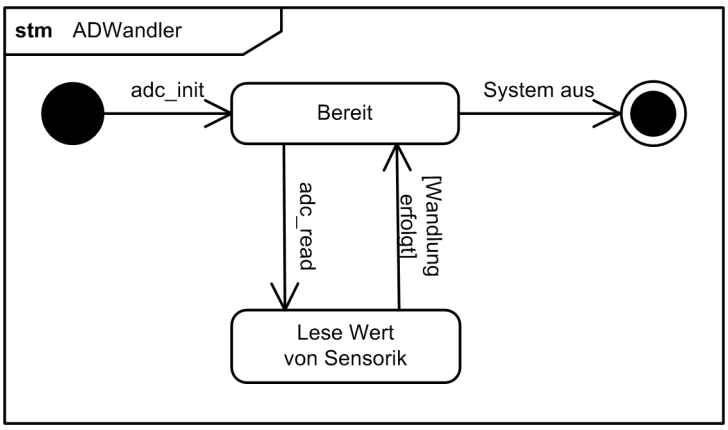


Abbildung 4.16: Zustandsdiagramm ADWandler

Das Zustandsdiagramm in Abbildung 4.16 ähnelt dem der Kollisionsdetektion. Wie die Kollisionsdetektion wartet der ADWandler darauf, dass dessen Funktion *adc_read* aufgerufen wird. Anschließend wechselt er in einen Zustand in dem er den aktuellen Wert von der Sensorik einliest und springt zurück in den Wartezustand.

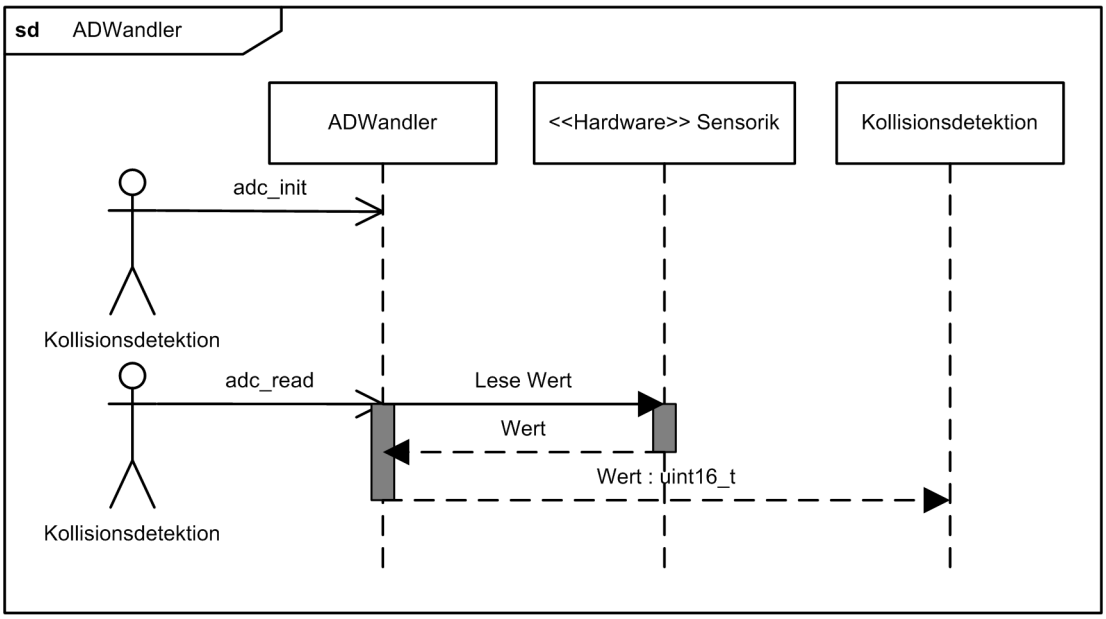


Abbildung 4.17: Sequenzdiagramm ADWandler

Das Sequenzdiagramm in Abbildung 4.17 zeigt dass nur eine Komponente, nämlich die Kollisionsdetektion, auf den ADWandler zugreift. Als erstes initialisiert sie den ADWandler durch Aufrufen der Funktion *adc_init*. Anschließend kann die Kollisionsdetektion über die Funktion *adc_read* vom ADWandler Werte einlesen. In diesem Fall liest der ADWandler zunächst Analogwerte von der Sensorik, digitalisiert diese und gibt sie der Kollisionsdetektion zurück.

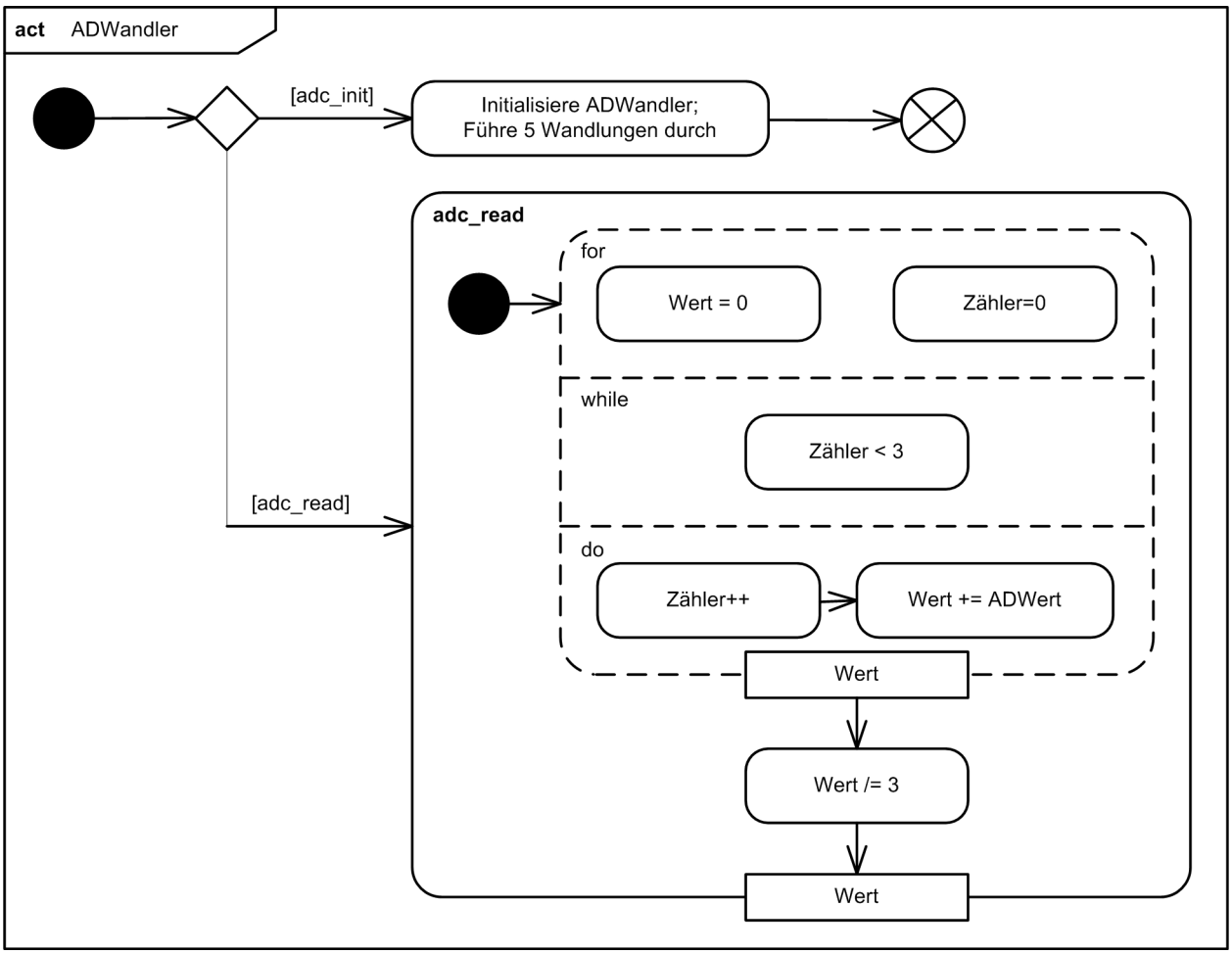


Abbildung 4.18: Aktivitätsdiagramm ADWandler

Abbildung 4.18 zeigt die Arbeitsweise der Funktionen *adc_init* und *adc_read*.
In *adc_init* wird zunächst der ADWandler durch setzen verschiedener Register initialisiert. Anschließend werden 5 AD-Wandlungen durchgeführt um die korrekte Funktionsweise zu gewährleisten.
adc_read führt 3 Wandlungen durch und gibt das arithmetische Mittel der gelesenen Werte zurück. Diese Mittelung gleicht kleine Spannungsschwankungen aus und verringert so die Wahrscheinlichkeit einer Fehlmeldung.

4.2.4 CANBusController

Der *CANBusController* arbeitet als Hilfskomponente zum vereinfachten Zugriff auf die von Atmel® veröffentlichte *CANLib*. Abbildung 4.19 zeigt dessen Struktur, die funktionale Beschreibung enthält Tabelle 4.7.

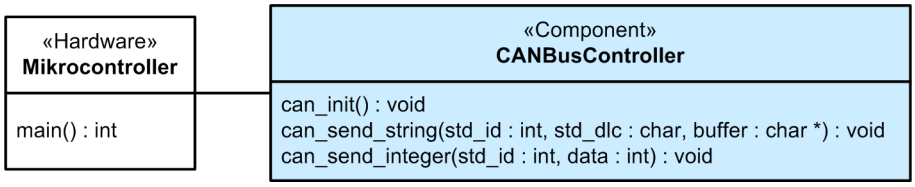


Abbildung 4.19: Spezifikation CANBusController

Name	CANBusController
Beschreibung	• *can_init:* Initialisiert den CANBusController • *can_send_string:* Sendet eine Zeichenkette als CAN-Telegramm • *can_send_integer:* Sendet eine Integer-Zahl als CAN-Telegramm
Einschränkungen	N/A
Empfängt	Zu sendende Nachrichten als Integer oder String
Rückgabe	N/A
Sendet	CAN-Telegramme
Liest	N/A
Verändert	N/A
Regeln	N/A
Vorraussetzungen	CAN-Bus muss initialisiert sein
Ergebnis	Der CAN-Bus wurde initialisiert und CAN-Telegramme wurden gesendet.

Tabelle 4.7: Funktionsspezifikation CANBusController

Die einzige Komponente die auf den CANBusController zugreift ist der Mikrocontroller. Dieser führt zunächst die Funktion *can_init* aus um anschließend über die Funktionen *can_send_integer* bzw. *can_send_string* CAN-Telegramme zu versenden.

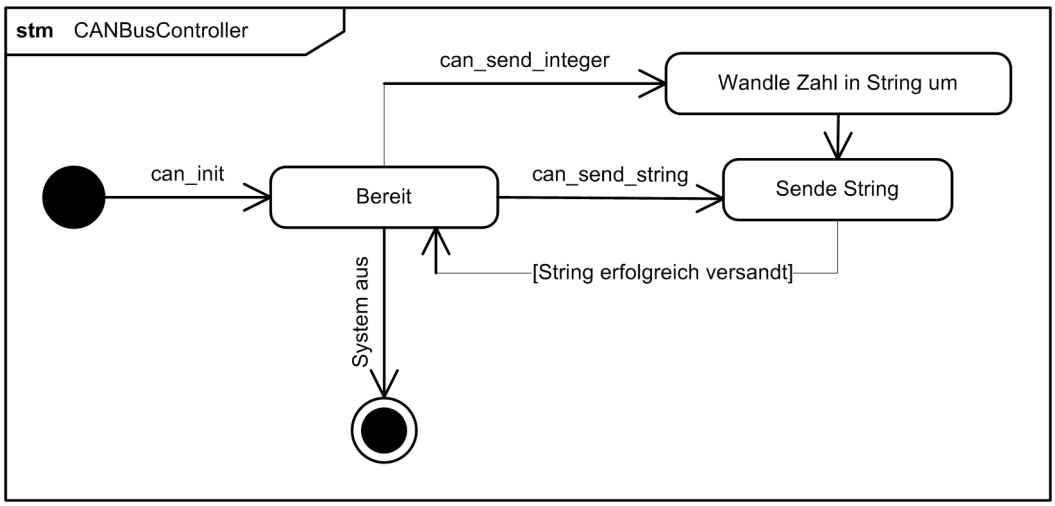


Abbildung 4.20: Zustandsdiagramm CANBusController

Das Zustandsdiagramm in Abbildung 4.20 stellt den Ablauf grob dar.

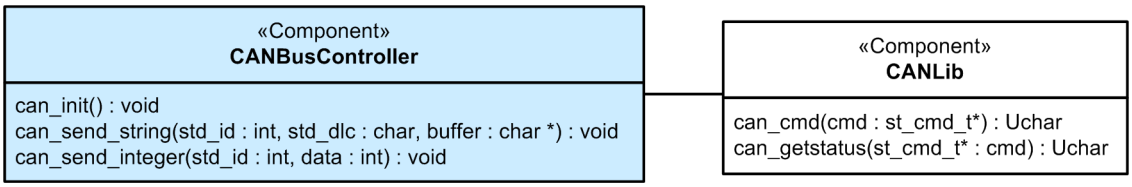


Abbildung 4.21: Realisierung CANBusController

Abbildung 4.21 zeigt die Realisierung der Komponente mit Hilfe der Atmel® *CANLib*.

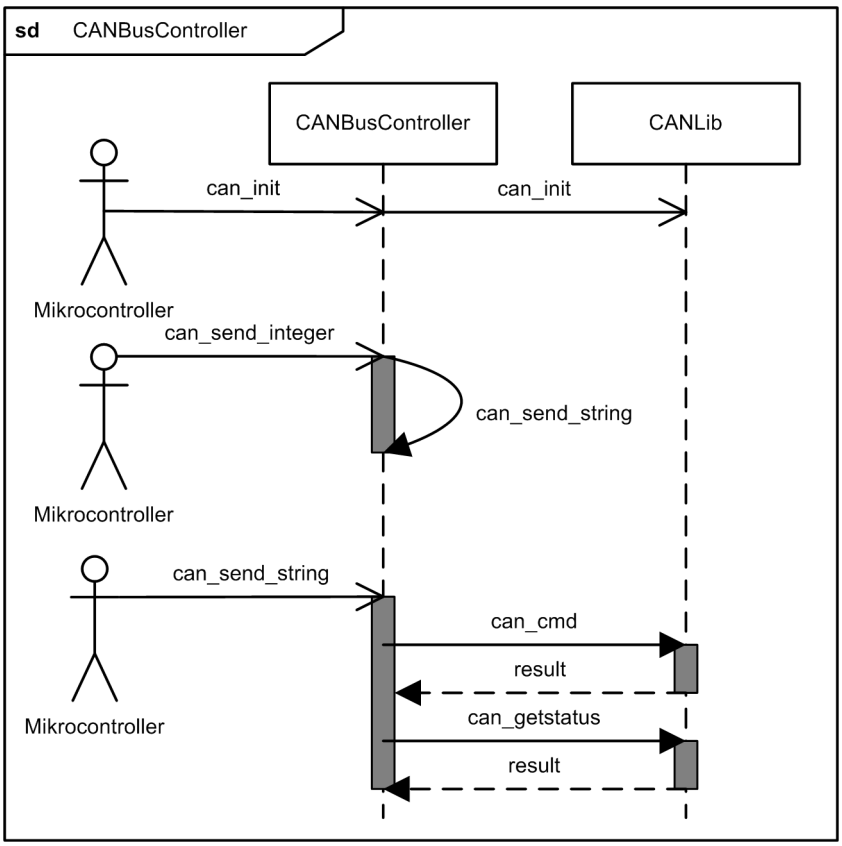


Abbildung 4.22: Sequenzdiagramm CANBusController

Das Sequenzdiagramm in Abbildung 4.22 verdeutlicht diese im zeitlichen Ablauf. Beim Aufruf der Funktion *can_send_integer*, wird die Zahl zunächst in einen String umgewandelt und anschließend versandt. Das Aktivitätsdiagramm in Abbildung 4.23 zeigt den Ablauf in den einzelnen Funktionen:
In *can_send_integer* wird zunächst ein 2-Byte-Puffer angelegt. Die oberen 8 Bit der Zahl werden ins erste Byte und die unteren 8 Bit ins zweite Byte geschrieben. Anschließend wird der Puffer mit der Funktion *can_send_string* versandt.
can_send_string baut zunächst einen struct mit allen fürs Senden des CAN-Telegramms nötigen Informationen zusammen. Nun versucht der CANBusController solange die Nachricht zu versenden, bis sie vom Bus akzeptiert wurde. Zuletzt wird solange gewartet bis das Telegramm auch wirklich versandt wurde. So wird sichergestellt, dass die Daten auch wirklich am Bus angekommen sind.

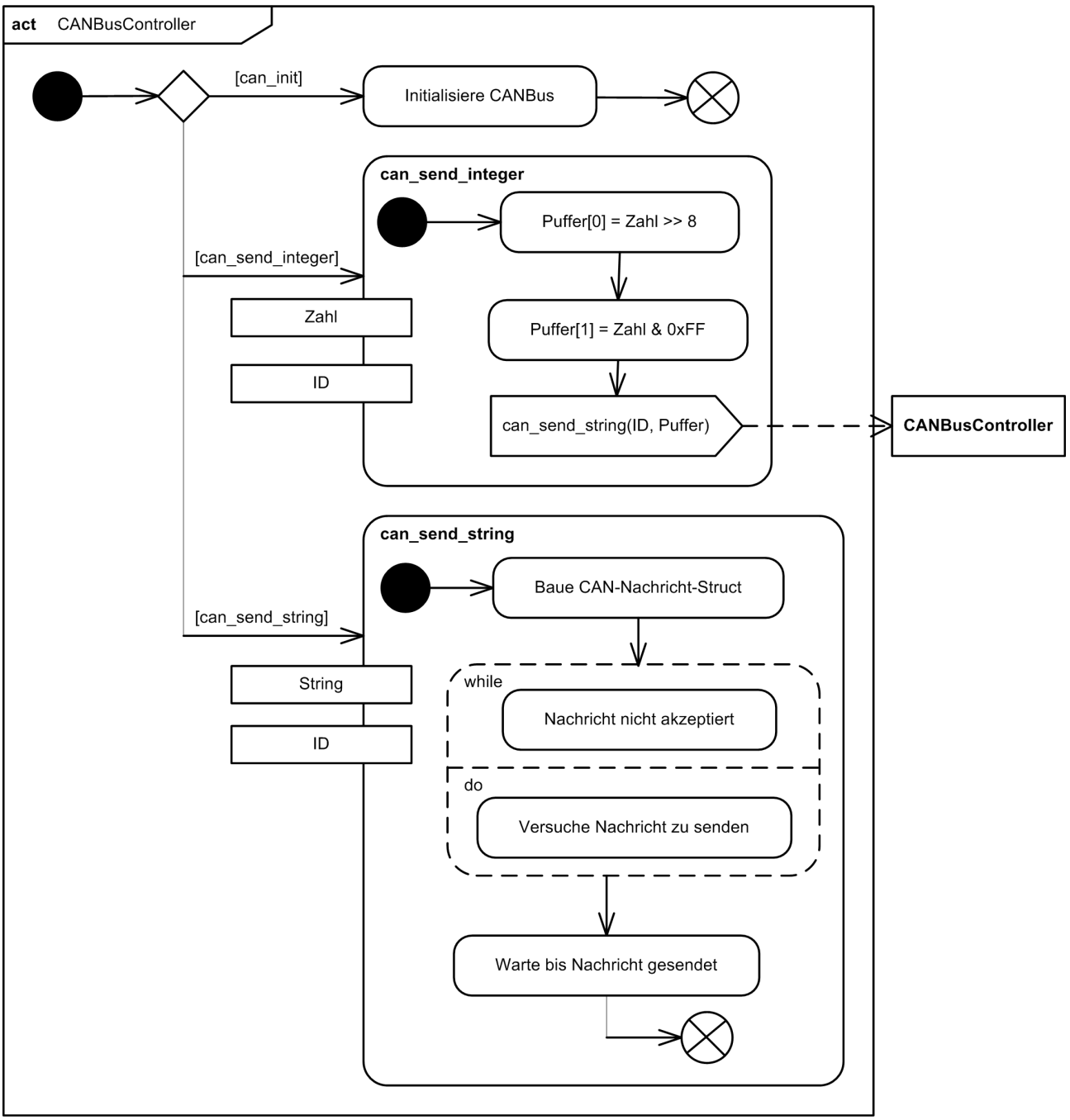


Abbildung 4.23: Aktivitätsdiagramm CANBusController

4.2.5 Kalibrierer

Der *Kalibrierer* ist die für die Kalibrierung der Sensorik zuständige Treiberkomponente.

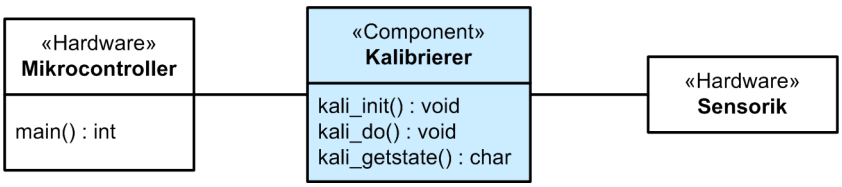


Abbildung 4.24: Spezifikation Kalibrierer

Das Klassendiagramm in Abbildung 4.24 zeigt dessen Funktionen, Tabelle 4.8 beschreibt diese genauer. Der Kalibrierer ist nur während der Initialisierungsphase des Antikollisionssystems aktiv. Er greift über 2 Digitaleingänge auf den aktuellen Kalibrierungsstatus der Sensorik zu und kalibriert diese entsprechend.

Name	Kalibrierer
Beschreibung	• *kali_init:* Initialisiert den Kalibrierer • *kali_do:* Führe Sensorkalibrierung durch • *kali_getstate:* Gib den aktuellen Sensorkalibrierungsstatus zurück
Einschränkungen	Während des Betriebs kann die Sensorik nicht mehr nachjustiert werden.
Empfängt	N/A
Rückgabe	N/A
Sendet	N/A
Liest	Kalibrierungsstatus über 2 Digital-Eingänge
Verändert	Sensorkalibrierung mit Hilfe des PotiTreibers
Regeln	N/A
Vorraussetzungen	Digitalpotentiometer und Eingänge sind initialisiert.
Ergebnis	Der Kalibrierer wurde initialisiert und die Sensorik wurde kalibriert.

Tabelle 4.8: Funktionsspezifikation Kalibrierer

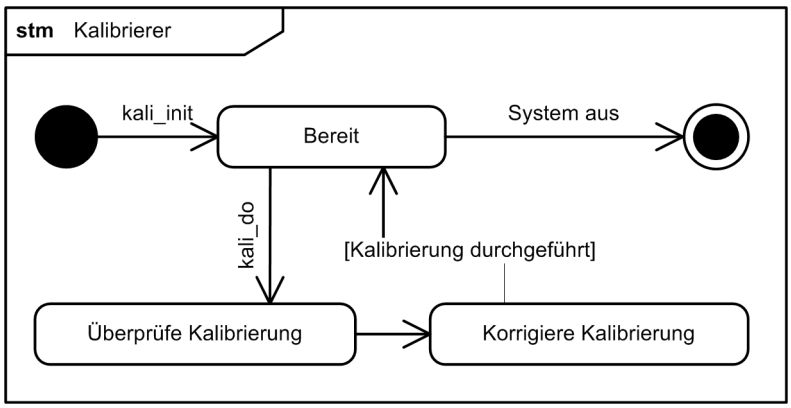


Abbildung 4.25: Zustandsdiagramm Kalibrierer

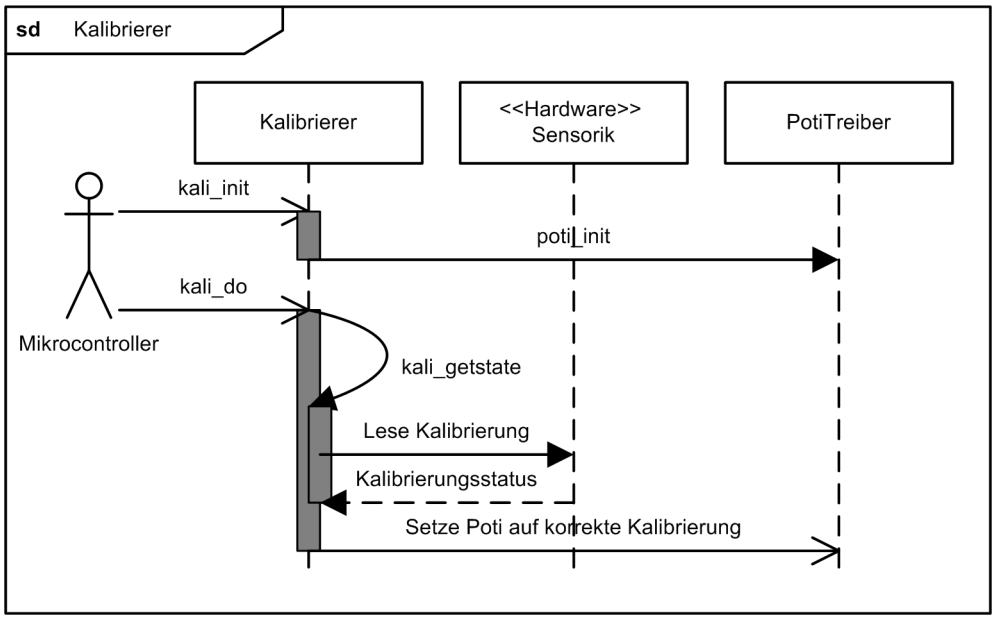


Abbildung 4.26: Sequenzdiagramm Kalibrierer

Die Abbildungen 4.26 und 4.25 zeigen in Form eines Sequenz- bzw. Zustandsdiagramms die Funktionsweise des Kalibrierers. Während der Initialisierungsphase ruft der Mikrocontroller die Funktion *kali_init* auf um den Kalibrierer funktionsbereit zu machen. Nachdem alle Komponenten initialisiert wurden, kalibriert er mit Hilfe der Funktion *kali_do* die Sensorik. Hierfür liest der *Kalibrierer* zunächst den momentan Kalibrierungsstatus aus und führt anschließend eine Korrektur des Digitalpotentiometers mit Hilfe des *PotiTreibers* durch.

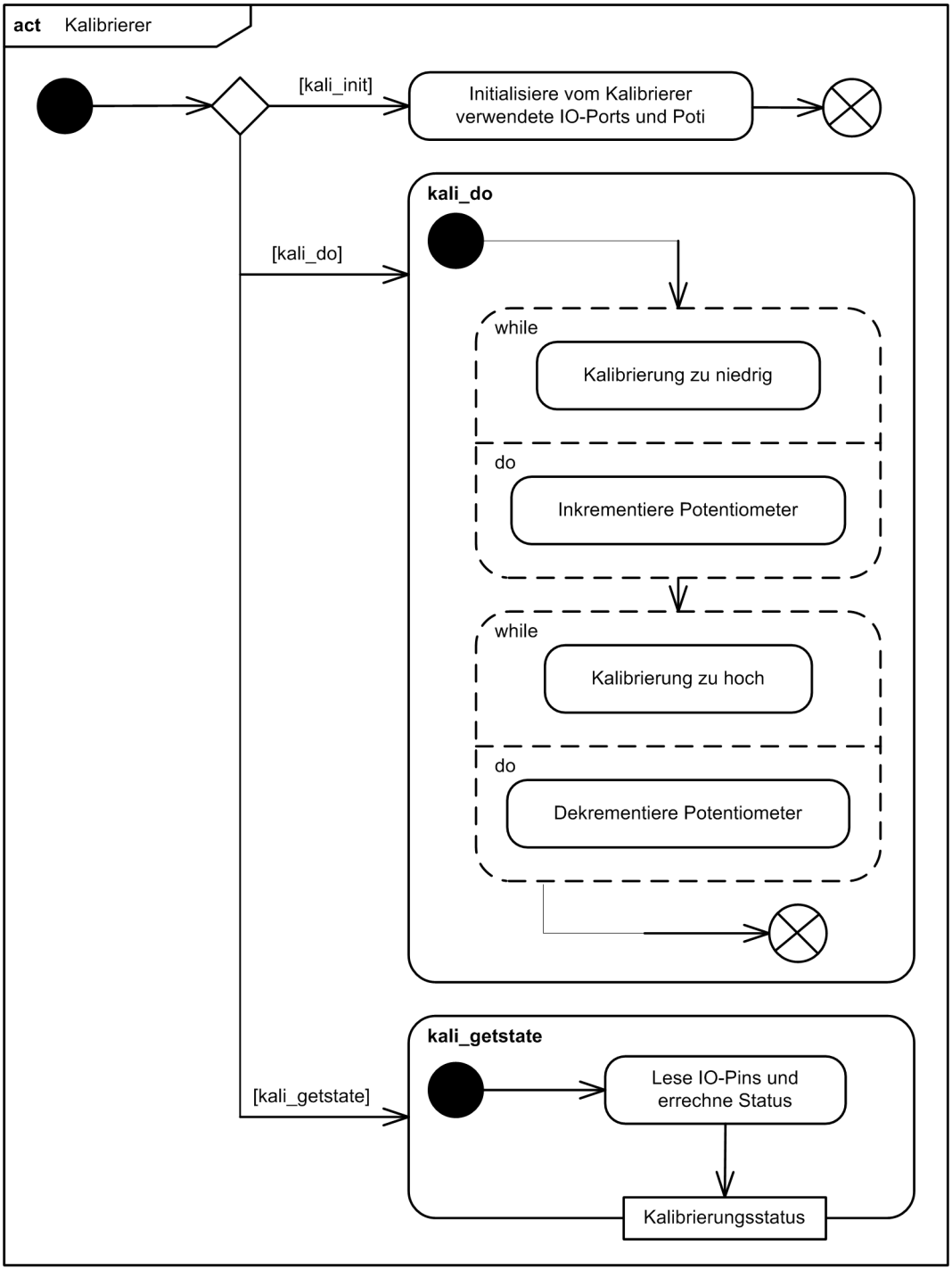


Abbildung 4.27: Aktivitätsdiagramm Kalibrierer

Den detailierten Ablauf zeigt das Aktivitätsdiagramm in Abbildung 4.27. Zunächst wird in einer Schleife der Widerstand des Potentiometers solange inkrementiert, solange die Referenzspannung für den Mittelwertbildner zu niedrig ist. Anschließend wird der gleiche Vorgang noch einmal in die Gegenrichtung durchgeführt.

4.2.6 PotiTreiber

Der *PotiTreiber* ist die Treiberkomponente, die sich um die Ansteuerung des Digitalpotentiometers kümmert. Wie das Klassendiagramm in Abbildung 4.28 zeigt besitzt er drei Funktionen: Eine Initialisierungsfunktion, eine zum Inkrementieren und eine zum Dekrementieren des Widerstandes.

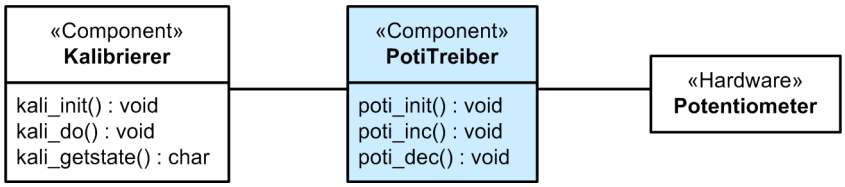


Abbildung 4.28: Spezifikation PotiTreiber

Name	PotiTreiber
Beschreibung	• *poti_init:* Initialisiere das Digitalpotentiometer. • *poti_inc:* Inkrementiere den Wert des Potis. • *poti_dec:* Dekrementiere den Wert des Potis.
Einschränkungen	Der Widerstand kann nicht direkt auf einen bestimmten Wert gesetzt werden.
Empfängt	inkrementieren oder dekrementieren
Rückgabe	N/A
Sendet	inkrementieren/dekrementieren an Potentiometer
Liest	N/A
Verändert	Widerstand des Potentiometers
Regeln	N/A
Vorraussetzungen	Ausgänge zur Ansteuerung des Digitalpotentiometers sind initialisiert.
Ergebnis	Der PotiTreiber wurde initialisiert und der Wert des digitalen Widerstands wurde über die Digitalausgänge inkrementiert, bzw. dekrementiert.

Tabelle 4.9: Funktionsspezifikation PotiTreiber

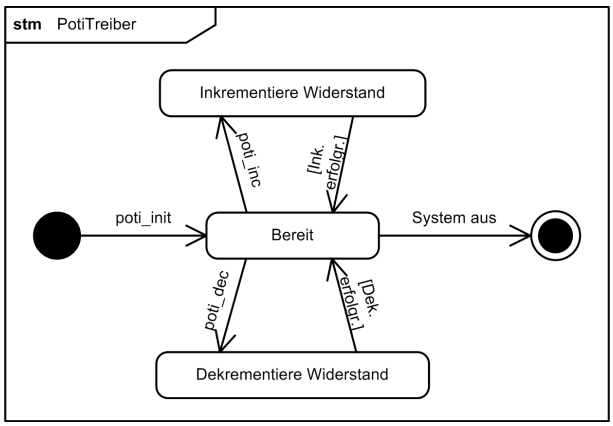


Abbildung 4.29: Zustandsdiagramm PotiTreiber

Die Realisierung des Treibers zeigen die Diagramme in den Abbildungen 4.30 und 4.31.

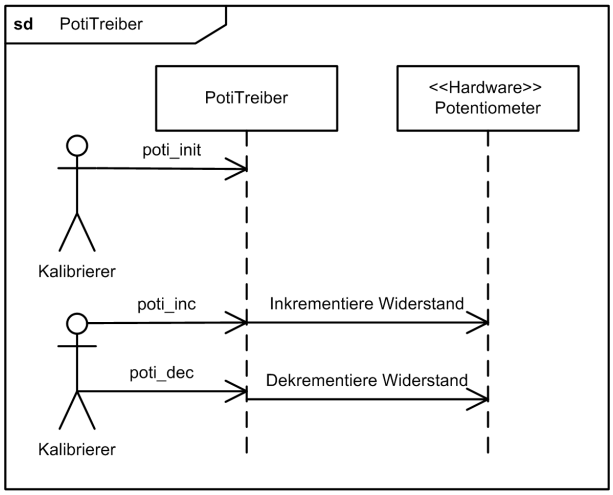


Abbildung 4.30: Sequenzdiagramm PotiTreiber

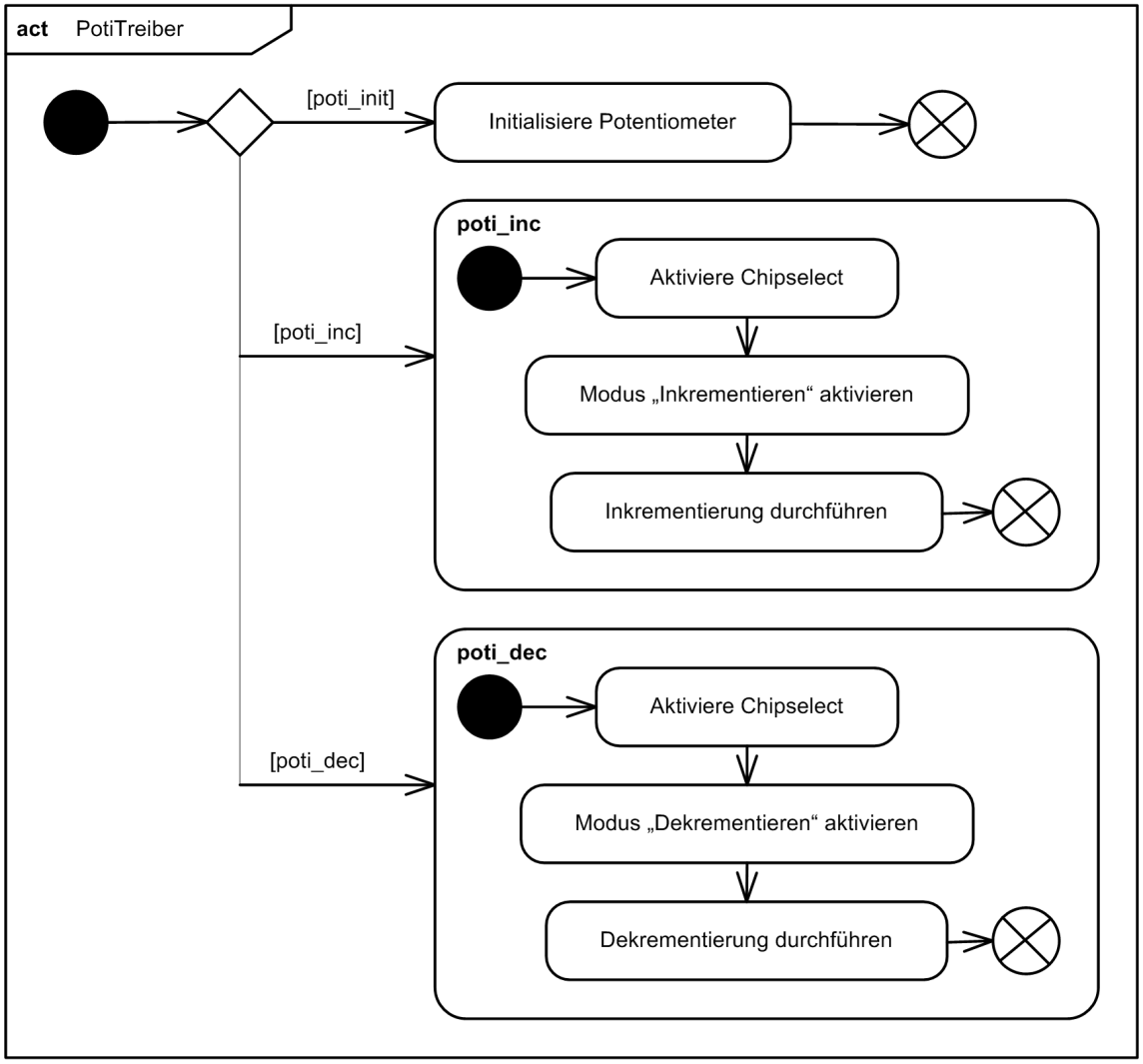


Abbildung 4.31: Aktivitätsdiagramm PotiTreiber

Der Kalibrierer ruft, als einziger auftretender Akteur, zunächst die Initialisierungsfunktion auf. Anschließend kann über die Funktionen *poti_inc* und *poti_dec* der Wert des digial einstellbaren Widerstands inkrementiert bzw. dekrementiert werden.
Die beiden Funktionen zum Einstellen des Widerstands arbeiten dabei ähnlich. Zunächst wird der sogenannte *Chipselect* des Chips aktiviert, d.h. er reagiert nun auf Befehle. Anschließend wird über einen weiteren Digitalausgang der Betriebsmodus (Inkrementieren oder Dekrementieren) festgelegt. Ein Flankenwechsel am dritten Digitalausgang führt den Befehl aus.

5 Code-Modellierung

Nach der Systembeschreibung folgt in MARMOT das sogenannte "Embodiment". Das Ziel dieses Schrittes ist es, zunächst die ursprüngliche Komponentenhierarchie (siehe Abbildung 4.1) auf die Software zu reduzieren und anschließend in Sourcecode zu übersetzen.

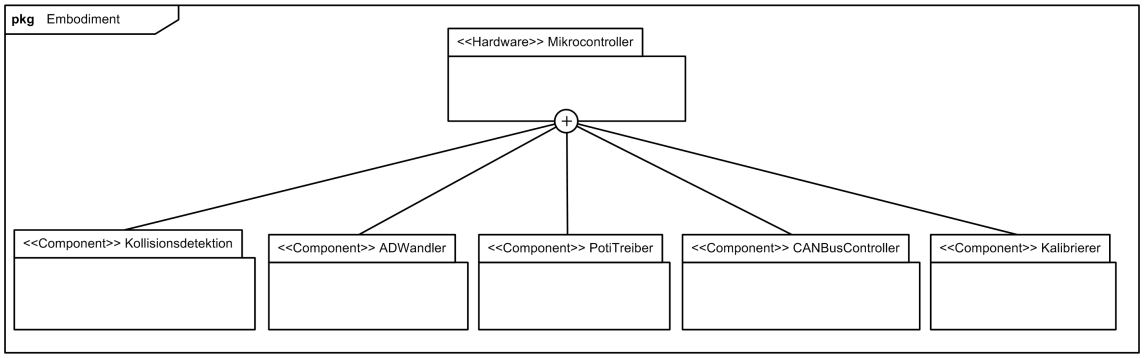


Abbildung 5.1: Paketdiagramm Software-Komponenten

Abbildung 5.1 zeigt die Hierarchie der Software-Komponenten.
Da es sich hierbei um ein in C geschriebenes eingebettetes System handelt, können die im letzten Kapitel beschriebenen Klassen nicht in wirkliche Klassen übersetzt werden. Stattdessen wird jede Komponente durch ein Sourcecode-Paar, c-Datei + Header-Datei, repräsentiert.

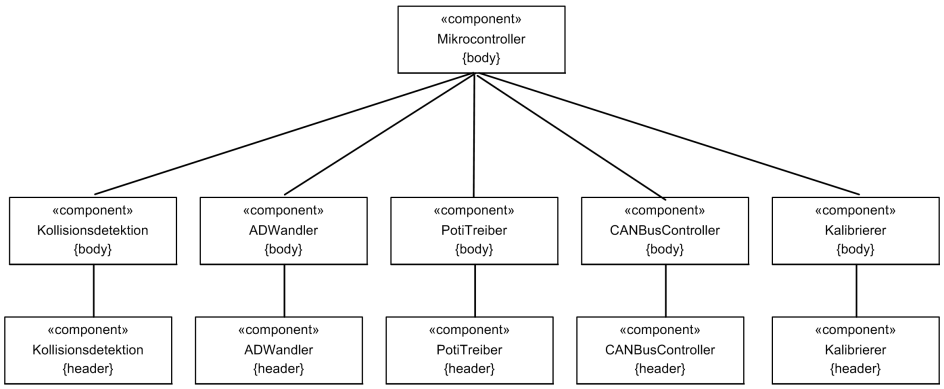


Abbildung 5.2: Code-Struktur

In Abbildung 5.2 wird die resultierende Code-Struktur grafisch dargestellt.

6 Implementierung und Test

6.1 Implementierung

Mit der nun fertigen Code-Struktur konnte die Mikrocontroller-Software implementiert werden. Zunächst wurde für jede in der Systembeschreibung spezifizierte Komponenten-Methode der Rumpf einer C-Funktion geschrieben. Anschließend wurden die Funktionen wie durch die Aktivitäts- und Sequenzdiagramme beschrieben implementiert.
Dieser Vorgang hätte im Groben von einem Codegenerator durchgeführt werden können. Da der Aufwand der automatischen Codegenerierung, vor allen Dingen wegen der Besonderheiten der Mikrocontroller-Plattform, jedoch für diese Projektarbeit verhältensmäßig zu groß gewesen wäre, wurden alle Implementierungen von Hand durchgeführt.

6.2 Test

Die einzelnen Komponenten wurden, soweit möglich, alle zunächst einzeln und am Ende im Zusammenspiel getestet.

6.2.1 Test der Komponenten

Für den Test der Komponenten wurde die *main*-Funktion modifiziert und die Codebasis um eine UART-Komponente zum Debuggen über eine serielle Schnittstelle erweitert. Diese Komponente stammt aus der AVRlib [9] und soll deshalb hier nicht weiter betrachtet werden.

ADWandler

Der *ADWandler* wurde mit Hilfe einer durch ein Labornetzteil erzeugten Referenzspannung getestet. In einer Endlosschleife wurde die aktuelle Spannung ausgelesen und der erhaltene Wert auf der seriellen Schnittstelle ausgegeben. Die Tests verliefen auf Anhieb erfolgreich, d.h. der ADWandler lieferte bei einer Spannung zwischen 0 und 5V linear Werte zwischen 0 und 1024.

PotiTreiber

Zum Test des *PotiTreibers* wurde der Widerstand des Digitalpotentiometers in einer Endlosschleife langsam bis auf dessen Maximum erhöht und anschließend wieder bis auf dessen Minimum dekrementiert. Mit Hilfe eines Multimeters konnte diese Änderung verifiziert werden.

CANBusController

Mit Hilfe eines USB2CAN-Interfaces konnte der *CANBusController* getestet werden. Da dieser nur als Hilfskomponente für die bereits von Atmel vorgegebene CANLib arbeitet, wurden nur grundlegende Tests durchgeführt, d.h. nur überprüft ob CAN-Telegramme versendet werden.

Kalibrierer

Um den *Kalibrierer* zu testen, wurden die LEDs auf der Sensorplatine verwendet. Diese zeigen an, ob der Mittelwertbildner zu hoch oder zu niedrig kalibriert ist. Leuchten beide, ist die Kalibrierung ideal, d.h. der Mittelwertbildner liefert ein Referenzsignal das das RADAR-Grundsignal am Instrumentierungsverstärker auf 0V absenkt. Zusätzlich wurde das Ausgangssignal am Instrumentierungsverstärker mit dem Oszilloskop gemessen.

Kollisionsdetektion

Da die *Kollisionsdetektion* die Hauptfunktionalität des Systems darstellt, wird deren Test im folgenden Abschnitt beschrieben.

6.2.2 Test des Gesamtsystems

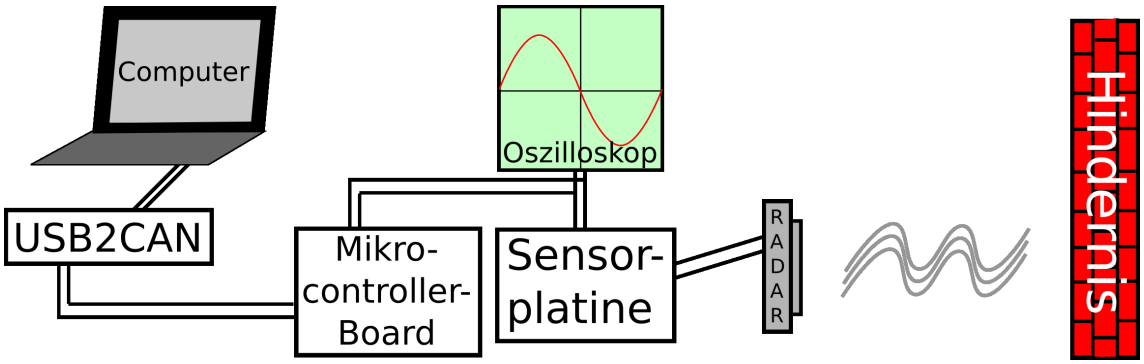


Abbildung 6.1: Versuchsaufbau Test Gesamtsystem

Um das Gesamtsystem, und damit vor allen Dingen auch die Kollisionsdetektion, zu testen, wurde der in Abbildung 6.1 dargestellte Versuchsaufbau verwendet. An die Sensorplatine ist, neben dem Mikrocontroller, auch noch ein Oszilloskop angeschlossen um das RADAR-Signal kontrollieren zu können. Das im letzten Abschnitt bereits erwähnte USB2CAN-Interface wird wieder verwendet um die CAN-Telegramme vom CAN-Bus zu lesen. Da der RADAR-Sensor nur sich zu ihm relativ bewegende Objekte registriert und er im Versuchsaufbau fest montiert ist, wird zum Testen das Hindernis bewegt.
Wie bereits erwartet wurden metallische Objekte am Besten, d.h. auf größte Reichweite, erkannt. Eine Metallplatte konnte auf 2 Meter Entfernung erfolgreich erkannt werden.

Als völlig ungeeignet haben sich Schaumstoffe und Pappe erwiesen; sie reflektieren die RADAR-Wellen nicht.

6.3 Migration auf eine andere Plattform

Im Laufe der Projektarbeit wurde mit einer Neukonzeptionierung des ConceptCars begonnen. Die gesamte Elektronik wurde aufgegeben und soll nun neu entwickelt werden. Diese Umbaumaßnahmen verhinderten, dass das Antikollisionssystem auf der eigentlichen Zielplattform, dem ConceptCar, getestet werden konnte. In einem weiteren Projekt wird am Fraunhofer IESE ein Gabelstabler im Modellformat entwickelt, der verschiedene Aufgaben autonom durchführen soll. Aktuell kann er über eine Bluetoothverbindung mit Hilfe eines PC-Lenkrads ferngesteuert werden. Zum Testen des Antikollisionssystems modifizierte ich es so, dass man es als Fahrassistenzssystem für den Gabelstabler nutzen kann. Als einzige Veränderung tauschte ich hierzu die CAN-Komponente durch ein "1-Pin-Warn-System" aus. Statt Kollisionswarnungen als CAN-Telegramm zu versenden, wird nun ein Flankenwechsel an einem Digitalausgang durchgeführt. Dieser Digitalausgang ist mit einem interruptgesteuerten Digitaleingang des Gabelstabler-Steuersystems verbunden. Sobald ein Flankenwechsel auf dem Eingang ankommt, wird eine Interrupttroutine aufgerufen, die den Gabelstapler zum Stehen bringt. Abbildung 6.2 zeigt ein Bild der neue Zielplattform mit montiertem Antikollisionssystem.

Abbildung 6.2: Gabelstapler mit Antikollisionssystem

7 Installation

7.1 Hardwareinstallation

Die für den Betrieb notwendige Versorgungsspannung von 12V Gleichstrom wird, wie in Abbildung 3.3 zu sehen, an der Mikrocontrollerplatine am zweipoligen Steckverbinder SV1 angelegt. Mikrocontroller- und Sensorplatine werden über ein 10-poliges Flachbandkabel an SV2 bzw. SV4 (siehe Abbildungen 3.3 und 3.5) miteinander verbunden. Der RADAR-Sensor wird an den Anschluss ST1 der Sensorplatine angeschlossen.
Die genauen Pinbelegungen sind auf den erwähnten Schaltplänen zu sehen.

7.2 Softwareinstallation

Die Mikrocontroller-Software wird mit Hilfe des avr-gcc und "make" kompiliert und anschließend mit avrdude über "make program" in den Flash-Speicher des AT90CAN128 geschrieben. Hierfür wird ein STK200-kompatibler Programmieradapter wie er auf [10] beschrieben ist benötigt. Dieser wird an den ICSP-Anschluss der Olimex-Platine und den Parallelport des Computers angeschlossen. Im Makefile muss evtl. die Adresse des benutzten Parallelports angepasst werden (z.B. *lpt1* unter Windows statt */dev/parport0* unter Linux).

7.3 Fahrzeugsteuerung

Die Fahrzeugsteuerung muss auf CAN-Telegramme mit der Adresse 0x139 reagieren. Empfängt sie solche, sind Hindernisse in Reichweite und das Fahrzeug sollte sofort den Motor ausschalten und bremsen.

8 Ergebnis

8.1 Fazit

Diese Projektarbeit hat gezeigt, das es möglich ist ein komponentenbasiertes Antikollisionssystem für ein Kraftfahrzeug zu entwickeln. Die dabei entwickelten Komponenten wie z.B. der ADWandler oder der CANBusController können mühelos auch für andere Entwicklungen wiederverwendet werden. Bereits vorhandene Komponenten wie die Atmel® CANLib wurden integriert und somit wertvolle Entwicklungszeit eingespart.

Leider konnte aufgrund des günstigen Sensors kein sehr gutes Ergebnis bei der Kollisionserkennung erreicht werden. Objekte die sich weiter als 2m vor dem Fahrzeug befinden können nicht mehr einwandfrei erfasst werden. Für eine reale Lösung in einem Kraftfahrzeug benötigt man deshalb weitaus teurere Sensorik.

Die gezwungene Migration auf den Gabelstapler zeigt, dass es dank des komponentenbasierten Ansatzes sehr einfach war das System für eine neue Zielplattform umzubauen. Der Tausch einer einzigen Komponente verbunden mit minimalen Hardwareänderungen ermöglichten es, in einer völlig neuen Umgebung zu arbeiten.

8.2 Ausblick

Fahrassistenzssysteme für Kraftfahrzeuge, zu denen auch dieses Antikollisionssystem gehört, werden eine große Rolle im Auto der Zukunft spielen. Durch intelligente Sensorik können Unfälle früh vorhergesehen, und der Fahrer rechtzeitig gewarnt werden. Die steigende Rechenleistung moderner Computer ermöglicht es, Hindernisse über vielfältige Art und Weise zu detektieren. Neben einfachen RADAR-Systemen werden in Zukunft auch kameragestützte Stereo-Bilderkennungssysteme eine große Rolle spielen. Diese erstellen ein 3D-Modell der Umgebung und können so nicht nur Hindernisse effektiv erkennen, sondern auch das Fahrverhalten anderer Fahrzeuge analysieren und so Unfällen effektiv vorbeugen.

9 Anhang

9.1 CD-Rom

- Quellcode der Mikrocontrollerapplikation, kompilierbar mit avr-gcc
- Quellcode des während dieser Arbeit entwickelten Software-Oszilloskops
- Dieses Dokument als PDF-Datei

9.2 Quellcode der Mikrocontrollerapplikation

9.2.1 Mikrocontroller

Listing 9.1: main.c

```
1  #include <avr/io.h>
2  #include <adc.h>
3  //#include <can.h>
4  #include <parwarn.h>
5  #include <poti.h>
6  #include <kali.h>
7  #include <coldet.h>
8  #include <stdint.h>
9  #include <stdio.h>
10 #include <defines.h>
11 #include <ctype.h>
12 #include <util/delay.h>
13
14
15 /**
16  * main
17  *
18  * main-function - called on poweron/reset
19  */
20 int main(void)
21 {
22     uint16_t colstate;
23
24     // initialize all components
```

```
25      coldet_init();
26      //can_init2();
27      parwarn_init();
28      kali_init();
29
30      // calibrate the sensor
31      kali_do();
32
33      // infinite loop
34      for (;;)
35      {
36          // get collision state and send a warning if needed
37          colstate = coldet_getstate();
38          if (colstate > 0)
39          {
40  //          can_send_integer(0x139, colstate);
41              parwarn_send();
42          }
43      }
44
45      return 0;
46  }
```

9.2.2 Kollisionsdetektion

Listing 9.2: coldet.h

```
1  #ifndef _COLDET_H_
2  #define _COLDET_H_
3
4  #include <avr/io.h>
5
6  #define NOCOLLISION 0
7  uint16_t coldet_getstate(void);
8  void coldet_init(void);
9
10 #endif
```

Listing 9.3: coldet.c

```
1  #include <coldet.h>
2  #include <adc.h>
3
4  /**
```

```
 * coldet_init
 *
 * initializes all components needed by the collision detection
 */
void coldet_init(void)
{
    // initialize adc
    adc_init();
}

/**
 * coldet_getstate
 *
 * calculates and returns the current state of the collision detection
 *
 * @returns collision state
 */
uint16_t coldet_getstate(void)
{
    uint16_t value;

    // read current data from adc
    value = adc_read();

    // if value is higher than 25, we detected something in front of us
    // else, return "no collision"
    if (value > 25)
        return value;
    else
        return NOCOLLISION;
}
```

9.2.3 ADWandler

Listing 9.4: adc.h

```
#ifndef _ADC_H_
#define _ADC_H_

void adc_init(void);
uint16_t adc_read(void);

#endif
```

Listing 9.5: adc.c

```
#include <avr/io.h>
#include <adc.h>

/**
 * adc_init
 *
 * initializes the analog/digital-converter
 */
void adc_init(void)
{
    uint8_t i;
    ADCSRA = (1<<ADEN) | (1<<ADPS2) | (1<<ADPS1) | (1<<ADPS0);
    ADMUX = 0;
    ADMUX |= (1<<REFS0);

    ADCSRA |= (1<<ADSC);

    for (i=0; i<5; i++)
        while (ADCSRA & (1<<ADSC));
}

/**
 * adc_read
 *
 * does three conversions and returns the average as uint16
 *
 * @returns converted analog-data
 */
uint16_t adc_read(void)
{
    uint8_t i;
    uint16_t result = 0;

    for (i=0; i<3; i++)
    {
        ADCSRA |= (1<<ADSC);
        while (ADCSRA & (1<<ADSC));

        result += ADCW;
    }

    result /= 3;

    return result;
}
```

9.2.4 CANBusController

Listing 9.6: can.h

```
#ifndef _CAN_H_
#define _CAN_H_

#include <can_lib.h>
#include <config.h>

void can_init2(void);
void can_send_integer(unsigned int std_id, uint16_t data);
void can_send_string(unsigned int std_id, unsigned char std_dlc,
        unsigned char * buffer);

#endif
```

Listing 9.7: can.c

```
#include <can.h>

/**
 * can_init2
 *
 * initialize the can bus driver
 */
void can_init2(void)
{
    CLKPR = CONF_CLKPR_RST;
    CLKPR = CONF_CLKPR;
    can_init();
}

/**
 * can_send_integer
 * @param std_id can message id
 * @param data data to send
 *
 * sends an integer value as a can telegram
 */
void can_send_integer(unsigned int std_id, uint16_t data)
{
    unsigned char buffer[2];
```

```
25
26      buffer[0] = (data >> 8);
27      buffer[1] = (data & 0xFF);
28
29      can_send_string(std_id, 2, buffer);
30  }
31
32  /**
33   * can_send_string
34   * @param std_id can message id
35   * @param std_dlc buffer character count
36   * @param buffer pointer to a character buffer containing the can
37   * message
38   *
39   * sends a defined count of characters of a buffer as a can telegram
40   */
41  void can_send_string(unsigned int std_id, unsigned char std_dlc,
42          unsigned char * buffer)
43  {
44      st_cmd_t msg;
45
46      msg.pt_data = &buffer[0];
47      msg.status = 0;
48      msg.handle = 0;
49      msg.dlc = std_dlc;
50      msg.id.std = std_id;
51
52      msg.cmd = CMD_TX;
53      while(can_cmd(&msg) != CAN_CMD_ACCEPTED);
54      while(can_getstatus(&msg) != CAN_STATUS_COMPLETED);
55  }
```

9.2.5 Kalibrierer

Listing 9.8: kali.h

```
1  #ifndef _KALI_H_
2  #define _KALI_H_
3
4  #include <avr/io.h>
5
6  #define PORT_KALI    PORTB
7  #define PIN_KALI     PINB
8  #define PINNUM_KALI1     5
9  #define PINNUM_KALI2     6
```

```
10 #define DDR_KALI    DDRB
11 #define PIN_KALI1   PINB5
12 #define PIN_KALI2   PINB6
13
14 void kali_init(void);
15 void kali_do(void);
16 char kali_getstate(void);
17
18 #endif
```

Listing 9.9: kali.c

```
1  #include <kali.h>
2  #include <poti.h>
3  #include <stdio.h>
4
5  #include <util/delay.h>
6
7  /**
8   * kali_init
9   *
10  * initializes the "kalibrator", a calibration unit for the radar
11  * sensor.
12  */
13 void kali_init(void)
14 {
15     // set pins as inputs
16     DDR_KALI &= ~(1 << PINNUM_KALI1);
17     DDR_KALI &= ~(1 << PINNUM_KALI2);
18
19     // initialize potentiometer
20     poti_init();
21 }
22
23 /**
24  * kali_do
25  *
26  * do one calibration
27  */
28 void kali_do(void)
29 {
30     while ((PIN_KALI & (1<<PIN_KALI2)))
31     {
32         // decrement potentiometer
33         poti_dec();
```

```
34          _delay_ms(10.0);
35      }
36
37      while ((PIN_KALI & (1<<PIN_KALI1)))
38      {
39          // increment potentiometer
40          poti_inc();
41          _delay_ms(10.0);
42      }
43
44      // needed for better results
45      poti_dec();
46  }
47
48  /**
49   * kali_getstate
50   *
51   * @returns the current calibration status as a 2 byte value
52   */
53  char kali_getstate()
54  {
55      char status;
56
57      status = ((PIN_KALI & (1<<PIN_KALI2)) << 1)
58              | ((PIN_KALI & (1<<PIN_KALI1)) << 0);
59
60      return status;
61  }
```

9.2.6 PotiTreiber

Listing 9.10: poti.h

```
1  #ifndef _POTI_H_
2  #define _POTI_H_
3
4  #include <avr/io.h>
5
6  #define PORT_POTI        PORTB
7  #define PINNUM_POTICS    0
8  #define PINNUM_POTIMODE  3
9  #define PINNUM_POTICMD   4
10 #define DDR_POTI         DDRB
11
12 void poti_init(void);
```

```
13 void poti_inc(void);
14 void poti_dec(void);
15
16 #endif
```

Listing 9.11: poti.c

```
1  #include <poti.h>
2  #define nop() __asm volatile ("nop")
3
4  /**
5   * poti_init
6   *
7   * initializes the digital I/O-Pins used by the digital
8   * potentimeter by setting them as outputs
9   */
10 void poti_init(void)
11 {
12     // set pins as output
13     DDR_POTI |= (1 << PINNUM_POTICS) | (1 << PINNUM_POTIMODE)
14                 | (1 << PINNUM_POTICMD);
15 }
16
17 /**
18  * poti_inc
19  *
20  * increments the potentiometer's resistance by one step
21  */
22 void poti_inc(void)
23 {
24     // activate chipselect (active low)
25     PORT_POTI &= ~(1 << PINNUM_POTICS);
26     // set increment-mode
27     PORT_POTI |= (1 << PINNUM_POTIMODE);
28     // do an incrementation
29     PORT_POTI |= (1 << PINNUM_POTICMD);
30     nop();
31     PORT_POTI &= ~(1 << PINNUM_POTICMD);
32     // deactivate chipselect
33     PORT_POTI |= (1 << PINNUM_POTICS);
34 }
35
36 /**
37  * poti_dec
38  *
```

```
39  * decrements the potentiometer's resistance by one step
40  */
41 void poti_dec(void)
42 {
43     // activate chipselect (active low)
44     PORT_POTI &= ~(1 << PINNUM_POTICS);
45     // set decrement-mode
46     PORT_POTI &= ~(1 << PINNUM_POTIMODE);
47     // do a decrementation
48     PORT_POTI |= (1 << PINNUM_POTICMD);
49     nop();
50     PORT_POTI &= ~(1 << PINNUM_POTICMD);
51     // deactivate chipselect
52     PORT_POTI |= (1 << PINNUM_POTICS);
53 }
```

9.2.7 ParWarn

Listing 9.12: parwarn.h

```
1  #ifndef _PARWARN_H_
2  #define _PARWARN_H_
3
4  #include <avr/io.h>
5
6  #define PORT_PARWARN    PORTB
7  #define PORTNUM_PARWARN PORTB1
8  #define DDR_PARWARN     DDRB
9
10 void parwarn_send(void);
11 void parwarn_init(void);
12
13 #endif
```

Listing 9.13: parwarn.c

```
1  #include <parwarn.h>
2  #include <util/delay.h>
3
4
5  /**
6   * parwarn_send
7   *
8   * sends a collision warning to an other microcontroller by sending
```

```
 * a low to high transition
 */
void parwarn_send(void)
{
    char i;

    // enable pin
    PORT_PARWARN |= (1 << PORTNUM_PARWARN);

    // wait 150 ms
    for (i=0; i<10; i++)
        _delay_ms(15);

    // disable pin
    PORT_PARWARN &= ~(1 << PORTNUM_PARWARN);
}

/**
 * parwarn_init
 *
 * initializes the parallel collision warning.
 * Set the used pin as output and disable it
 */
void parwarn_init(void)
{
    // pin is an output
    DDR_PARWARN |= (1 << PORTNUM_PARWARN);
    // disable pin
    PORT_PARWARN &= ~(1 << PORTNUM_PARWARN);
}
```

Literaturverzeichnis

[1] *http://www.destatis.de.*

[2] *http://www.bosch-acc.de.*

[3] *http://www.uml.org.*

[4] *http://www.atmel.com/products/CAN/.*

[5] *http://www.olimex.com.*

[6] *http://www.sprut.de/electronic/switch/minus.html.*

[7] *http://www.matwei.de.*

[8] *http://www.povray.org.*

[9] *http://procyonengineering.com/avr/avrlib/.*

[10] *http://rumil.de/hardware/avrisp.html.*

[11] *http://de.wikipedia.org.*

[12] Colin Atkinson, Joachim Bayer, Christian Bunse, Erik Kamsties, Oliver Laitenberger, Roland Laqua, Dirk Muthig, Barbara Paech, Jürgen Wüst, and Jörg Zettel. *Component-based product line engineering with UML.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[13] Frank Böhr. *Anwendung und Evaluierung verhaltensbasierter Ansätze zur Entwicklung verlässlicher Automotivesysteme.* 2006.

Glossar

ASR *Antischlupfregelung* – sorgt bei einem Kraftfahrzeug dafür, dass die Räder beim Beschleunigen nicht durchdrehen. Sie ist auch als Traktionskontrolle aus der Formel 1 bekannt.

CAN-Bus *Controller Area Network* – ist ein asynchrones, serielles Bussystem, das 1983 von Bosch für die Vernetzung von Steuergeräten im Automobil entwickelt und 1985 zusammen mit Intel vorgestellt wurde [11].

ESP *Elektronisches Stabilitätsprogramm* – ist ein von Bosch entwickeltes System um in einem KFZ einzelne Räder gezielt abbremsen zu können.

LIDAR *Light Detection and Ranging* – ist ein dem RADAR ähnliches Ortungsverfahren, dass statt mit elektromagnetischen Wellen mit Laserstrahlen arbeitet.

RADAR *Radio Detection and Ranging* – ist ein Ortungsverfahren das eine elektromagnetische Welle im Mikrowellenband aussendet, das von der Umwelt reflektierte Signal ("Echo") empfängt und auswertet.

Schmitt-Trigger – ist ein Schwellwertschalter, der sobald eine bestimmte Spannung an seinem Eingang überschritten wird einen High-Pregel an den Ausgang legt und sobald eine zweite vordefinierte Spannung unterschritten wird auf einen Low-Pegel schaltet.