Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Masterarbeit

Heiko Wilken

Laserscanner-basierte Objekterkennung für ein Fahrspurführungssystem auf einer Multiprozessor FPGA-Plattform

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Heiko Wilken

Laserscanner-basierte Objekterkennung für ein Fahrspurführungssystem auf einer Multiprozessor FPGA-Plattform

Masterarbeit eingereicht im Rahmen der Masterprüfung
im Studiengang Master Informatik
Department Informatik
in der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr.-Ing. Bernd Schwarz
Zweitgutachter : Prof. Dr. rer. nat. Reinhard Baran

Abgegeben am 27. Januar 2012

Heiko Wilken

Thema der Ausarbeitung

Laserscanner-basierte Objekterkennung für ein Fahrspurführungssystem auf einer Multiprozessor FPGA-Plattform

Stichworte

MPSoC, SoC, FPGA, RTOS, FAUST, LIDAR, MicroBlaze, Xilkernel

Kurzzusammenfassung

Diese Masterarbeit beschreibt die Konfiguration eines Dual-MicroBlaze Systems mit einer Shared-Memory Architektur und die Implementierung eines Software-Interfaces zur Laserscanner-basierten Objekterkennung. Das Ziel der System-on-Chip Entwicklung ist es, Steuerungs- und Regelungsfunktionen in ein Mikrocontrollersystem zu integrieren, wobei die Threads eines Echtzeitbetriebssystems (RTOS) mit parallelen Hardware-Beschleunigermodulen kombiniert werden und somit algorithmische Aufgaben auch bei niedrigen Taktfrequenzen berechnet werden können. Das Software-Interface empfängt die Abstandswerte vom Laserscanner aus einem 90° Scanbereich mit einer Winkelauflösung von 1,05°. Eine nachgelagerte Segmentierung führt mit einem Schwellwertverfahren ein Clustering der Messwerte durch und generiert somit ein Abbild der Umgebung. Die Messdatenerfassung und die Segmentierung werden in ein POSIX Thread Modell mit einem prioritätenbasierten Scheduling integriert.

Heiko Wilken

Title of the paper

Laser scanner-based object detection for a lane guidance system on a multiprocessor FPGA platform

Keywords

MPSoC, SoC, FPGA, RTOS, FAUST, LIDAR, MicroBlaze, Xilkernel

Abstract

This thesis describes the configuration of a Dual-MicroBlaze system with a shared memory architecture and the implementation of a software interface for the laser scanner based object recognition. The goal of system-on-chip development is the integration of control and regulation functions into a microcontroller system, where the threads of a real-time operating system (RTOS) will be combined with parallel hardware accelerator modules and thus algorithmic tasks can also be calculated at low frequencies. The software interface receives the distance values from the laser scanner from a 90 degree scan range with an angular resolution of 1.05 degree. A downstream segmentation executes the clustering of the measured values with a thresholding procedure and so an image of the environment will be generated. The data acquisition and the segmentation are integrated into a POSIX thread model.

Inhaltsverzeichnis

1. Einleitung

Bereits 1965 wurde durch das Mooresches Gesetzt prognostiziert, dass sich die Anzahl der Transistoren auf einem Chip alle zwei Jahre verdoppelt [50]. Jedoch erfordert der stetig steigende Bedarf an Rechenleistung ein Umdenken im Systementwurf. Die Entwicklung von eingebetteten Systemen wird durch die wachsende Systemkomplexität und den steigenden Anforderungen gekennzeichnet. Eine Erhöhung der Taktfrequenz bei Singlecores zur Erzielung von Durchsatzsteigerungen ist aufgrund von Stromverbrauch und Wärmeabgabe nicht realisierbar. Des Weiteren steigen die Speicherzugriffszeiten nicht im gleichen Umfang wie die CPU Geschwindigkeiten und somit entsteht ohne eine effiziente Cache-Architektur ein Flaschenhals beim Speicherzugriff [57]. Abb. 1.1 zeigt die heutige Lage, nämlich eine gleichbleibende Taktfrequenz trotz steigender Transistoranzahl. Um den leistungsbezogenen Anforderungen von heutigen Anwendungen gerecht zu werden, werden Multiprozessorsysteme eingesetzt. Vor allem in den Bereichen der multimedialen Unterhaltungselektronik und der Automobilindustrie werden Multiprozessoren für Echtzeitsysteme zur Verarbeitung von hohen Datenmengen integriert.

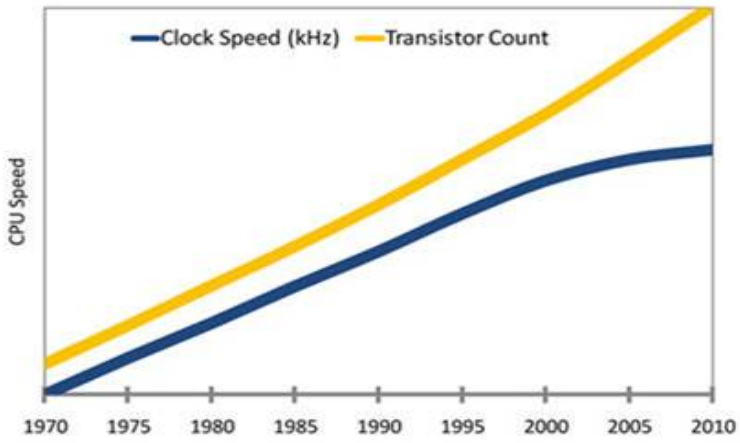


Abb. 1.1.: Mooresches Gesetz im Vergleich zu heutigen Taktfrequenzen [36]

Durch den Einsatz von FPGA-basierten „Multiprocessor System-on-Chips (MPSoC)“ wird ein erhöhter Systemdurchsatz erreicht, indem verschiedene Programme parallel auf mehreren Prozessoren ausgeführt werden. Die FPGA Technologie nutzt durch die getrennte Konfiguration der Hardwarebeschleunigermodule eine Parallelität auf Hardwareebene und bietet dadurch eine hohe Flexibilität. Das Ziel der System-on-Chip Entwicklung ist es, Steuerungs- und Regelungsfunktionen in ein Mikrocontrollersystem zu integrieren, wobei die Threads eines Echtzeitbetriebssystems (RTOS) mit parallelen Hardware-Beschleunigermodulen kombiniert werden. Bei FPGA-basierten SoCs werden durch die Logik- und Arithmetikelemente sowie die internen Speichermodulen, Ressourcen zur Verfügung gestellt, die algorithmische Aufgaben auch bei niedrigen Taktfrequenzen berechnen [28].

Beim „HW/SW-Codesign“ werden ressourcenlastige Softwareteile extrahiert und in spezielle Hardwarestrukturen implementiert. Mit der Hinzunahme von zusätzlicher Hardware wird der Durchsatz und damit die Beschleunigung des Gesamtsystems gesteigert [44]. Die HW/SW Partitionierung ist bei der Implementierung von Multiprozessorsystemen von wichtiger Bedeutung. Durch die Auslagerung von sequentiell ablaufenden Befehlsfolgen auf verschiedene Prozessoren werden Aufgaben parallel ausgeführt. Die gleichzeitige Reduzierung der Prozessorauslastung minimiert den Energiebedarf des Systems, wobei die Leistungssteigerung von Multiprozessorsystemen hierbei von der Parallelisierbarkeit der Software abhängt [61]. Dies wird durch das Amdahlsche Gesetz, welches aussagt, dass ein Programm nie vollständig parallel ausgeführt werden kann, da einige Teile, wie Initialisierung und Speicher Allokation stets sequentiell ausgeführt werden müssen, bestätigt [1]. Der zusätzlich entstehende Overhead muss durch eine Durchsatzsteigerung kompensiert werden, was bei parallelen oder nebenläufigen Programmen durch eine Auslagerung der Aufgaben in Threads erreicht wird. Der Einsatz von parallelen Multiprozessoranwendungen eignet sich nur für Problematiken, die in einem sequentiellen Programm wenige Reihenfolge- und Datenabhängigkeiten aufweisen, da nicht skalierbare und stark abhängige Programmsequenzen nicht parallelisierbar sind [8].

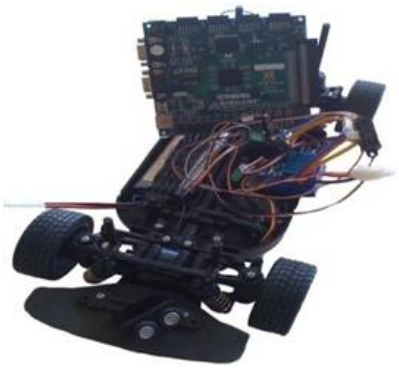


Abb. 1.2.: FAUST SoC-Fahrzeug

Abb. 1.3.: Sensor Controlled Vehicle

In dieser Arbeit wird die Entwicklung einer Laserscanner-basierten Objekterkennung und deren Integration in ein FPGA-basiertes Multiprozessorsystem vorgestellt. Das aus zwei MicroBlaze Softcore bestehende MPSoC wird als Fahrzeugsteuerungsplattform in einem autonomen SoC-Fahrzeug eingesetzt (vgl. Abb. 1.4). Der erste Prozessor dient als Controller und übernimmt die Steuerung und Koordinierung des Fahrzeugs. Hierzu zählen Aufgaben wie die Parametrierung der Fahrspurführung und der Geschwindigkeitsregelung, sowie die Initiierung der Objekterkennung. Durch die asymmetrische Prozessorarchitektur führen beide Prozessoren eigenständige Software-Module aus und kommunizieren über eine direkte FSL Interprozessorkommunikation. Der zweite MicroBlaze ist für die Erfassung der Laserscanner Messdaten und die nachgelagerte Objekterkennung zuständig, die durch ein Kommando vom ersten MicroBlaze gestartet wird. Eine Kopplung beider Prozessoren mit dem „Multi-Port Memory Controller“ gewährleistet den Zugriff auf gemeinsame Speicherressourcen, die durch einen Mutex synchronisiert werden. Die vorliegende Arbeit erläutert sowohl die Konfiguration des Dual-MicroBlaze Systems als auch die Implementierung des Sensorik Interfaces zur Erfassung von Abstandswerten. Die Entwicklung der auf die Abstandswerte angewandte Segmentierung basiert auf der Objekterkennung des SCV-Fahrzeugs, die auf einem GEME-2000 Industrierechner unter einem QNX Echtzeitbetriebssystem ausgeführt wird [62].

1.1. Zielsetzung

Im Rahmen dieser Masterarbeit wurde eine Laserscanner-basierte Objekterkennung auf einer Multiprozessor FPGA-Plattform entwickelt. Die Arbeit soll zum Aufbau einer MPSoC-basierten Fahrzeugsteuerungsplattform beitragen, die sukzessive zu einem Fahrspurführungs- und Kollisionsvermeidungssystem für das SoC-Fahrzeug weiterentwickelt wird. Zum ersten Einsatz kommt die Laserscanner-basierte Objekterkennung bei der Kollisionsvermeidung in einem automatischen Ausweichassistenten. Mit dem Hauptziel, die System-on-Chip Technologie für eingebettete Systeme und das HW/SW-Codesign zu durchdringen, wurde eine Dual-MicroBlaze Konfiguration mit einer Shared Memory Architektur entwickelt (vgl. Abb. 1.4). Die auf einem Spartan-3A DSP FPGA implementierte MPSoC-Plattform besteht aus zwei MicroBlazes, die eine direkte FSL Verbindung zur Interprozessorkommunikation nutzen. Durch den „Multi-Port Memory Controller“ werden beide Prozessoren mit dem gemeinsamen Speicher gekoppelt, welcher mit einem XPS Mutex synchronisiert wird (vgl. Kap. 5).

In einem POSIX Thread Modell des Xilkernel RTOS wird die Software für die Messdatenerfassung und die Objekterkennung auf MicroBlaze_0 ausgeführt. Der als Controller arbeitende MicroBlaze_1 ist für die Steuerung der Objekterkennung und für die Ausgabe von Informationen über den JTAG UART zuständig. Die vom Anwender über Push Buttons bestimmten Kommandos werden über den FSL-Bus an MicroBlaze_0 gesendet. Die Zuteilung der CPU an die Threads auf MicroBlaze_0 erfolgt durch ein prioritätenbasiertes Scheduling (vgl. Kap. 8).

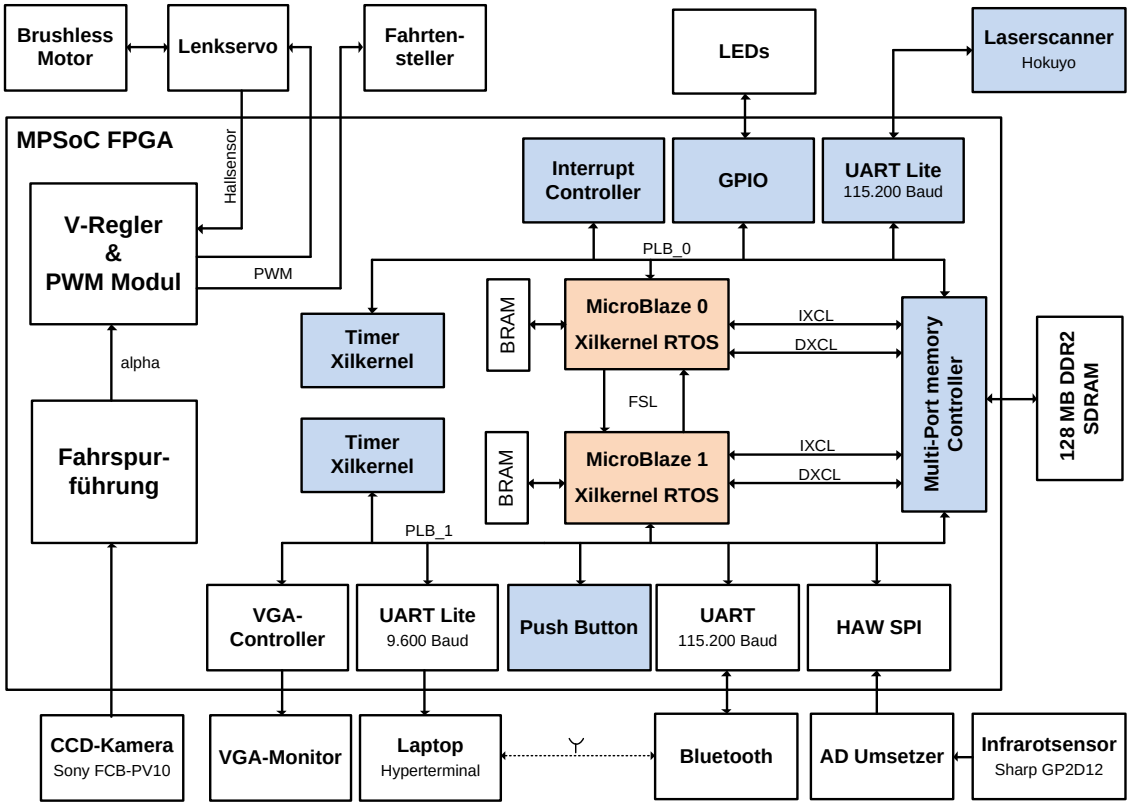


Abb. 1.4.: Dual-MicroBlaze Konfiguration mit einer Shared-Memory Architektur. Die farbig markierten Komponenten stellen den Hauptschwerpunkt dieser Arbeit dar.

Im Forschungsprojekt „Fahrerassistenz- und Autonome Systeme (FAUST)“ der Hochschule für Angewandte Wissenschaften Hamburg wurde das autonome SoC-Fahrzeug entwickelt [29]. Diese Masterarbeit wurde im Rahmen des FAUST Projekts durchgeführt und dient der sukzessiven Weiterentwicklung des Fahrzeugs. Mit dem Ziel der autonomen Fahrspurführung, werden die in vorrangegangen Arbeiten implementierten Komponenten in ein Dual-MicroBlaze System integriert. Die Durchdringung der Multiprozessortechnologie für FPGA-basierte Systeme, sowie die Entwicklung und Partitionierung der MicroBlaze Software, sind Schwerpunkte dieser Arbeit. Mit dem Einsatz von FPGA basierter Hardware können im Gegensatz zu herkömmlichen Prozessorsysteme, die nur wenige Operanden parallel verarbeiten, alle Berechnungen völlig parallel stattfinden. Spezifische Aufgaben können mit einem FPGA in gleicher Zeit, aber mit wesentlich geringeren Taktraten berechnet werden. Der Einsatz von MPSoCs mit Echtzeitbetriebssystemen hat den Vorteil, dass die Parallelität auf Hardware Ebene mit einer Software Parallelität ergänzt wird und somit der Systemdurchsatz steigt. Für diese Masterarbeit werden drei Ziele definiert:

- Als Fahrzeugsteuerungsplattform für das SoC-Fahrzeug wird eine Dual-MicroBlaze Konfiguration auf einem Spartan-3A DSP FPGA entwickelt (vgl. Abb. 1.4). Der Schwerpunkt liegt in der Durchdringung der „Multiprocessor System-on-Chip“ Technologie, welche in den letzten Jahren zunehmend an Bedeutung gewonnen hat [73]. Zwei MicroBlaze Softcore Prozessoren nutzen einen gemeinsamen Speicher und kommunizieren über eine direkte FSL Verbindung. Sowohl die Konfiguration der Daten- und Instruktionscaches als auch die MPMC Kopplung der Prozessoren mit dem gemeinsamen Speicher und dessen Synchronisation sind Teile dieser Masterarbeit (vgl. Kap. 5). Durch die Integration der Laserscanner-basierten Objekterkennung in das Dual-MicroBlaze System, wird eine Evaluierung der MPSoC-Plattform durchgeführt (vgl. Kap. 8).
- Die zweite Zieldefinition beinhaltet die Integration des Laserscanners URG-04LX in das Dual-MicroBlaze System, wobei das serielle SCIP 2.0 Kommunikationsprotokoll als Software Interface auf MicroBlaze_0 implementiert wird. Die kodierten Abstandswerte werden in einem 90° Scanbereich durch ein vom MicroBlaze_1 initiiertes Kommando über die RS232 Schnittstelle empfangen und ausgewertet (vgl. Kap. 6). In Hinblick auf den Einsatz im SoC-Fahrzeug zur automatischen Kollisionsvermeidung erfolgt eine messtechnische Analyse des Timings und der Genauigkeit des Laserscanners.
- Die Implementierung einer Objekterkennung, die auf dekodierte Abstandswerte eine Segmentierung durchführt, wird als drittes Ziel definiert. Durch ein Schwellwertverfahren werden aus den einzelnen Messwerten Objekte generiert und somit ein Abbild der Umgebung erstellt (vgl. Kap. 7). Als Grundlage dient die Objekterkennung des „Sensor Controlled Vehicles“, welche auf einem QNX-basierten GEME 2000 Industrierechner eingesetzt wird [62]. Durch eine messtechnische Analyse wird die Einsatzfähigkeit der Objekterkennung auf einer eingebetteten System-on-Chip Plattform mit einer Taktfrequenz von 62,5 MHz geprüft. In dieser Masterarbeit wird die erste Funktionsanalyse der SCV-Objekterkennung durch eine Implementierung in Software durchgeführt. Mit den gewonnen Ergebnissen kann in nachfolgenden Arbeiten eine Partitionierung der Software in Hardware stattfinden und somit die Ausführungsintervalle verkürzt werden.

1.2. Kapitelübersicht

- **1. Kapitel:** Dieser Abschnitt beinhaltet die Einleitung zum Thema „System-on-Chip“ und definiert die Ziele der Masterarbeit.
- **2. Kapitel:** In diesem Kapitel werden die Grundlagen der Laser Technologie und der Einsatz von Umfelderfassungsensorik im Automobil vorgestellt.
- **3. Kapitel:** Das autonome SoC-Fahrzeug wird sowohl anhand des Fahrzeugaufbaus mit Sensorik und Aktorik als auch anhand der grundlegenden Übersicht zum Fahrspurführungs-SoC beschrieben.
- **4. Kapitel:** Die Konzeptionierung der Laserscanner-basierten Objekterkennung und die Entscheidungskette zur SW/SW Partitionierung wird präsentiert. Dies beinhaltet unter anderem die Vorstellung der Subsumption Architektur und des Xilkernel RTOS.
- **5. Kapitel:** Dieser Abschnitt erläutert die Konfiguration des gekoppelten Dual-MicroBlaze Prozessorsystems, welches auf einer Shared Memory Architektur basiert und eine FSL-Interprozessorkommunikation nutzt. Der Schwerpunkt liegt in der Parametrierung der MicroBlaze Prozessoren und des „Multi-Port Memory Controllers“. Die Auswirkung der Daten- und Instruktionscaches auf das Timing-Verhalten des Gesamtsystems wird evaluiert.
- **6. Kapitel:** Die Charakterisierung des Laserscanners URG-04LX und die Vorstellung des SCIP 2.0 Kommunikationsprotokolls erfolgt im sechsten Kapitel. Dazu zählt die Implementierung des MicroBlaze Software-Interfaces zur Ermittlung und Dekodierung der Abstandswerte in einem 90° Scanbereich. Anhand einer messtechnischen Analyse wird das Timing und die Genauigkeit der Abstandswerte abhängig von der Entfernung analysiert.
- **7. Kapitel:** In diesem Kapitel wird die Implementierung der Segmentklassifizierung zur Objekterkennung beschrieben. Aus den dekodierten Abstandswerten wird mit einem Schwellwertverfahren ein Modell der Umgebung generiert. Zusätzlich wird die Genauigkeit der implementierten Objekterkennung in Bezug auf den Einsatz im SoC-Fahrzeug bewertet.
- **8. Kapitel:** Das achte Kapitel erläutert die Integration der Laserscanner-basierten Objekterkennung in das Dual-MicroBlaze System, wobei die Funktionen zur Messdatenerfassung und zur Segmentierung in POSIX Threads ausgelagert werden. Das prioritätenbasierte Scheduling des POSIX Thread Modells sowie die Software Partitionierung auf zwei MicroBlazes wird vorgestellt.
- **9. Kapitel:** Zur Bewertung und Evaluierung wird in diesem Kapitel eine messtechnische Analyse durchgeführt, die die Messung der seriellen Übertragungszeit und der Ausführungsintervalle der Funktionen beinhaltet. Abschließend erfolgt eine Validierung des Schwellwerts und die Vorstellung des Ressourcenbedarfs auf einem Spartan-3A DSP FPGA.
- **10. Kapitel:** Die Präsentation der Ergebnisse und die Formulierung des Ausblicks für weiterführende Arbeiten bildet den Abschluss dieser Masterarbeit.

2. Prinzipien der Lasertechnologie

Das Wort LASER ist ein Akronym für „Light Amplification by Stimulated Emission of Radiation“ und wurde erstmals im Jahre 1959 öffentlich erwähnt [25]. Das Licht eines Lasers wird aufgrund der Einfarbigkeit als monochromatisch bezeichnet und unterscheidet sich von einer normalen Lichtquelle durch die konstante Wellenlänge und dem sehr engen Frequenzspektrum. Des Weiteren erzeugt ein Laser ein paralleles Lichtbündel, bei dem alle Lichtstrahlen in die gleiche Richtung ausgesendet werden. Die einzelnen Wellen des gebündelten Laserstrahls haben nahezu eine identische Phasenlage und sind somit kohärent. Es wird zwischen räumlicher und zeitlicher Kohärenz unterschieden, wobei bei der räumlichen Kohärenz alle Wellen im Querschnitt eines Laserstrahls phasensynchron sind. Wenn alle Wellen nach einer bestimmten Distanz immer noch in Phase liegen, spricht man von zeitlicher Kohärenz. Für eine weitreichende Entfernungsmessung mit einem Laserscanner wird ein Laser mit einer hohen zeitlichen Kohärenz und einer großen Kohärenzlänge benötigt [40].

Die Entstehung der Laserstrahlen erfolgt mittels stimulierter Emission, bei der ein bestehendes Photon zur Aussendung eines weiteren identischen Photons angeregt wird. Da jedes Photon die gleichen Eigenschaften wie Richtung und Frequenz hat, erfolgt eine Bündelung der Strahlen und somit eine Verstärkung der Strahlung. Ein optischer Resonator speichert die Verstärkung, indem er für eine Rückkopplung des emittierten Lichts durch zwei gegenüberliegende Spiegel sorgt. Die Strahlen werden durch Reflexion mehrfach durch das bei Diodenlasern aus meist Kristall oder Glas bestehende Medium geleitet, wobei die Photonen weitere stimulierte Emissionen erzeugen [16].

Bei Laserscannern werden meist Laserdioden zur Aussendung des Laserstrahls genutzt und das reflektierte Licht wird durch eine Photodiode oder eine Lateraldiode in elektrischen Strom umgewandelt. Laserscanner werden je nach Signalform unterschieden:

- **Pulslaser:** Die Photonen werden nicht kontinuierlich emittiert, sondern in bestimmten Perioden. Zur Erzeugung der Pulse wird im Resonator ein optischer Schalter verwendet, der das Licht nur im geöffneten Zustand durchlässt. Dies hat den Vorteil, dass wesentlich mehr Energie freigesetzt wird und somit der Laserstrahl mehr Intensivität besitzt. Jedoch muss das speichernde Medium eine größere Verstärkungsbandbreite aufweisen. Gemäß der Fourier-Transformation bestimmt die Dauer eines Pulses die Breite des erzeugten elektromagnetischen Spektrums, sodass je kürzer ein Puls desto breiter sein Spektrum [40]. Femtosekundenlaser sind die leistungsstärksten Laser, die eine Pulsdauer von unter einer Pikosekunde und eine Leistung im Kilowatt Bereich erzielen [17].
- **Dauerstrich-Laser:** Im Gegensatz zum Pulslaser sendet der cw-Laser einen kontinuierlichen Laserstrahl, dessen Wellen monochromatisch und kohärent sind. Die durchschnittliche Leistung von 20 Watt liegt bei cw-Laser aufgrund der kleineren Intensivität unterhalb der Leistung von Pulslasern (ca. 60 Watt) [16].

Die „Light Detection and Ranging (LIDAR)“ Technologie ist ein optisches Messverfahren zur räumlichen Messung der Entfernungen von Objekten. Ein periodisch oder kontinuierlich abgestrahlter Laserimpuls wird durch eine rotierende Sendeeinheit oder einen Spiegel in die gewünschte Richtung abgelenkt und ein ebenfalls rotierender Empfänger erfasst den vom Hindernis reflektierten Laserstrahl. Die meisten Laserscanner verfügen über eine interne CPU, die sowohl die Berechnung der Distanzen als auch die Filterung der Rohdaten vornimmt. Laserscanner verwenden zur Entfernungsmessung je nach Anwendungsgebiet verschiedene Verfahren [69][10][80]:

- **Laufzeitmessung:** Bei der „Time of Flight“ Messung wird ein oder mehrere Laserstrahlen in kurzen Impulsen ausgesendet und von einem Hindernis reflektiert. Die Laufzeit Δt des Laserstrahls repräsentiert die doppelte Distanz d zum Hindernis.

$$d = \frac{c_0 * \Delta t}{2} \qquad c_0 = Lichtgeschwindigkeit \tag{2.1}$$

  In der Automobilelektronik wird das „Time of Flight“ Verfahren aufgrund der geringen Reaktionszeit und der großen Reichweite zur Hinderniserkennung eingesetzt [31]. Jedoch ist wegen der erforderlichen Messung von geringen Zeiten eine hohe Abtastfrequenz der internen Zähler erforderlich. Die Entfernungsauflösung ist abhängig von der Zählfrequenz des Laserscanners und liegt bei diesem Verfahren meist im unteren Zentimeterbereich. Das „Time of Flight“ Messverfahren wird ebenfalls bei TOF Kameras zur dreidimensionalen Objekterfassung eingesetzt, wobei für jeden Bildpunkt die Distanz zum Objekt berechnet wird. Der Vorteil von TOF Kameras ist, dass keine Abtastung stattfindet, sondern der gesamte Sichtbereich aufgenommen wird [26].

- **Phasenverschiebung:** Die Amplitude eines kontinuierlich ausgestrahlten Laserstrahls wird mit mehreren sinusförmigen Schwingungen moduliert und durch den Vergleich der Phasenlage des gesendeten und des empfangenen Signals wird die von der Entfernung abhängige Phasendifferenz ϕ bestimmt. Die Entfernung zum Hindernis ist ebenfalls abhängig von der Modulationsperiode T:

$$d = \frac{c_0 * T}{4\pi} * (\phi + 2\pi) \qquad \phi = \frac{\Delta t}{T} * 2\pi \tag{2.2}$$

  Im Vergleich zur Laufzeitmessung kann mit diesem Verfahren aufgrund des geringeren messtechnischen Aufwands, da die internen Zähler nicht so hoch getaktet werden müssen, eine höhere Auflösung im Millimeterbereich erreicht werden. Jedoch ist bei größeren Distanzen eine Eindeutigkeit der Signale nicht gegeben.

- **Triangulation:** Bei der Lasertriangulation wird ein Laserstrahl auf ein Messobjekt gerichtet und das reflektierte Licht mit einem CCD-Sensor aufgenommen (vgl. Abb. A). Durch trigonometrische Winkelfunktionen wird abhängig vom Einstrahlwinkel die Entfernung zum Objekt berechnet. Tritt eine Änderung der Position ein, so wird auch der Einstrahlwinkel verändert. Die Lasertriangulation eignet sich aufgrund des rein mathematischen Verfahrens und den konstanten Vorgaben nur für geringe Entfernungen. Vor allem in 3D-Sensoren zur Objekterkennung oder zur Kartenbildung werden Triangulationsverfahren eingesetzt [18]

2.1. Einsatz von Umfelderfassungsensorik im Automobil

Zur Erhöhung der aktiven Verkehrssicherheit und des Fahrkomforts ist die Anzahl an Sensorik in der Automobilbranche in den letzten Jahren stetig gestiegen. Eine Vielzahl von Kontroll- und Umgebungsinformationen werden von Sensoren erfasst und durch komplexe Regel- und Steueralgorithmen zur Erhöhung der Sicherheit und des Komforts genutzt (vgl. Abb. 2.1). Die im Automobil eingesetzten Sensoren, die der Vorausschau und der Umgebungserfassung dienen, werden vorwiegend in die Kategorie der aktiven Sensoren eingegliedert [59]. Als erste Umfelderfassungssensorik wurden Ultraschallsensoren zur Abstandsmessung eingesetzt, welche sich jedoch aufgrund der kurzen Reichweite nur zur Erfassung von Objekten in unmittelbarer Nähe des Automobils eignen [79]. Für weiter entfernte Objekte werden RADAR und LIDAR Sensoren eingesetzt, die eine Reichweite von bis zu 200 Metern abdecken und vorwiegend in Abstandsregeltempomaten (Adaptive Cruise Control ACC) eingesetzt werden. Aus der Sensordatenfusion von aufeinanderfolgenden Abtastungen, der in Abb. 2.1 dargestellten Erfassungsbereiche, wird die Bewegung und die Entfernung eines Objektes bestimmt.

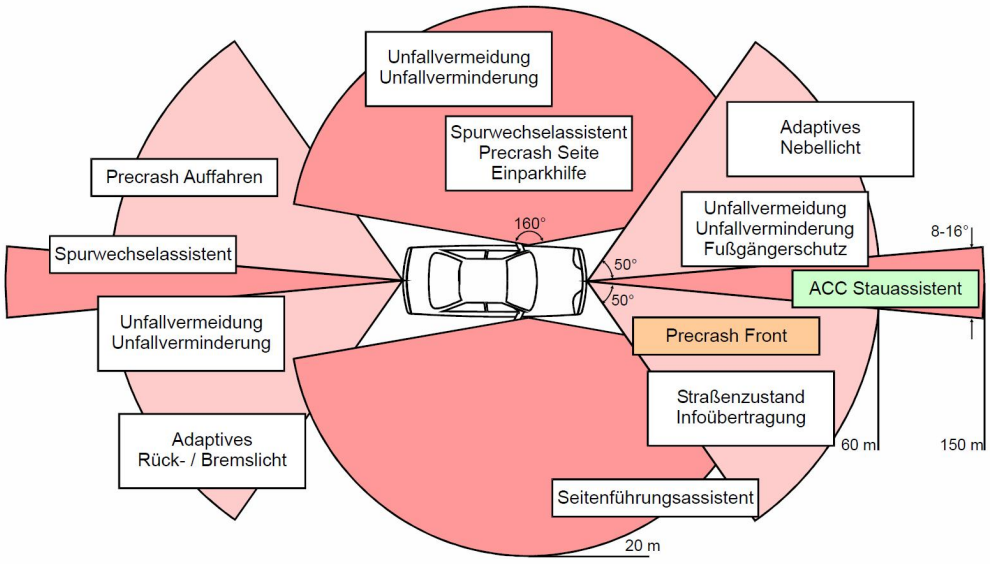


Abb. 2.1.: Anwendungsbereiche von LIDAR Sensorik im Automobil [70]

LIDAR Sensoren sind im Vergleich zum RADAR Sensor in der Automobilbranche weitestgehend noch nicht etabliert, da die Radarwellenausbreitung gegenüber Witterungseinflüssen wie Regen oder Schnee unempfindlicher ist [69]. Die RADAR Technologie nutzt zur Entferungsmessung gebündelte elektromagnetische Wellen im Mikrowellenbereich und arbeiten mit einer Leistung von ca. 10 mW. Die im Automobil eingesetzten Radarsensoren haben somit keine gesundheitlichen Auswirkungen auf Personen, wenn die Richtlinien nach DIN-VDE 0848 Teil 2 eingehalten werden [15].

Die Anwendungsgebiete von Laserscannern beschränken sich nicht nur auf die Automobilbranche. Ebenso werden sie in fahrerlosen Transportsystemen zur Navigation, in der Militättechnik zur Ortung und in der Sicherheitstechnik eingesetzt. Vor allem das 3D-Laserscanning hat in den letzten Jahren im Bereich der Topographischen Vermessungen an Bedeutung gewonnen. Airborne Laserscanning Systeme werden in Flugzeugen oder

Helikoptern eingebaut und erstellen hochauflösenden Geländemodelle mit einer Höhengenauigkeit von bis zu 10 cm [76]. Ein weiteres Beispiel sind die LKW-Maut Brücken, die mit 3D-Laserscannern Fahrzeuge klassifizieren [13]. Nachfolgend werden LIDAR Sensoren vorgestellt, die in der Automobilbranche zum Einsatz kommen.

Hella IDIS® - Adaptive Cruise Control

Die LIDAR-basierte automatische Abstandsregelung von Hella wird seit 2006 optional im Chrysler 300C eingesetzt [31]. Ein Infrarot-Lichtsensor erfasst vorausfahrende Fahrzeuge und ermittelt mit der Lichtlaufzeitmessung den Abstand sowie die Relativgeschwindigkeit. Der Abstand zum vorrausfahrenden Fahrzeug wird in Abhängigkeit zur eigenen Ist-Geschwindigkeit durch die Anpassung der Motorleistung und der Bremskraft konstant gehalten. Hierfür werden über den CAN Bus zusätzlich Informationen vom Geschwindigkeitssensor und Lenkwinkelsensor eingelesen.

Abb. 2.2.: Hella LIDAR Sensor [80]

Abb. 2.3.: Hella IDIS ACC [51]

Der Lidarsensor von Hella ist ein Multi-Beam Laserscanner, welcher mit maximal 16 unabhängigen Sende- und Empfangskanälen arbeitet. Über ein Multiplexverfahren werden die einzelnen Laserdioden getrennt angesteuert und das reflektierte Licht über PIN Dioden erfasst. Die empfangenen Rohdaten werden durch eine Vorfilterung auf maximal 48 Objekte beschränkt. Ein nachgelagertes Tracking-Modul beobachtet die Verkehrssituation und bestimmt aus den vorgefilterten Messdaten, abhängig von der Sichtweite und der erkannten Fahrspur, diejenigen Objekte welche eine Gefahrensituation verursachen könnten. Das ACC-System regelt dementsprechend die Motorleistung oder führt ein automatisches Bremsmanöver ein, welches aus Sicherheitsgründen ein Viertel der maximalen Bremskraft aufwenden darf [65].

Charakterisierung des Hella IDIS®	
Reichweite	200 m
Arbeistbereich	150 m
Frequenz	8 Hz (120 ms)
Öffnungswinkel	12° or 16°
Auflösung	0.1 m
Genauigkeit	± 1 %
Interface	CAN

Tab. 2.1.: Betriebskenngrößen des Hella-ACC [31]

Laserscanner Ibeo ALASCA XT

Der Multi-Beam Laserscanner erfasst in der Vertikalen über vier parallele Kanäle gleichzeitig bis zu vier reflektierte Lichtstrahlen. Durch das mehrzeilige Scannen hat der ALASCA XT einen vertikalen Öffnungswinkel von 3.2° und kann hiermit den Nickwinkel des Fahrzeuges kompensieren (vgl. Abb. 2.4). Des Weiteren wird durch die Multi-Echo Technologie, die pro Laserpuls vier Echosignale empfängt, eine zuverlässige Objekterkennung gewährleistet, welche nicht durch schlechte Witterung wie Regen oder Schnee beeinflusst wird [19].

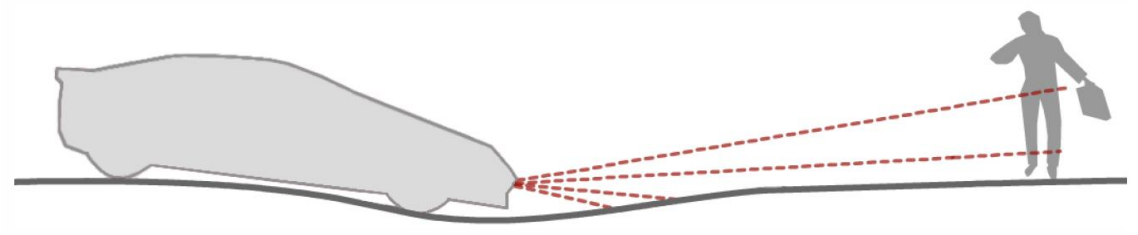
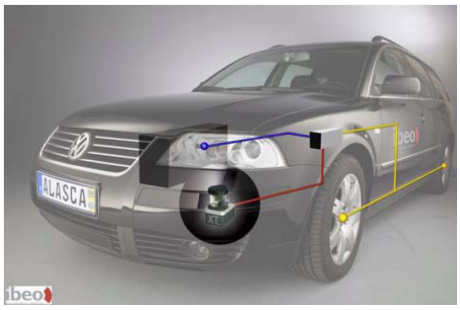


Abb. 2.4.: ALASCA XT Multi-Layer Technologie zur Kompensation des Nickwinkels [35]

Das Umfeld eines Fahrzeuges ist je nach Situation mehr oder weniger relevant. Die resultierenden Messdaten aus dem 240° Scanbereich werden abhängig von der Frequenz, mit einer konfigurierbaren Winkelauflösung eingeschränkt. Bei einer Frequenz von 25 Hz scannt der ALASCA XT die relevante Bereiche von $\pm 16°$ mit einer Winkelauflösung von 0.25° hingegen werden nicht relevante Bereiche, wie die Fahrzeugseite, mit einer Auflösung von 1° gescannt. Zur Datenreduktion empfängt die Ëlectronic Control Unit (ECU)“ vorgefilterte Messdaten und generiert aus den zusammenhängenden Abstandswerten Objekte. Die Positionsverfolgung von sich bewegenden Objekten erfolgt über einen Kalman-Filter. Über den CAN Bus werden die Objektkoordinaten an den Fahrzeug Computer übertragen, wobei auf Grund der Vielzahl an Sensoren im Automobil, ein allgemeingültiges Koordinatensystem durch die DIN 70000 Norm definiert wurde [35][14].



Charakterisierung des ALASCA XT	
Reichweite	0.3 - 200 m
Frequenz	8 - 40 Hz
Horizontaler Scanwinkel	240°
Vertikaler Scanwinkel	3.1°
Auflösung	0.125° - 1.0°
Genauigkeit	$\pm$ 1 %
Interface	ECU CAN

Abb. 2.5 & Tab. 2.2: Betriebskenngrößen des ibeo ALASCA XT Laserscanners [35]

Liegt eine detaillierte Umfelderfassung vor, kann der ALASCA XT für verschiedene Anwendungen eingesetzt werden. Durch den großen Sichtbereich kann der Laserscanner sowohl für eine automatische Abstandsregelung (ACC) als auch für Stop & Go Systeme genutzt werden. Letztere reduzieren den Kraftstoffverbrauch indem sie den Motor bei bestimmten Bedingungen abschalten. Aufgrund der guten Erkennung im Nahbereich werden zukünftige ACC Systeme über eine Stop & Go Funktionalität verfügen [80].

3. Autonomes SoC-Fahrzeug

Das autonome SoC-Fahrzeug wurde im Rahmen des Projekts „Fahrerassistenz- und Autonome Systeme (FAUST)“ an der Hochschule für angewandte Wissenschaften Hamburg entwickelt. Mit dem Ziel die „System-on-Chip“ Technologie, das SW/HW-Codesign, die modellgeriebene Entwicklung mit Matlab/Simulink und die automatische Codegenerierung zu durchdringen, wurde ein automatisches Fahrspurführungssystem auf einem FPGA implementiert. Gerade FPGA-basierte SoC's stellen mit den Logik- und Arithmetikelementen sowie dem internen Speicher die Ressourcen für algorithmische Aufgaben mit hohen Datenraten auch bei niedrigen Taktfrequenzen zur Verfügung. Zusätzlich sind im FPGA konfigurierbare Mikrocontroller mit einer Vielzahl an Peripheriefunktionen verfügbar. Für die Umgebungserfassung nutzt das SoC-Fahrzeug einen Laserscanner und vier Infrarotsensoren zur Hinderniserkennung, Hallsensorpulse für die Geschwindigkeitsermittlung, sowie eine CCD Kamera zur Fahrspurerkennung. Basierend auf einem Tamiya TT-01 Chassis wurde das SoC-Fahrzeug sukzessive zu einem autonom fahrenden Modellfahrzeug im Maßstab 1:10 entwickelt.

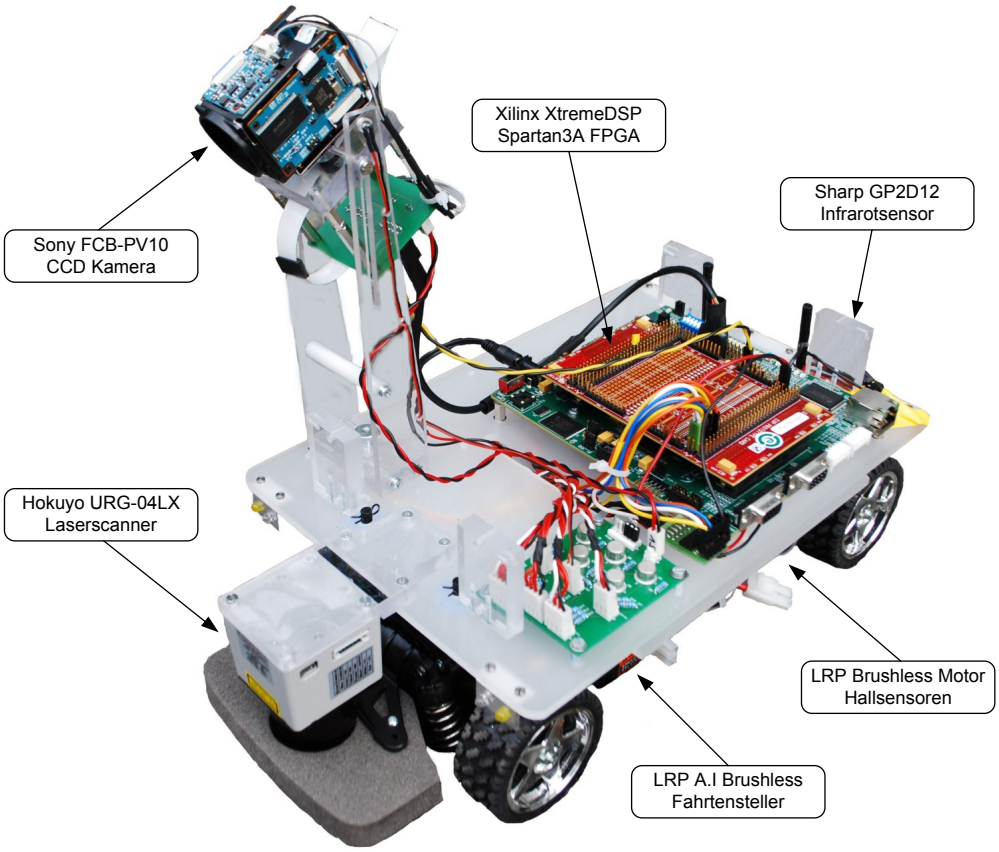


Abb. 3.1.: Autonomes SoC-Fahrzeugs mit Sensorik und Aktorik

3.1. Fahrzeugaufbau mit Sensorik und Aktorik

Das SoC-Fahrzeug basiert auf einem Tamiya TT-01 Chassis mit Allradantrieb und setzt eine 40 MHz Fernsteuerung ein, die die Steuerkommandos direkt an den Empfänger sendet. Die Aufgabe des Empfängers besteht darin, die Kommandos auszuwerten und das jeweils entsprechende PWM Signal an den Lenkservo oder den Fahrtensteller zu senden. Beim SoC-Fahrzeug wurde für den autonomen Fahrbetrieb die PWM Ausgänge des Empfängers mit dem FPGA Board gekoppelt, sodass alle Fernsteuerungssignale direkt in das FPGA geleitet und dort entsprechend ausgewertet werden (vgl. Abb. 3.2). Mit einem Schalter an der Fernsteuerung kann vom autonomen in den manuellen Fahrbetrieb gewechselt werden, wobei durch eine PWM Auswertung im FPGA der manuelle Fahrbetrieb erkannt wird und alle Fernsteuerungssignale direkt an den Fahrtensteller oder den Lenkservo weitergeleitet werden.

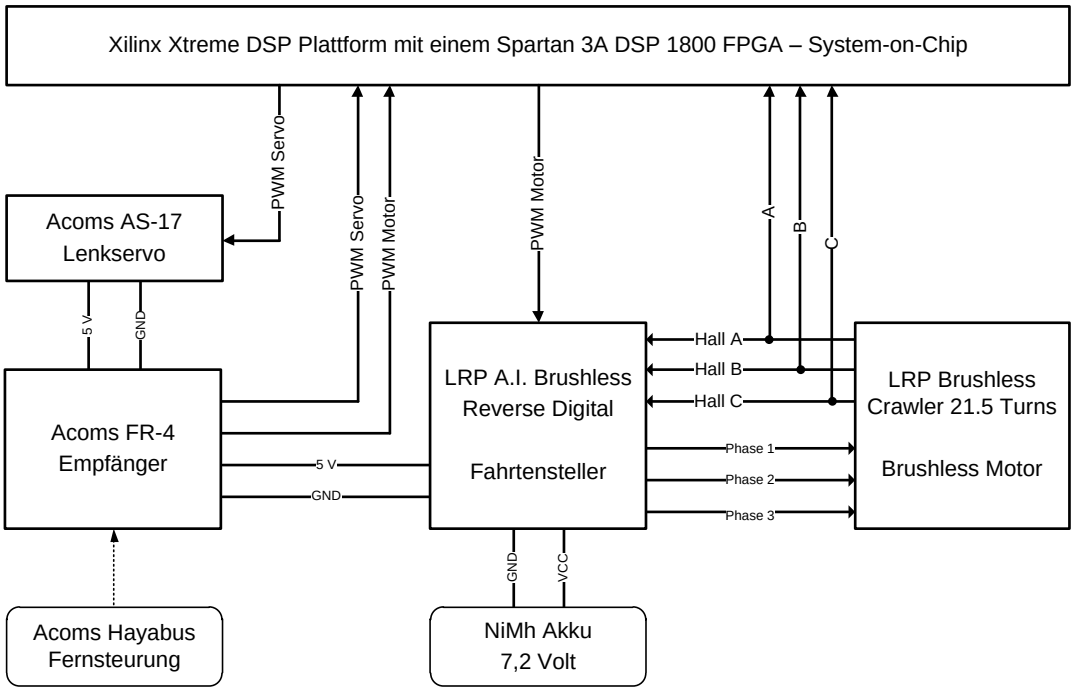


Abb. 3.2.: Autonomer Fahrmodus durch Kopplung der Fahrzeugelektronik mit dem FPGA Board

Mit dem Ziel am Carolo Cup für autonome Fahrzeuge teilzunehmen, wurde das SoC-Fahrzeug entsprechend des aktuellen Reglements entwickelt [11]. Über eine LED Platine werden Brems-, Blink-, Abblend- und Rückfahrlicht entsprechend der Situation geschaltet. Die Fahrzeugma0e betragen:

Länge	457 mm
Breite	202 mm
Höhe	110 mm
Radstand	257 mm

Tab. 3.1.: Fahrzeugmaße des SoC-Fahrzeugs

3.1.1. Anbindung des Laserscanners Hokuyo URG-04LX

Der Hokuyo URG-04LX ist ein Einstrahliger 2D-Laserscanner, dessen Laserdiode eine Wellenlänge von 728 nm hat. Bei einer Frequenz von 10 Hz werden die Laserpulse mit einer Periode von 100 ms ausgesendet. Durch eine Photodiode wird der reflektierte Laserstrahl empfangen und anhand der Phasenverschiebung des modulierten Strahls die Entfernung zum Hindernis berechnet (vgl. Kap. 2). Der Laserscanner arbeitet nach dem „Amplitude Modulated Continuous Wave“ Prinzip, wobei bei konstanter Frequenz die Amplitude und somit die Intensität des Lichts moduliert wird [16]. Durch eine minimale Winkelauflösung von 0,35° werden die gemessenen Abstandswerte in zweidimensionale Polarkoordinaten übermittelt. Der Laserscanner hat bei einer Auflösung von 1 mm eine maximale Reichweite von 5.600 mm (vgl. Tab. 3.3). Jedoch wird lediglich im Bereich von 60 mm - 4.095 mm eine Genauigkeit der Abstandswerte garantiert [33].

Beschreibung	Kenngröße
Maximale Reichweite	20 - 5.600 mm
Garantierte Messreichweite	60 - 4.095 mm
Scanfrequenz	10 Hz
Horizontaler Scanwinkel	240°
Winkelauflösung	0,3515625°
Anzahl Scannschritte	682
Auflösung	1 mm
Genauigkeit (20 - 1.000mm)	± 10 mm
Genauigkeit (1.000 - 4.000mm)	± 1 %
Interface	RS232 & USB
Versorgungsspannung	5 Volt

Abb. 3.3.: Betriebskenngrößen des Laserscanners Hokuyo URG-04LX [33]

Die Berechnung des korrespondierenden Winkels zu einem gemessenen Abstandswert ergibt sich aus der vom Hersteller angegebenen Anzahl an Schritten. Über die komplette Rotation des Scankopfs ergeben sich für 360° insgesamt 1024 Schritte. Diese werden in drei Regionen untergliedert, wobei die erste Region den eigentlichen Scanbereich mit den Schritten 44 - 725 in einem Winkel von 240° umfasst. In einem Winkel von 15° werden die Schritte 0 - 44 und 725 - 768 als Toleranzbereich für den Scanvorgang verwendet. Die restlichen Schritte liegen im nicht scannende Bereich des Laserscanners (vgl. Abb. 3.5). Bei einen Öffnungswinkel von 240° und einer Winkelauflösung von 0,35° besteht eine Scanzeile aus 682 Messpunkten. Durch ein vom Host initiiertes Kommando werden die gewünschten Messpunkte kodiert über die serielle Schnittstelle an den Initiator übertragen (vgl. Kap. 6.1).

Da der Laserscanner mit 5 Volt TTL Pegel arbeitet, werden die Tx und Rx Signale der RS232 Schnittstelle über einen Pegelwandler an die Pins des FPGA Boards angeschlossen (vgl. Abb. 3.4). Der ST-3232 Pegelwandler wandelt die 5 Volt Ausgangspannung des Laserscanners in eine 3,3 Volt Eingangsspannung. Dies ist notwendig, da die Standard I/Os des FPGA Boards nicht 5 Volt tolerant sind, sondern lediglich Eingangsspannungen von 3,3 V oder 2,5 V tolerieren.

Sowohl der Laserscanner als auch der Pegelwandler nutzen eine externe Spannungsversorgung, die auf dem SoC-Fahrzeug von einer Spannungsversorgungsplatine bereit gestellt wird. Zwei in Reihe geschaltete 7,2 Volt Akkus werden durch einen Gleichspannungswandler in 5 V und 3,3 V gewandelt. Der Laserscanner hat bei Inbetriebnahme eine Leistungsaufnahme von 800 mA und während dem Betrieb einen durchschnittlichen Stromverbrauch von 500 mA.

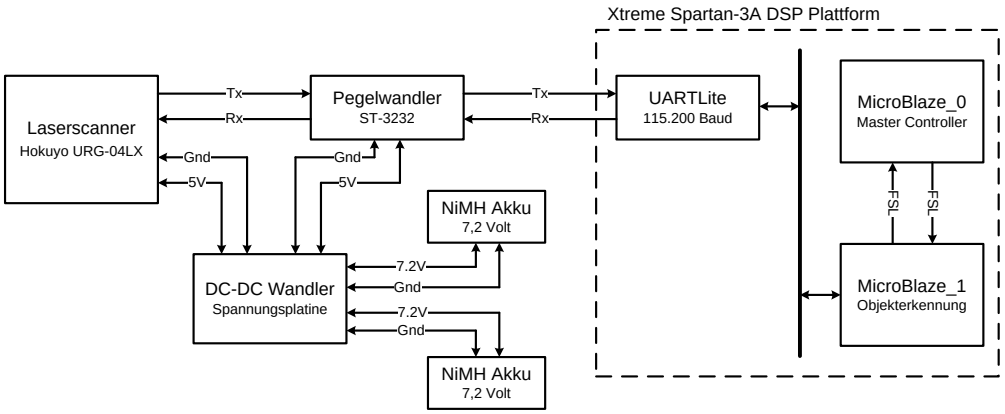


Abb. 3.4.: Kopplung des Laserscanners mit der Xtreme Spartan-3A DSP Plattform

Aus der Drehgeschwindigkeit des Scankopfmotors von 600 rpm folgt, dass sich der Scankopf pro Sekunde zehn Mal dreht und somit eine Frequenz von 10 Hz hat. Die Scanperiode von 100 ms setzt sich aus 66,7 ms echtem Scannen und 33,3 ms Totzeit, in der sich der Spiegel im nicht scannenden Bereich befindet, zusammen. Dem aktiven Scannen folgt ein Toleranzbereich von 4,2 ms. Anschließend befindet sich der Laserscanner 25 ms im nicht scannenden Bereich (vgl. Abb. 3.5).

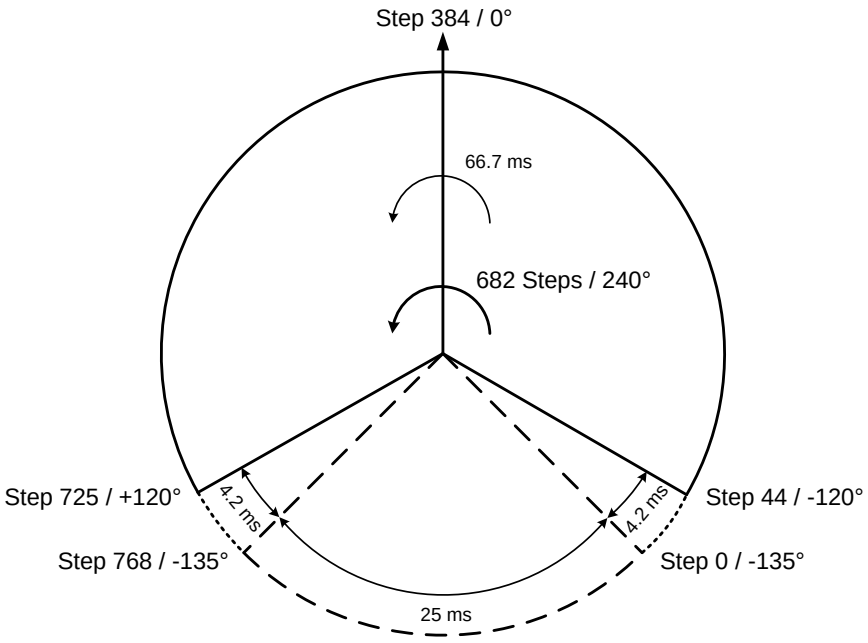


Abb. 3.5.: Laserscanbereich und korrespondierende Schritte zur Winkelauflösung

Xilinx Xtreme DSP Plattform mit einem Spartan-3A DSP 1800 FPGA

Field Programmable Gate Arrays (FPGA) werden seit Anfang der 90er Jahre zunehmend in der digitalen Signalverarbeitung eingesetzt. Sie bestehen aus einer Vielzahl von Logik- und Arithmetikblöcken, die mit einer Hardware-Beschreibungssprache, wie VHDL oder Verilog, zu anwendungsspezifischen Schaltungen synthetisiert werden. Die Eingangssignale aus Sensoren werden mit ihren hohen Datenmengen direkt ohne Zwischenspeicherung mit Additionen und Multiplikationen parallel in Pipeline-Strukturen verarbeitet. Somit lassen sich Beschleunigermodule für die Verarbeitung von großen Datenmengen bei geringen Taktfrequenzen entwickeln.

Die „XtremeDSP Starter Platform“ ist für die Entwicklung und Evaluierung von DSP Anwendungen konzipiert. Ausgestattet mit einem „Spartan-3A DSP 1800A“ FPGA stellt die Plattform ausreichend Ressourcen für das Fahrzeugführungssystem zur Verfügung (vgl. Kap. 9.4). Zahlreiche Peripherieelemente und Interfaces ergänzen das Board mit Schnittstellen zur Kommunikation [84]:

- 125 MHz LVTTL SMT Oszillator
- 128 MB Micron DDR2 SDRAM
- 128 MBit Intel Parallel BPI Flash und 64 MBit SPI Flash
- Ethernet PHY, RS232 Port, VGA, JTAG, SystemACE Module
- 8 LEDs, 8 DIP Switches, 4 Push Buttons
- 2 Digilent 6-pin Header (PMODs)
- 2 EXP Expansion Connectors mit insgesamt 132 Pins

Der Spartan-3A DSP FPGA ist eine Weiterentwicklung des Spartan-3A FPGAs, der bei der DSP Reihe sowohl mit zusätzlichen BRAM Speicher als auch mit Xtreme DSP48A Slices erweitert wurde. Letztere erhöhen den Durchsatz bei algorithmischen Funktionen durch einen internen mit 250 MHz getakteten 18x18 Multiplizierer [89]:

- 16.640 Slices mit 33.280 LUTs und 33.280 Flip-Flops
- 84 DSP48A Slices mit je einem 18x18 Bit Multiplizierer, einem 18 Bit Pre-Adder und einem 48 Bit Post-Adder, sowie Register zur Akkumulation
- 84 BRAM Blöcke zu je 18 KBit (1.512 KBit On-Chip BRAM)
- 519 User I/Os zum Verbinden der externen Peripherie
- 8 Digital Clock Managers (DCMs)

Über die „EXP Expansion Connectors“ wird das FPGA Board mit einer Dual-Slot Aufsteckplatine gekoppelt, die 132 zusätzliche PINs aus dem FPGA führt und für den Anschluss von Sensorik und Aktorik verwendet wird. Insgesamt stehen 84 User I/Os und 24 Power I/Os mit 2.5 V oder 3.3 V zur Verfügung. Sowohl die CCD Kamera als auch der Laserscanner werden über diese PINs angeschlossen, wobei ein Pegelwandler die 3.3 V Boardspannung auf 5 V Sensorikspannung anpasst.

Bürstenloser Elektromotor mit Fahrtensteller und Hallsensoren

Für den Antrieb nutzt das SoC-Fahrzeug einen Brushless Motor, der über einen Fahrtensteller durch drei Phasen gesteuert und mit Spannung versorgt wird. Die Motorleistung wird mit einem Differentialgetriebe über die Antriebswelle auf alle vier Räder verteilt. Um die unterschiedliche Drehzahl der Räder in Kurvenfahrten auszugleichen, werden die Achsen über ein Achsdifferential synchronisiert. Durch die Brushless Technologie wird das Magnetfeld im Motoranker nicht elektrisch, sondern über Dauermagnete erzeugt. Dies hat den Vorteil, dass im Gegensatz zu einem Bürstenmotor kein Verschleiß durch die mechanischen Kohlebürsten entsteht.

Abb. 3.6.: LRP Crawler Brushless Motor mit A.I Brushless Reverse Digital Fahrtensteller [43]

Der Crawler Brushless Motor wurde für präzise Regelbarkeit bei niedrigen Geschwindigkeiten entwickelt und eignet sich aus diesem Grund, gut für den Einsatz im SoC-Fahrzeug, bei dem die maximale Geschwindigkeit bei ca. 5 m/s liegt. Der Fahrtensteller stellt über das empfangene PWM Signal die Drehzahl am Motor ein und empfängt die aktuelle Position des Rotors durch die Hallsensoren im Motor (vgl. Abb. 3.2). Eine Motorwellenumdrehung entspricht einer Hallsensor Periode, was bedeutet, dass je größer die Periodendauer desto langsamer dreht sich der Motor. Durch die gefahrene Strecke pro Umdrehung und durch die Messung der Periodendauer wird die aktuelle Ist-Geschwindigkeit ermittelt. Eine Auswertung der Phasenverschiebung der drei Hallsensor Phasen bestimmt die Drehrichtung des Motors [74].

Drehzahl	11.520 rpm
Leistung	100 Watt
Wicklung	Stern
Wirkungsgrad	90 %
Spezifische RPM/Volt	1.600 kV

Tab. 3.2.: Betriebskenngrößen des LRP Crawler Brushless 21,5 Turns Motors [43]

Sowohl der Motor als auch der Empfänger werden über den Fahrtensteller mit Spannung versorgt, der hierfür einen getrennten 7,2 Volt Akku nutzt. Das FPGA Board, die Sensorik und die Aktorik werden über zwei weitere 7,2 Volt Akkus versorgt, die über eine Spannungsversorgungsplatine in Reihe geschaltet werden und mit einem Spannungswandler wird die Gesamtspannung sowohl auf 7,2 Volt als auch auf 5 Volt übersetzt.

CCD Kamera FCB-PV10 zur Fahrspurerkennung

Die Sony FCB-PV10 besitzt einen 1/4-Type Interline-Transfer-CCD Bildsensor, der die Bilder in einer VGA-Auflösung mit 640x480 Pixeln und einer Bildrate von 29,97 Bildern pro Sekunde liefert (vgl. Tab. 3.7). Das Datenformat entspricht nach der ITU-Rec. BT.656 der Form YCbCr 4:2:2. Mit einem FFC (Flat Flexible Cable) wird die Kamera direkt an das Extension Board angeschlossen. Über diese Verbindung kann das FPGA direkt auf die Videodaten der Kamera zugreifen und zur Parametrierung der Kameraeigenschaften wird eine zusätzliche serielle Verbindung über einen UART genutzt. Des Weiteren verfügt die Kamera über verschiedene Ausgabenmodi, die sich in der Anzahl der in einem Takt übertragenen Bits und den Taktraten unterscheiden [68]. Für das Soc-Fahrzeug wurde zur Fahrspurerkennung der 8 Bit Interlace Modus mit einer Taktrate von 27 MHz gewählt.

VGA Auflösung	640x480 Pixel
Bildrate	29,97 pro Sekunde
Blickwinkel	46°
Optischer Zoom	10 x
Serial Interface	VISCA Protokoll
Video Output	YUV 4:2:2
Versorgungsspannung	6 V - 12 V

Abb. 3.7 & Tab. 3.3: Betriebskenngrößen der Sony FCB-PV10 CCD Kamera [68]

Infrarotsensoren Sharp GP2D12 zur seitlichen Abstandserkennung

Der Infrarotsensor hat eine Reichweite von 80 cm und wird im SoC-Fahrzeug zur seitlichen Abstandsmessung eingesetzt. Die Entfernung wird durch eine analoge Spannung am Ausgang übermittelt, wobei diese abhängig von der Distanz im Bereich von 0.4 V und 2.6 V liegt und über einen AD Wandler digitalisiert wird. Bestehend aus einem Sender und einem Empfänger arbeitet der Sensor nach dem Triangulationsprinzip (vgl. Kap. 2). Ein modulierter Infrarotstrahl wird durch ein Objekt reflektiert und mit einem optischen Positionssensor (PSD) empfangen. Zur Bestimmung der Entfernung ist nicht die Intensivität des reflektierten Lichtes entscheidend, sondern der Reflexionswinkel. Dies hat den Vorteil, dass der Sensor unabhängig von der Oberflächenstruktur und der Reflexionseigenschaft des Objektes ist.

Reichweite	10 - 80 cm
Frequenz	25 Hz / 40 ms
Toleranz	0,5 cm
Ausgangsspannung	0,4 V - 2,6 V
Versorgungsspannung	5 V

Abb. 3.8 & Tab. 3.4: Betriebskenngrößen des Infrarotsensors Sharp GP2D12 [66]

3.2. Übersicht zum Fahrspurführungs-SoC

Das SoC-Fahrzeug als Echtzeit-Bildverarbeitungsplattform basiert auf einer Multiprozessorarchitektur, die zwei identisch konfigurierte MicroBlaze Softcore Prozessoren mit einer Shared Memory Architektur besitzt (vgl. Kap. 5). Über zwei getrennte PLB Bussysteme wird die in Kap. 3.1 beschriebene Sensorik und Aktorik mit den MicroBlazes gekoppelt (vgl. Abb. 3.9). Das Fahrspurführungssystem besteht aus einer Videopipeline, die als Beschleuniger Modul implementiert wurde und eine direkte Verbindung zur CCD Kamera hat (vgl. Kap. 3.2.2). Ebenso ist die Geschwindigkeitsregelung als autarkes System realisiert, wobei durch die Kopplung mit der Fahrspurführung der Lenkwinkel α an das PWM Modul übertragen wird und somit die Ansteuerung des Lenkservos erfolgt. Des Weiteren ist die Geschwindigkeitsregelung zur Parametrierung mit dem PLB gekoppelt.

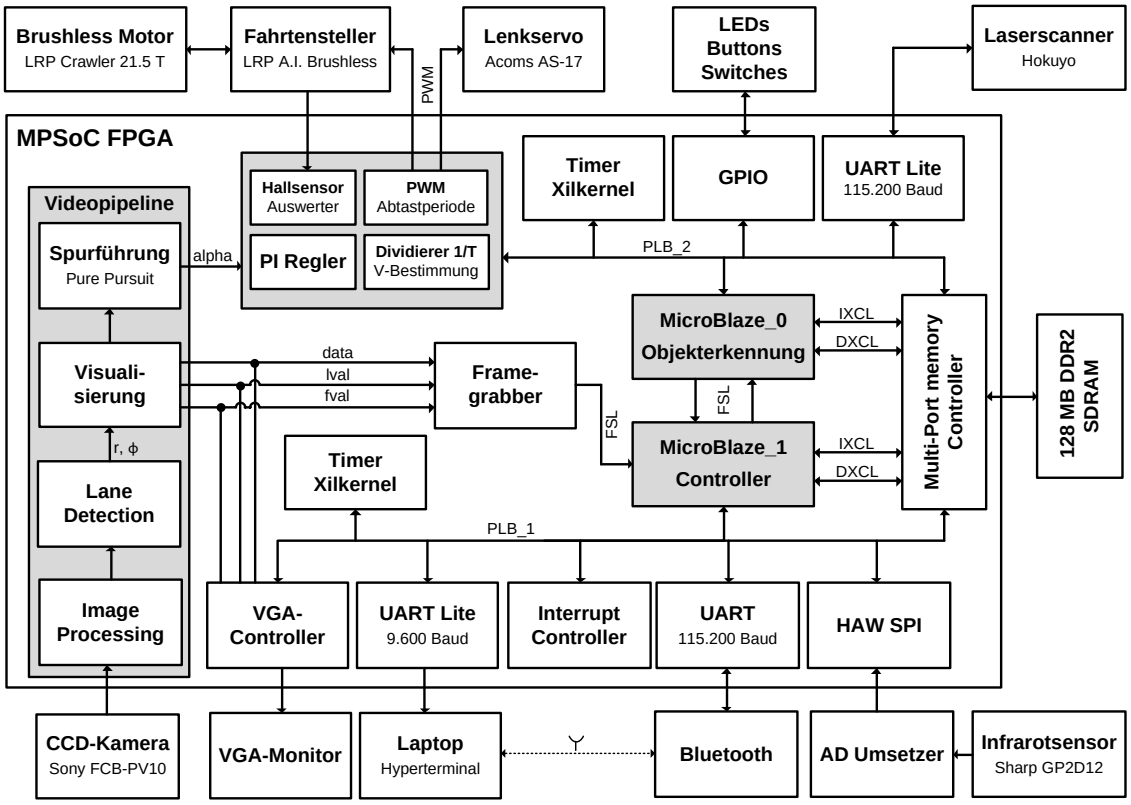


Abb. 3.9.: FPGA-basierte MPSoC Plattform mit Videopipeline und Geschwindigkeitsregelung als IP-Core. Dual-MicroBlaze mit Shared Memory Architektur, wobei MicroBlaze_0 für die Objekterkennung zuständig ist und MicroBlaze_1 als Controller arbeitet

Im Rahmen des FAUST Projekts wurden die einzelnen Module für das SoC-Fahrzeug entwickelt. Die Aggregation dieser Komponenten wird integriert in die Entwicklung der Laserscanner-basierten Objekterkennung. Für die softwareseitige Implementierung wird ein unabhängiger MicroBlaze_0 verwendet, der als exklusive Aufgabe die Messdatenaufbereitung und die Objekterkennung hat. Nachfolgend werden die zentrale Komponenten wie das Fahrspurführungssystem, welches den Lenkwinkel *alpha* bestimmt und die Geschwindigkeitsregelung mit einem PWM Modul erläutert. Für die Übertragung von Messwerten und Parametern im Fahrbetrieb verwendet das SoC-Fahrzeug als Telemetrie Komponente eine Bluetooth Verbindung.

3.2.1. MicroBlaze Softcore Prozessor

Der Xilinx MicroBlaze ist ein Softcore Prozessor mit einer 32 Bit Harvard RISC Architektur (vgl. Abb. 3.10). Er verfügt über zweiunddreißig 32 Bit breite „General Purpose Register“, die die Daten je nach Konfiguration entweder im Big-Endian oder im Little-Endian Format repräsentieren. Die getrennten 32 Bit breiten Adress- und Instruktionsspeicher werden über den „Local Memory Bus (LMB)“ mit dem On-Chip BRAM gekoppelt. Dies hat den Vorteil, dass auf den lokalen BRAM Speicher mit einer Latenz von einem Taktzyklus zugegriffen wird. Die Instruktionsabarbeitung erfolgt je nach Konfiguration in einer 3-5 stufigen Pipeline. Zur Durchsatzsteigerung wird der MicroBlaze mit einem optionalen Daten- und Instruktionscache ausgestattet, der über den „Xilinx Cache Link (XCL)“ mit dem Speicher Controller (meist der MPMC) gekoppelt wird und darüber Zugriff auf die „Cache Area“ im externen Speicher gewährleistet [91].

Mit dem „Xilinx Embedded Development Studio (EDK)“ werden mehrere Instanzen des MicroBlazes in ein FPGA-basiertes MPSoC System integriert, wobei der „Base System Builder (BSB)“ die grundlegende Konfiguration der MicroBlazes übernimmt. Über das SDK kann jeder Prozessor zusätzlich mit einem preemptiven RTOS Kernel, dem sogenannten Xilkernel, ausgestattet werden. Dieser unterstützt den POSIX Standard und ist als freie Software Bibliothek verfügbar [81]. Ein Vorteil von Softcore Prozessoren ist die Flexibilität und die Rekonfigurierbarkeit. Entsprechend dem Einsatzzweck, kann der Entwickler den Prozessor mit weiteren konfigurierbaren Features ausstatten, wie beispielsweise mit einer „Floating Point Unit“ oder einer „Debug Logic“ [91].

Die für das SoC-Fahrzeug gewählte MicroBlaze Konfiguration und die Anbindung an den externen Speicher über den MPMC Controller wird in Kap. 5.2 erläutert.

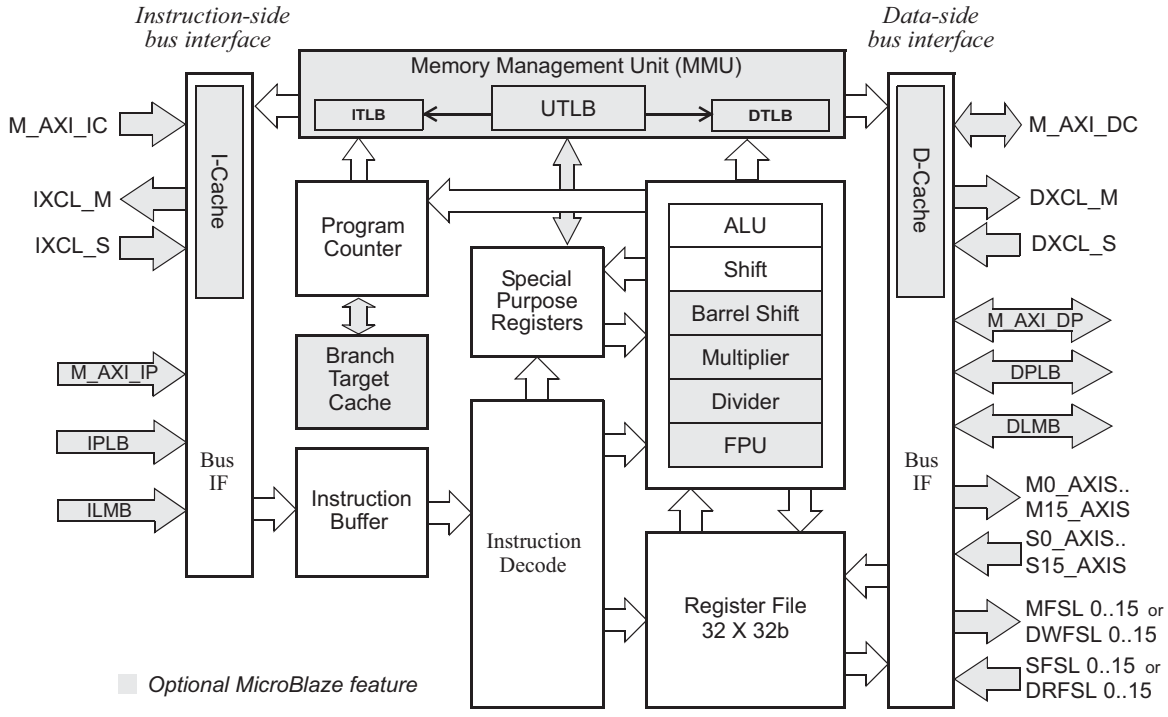


Abb. 3.10.: MicroBlaze Blockdiagramm mit markierten Komponenten als optionale Features [91]

3.2.2. Das Fahrspurführungssystem

In vorangegangen Arbeiten wurde das Fahrspurführungssystem sukzessive entwickelt, wobei die Video-basierte Fahrspurerkennung mit einer Hough-Transformation zur Approximation und einer projektiven Transformation in [46] entwickelt wurden. Eine Erweiterung der Bildverarbeitungsplattform und die Implementierung der Spurführungsalgorithmen wurde in [63] und [55] vorgestellt. Im Rahmen dieser Masterarbeit wird eine Laserscanner-basierte Objekterkennung entwickelt, die in nachfolgenden Arbeiten in das bestehende Fahrspurführungssystem integriert wird.

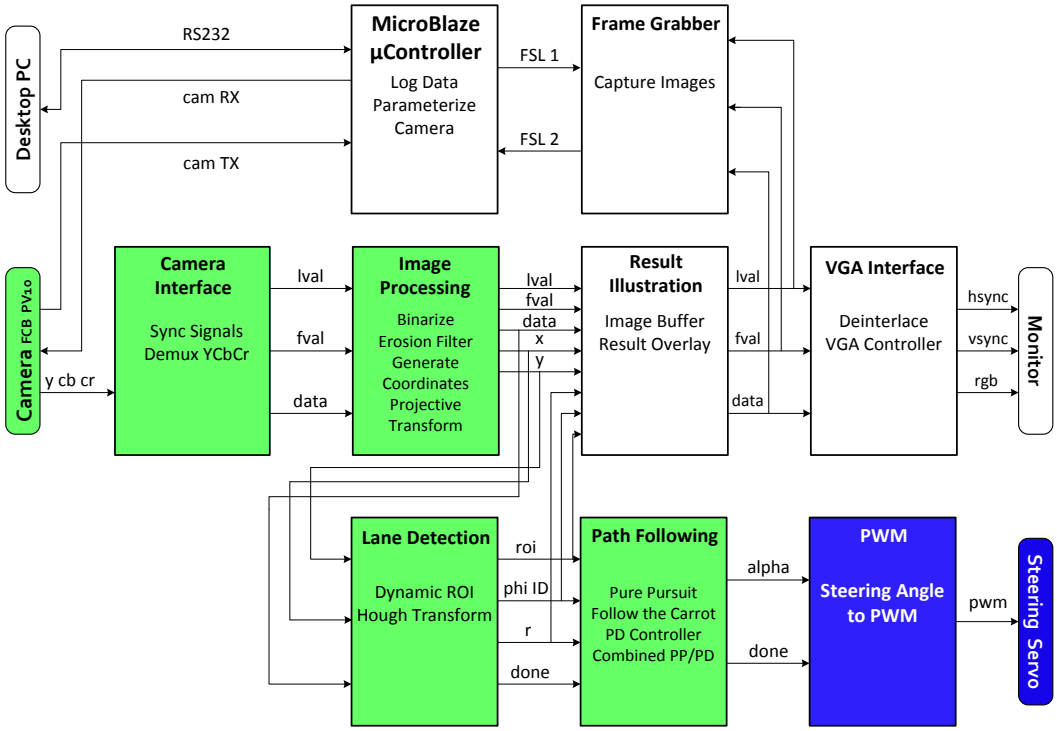


Abb. 3.11.: Schnittstellen und Module der Videopipeline für das Fahrspurführungs-SoC [63]

- **Camera Interface:** Die Halbbilder des Kameradatenstroms werden im „Camera Interface“ mit den Synchonisationssignalen *lval* und *fval* ergänzt. Des Weiteren werden die Pixelfarbwerte im YCbCr Format in 8 Bit Grauwerte transformiert.
- **Image Processing:** Gekoppelt mit dem „Camera Interface“ empfängt das „Image Processing“ Modul die Grauwertbilder und führt zur Datenreduktion eine Grauwertskalierung durch. Anschließend wird mit einem Schwellwertverfahren das Graustufenbild binarisiert und die zusammenhängenden hellen Flächen durch morphologische Operationen erodiert. Die x- und y-Koordinaten der einzelnen Pixel werden nach der Filterung mit einem Pixelzähler berechnet.
- **Lane Detection:** In diesem Modul wird die Lage der Fahrspur als Abstand r und Winkel Φ approximiert. Hierfür werden die eingehenden Pixel durch eine dynamische „Region of Interest (ROI)“ geleitet, die feststellt, ob die Pixelkoordinaten im relevanten Bereich liegen. Ist dies der Fall, werden die Pixelkoordinaten in das ROI-Koordinatensystem transformiert (vgl. Abb. 3.13). Eine Hough-

Transformation trägt die transformierten Koordinaten in die Hough-Ebene ein und gibt die Parameter r und Φ der approximierten Geraden in Hesse'scher Normalform aus.

- **Path Following:** Zur Bestimmung des Lenkwinkelsollwertes wird in diesem Modul aus den Geradenparametern und den ROI-Koordinaten sowohl der „Look-Ahead-Point (LAP)" als auch der Fahrzeugabstand zur Fahrspurmarkierung bestimmt. Der durch Spurführungsalgorithmen bestimmte Lenkwinkel wird an das PWM Modul der Geschwindigkeitsregelung weitergeleitet. Zur Spurführung wurden verschiedene Algorithmen implementiert, wie beispielsweise der Pure-Pursuit oder der Follow-The-Carrot. Des Weiteren wurde eine PD-Regelung implementiert, die den Abstand zur Fahrspurmarkierung regelt, wobei aus der Regelabweichung der Soll-Lenkwinkel bestimmt wird.
- **Result Illustration:** Dieses Modul dient der Ausgabe der bearbeiteten Halbbilder auf einem VGA Monitor. Hierfür werden die erkannten Geraden mit dem perspektivisch entzerrten Bild überlagert, indem alle projektiv veränderten Pixel in einen Bildzwischenspeicher eingetragen werden. Zur Anzeige auf einem VGA Monitor werden die Halbbilder zeilenverdoppelt und im VGA Interface auf das Format 640 x 480 Pixel gebracht. Im „Frame Grabber" Modul werden die Pixel eines Bildes zwischengespeichert und über eine RS232 Schnittstelle an den PC übertragen.

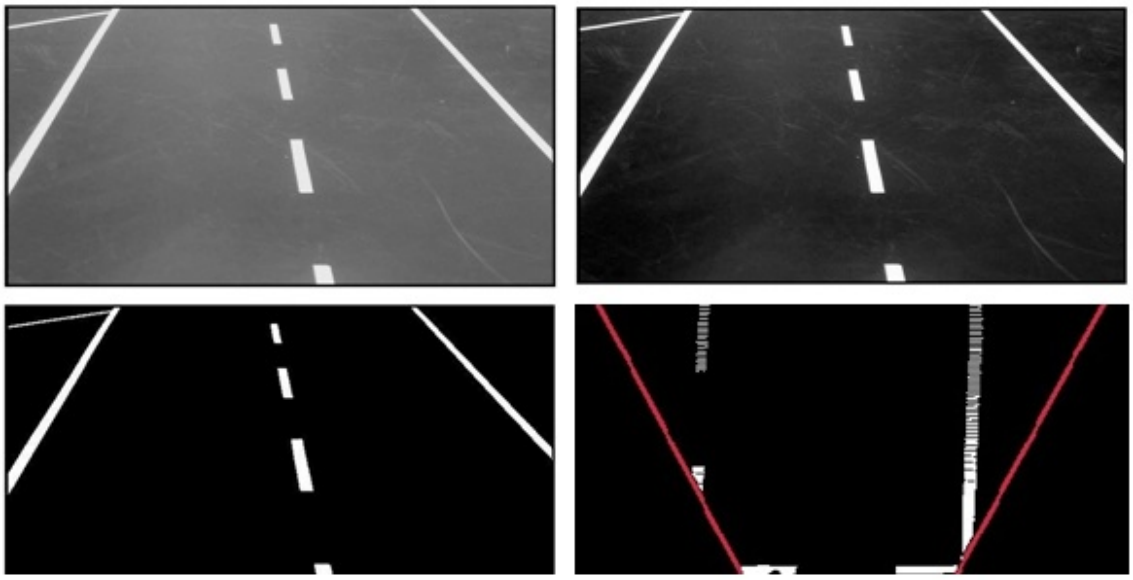


Abb. 3.12.: Das originale Kamerabild (oben links) wird kontrastverstärkt (oben rechts) und anschließend in ein Binärbild transformiert (unten links). Nach der perspektivischen Entzerrung wird die Fahrzeuglage in der Vogelperspektive bestimmt (unten rechts). [63]

Mit dem weiterreichenden Ziel eine automatische Hinderniserkennung und ein Ausweichmanöver zu implementieren, muss in nachfolgenden Arbeiten die Fahrspurführung angepasst werden. Die in dieser Arbeit implementierte Objekterkennung erkennt Hindernisse und transformiert die Koordinaten in das Fahrzeugkoordinatensystem (vgl. Abb. 3.13). Jedoch muss für eine Integration in das Fahrspurführungssystem zwei Konzepte in Betracht

gezogen werden, sodass Objekte auf einer Fahrspur erkannt werden und ein Ausweichmanöver eingeleitet werden kann:

- Der erste Ansatz ist die Verdopplung der ROI, was den Vorteil hat, dass beide Fahrspurmarkierungen erkannt werden und bei der Identifizierung eines Hindernisses der ROI dynamisch auf die gegenüberliegenden Markierung umgeschaltet wird. Jedoch erfordert dieser Ansatz eine Kamera mit einem größeren Öffnungswinkel als die Jetzige. Vor allem in Kurvenfahrten ist sonst die Erkennung beider Markierung nicht garantiert.
- Ein zweiter Ansatz ist die Vergrößerung des Soll-Abstandes zur rechten Fahrspurmarkierung. Hierfür wird durch die Laserscanner-basierte Objekterkennung die Breite des detektierten Objektes berechnet und der Abstand zur rechten Markierung angepasst, sodass der ROI die linke Fahrspurmarkierung erkennt. Jedoch hat eine Manipulation des Lenkwinkels zur Folge, dass bei Nichterkennung der gegenüberliegende Fahrspurmarkierung, das Fahrzeug die Fahrspur überfährt.

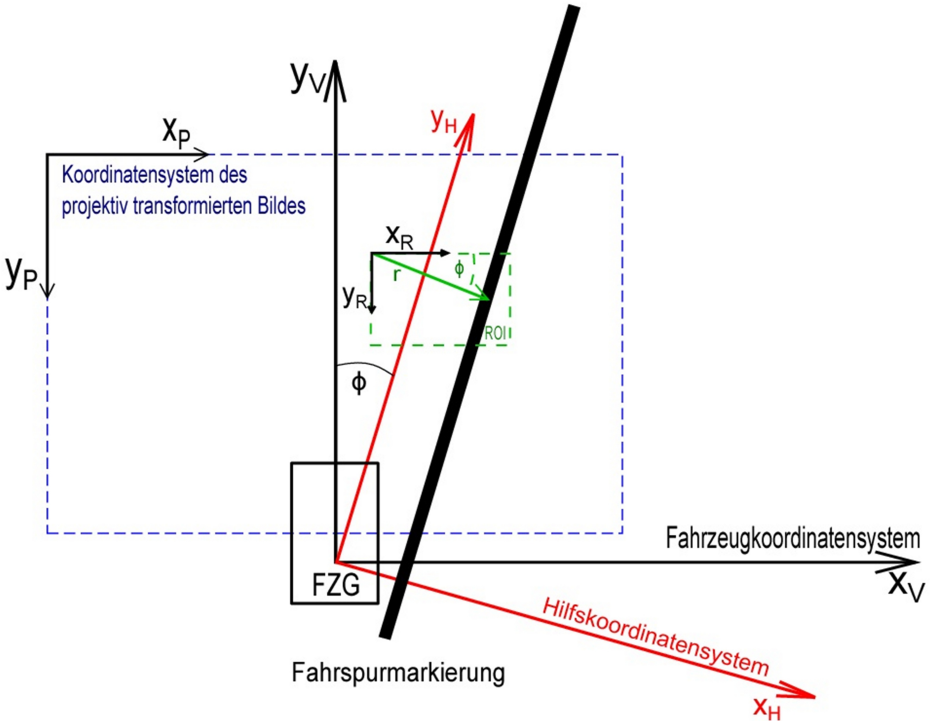


Abb. 3.13.: Verknüpfung der verschiedenen Koordinatensysteme des SoC-Fahrzeugs [63]

3.2.3. Geschwindigkeitsregelung und Geschwindigkeitsermittlung

Die Geschwindigkeitsregelung und die Ansteuerung des Fahrtenstellers über ein PWM Modul sind in einem Beschleuniger Modul integriert (vgl. Abb. 3.9). In [6] wurde die Entwicklung der V-Regler Pipeline vorgestellt, die zur Einsparung von FPGA Ressourcen in [74] angepasst und durch ein Prozessor-Element erweitert wurde. Die Bestimmung der Ist-Geschwindigkeit basiert auf der Auswertung der Hallsensoren des Brushless Motors (vgl. Kap. 3.1.1). Im SoC-Fahrzeug arbeitet die Geschwindigkeitsregelung als autarkes System, wobei die Soll-Geschwindigkeit vom MicroBlaze über die Kopplung mit dem

PLB Bus in einem Software Register bereitgestellt wird. Der vom Fahrspurführungssystem berechnete Soll-Lenkwinkel wird direkt dem PWM Modul übermittelt, welches den Duty-Cycle berechnet und den Lenkservo ansteuert (vgl. Abb. 3.14). Die drei Hallsensorpulskanäle zur Geschwindigkeitsermittlung werden durch das Extension Board direkt über externe PINs in das FPGA geführt.

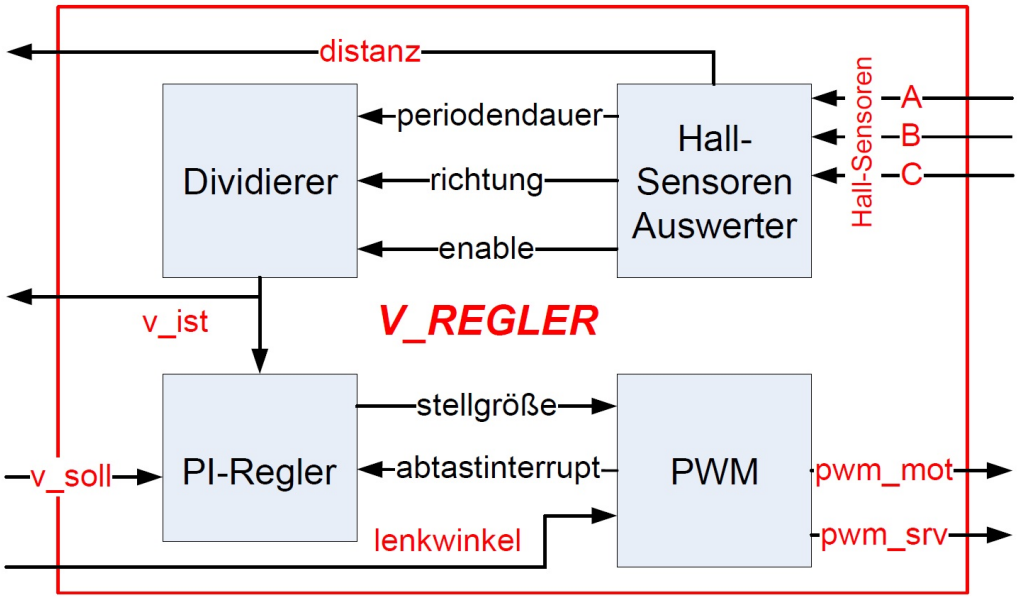


Abb. 3.14.: Beschleuniger Modul zur Geschwindigkeitsbestimmung- und Regelung [74]

- **Hallsensoren Auswerter:** Dieses Modul bestimmt die Periodendauer eines Hallsensorpulses, die gefahrene Strecke und die Fahrtrichtung. Mit einem Timer wird die Zeit zwischen zwei aufsteigenden Flanken eines Hallsensorpulskanals gemessen und die Periodendauer an das Dividerer Modul zur Ist-Geschwindigkeitsberechnung übermittelt, welcher über ein Enable Signal vom Auswerter aktiviert wird.
- **Dividierer:** Da der Spartan-3A DSP FPGA keine dedizierten Dividierer-Ressourcen zur Verfügung stellt, wurde ein „Division by Trial Subtraction“ Dividierer als Multi-Zyklus Datenpfad entwickelt, welcher aus der zurückgelegten Strecke und der Periodendauer die Ist-Geschwindigkeit ermittelt. Die konstante Strecke, die aus der Umdrehung eines Rades folgt, wird durch die vom Hallsensor Auswerter übermittelte Periodendauer geteilt.
- **PI-Regler:** Der Geschwindigkeitsregler empfängt die Ist-Geschwindigkeit vom Dividierer und die Soll-Geschwindigkeit vom MicroBlaze, wobei die vom PWM Modul generierte Abtastperiode den Regler durch einen Interrupt startet. Aus der resultierenden Regelabweichung wird die Stellgröße berechnet und an das PWM Modul gesendet. Die Geschwindigkeitswerte der Stellgröße liegen im Intervall von -400 cm/s und 400 cm/s.
- **PWM Modul:** Dieses Modul wird zur Ansteuerung des Fahrtenstellers und des Lenkservos genutzt. Abhängig von der Stellgröße wird bei einer Periode von 17 ms ein PWM Signal mit einem Duty-Cycle zwischen 5% und 10% generiert.

4. Konzept der Laserscanner-basierten Objekterkennung

Die Laserscanner-basierte Objekterkennung empfängt die kodierten Rohdaten vom Laserscanner und eine nachgelagerte Segmentierung nutzt ein Schwellwertverfahren zum Clustering der Abstandswerte. Das grundlegende Konzept der Objekterkennung wurde aus dem Ausweichassistenten des „Sensor Controlled Vehicle (SCV)“ abgeleitet [62][54]. In dieser Arbeit wird zur ersten Funktionsanalyse die Objekterkennung als Software-Modul auf einem FPGA-basierten Multiprozessorsystem implementiert, wobei das bestehende Software-Konzept des SCV-Fahrzeugs auf eine Portierbarkeit geprüft wird:

- Die Objekterkennung des SCV-Fahrzeugs wird auf einem QNX-basierten GEME-2000 Industrierechner mit einer Taktfrequenz von 650 MHz ausgeführt. In dieser Arbeit wird analysiert, ob eine Portierung der Objekterkennung auf eine SoC-basierte Mikrocontroller Plattform realisierbar ist.
- Mit einer SW/SW-Partitionierung auf zwei MicroBlazes soll die von 650 MHz auf 62,5 MHz reduzierte Taktfrequenz durch eine Parallelisierung kompensiert werden.
- Der auf dem SCV-Fahrzeug eingesetzte SICK Laserscanner mit einer Frequenz von 20 Hz wird durch einen URG-04LX Laserscanner mit einer Frequenz von 10 Hz ersetzt.
- Durch eine messtechnische Analyse werden die Bearbeitungsintervalle analysiert und Konzepte für eine Hardware Partitionierung vorgestellt.

Der Entwurf von Systemen, die sowohl aus Hardware- als auch aus Software Komponenten bestehen, wird Hardware / Software Codesign genannt. Hierbei werden ressourcenlastige Softwareteile extrahiert und in spezielle Hardwarestrukturen implementiert. Mit der Hinzunahme von zusätzlicher Hardware wird der Durchsatz und damit die Beschleunigung des Gesamtsystems gesteigert [44][61].

Ein Nachteil der Auslagerung in Hardware tritt bei stark sequentiellen Algorithmen ein. Sind Operationen abhängig von vorrangegangen Ergebnissen tritt das Pipelining ein, wobei die Befehle in Teilaufgaben zerlegt und parallel ausgeführt werden. Bei komplexen sequentiellen Algorithmen hat diese Architektur einen Anstieg der Pipeline Stufen und somit auch einen Anstieg der FPGA Ressourcen zur Folge. Da Mikroprozessoren meist höher getaktet sind als FPGAs, ist der erhöhte Implementierungsaufwand nicht vorteilhaft [44]. Aufgrund der Laserscanner Frequenz von 10 Hz und der Abhängigkeit der Verarbeitungsschritte wird bei der Implementierung der Objekterkennung von einer Extrahierung in Hardware abgesehen. Stattdessen wird eine SW/SW-Partitionierung eingesetzt, die durch die Auslagerung von Teilaufgaben auf zwei MicroBlazes eine Parallelität gewährleistet, wobei die Erfassung der Messdaten und die darauf angewandte Segmentierung in ein POSIX Thread-Modell des Xilkernels RTOS integriert werden (vgl. ßkap8).

Diese Arbeit dient als erste Funktionsanalyse und kann in nachfolgenden Arbeit durch eine Hardware Partitionierung erweitert werden. Das Software-Modul der Laserscanner-basierte Objekterkennung besteht aus vier Komponenten, welche jeweilig mit den Ergebnissen der vorangegangenen arbeitet (vgl. Abb. 4.1).

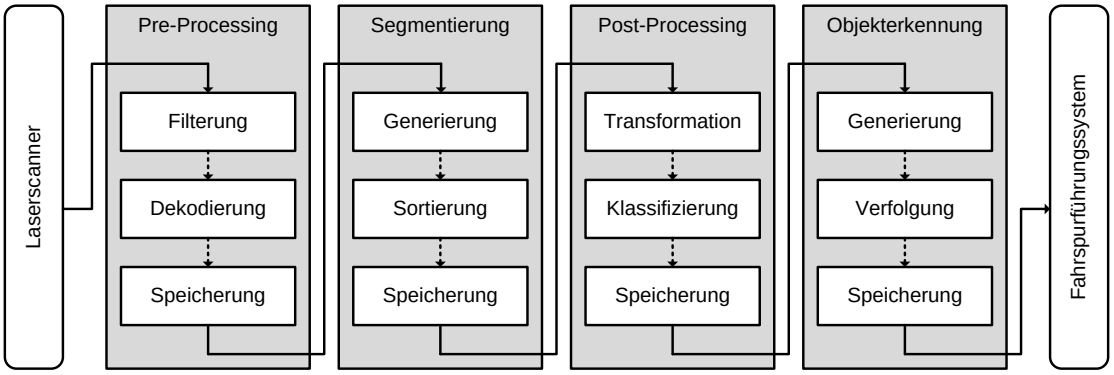


Abb. 4.1.: Struktur und Ablauf der Laserscanner-basierten Objekterkennung

Pre-Processing:
Die erste Komponente empfängt zyklisch die Rohdaten vom Laserscanner, wobei durch einen Filteralgorithmus das Messrauschen innerhalb der Laserscanwerte entfernt wird. Anschließend werden die in zwei Byte dekodierten Abstandswerte entschlüsselt (vgl. Kap. 6). Die für die Objekterkennung nicht relevanten Abstandswerte werden durch eine nachgelagerte Reichweitenbeschränkung entfernt (vgl. Kap. 6.2.2). Dies reduziert den Speicherbedarf und erhöht die Ausführungszeit der Segmentierung.

Segmentierung:
Die Hauptaufgabe der Segmentierung ist das Auffinden von zusammenhängenden Segmenten innerhalb der Abstandswerte. Durch den sequentiellen Vergleich von aufeinanderfolgenden Abstandswerten mit einem Schwellwert werden Segmente generiert, welche anhand von drei Punkten sowohl in Polarkoordinaten als auch in kartesischen Koordinaten in einer Liste gespeichert werden (vgl. Kap. 7.1).

Post-Processing:
Eine Überprüfung der abgespeicherten Segmente dient der Klassifikation von Objekten, die für das Fahrspurführungssystem relevant sind. Anschließend werden die Polarkoordinaten der Segmente in das kartesische Fahrzeugkoordinatensystem transformiert und die Breite des Objekts berechnet (vgl. Kap. 7.1.1).

Objekterkennung:
Diese Komponente empfängt die generierten Segmente und erzeugt Objekte, welche durch einen Tracking Algorithmus verfolgt werden. Die Informationen aus Segmenten zwei aufeinander folgender Scans werden zur Erstellung bzw. Aktualisierung von dynamischen, sich bewegenden Objekten verwendet. Zur Berechnung des Kollisionszeitpunktes wird außerdem der Bewegungsvektor des Objektes, spezifiziert durch die Geschwindigkeit und den Abstand, berechnet. In der aktuellen Konfiguration des SoC-Fahrzeugs werden statische Segmente berücksichtigt und es erfolgt keine Objektverfolgung (vgl. Kap. 7.2).

4.1. Die Subsumption Architektur

Die Subsumption Architektur wurde im Jahre 1986 für den Einsatz in mobilen Robotern entwickelt und basiert auf der Unterteilung des Gesamtverhaltens in mehrere hierarchisch angeordnete Module [9]. Jede Komponente ist für eine spezifische Aufgabe zuständig und kommuniziert über eine Kopplung mit den darüber und darunter liegenden Modulen. Das komplexe Verhalten des Gesamtsystems entsteht durch die Überlagerung von mehreren Modulen, die parallel und asynchron zueinander arbeiten, wobei die unterste Schicht für die Ansteuerung der Aktorik und den Empfang der Sensorik zuständig ist (vgl. Abb. 4.2). Die Module auf höheren Schichten können aufgrund ihrer größeren Priorität die Module auf unteren Ebenen unterbrechen. Dies hat den Vorteil, dass je nach Situation, verschiedene Aufgaben mit unterschiedlichen Verhalten ausgeführt werden. Hierzu zählt beispielsweise die Fahrspurführung, die durch eine Kollisonsvermeidung oder ein Ausweichmanöver unterbrochen oder rekonfiguriert wird.

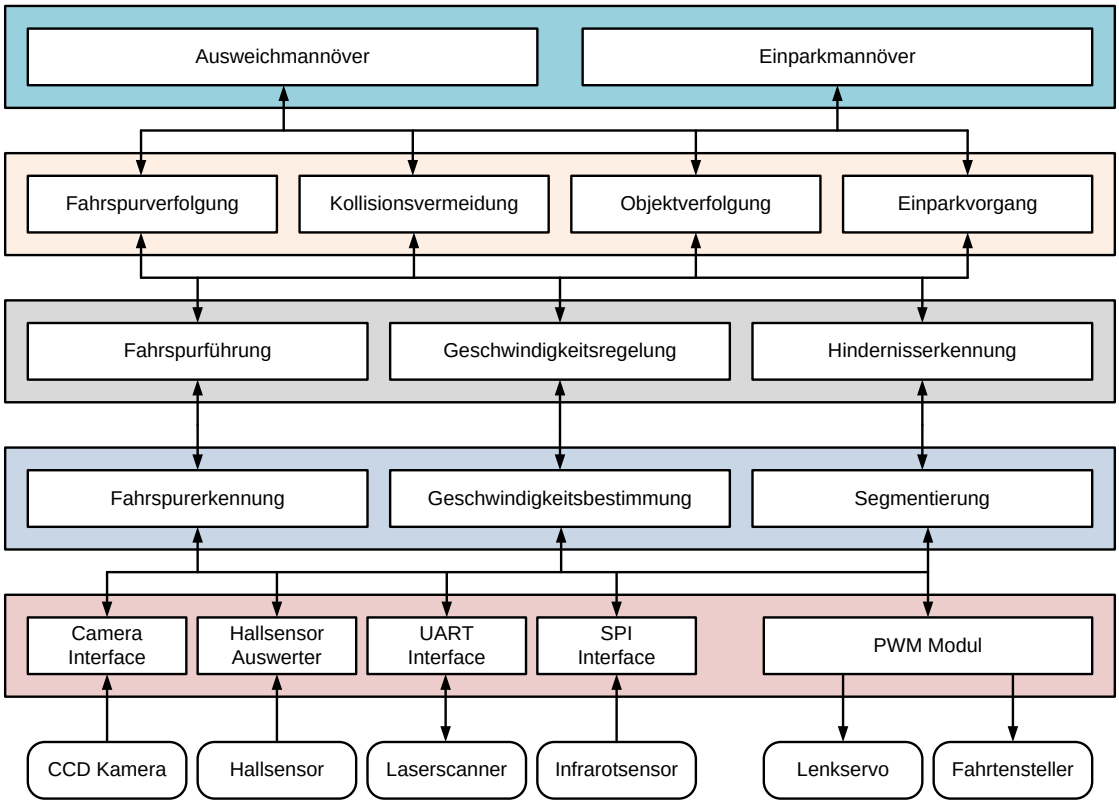


Abb. 4.2.: Funktionsarchitektur des SoC-Fahrzeugs nach der Subsumption Architektur

Reaktive Verhaltensteuerungen, wie die Subsumption Architektur, reagieren im Gegensatz zu planbasierten Verfahren direkt auf die Veränderung der Umwelt. Sie besitzen keinen Plan und kein Modell der Umwelt. Bei den meisten reaktiven Systemen ist die Sensorik direkt über eine Steuerungseinheit mit der Aktorik gekoppelt und im Vergleich zur deliberativen Verhaltensteuerung, wobei ein detaillierter Plan schrittweise abgearbeitet wird und ein vollständiges Weltmodell erstellt wird, reagieren reaktive Systeme lediglich abhängig von der Umfelderfassung. Reaktive Systeme befinden sich in ständiger Kommunikation mit der Umwelt, wobei sich das System der Umwelt unterordnet. Die

gemessenen Sensorwerte werden nach einer Verarbeitung direkt an die Umwelt zurückgeführt [64]. Beim SoC-Fahrzeug ist beispielsweise die Auswertung der Hallsensorpulse und die daraus resultierende Ansteuerung des Motors über das PWM Modul, eine direkte Rückführung der Sensorgrößen.

In manchen Situationen ist die direkte Kopplung der Sensorik und Aktorik ein Nachteil der Subsumption Architektur. Der Roboter oder das Fahrzeug erstellt kein internes Weltmodell und kann somit keine Langzeitstrategien planen [60]. Jedoch ist dieser Nachteil für das SoC-Fahrzeug hinfällig, da keine Kartografie der Umwelt stattfindet, sondern nur die in Sichtweite liegende Fahrspur betrachtet wird und ausschließlich in Abhängigkeit der momentanen Situation reagiert wird.

Für viele Aufgaben ist die rein reflexartige Entscheidung über ausführende Aktionen auf Basis von momentanen Sensordaten vollkommen ausreichend, hierzu zählt beispielsweise die Kollisionsvermeidung im SoC-Fahrzeug. Jedoch gibt es Anwendungsgebieten bei denen ein detailliertes Weltmodell erforderlich ist, wobei die Grundlage für zielgerichtete Aktionen, die Speicherung von Informationen über den Zustand der Umwelt ist. Das sogenannte Weltmodell wird aus den über längere Zeit aufgezeichneten Sensordaten erzeugt. Autonome Fahrzeuge oder Roboter besitzen meist zwei Kartentypen, wobei die lokale Karte aus der Fusion der Sensordaten gewonnen wird und dem aktuellen Sichtbereich entspricht. Eine globale Karte entspricht dem Weltmodell und dient der Navigation und der Positionsbestimmung in der Umwelt. Für die Erzeugung des Weltmodells gibt es in der Literatur und in Veröffentlichungen verschiedene Ansätze [27][41].

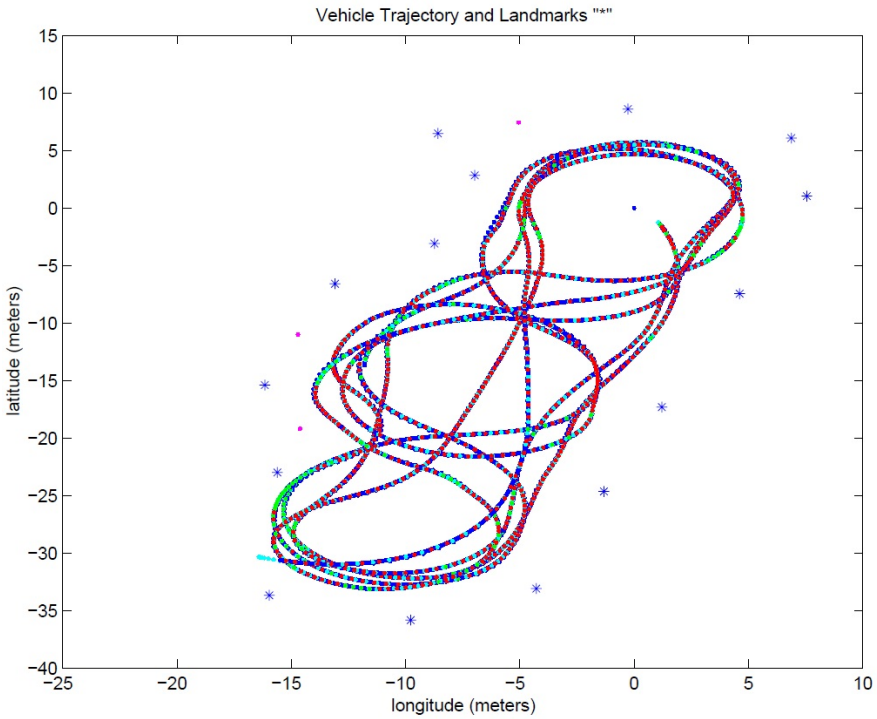


Abb. 4.3.: Ergebnis einer autonomen Kartografie zur Navigation mit künstlichen Landmarken [27]

4.2. Methoden der Segmentierung

Unter Segmentierung versteht man das Extrahieren von Merkmalen und die Vereinigung von zusammenhängenden Regionen innerhalb eines Bildes oder eines Scans. Eine vollständige Segmentierung ist gegeben, wenn jedes Pixel eindeutig einem Segment zugeordnet ist. Im Bereich der digitalen Bildverarbeitung wird die Segmentierung zur Klassifizierung von Objekten innerhalb eines Bildes genutzt. Diese Ansätze werden ebenfalls auf die Segmentierung von Laserscandaten angewandt, wobei anstatt einen Bildes mit Pixelinformationen eine Menge von Abstandswerten betrachtet wird. Nachfolgend werden aus der Bildverarbeitung bekannte Segmentierungsverfahren vorgestellt und anschließend für den Einsatz beim Laserscanning bewertet [38][23][71].

- **Punktorientiertes Verfahren:** Bei dieser Segmentierung wird für jeden Bildpunkt einzelnd entschieden, ob er zu einem Segment zugeordnet wird oder nicht, wobei meist der Grauwert des isolierten Pixels das Kriterium für die Entscheidung darstellt. Das verbreitetste punktorientierte Verfahren ist das Schwellwertverfahren. Anhand eines Histogramms wird die Grauwertverteilung eines Bildes analysiert und Segmente klassifiziert. Innerhalb des Histogramms wird der Schwellwert meist direkt aus den erkennbaren minimalen Extremwerten berechnet und anhand der lokalen Minima und Maxima werden die Objekte klassifiziert. Treten innerhalb eines Bildes mehrere Extremwerte auf, ist für eine optimale Segmentierung eine Anpassung des Schwellwertes vorteilhaft. Im Gegensatz zum globalen Schwellwert wird beim dynamischen Schwellwert eine Nachbarschaftsbeziehung betrachtet, welche das Bild in Regionen unterteilt, wobei jede Region einen eigenen Schwellwert abhängig von den Nachbarn besitzt.
- **Kantenorientiertes Verfahren:** Hierbei wird das Bild nach Kanten durchsucht, die meist als Trennung zwischen zwei Pixelregionen liegen. Da die kantenorientierte Segmentierung eine sequentielle Methode ist, kann dieses Verfahren nicht parallel auf alle Pixel angewandt werden, sondern ist abhängig von vorangegangen Berechnungsschritte. Die meisten kantenorientierten Segmentierungsalgorithmen führen im ersten Schritt eine Glättung zur Signalrauschunterdrückung durch. Anschließend werden Kanten oder Linien im Bild anhand eines Filters detektiert, wie beispielsweise mit dem Sobel-Operator oder dem Laplace-Operator.
- **Regionorientiertes Verfahren:** Bei dieser Methode werden Pixel aufgrund ihres Grauwerts einem Objekt zugeordnet, unabhängig von den Nachbarpixeln. Die regionorientierte Segmentierung sucht zusammenhängende Objekte, indem Pixelmengen als Gesamtheit betrachtet werden und somit können einzelne isolierte Bereiche entstehen. Einer der bekanntesten Algorithmen ist das „Split and Merge“ Prinzip, wobei im ersten Schritt das ganze Bild als eine Region betrachtet wird. Anschließend wird die Region ausgehend von einem Homogenitätskriterium, wie beispielsweise einem Schwellwert, in kleinere Regionen unterteilt. Die entstandene kleinste Region wird mit der Nachbarregion verglichen und bei hinreichender Übereinstimmung werden beide Regionen vereinigt. Die Segmentierung ist beendet, wenn nach einer Iteration alle Regionen eine definierte Mindestgröße besitzen, die eine bestimmte Anzahl an Pixeln darstellt.

Für die Laserscanner-basierte Objekterkennung wird das punktorientierte Verfahren eingesetzt, da sowohl das kantenorientierte als auch das regionorientierte Verfahren sequentiell sind und stark von den vorrangegangen Berechnungen abhängen. Hingegen kann bei der punktorientierten Segmentierung eine Parallelisierung aufgrund der direkten Nachbarschaftsbeziehung stattfinden, indem mehrere Abstandswerte zur gleichen Zeit betrachtet werden. Ein Problem besteht darin, dass bei einer parallelen Berechnung, die Ergebnisse am Ende zusammengeführt werden müssen, da die in der Umwelt zusammenhängenden Objekte in zwei oder mehr Segmente zerlegt werden. Aus diesem Grund ist eine Parallelisierung von Abstandswerten nur vorteilhaft, wenn es keine Abhängigkeiten zwischen den zu parallelisierenden Werten gibt.

Bei der Implementierung der Objekterkennung im Rahmen dieser Masterarbeit wird sowohl auf ein statisches als auch auf ein dynamisches Schwellwertverfahren zurückgegriffen, welches abhängig von der Distanz arbeitet. Der sogenannte „Successive Edge Following“ Algorithmus arbeitet direkt auf den in Polarkoordinaten vorliegenden Abstandswerten, wobei die Distanz eines Messwerts sukzessive mit der Distanz des nachfolgenden Messwerts verglichen wird [67]. Benachbarte Laserscandaten gehören zu einem Segment, wenn die Abstandswerte einen bestimmten Schwellwert nicht überschreiten (vgl. Abb. 4.4). Da die Öffnung des Abstrahlwinkels abhängig von der Distanz größer wird, muss der Schwellwert ebenfalls abhängig von der Entfernung angepasst werden. Ist dies nicht der Fall, werden nicht zusammenhängende Objekte, die mit einem minimalen Abstand zueinander positioniert sind, als ein Objekt identifiziert.

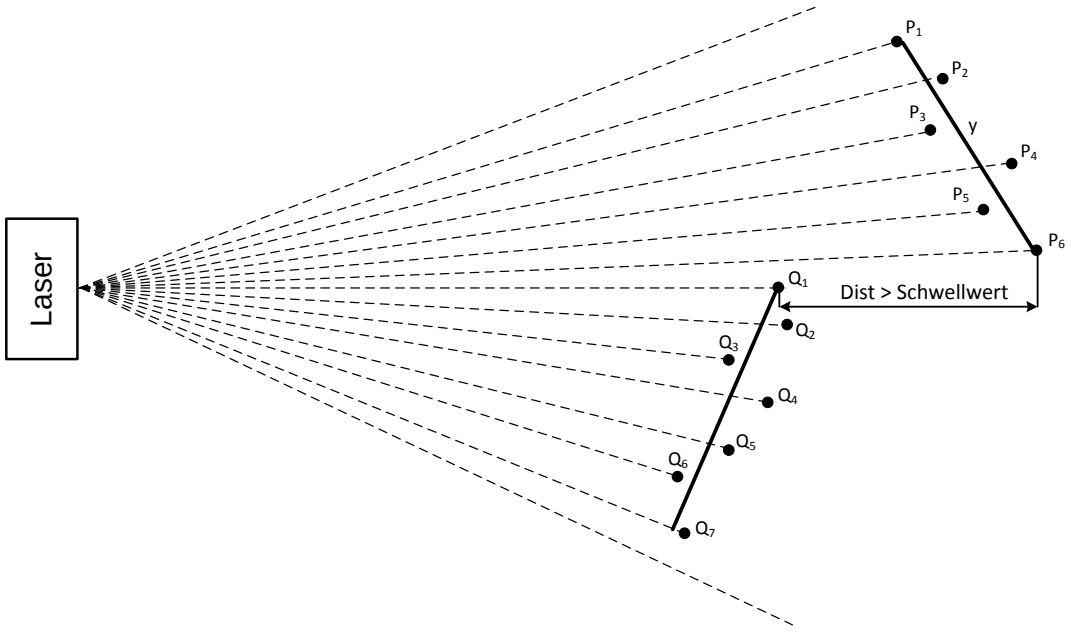


Abb. 4.4.: Generierung von Segmenten bei Überschreitung des Schwellwerts

Beim „Successive Edge Following“ Verfahren wird keine Vorverarbeitung der Daten und keine Transformation der Koordinaten in das kartesische Koordinatensystem benötigt. Dies ist ein Vorteil gegenüber anderen Verfahren, die auf kartesischen Koordinaten arbeiten oder einen rekursiven Algorithmus verwenden [67]. Bei dem für die Masterarbeit angewandten Segmentierungsalgorithmus wird davon ausgegangen, dass aus einer Menge von Punkten $P = (x_1, y_1)...(x_n, y_n)$ eine Gerade $y = m * x + b$ approximiert wird (vgl. Abb. 4.4).

4.3. Objektverfolgung mit Tracking-Verfahren

Unter der Objektverfolgung versteht man die zeitliche Zuordnung von aufeinanderfolgenden Messwerten anhand der Bewegung von Objekten. Basierend auf der Objekterkennung werden mit Tracking-Algorithmen die Bewegung von dynamischen Objekten berechnet. Da für das SoC-Fahrzeug die Objektverfolgung im Gegensatz zur Erkennung ein untergeordnetes Ziel darstellt, wird im Nachfolgenden das Tracking nur grob beschrieben. Nach Blackman und Popoli lautet die allgemeine Definition für Tracking [5]:

> *The target tracking objective is to collect sensor data from a field of view containing one or more potential targets of interest and to then partition the sensor data into sets of observations, or tracks, that are produced by the same sources.*

Tracking-Algorithmen bestimmen die Trajektorie eines beweglichen Objekts, indem sie in allen aufeinanderfolgenden Scans die Position des Objektes finden, wobei die Gültigkeit eines Objekten von der Sichtweite des Lasers oder der Kamera abhängt. Ein Problem von Tracking-Algorithmen ist die Aktualisierung von Objekten anhand aktueller Messdaten. Wird ein Objekt in einem nachfolgenden Scan nicht mehr erkannt, dann kann dies verschiedene Ursachen haben: Das Objekt ist entweder aus der Sichtweite des Laserscanners oder es wurde durch die Segmentierung mit einem anderen Segment vereinigt. Die Zuverlässigkeit einer Objektverfolgung ist dadurch gekennzeichnet, dass die Zuordnung von Segmenten zu einem Objekt trotz unterschiedlicher Randbedingungen, wie Verdeckung eines Objekts oder ein Wechsel der Perspektive, stattfindet.

Die im Fahrzeugumfeld eingesetzten Objektverfolgungsalgorithmen basieren meist auf dem Kalman-Filter, der Rückschlüsse auf den Zustand von Objektparametern, wie Geschwindigkeit und Position, anhand von vorliegenden fehlerbehafteten Beobachtungen berechnet. Umfelderkennende Systeme sind auf verlässliche Informationen bezüglich der Dynamik eines Objekte im Umfeld angewiesen. Aktuelle Systemen verwenden zur Zustandsschätzung anstatt dem Kalman-Filter einen Partikel Filter. Die sogenannte „Sequentielle Monte-Carlo-Methode (SMC)“ gehört zur Klasse der stochastischen Verfahren, die durch die Bestimmung der Wahrscheinlichkeit den Systemzustand des dynamischen Systems bestimmt [77].

Das Tracking wird in vielen verschiedenen Gebieten eingesetzt, wie beispielsweise bei der Videoüberwachung von öffentlichen Plätzen oder bei der Fußgängererkennung in zukünftigen Fahrerassistenzsystemen [21]. Der Einsatz einer Objektverfolgung im SoC-Fahrzeug ist nicht vorteilhaft, da durch die enge Bahnführung und aufgrund des schmalen Öffnungswinkels des Laserscanners eine zuverlässige Verfolgung nicht gegeben ist. Des Weiteren hat die Laserscanner Frequenz von 10 Hz zur Folge, dass lediglich alle 100 ms die Objekte aktualisiert werden. Geht man davon aus, dass das SoC-Fahrzeug mit einer Geschwindigkeit von 2 m/s der Fahrspur folgt, dann wird pro Abtastfrequenz eine Strecke von 20 cm zurückgelegt. Um ein Objekt eindeutig zu identifizieren und der reellen Welt zuzuordnen sind mindestens zwei aufeinanderfolgende Scans notwendig, was einer zurückgelegten Strecke von 40 cm entspricht. Aus diesem Grund wird für das im Fahrspurführungssystem integrierte Ausweichmanöver ausschließlich statische Segmente verwendet.

4.4. Der Xilkernel als Echtzeitbetriebssystem

In einem eingebetteten System muss die Software Echtzeitanforderungen erfüllen, wie beispielsweise das Einhalten eines garantierten Zeitverhaltens. Die anwendungsspezifische Konfiguration der Hardware ist ein Vorteil von eingebetteten System, jedoch erfordert die steigende Komplexität und die hohen Leistungsanforderungen eine Auslagerung in Software. Da herkömmliche Betriebssysteme den zeitlichen Determinismus nicht garantieren, werden Echtzeitbetriebssystemen eingesetzt, die mit zusätzlichen Echtzeit-Funktionen ein vorhersehbares Zeitverhalten bieten und somit Antwortzeiten garantieren. Dies wird vor allem bei harten Echtzeitsystemen gefordert, bei denen die Antwort zu einer genau definierten Zeit vorliegen muss, da sonst schwerwiegende Systemausfälle die Folge sind [4].

Die Prozesse in einem Echtzeitbetriebssystem arbeiten zeitlich determiniert und können zu jeder Zeit unterbrochen werden. Der RTOS Scheduler nutzt zur Suche nach dem nächsten Thread anstatt eines Arrays eine Prozess Tabelle, die stets aktuell ist und somit eine konstante Suchzeit hat. Der Nachteil, dass die Suche nach dem nächsten Prozess mehr Zeit in Anspruch nimmt als bei einer Array Implementierung, wird bei Echtzeitbetriebssystemen akzeptiert, da die Zeit für einen Taskwechsel vorhersehbar ist [47].

Die meisten Echtzeitbetriebssysteme basieren auf der Mikrokernel Architektur, bei der im Gegensatz zum monolithischen Kernel einzelne Betriebssystemkomponenten in eigene Prozesse ausgelagert werden. Der eigentliche Kernel implementiert die Funktionen zur Prozess- und Speicherverwaltung, sowie Funktionen zur Interprozesskommunikation und Synchronisation. Die ausgelagerten Komponenten laufen im Benutzer Modus und haben keinen direkten Zugriff auf den Kernel, sondern kommunizieren über die Interprozesskommunikation [72]. Hierzu zählt beispielsweise die Treiberverwaltung und das Dateisystem. Ein Vorteil des Mikrokernels ist die gute Ausfallsicherheit, welche durch die modulare Architektur garantiert wird. Bei Ausfall einer Komponente, wie beispielsweise einem Hardware Treiber, ist das Gesamtsystem weiterhin arbeitsfähig.

Auf beiden MicroBlazes des MPSoC's wird als Echtzeitbetriebssystem der Xilinx Xilkernel ausgeführt, welcher den POSIX Standard unterstützt und als freie Software Bibliothek verfügbar ist. Das preemptive Xilkernel RTOS verfügt über zwei statische Schedulingstrategien, die über das „Board Support Package (BSP)“ festgelegt werden [81]:

- Das Round-Robin Verfahren *SCHED_RR* arbeitet mit einer einzigen „Ready-Queue“, wobei alle Threads gleichberechtigt behandelt werden. Die Abarbeitung der Threads erfolgt nach einem definierten Zeitintervall (Time-Slice), das durch den Parameter *SYSTMR_INTERVAL* konfiguriert wird.
- Beim prioritätenbasierten Scheduling *SCHED_PRIO* wird pro Priorität eine „Ready-Queue“ eingesetzt, wobei ein höher priorisierter Thread stets das Vorzugsrecht vor einem niedrig priorisierten Thread hat. Wenn alle Threads die gleiche Priorität besitzen, werden diese nach den Round-Robin Verfahren abgearbeitet.

Beim Fahrspurführungs-SoC arbeitet MicroBlaze_0 mit einem prioritätenbasierten Scheduling und MicroBlaze_1 nach dem Round-Robin Verfahren (vgl. Kap. 8). Die zugrunde liegende Multiprozessor Architektur und die Aufgaben der einzelnen Prozessoren werden in Kap. 5 beschrieben.

4.5. Entwurf des Ausweichmanövers zur Kollisionsvermeidung

Die Laserscanner-basierte Objekterkennung ist die Grundlage für das Ausweichmanöver zur Kollisionsvermeidung im Soc-Fahrzeug. Wird ein auf der Fahrspur befindliches Objekt durch die Segmentierung erkannt, dann leitet der Controller MicroBlaze_1 das Ausweichmanöver ein. Im Vergleich zu einem Bremsmanöver verlässt das Fahrzeug beim Ausweichen seine ursprüngliche Trajektorie um einen seitlichen Versatz nach links bzw. rechts. Nachfolgend wird der Entwurf einen Ausweichmanövers auf Basis von Abstandswerten erläutert, wobei durch das Carolo-Cup Regelwerk die Fahrspur- und Hindernisabmessungen bekannt sind [11]. Die Implementierung des Ausweichvorgangs ist kein Teil dieser Arbeit.

Im Rahmen des Carolo-Cups werden Hindernisse sowohl statisch als auch dynamisch auf eine Fahrspur gestellt. Die Detektion eines Hindernisses erfordert einen Spurwechsel, welcher innerhalb von 2 m abgeschlossen sein muss. Auf dem Rundkurs werden mehrere Hindernisse mit einem Mindestabstand von einen Meter stehen. Die zweite Aufgabe besteht darin, dynamischen Objekten auszuweichen, die sich mit einer maximalen Geschwindigkeit von 0.6 m/s auf der Fahrspur bewegen. Sowohl die Vorfahrtsbedingungen an einer Kreuzung als auch die Stoppschilder müssen beachten werden. Fahrsituationen, bei denen die gesamte Fahrbahn durch ein Hindernis blockiert ist, sind ausgeschlossen [11]. Für das SoC-Fahrzeug wird im ersten Schritt ein Ausweichmanöver für statische Hindernisse konzipiert

Das Ausweichmanöver wird durch den Controller MicroBlaze_1 eingeleitet, indem die Parametrierung der Fahrspurführung rekonfiguriert wird. Zu dieser Zeit ist durch die Objekterkennung die Position und die Breite des Hindernisses bekannt. Der Ausweichweg wird durch das Aneinanderreihen von vier Klothoidenkurven dargestellt, wobei jede Klothoide die Übergangskurve darstellt, die bei der Fahrt von einer Geraden in einen Kreisbogen entsteht (vgl. Abb. 4.5). Die Krümmung der Kurve ist hierbei stets linear und zunehmend. Durch das Aneinanderreihen von Klothoiden wird die Querbeschleunigung reduziert und das Fahrzeug stabiler.

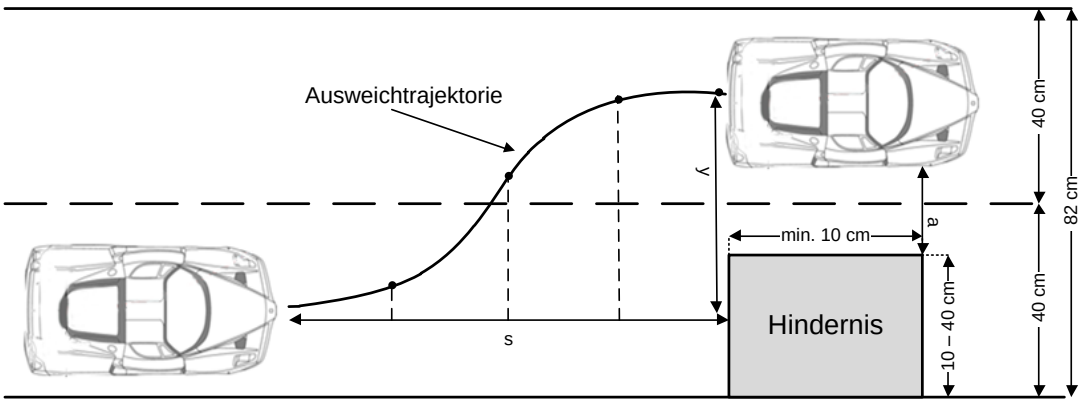


Abb. 4.5.: Ausweichmanöver durch Aneinanderreihen von vier Klothoiden

Die Koordinaten des Hindernisses werden zyklisch durch die Laserscanner-basierte Objekterkennung berechnet. Der einzuschlagende Lenkwinkel zum Erreichen des Zielpunktes wird wie folgt berechnet:

$$\alpha = \arctan \frac{y}{s} \tag{4.1}$$

Bei einem Ausweichvorgang wird ein definierter Sicherheitsabstand a zum Hindernis eingehalten. Die Hindernisbreite b_h wird durch die Segmentierung berechnet. Geht man davon aus, dass das Fahrzeug sich mittig auf das Hindernis zu bewegt, dann berechnet sich die Ausweichbreite y wie folgt. Dies ist gleichzeitig die y-Koordinate des Zielpunktes:

$$y = \frac{b_h}{2} + \frac{b_f}{2} + a \qquad b_f = Fahrzeugbreite \tag{4.2}$$

Die x-Koordinate des Zielpunktes ergibt sich aus den Laserscanner Messwerten. Das SoC-Fahrzeug erreicht die Ausweichbreite durch Abfahren der Klothoidekurven (vgl. Abb. 4.5). Die Gleichung wird durch ein Polynom dritten Grades angenähert [2]:

$$y(x) = c_0 + c_1 x + c_2 \frac{x^2}{2} + c_3 \frac{x^3}{6} \tag{4.3}$$

Die Konstante c_0 entspricht der Anfangsabweichung von der Fahrzeugmitte und c_1 ist die Anfangssteigung. Für die detailierte Berechnung der Krümmungsänderungskonstanten c_2 und c_3 wird auf [2] und [12] verwiesen.

Wurde ein Objekt auf der Fahrspur detektiert und der MicroBlaze Controller hat das Ausweichmanöver eingeleitet, dann muss aufgrund der Laserscanner Frequenz von 10 Hz die Geschwindigkeit verringert werden. Ist dies nicht der Fall, wird die Aktualisierung der Zielkoordinaten durch die Segmentierung nicht garantiert. Bei einer Geschwindigkeit von 2 m/s und einer minimalen Abtastperiode von 100 ms werden die Zielkoordinaten alle 20 cm neu berechnet. Geht man davon aus, dass auf einer Geraden das Hindernis erst 100 cm vor Erreichen detektiert wird, dann werden die Zielkoordinaten fünf Mal aktualisiert. Des Weiteren sind Kurvenfahrten aufgrund des Scanwinkels von 90 Grad ein großer Risikofaktor, da die Hindernisse zu spät gesichtet werden und somit ein Ausweichen bei hoher Geschwindigkeit nicht mehr möglich ist (vgl. Abb. 4.6). Um diesen Risikofaktor zu minimieren und eine Kollision zu vermeiden, wird zusätzlich ein Bremsmanöver implementiert, das ebenfalls vom Controller MicroBlaze_1 eingeleitet wird, wenn das Hindernis in unmittelbarer Nähe zum Fahrzeug detektiert wird.

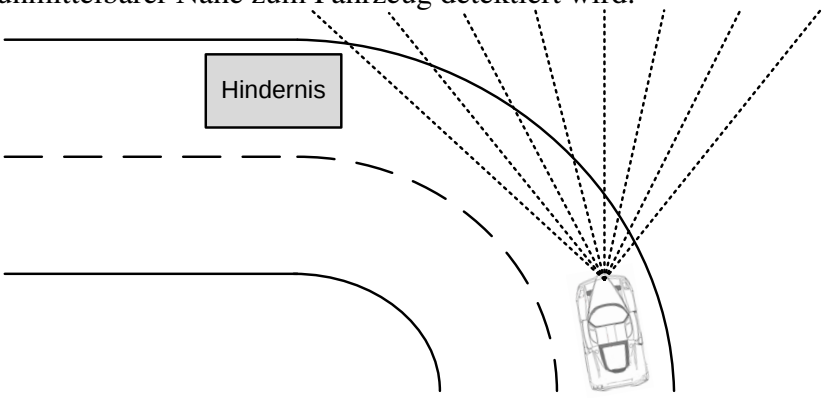


Abb. 4.6.: Nicht garantierte Detektion von Hindernissen während einer Kurvenfahrt

5. Dual-MicroBlaze Prozessor Konfiguration mit gemeinsamen Speicher

Im Vergleich zu einer Singlecore-Lösung mit erhöhter Taktfrequenz bieten FPGA-basierte MPSoCs eine Durchsatzsteigerung bei niedrigen Taktfrequenzen sowie die gleichzeitige Reduzierung des Energiebedarfs durch die Verringerung der Prozessorauslastung [44].

Der Einsatz von FPGA-basierten MPSoCs eignet sich vor allem für anwendungsspezifische und berechnungsintensive Aufgaben. Dies zeigt eine aktuelle Veröffentlichung des Fraunhofer-Instituts für Optronik, Systemtechnik und Bildauswertung, bei der die Performance und der Energiebedarf von einem MPSoC, einer GPU und einer CPU anhand einer Ultraschall-Computertomographie (USCT) und einer Objekterkennung verglichen wurde [22]. Die Ergebnisse zeigen, dass der mit 125 MHz getaktete MicroBlaze aufgrund der Hardware Parallelität eine schnellere Ausführungszeit bei der USCT Anwendung hat, als die mit 2,2 GHz getaktete AMD CPU (vgl. Tab. 5.1). Dieser Geschwindigkeitsvorteil lässt sich bei der Objekterkennung, trotz der drei MicroBlazes, nicht wiederholen. da die Nvidia GPU speziell für Grafikanwendungen ausgelegt ist und somit bei beiden Anwendungen die niedrigste Ausführungszeit hat. Jedoch hat die GPU auch den höchsten Energiebedarf und die niedrigste Energieeffizienz. Im Bereich von eingebetteten Multiprozessorsystemen ist ein hoher Durchsatz bei niedrigen Taktfrequenzen ebenso wichtig wie ein geringer Energiebedarf. Der für den Vergleich verwendete Virtex-4FX100 FPGA hat bei beiden Anwendung eine Ressourcenauslastung von ca 30 % [22].

Anwendung		Ausführungszeit	Energiebedarf	Energieeffizienz
USCT	CPU	150 μs	177 W	37 1/J
	GPU	3,23 μs	270 W	1147 1/J
	MPSoC	144 μs	3,61 W	1924 1/J
Objekterkennung	CPU	13,37 ms	168 W	0,45 1/J
	GPU	1,85 ms	230 W	2,35 1/J
	MPSoC	31,21 ms	2,25 W	14,24 1/J

Tab. 5.1.: Vergleich der Performance und des Energiebedarfs bei einem FPGA-basierten MPSoC, einer GPU und einer CPU, in einer USCT und einer Objekterkennung [22]

Eng gekoppelte Multiprozessoren sind asymmetrisch, wenn die Prozessoren unabhängig voneinander verschiedene Aufgaben bearbeiten. Durch einen Synchronisationsmechanismus wird der wechselseitige Zugriff auf gemeinsame Hardware Ressourcen gewährleistet. Da die Prozessoren unabhängig voneinander arbeiten, werden Ergebnisse und Statusinformationen über eine Interprozesskommunikation ausgetauscht, beispielsweise mit dem Einsatz des „Fast Simplex Link (FSL)“. Im Vergleich zu symmetrischen Multiprozessoren, die von einem zentralen Betriebssystem gesteuert werden, können bei asymmetrischen Multiprozessoren verschiedene Betriebssysteme implementiert werden. MPSoC's

bestehen vor allem im Bereich der multimedialen Anwendungen aus einer Vielzahl von IP-Blöcken, wozu sowohl Mikroprozessoren als auch Speicherelemente und die Kommunikationsinfrastruktur zählen. Bei Anwendungen mit vielen Kommunikationspartnern eignet sich die Vernetzung durch zentral gesteuerte Bussysteme nicht, da aufgrund der begrenzten Skalierbarkeit und der hohen Anzahl an globalen Kommunikationssignalen die zuvor gewonnene Parallelität durch den sequentiellen Buszugriff wieder serialisiert wird [37]. Aufgrund der sequentiellen Abarbeitung von Speicheranfragen durch den MPMC, entsteht in Folge von parallelen Speicherzugriffe ein Flaschenhals zwischen MPMC und externen Speicher. Dieser wird durch den Einsatz von lokalen BRAM Speicher und durch die Verwendung von Caches minimiert (vgl. Abb. 5.1).

5.1. Konzeptübersicht zum Dual-MicroBlaze System

Basierend auf zwei homogenen MicroBlazes, die an separate PLB Bussysteme angeschlossen sind und über den MPMC mit dem gemeinsamen Speicher gekoppelt sind, wird das MPSoC auf einem „Spartan-3A DSP“ FPGA implementiert (vgl. Abb. 5.1). Das Fahrspurführungssystem und die Geschwindigkeitsregelung wurden als Beschleuniger Modul in das MPSoC integriert, wobei durch die Kopplung mit PLB_1, Daten zur Parametrierung zwischen MicroBlaze_1 und dem IP-Block ausgetauscht werden. Durch die Asymmetrie der Prozessorarchitektur wurden die Aufgaben auf zwei MicroBlazes verteilt:

Prozessor	Aufgabenbeschreibung	Peripherie
MB_0	Steuerung des Laserscanners	UART Laser
	Empfang und Dekodierung von Abstandswerten	UART Laser
	Segmentierung und Transformation von Objekten	keine
	Segmente im gemeinsamen Speicher ablegen	MPMC, Mutex
	Steuerkommandos von MB_1 empfangen	FSL_{get}, LEDs
	Statusinformationen an MB_1 senden	FSL_{put}
MB_1	Steuerkommandos an MB_0 senden	FSL_{put}
	Statusinformationen von MB_0 empfangen	FSL_{get}
	Gemeinsamer Speicher auslesen	MPMC, Mutex
	Auswertung der Push Buttons	Push Buttons
	Statusausgabe auf RS232 Terminal	JTAG UART
	Parametrierung des Fahrspurführungs-IP	PLB

Tab. 5.2.: Aufgabenverteilung auf MicroBlaze_0 und MicroBlaze_1 (vgl. Kap. 8.1)

Die Objekterkennung auf MicroBlaze_0 wird durch ein FSL-Kommando von MicroBlaze_1 initiiert, wobei die von der Segmentierung detektierten Objekte im gemeinsamen Speicher abgelegt und über die FSL Kopplung dem Controller MicroBlaze_1 mitgeteilt werden (vgl. Kap. 8). Die nachfolgenden Kapitel erläutern sowohl die Konfiguration der MicroBlazes und deren Speicheranbindung über den MPMC als auch die Reduzierung der FPGA Ressourcen. Die für das MPSoC gewählten Parameter des MicroBlazes und des MPMCs sind in der MHS Datei B aufgelistet.

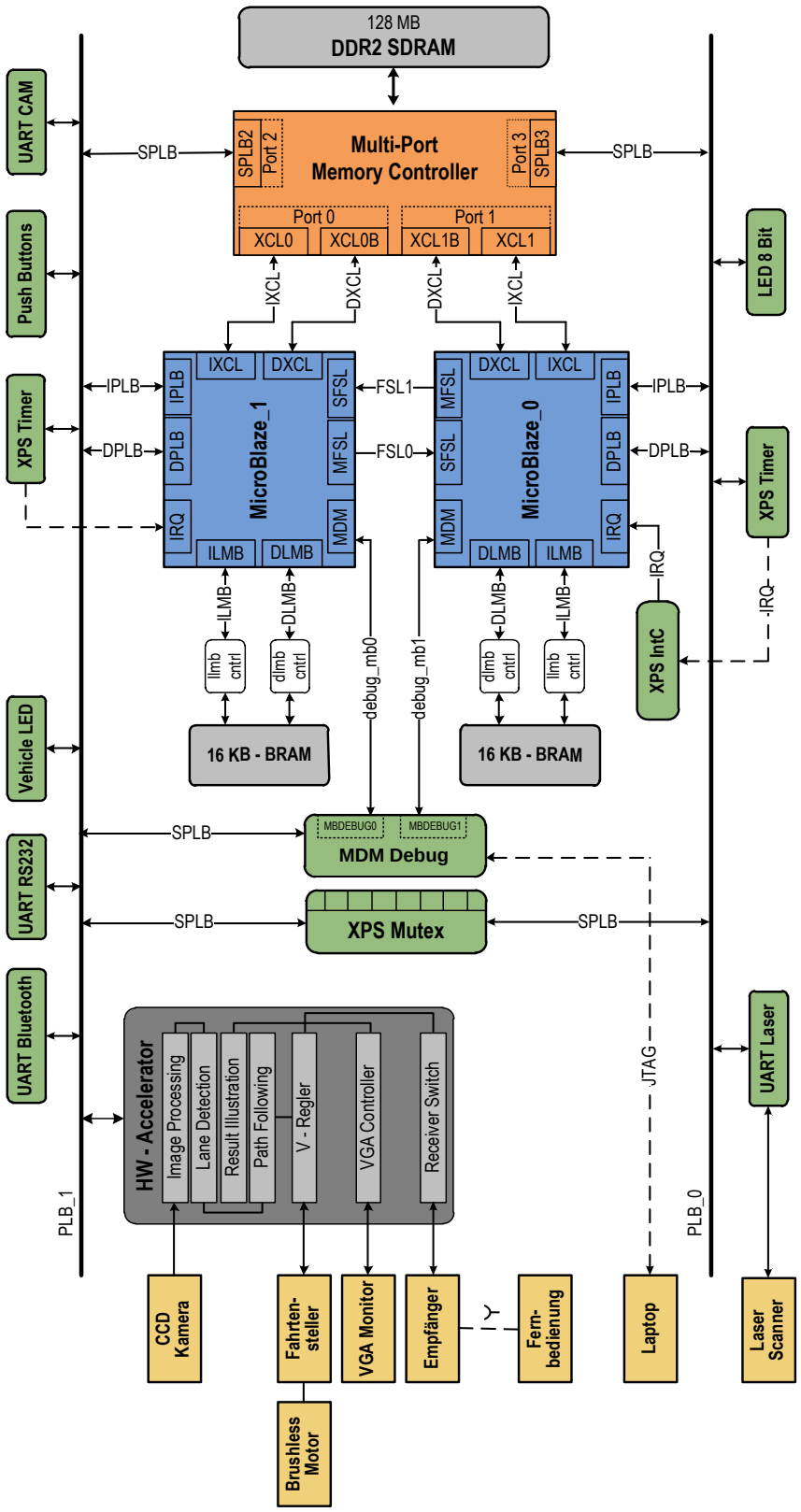


Abb. 5.1.: Dual-MicroBlaze MPSoC mit Shared Memory Architektur (vgl. Abb. 3.9). MicroBlaze_1 sendet abhängig von den Push Buttons Steuerkommandos über FSL_0 an MicroBlaze_0, der die Erfassung der Laserscandaten und die Segmentierung ausführt.

- **Multi-Port Memory Controller (MPMC)**: Für den Zugriff auf den gemeinsamen 128 MB großen DDR2-SDRAM Speicher werden die MicroBlazes über den „Xilinx Cache Link (XCL)“ mit dem MPMC gekoppelt, der acht konfigurierbare Input Ports (PIM) bietet. Die Zugriffsverwaltung übernimmt ein Arbiter, dessen Schema entweder nach dem Round-Robin oder nach einem prioritätenbasierten Verfahren arbeitet [92]. Jeder MPMC Port wird auf den gleichen Adressbereich des gemeinsamen Speichers gemapped, wobei der jeweils 8 Kilobyte große Daten- und Instruktionscache in MicroBlaze nahen BRAMs gespeichert wird. Des Weiteren ist der MPMC an beide PLB Bussystemen angeschlossen und bietet somit einen Zugang auf Speicherbereiche, die außerhalb der „Cache Area“ liegen. Die PLB Busse werden ausschließlich für Test- und Messvorgänge mit dem MPMC gekoppelt, da die Parameter *C_ICACHE_ALWAYS_USED* und *C_DCACHE_ALWAYS_USED* gesetzt sind (vgl. Kap. 5.3.1).
- **Lokaler BRAM**: Jeder MicroBlaze ist über zwei LMB Busse mit den ILMB- und DLMB BRAM Controller verbunden. Diese koppeln den MicroBlaze mit einem lokalen 16 KB großen BRAM Speicher, welcher für die Speicherung des Software-Bootsektors (*.vectors*) genutzt wird. Der „Local Memory Bus“ ist ein schneller synchroner „Single Master Bus“ zur Verbindung von durchsatzstarken Speicherbausteinen [83]. Aus diesem Grund wird der Software Heap und Stack ebenfalls im BRAM gespeichert.
- **XPS Mutex**: Zur Synchronisation von Zugriffen auf gemeinsame Speicherbereiche wird der Mutex IP eingesetzt, welcher den wechselseitigen Ausschluss gewährleistet und das System vor Deadlocks schützt. Durch die Kopplung der beiden PLB Bussysteme hat jede Komponente Zugriff auf den Mutex. Insgesamt verfügt der Mutex IP über acht PLB Interfaces und bietet eine maximal konfigurierbare Anzahl von 32 Memory Mapped Mutex Registern (vgl. Kap. 5.5)
- **XPS Timer**: Die Timer IPs sind als Peripheriekomponenten an die PLB Bussysteme angeschlossenen und werden bei der Software-Implementierung als Taktgeber für das Xilkernel RTOS verwendet. Des Weiteren werden sie zur Messung der Zugriffs- und Datentransferzeiten eingesetzt (vgl. Kap. 5.2.2).
- **XPS Uartlite**: Das MPSoC nutzt drei XPS Uartlite IPs zur seriellen Kommunikation mit dem Laserscanner, der Kamera und zur Übertragung von Daten an einen PC. Für das Bluetooth Telemetrie Modul wird ein zusätzlicher XPS Uart 16550 IP eingesetzt, der im Gegensatz zum Uartlite IP über Steuerleitungen für die CTS und RTS Signale zum Hardware-Handshake verfügt.
- **Debug Module**: Das „MDM Debug Module“ hat für das Software Debugging eine direkte Verbindung zu beiden MicroBlazes. Zusätzlich ist das MDM an PLB_1 angeschlossen und stellt darüber einen JTAG UART für MicroBlaze_1 zur Verfügung. Über die „XMD Konsole“ kann der Anwender auf beide Prozessoren zugreifen und diese mit TCL Kommandos steuern.
- **XPS GPIO**: Für die Software-Anwendungsbeispiele werden die „8 Bit LEDs“ und die „Push Buttons“ an PLB_0 und PLB_1 angeschlossen. Sie dienen der Signalisierung von Test- und Messergebnissen und werden zur Parametrierung und zur Steuerung der Laserscanner-basierten Objekterkennung eingesetzt (vgl. Kap. 8).

5.2. Parametrierung der MicroBlaze Konfiguration

Das homogene asymmetrische Multiprozessorsystem enthält zwei identisch konfigurierte MicroBlaze Prozessoren, die unterschiedliche Software Module ausführen und sich die zur Verfügung stehenden Hardware Ressourcen teilen. MicroBlaze_1 ist als Master Prozessor für die Steuerung und Parametrierung des SoC-Fahrzeugs zuständig. Die Laserscanner-basierte Objekterkennung wird auf dem zweiten MicroBlaze_0 ausgeführt und über eine direkte FSL Kopplung werden die detektierten Objekte dem Controller MicroBlaze_1 zur Entscheidungsfindung übermittelt. In diesem Kapitel wird die Konfiguration und die Parametrierung der MicroBlazes vorgestellt. Als Schwerpunkt gilt die Architektur der Caches und die Erhöhung des daraus resultierenden Durchsatzes. Der im „Xilinx Platform Studio„ integrierte „XPS Core Configurator“ stellt zur Konfiguration der MicroBlazes fünf vordefinierte Templates zur Verfügung:

- **Minimum Area**: Die MicroBlazes werden mit einer minimalen FPGA Ressourcenauslastung konfiguriert. Sowohl die Caches als auch die Debug Logik werden deaktiviert. Bei dieser Konfiguration werden keine BRAMs und keine DSP48A Slices genutzt, jedoch sinkt durch die Deaktivierung der Caches der Durchsatz der Prozessoren.
- **Maximum Performance**: Bei dieser Konfiguration werden 32 Kilobyte große Daten- und Instruktionscaches verwendet, wobei für den Instruktionscache zusätzlich acht „Victim Caches“ und ein „Stream Buffer“ instantiiert wird. Der Datencache nutzt die „Write Back Policy“ und hat ebenfalls acht „Victim Caches“. Durch die zusätzliche Aktivierung der „Floating Point Unit“ werden die auf dem Spartan-3A DSP FPGA verfügbaren BRAMs vollständig ausgenutzt. Aus diesem Grund ist dieses Template für die MPSoC Implementierung nicht anwendbar.
- **Maximum Frequency**: Die MicroBlazes werden mit der maximalen Systemfrequenz getaktet. Sowohl der Instruktionscache als auch der Datencache werden deaktiviert und es werden keine BRAMs verwendet. Jedoch bietet diese Konfiguration aufgrund der fehlenden Caches einen schlechteren Durchsatz als beispielsweise das „Maximum Performance“ Template.
- **Linux with MMU**: Mit 16 KB großen Caches und der Aktivierung der „Memory Management Unit“ ist dieses Template für den Einsatz in einem Linux-basierten System vorbereitet. Für den Instruktionscache werden acht „Victim Caches“ und ein „Stream Buffer“ instantiiert. Der Datencache nutzt die „Write Through Policy".
- **Typical**: Mit dieser Einstellung wird ein Kompromiss zwischen Performance, Taktfrequenz und Ressourcenbedarf gewählt. Die Caches sind 8 KB groß und nutzen keine „Stream Buffer“ und auch keine „Victim Caches“. Sowohl die MMU als auch die „Floating Point Unit“ ist deaktiviert.

Für die MPSoC Implementierung des SoC-Fahrzeugs wurde eine anwendungsspezifische MicroBlaze Konfiguration verwendet (vgl. MHS File B). In Tab. 5.3 sind die gewählten Konfigurationsparameter aufgelistet, wobei die einzelnen Komponenten in einer vorangegangenen Arbeit detailliert erläutert wurden [78]. Da in einem MPSoC die Prozessoren die meisten FPGA Ressourcen nutzen, wurde für die Spartan-3A DSP Konfiguration

der Bedarf an den 84 zur Verfügung stehenden BRAMs minimiert. Bei der Instantiierung eines MicroBlazes werden ausschließlich für die Daten- und Instruktionscaches, für die MMU und für die Sprungvorhersage (Branch Target Cache) Block RAMs verwendet [91]. Der zusätzliche BRAM Bedarf ist alleinig von den im MPSoC implementierten IP Blöcken abhängig, wie beispielsweise dem MPMC oder den anwendungsspezifischen Beschleuniger-Modulen (vgl. Tab. 9.5).

Komponente	Parameter	Konfiguration
Barrel Shifter	*C_USE_BARREL*	Aktiviert
Integer Multiplier	*C_USE_HW_MUL*	32 Bit Aktiviert
Integer Divider	*C_USE_DIV*	Deaktiviert
Floating Point Unit	*C_USE_FPU*	Deaktiviert
Machine Status Register	*C_USE_MSR_INSTR*	Aktiviert
Pattern Comparator	*C_USE_PCMP_INSTR*	Aktiviert
Pipeline	*C_AREA_OPTIMIZED*	5-Stufen
Branch Target Cache	*C_USE_BRANCH_TARGET_CACHE*	Deaktiviert
Processor Version Register	*C_PVR*	Deaktiviert
Memory Management Unit	*C_USE_MMU*	Deaktiviert
Bus Interface	*C_INTERCONNECT*	PLBv46
Debug Module	*C_DEBUG_ENABLED*	Aktiviert

Tab. 5.3.: Konfigurationsparameter für MicroBlaze_0 und MicroBlaze_1 [78]

Der „XPS Core Configurator“ bietet für die Implementierung von Bus- und Stream Interfaces einen automatischen Mechanismus, der die Instantiierung übernimmt. Bei Auswahl des PLB Busses als Bus Interface wird automatisch der XCL für den Cache Zugang verwendet. Als Stream Interface wird für die direkte Verbindung zwischen MicroBlaze_0 und MicroBlaze_1 der FSL Bus gewählt (vgl. Abb. 5.1). Da bei der Konfiguration stets ein Slave- und ein Master Interface instantiiert werden, benötigt jeder MicroBlaze lediglich ein „Stream Link“ [91].

5.2.1. MicroBlaze Daten- und Instruktionscaches

Das Ziel beim Einsatz eines Caches ist die Verringerung der Anzahl an Zugriffe auf den zu cachenden externen DDR2 Speicher. Jeder MicroBlaze wird mit einem „Direct-Mapped“ Daten- und Instruktionscache konfiguriert. Durch die MicroBlaze nahe BRAM Implementierung werden die Speicherzugriffszeiten minimiert und so der Durchsatz des Gesamtsystems erhöht. Bei der Entwicklung eines asymmetrischen MPSoCs besteht die Herausforderung in der Sicherstellung der Datencache Kohärenz, was bedeutet, dass ein Lesezugriff stets den Wert des letzten Schreibzugriffs liefern muss. Da die MicroBlazes einen gemeinsamen Speicher nutzen, können Daten im Cache gehalten werden, die gerade von einem anderen Prozessor geändert werden. Die aktuelle Version des MicroBlazes unterstützt keine hardwareseitige Cache Kohärenz und bietet auch kein entsprechendes

Protokoll. Aus diesem Grund muss entweder die „Cache Area“ der einzelnen Prozessoren so gewählt werden, dass keine Überlappung der Adressbereiche stattfindet und somit keine Inkohärenz Probleme auftreten oder es wird ein geeigneter Software Mechanismus implementiert [3]. Da durch die Asymmetrie der Prozessoren die Instruktionen nicht geteilt werden, bestehen die Inkohärenz Probleme lediglich beim Datencache. Erste Ansätze für die Implementierung eines Protokolls ist die Portierung des Snooping Verfahrens [49]. Hierbei beobachtet der Cache Controller den Bus hinsichtlich Speicherzugriffe und vergleicht die angeforderten Adressen mit denen in seinen Cache Zeilen. Tritt eine Übereinstimmung auf, ein sogenannter „Snoop Hit“, wird die betroffene Cache Zeile als ungültig markiert [75][32].

Die Daten- und Instruktionscaches der MicroBlazes werden bis auf die Adressen identisch konfiguriert. Über die „Cache Base Address“ und die „Cache High Adress“ wird der Adressbereich des externen Speichers in ein cachebares und ein nicht cachebares Segment geteilt (Direct Mapping). Wird der gesamte Adressbereich gecached, dann stimmen die Cache Adressen mit den Adressen des MPMCs überein und somit ist die „Cache Area“ ein Abbild des gesamten Speicherbereichs, wobei Kohärenzprobleme zwischen den Datencaches auftreten. Beim MPSoC des SoC-Fahrzeugs entstehen keine Inkohärenz Probleme, da die Cache Bereiche durch Anpassung der Adressen in getrennten Speicherregionen liegen und somit keine Überlappung stattfindet (vgl. Abb. 5.2). In Tab. 5.4 ist die aktuell gewählte Konfiguration der Caches von MicroBlaze_0 und MicroBlaze_1 erläutert, wobei die einzelnen Cache-Komponenten in vorangegangenen Arbeiten analysiert und detailiert beschrieben wurden [78].

Beschreibung	Cache Parameter	Konfiguration
Größe Instruktionscache	*C_ICACHE_BYTE_SIZE*	8 kB
Größe Datencache	*C_DCACHE_BYTE_SIZE*	8 kB
I-Cache Zeilenbreite	*C_ICACHE_LINE_LEN*	8 Wort
D-Cache Zeilenbreite	*C_DCACHE_LINE_LEN*	8 Wort
IXCL anstatt PLB verwenden	*C_ICACHE_ALWAYS_USED*	Aktiv
DXCL anstatt PLB verwenden	*C_DCACHE_ALWAYS_USED*	Aktiv
D-Cache Schreibstrategie	*C_DCACHE_USE_WRITEBACK*	Inaktiv
Stream Buffer	*C_ICACHE_STREAMS*	0
Victim Caches	*C_ICACHE_VICTIMS*	0
I-Cache Startadresse MB_0	*C_ICACHE_BASEADDR*	0x88000000
D-Cache Startadresse MB_0	*C_DCACHE_BASEADDR*	0x88000000
I-Cache Startadresse MB_1	*C_ICACHE_BASEADDR*	0x8A000000
D-Cache Startadresse MB_1	*C_DCACHE_BASEADDR*	0x8A000000
I-Cache Endadresse MB_0	*C_ICACHE_HIGHADDR*	0x89FFFFFF
D-Cache Endadresse MB_0	*C_DCACHE_HIGHADDR*	0x89FFFFFF
I-Cache Endadresse MB_1	*C_ICACHE_HIGHADDR*	0x8BFFFFFF
D-Cache Endadresse MB_1	*C_DCACHE_HIGHADDR*	0x8BFFFFFF

Tab. 5.4.: Konfiguration der Daten- und Instruktionscaches auf beiden MicroBlazes [78]

Ein im MicroBlaze aktivierter Cache wird genutzt, wenn er zusätzlich in Software mit *Xil_DCacheEnable()* und *Xil_ICacheEnable()* aktiviert wird. Hierfür wurde auf beiden MicroBlazes die Funktion *init_platform()* implementiert (vgl. Tab. 8.1).

Das Funktionsprinzip des Datencaches unterscheidet sich in der Schreibstrategie. Hierfür gibt es zwei Varianten, die durch *C_DCACHE_USE_WRITEBACK* bestimmt werden:

- **Write-Back:** Bei einem Schreibvorgang auf eine im Cache gespeicherte Adresse werden die Daten nicht sofort in den externen Speicher zurückgeschrieben, sondern lediglich der Cache wird aktualisiert. Die Daten werden erst nach dem Ersetzen der Cache Zeile zurückgeschrieben, sodass aufgrund der veralteten Daten im externen Speicher eine Inkohärenz zwischen den Datencaches entsteht [20]. Für eine Multiprozessorumgebung ist dieses Verfahren aufgrund den Inkonsistenzen nicht vorteilhaft, da ein Protokoll zur Vermeidung implementiert werden muss [49][56].
- **Write-Through:** Bei dieser Strategie werden bei einer Schreibanfrage sowohl die im Cache gespeicherten Daten als auch die Daten im externen Speicher aktualisiert. Bei einem „Cache Miss“ wird die fehlende Cache Zeile nicht in den Datencache geladen. Die Write-Through Strategie wird bei „General Purpose Computer“ aufgrund der hohen Busbelastung nur selten eingesetzt [32].

Für das SoC-Fahrzeug wurden die Caches der beiden MicroBlazes identisch konfiguriert (vgl. MHS File B). Als Schreibstrategie wird die „Write-Through Policy“ verwendet, da durch die stetige Aktualisierung der Daten keine Inkohärenzen auftreten. Die Verwendung der „Write-Through Policy“ hat zur Folge, dass keine Victim Caches instantiiert werden. Bei der Software-Implementierung muss darauf geachtet werden, dass sowohl der Instruktionscache als auch der Datencache explizit aktiviert werden. Ist dies nicht der Fall, wird die Ausführungszeit erhöht (vgl. Kap. 5.2.2).

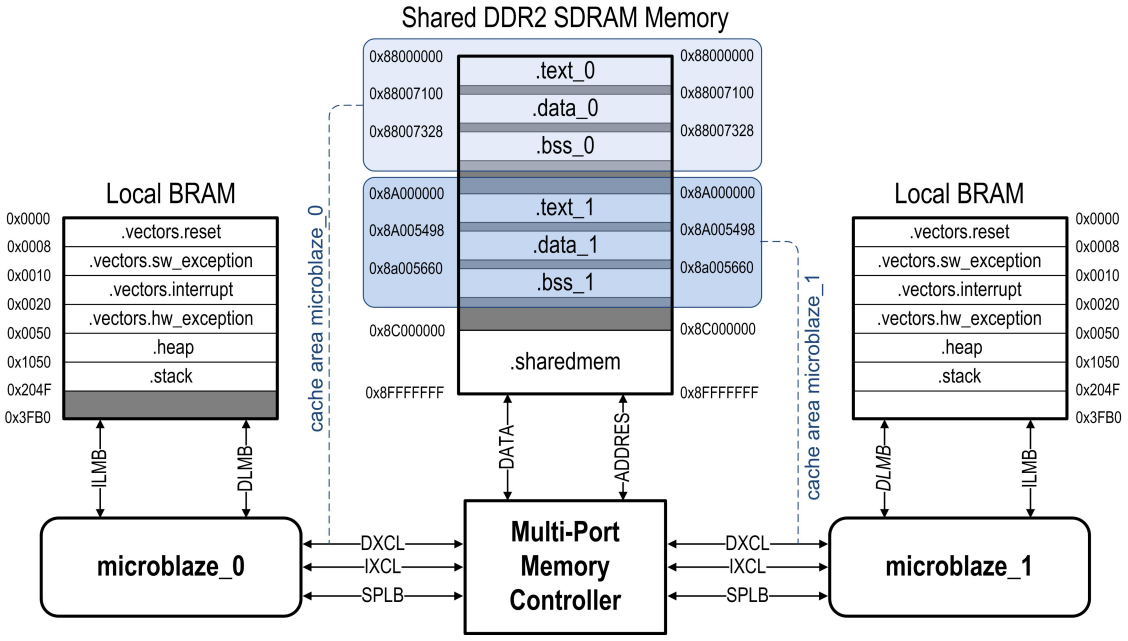


Abb. 5.2.: Memory Mapping der Dual-MicroBlaze Prozessor Konfiguration mit lokalem und externem Speicher. Getrennte Cache Bereiche zur Vermeidung von Inkohärenzen. Der bidirektionaler IXCL dient zur Cache Validierung und zum Leeren des Caches

5.2.2. Auswirkung der Caches auf die Ausführungsintervalle

Die Anzahl der Taktzyklen, die bei der Ausführung einer Funktion oder bei der Speicherung von Variablen benötigt wird, ist hauptsächlich vom Speicherort der Software und von der Benutzung von Caches abhängig. Zur Analyse dieser Zeiten wurde eine Software implementiert, die die benötigten Taktzyklen mit Hilfe eines „XPS Timers“ zählt. Hierbei wurde das Standalone Betriebssystem verwendet. Nach der Timer Initialisierung wird vor jeder Funktionsausführung der Timer gestartet und direkt danach wieder gestoppt. Die gezählten Zyklen werden mit *XTmrCtr_GetValue* gelesen und anschließend über den UART ausgegeben. Der Codeausschnitt 5.1 zeigt exemplarisch die Messung für eine *putfslx()* Funktion. Zur Durchschnittsermittlung wurde jede Messung fünf Mal ausgeführt.

```
XCACHE_ENABLE_ICACHE();
XCACHE_ENABLE_DCACHE();

for(i=0; i<3; i++){
                XTmrCtr_Start(&timer, 0);
                putfslx(0,FSL_SLOT_ID, FSL_NONBLOCKING);
                XTmrCtr_Stop(&timer,0);
                timer_cycle = XTmrCtr_GetValue(&timer,0);
                xil_printf("FSL PUT:\t Measure Clock Cycles with Caches: \t%d \r\n", timer_cycle);
        }

XCACHE_DISABLE_ICACHE();
XCACHE_DISABLE_DCACHE();
```

Listing 5.1: Codeauszug zum Messen der Taktzyklen bei aktiven und inaktiven Caches

Bei der ersten Messung war die *.data* und die *.bss* Sektion im lokalen On-Chip BRAM gespeichert. Nur die *.text* Sektion, welche den ausführbaren Code enthält, war im DDR2 Speicher abgelegt. Eine Sektionierung im BRAM hat den Vorteil, dass durch eine bessere Zugriffszeit ein erhöhter Durchsatz erreicht wird. Jedoch hat dies eine Begrenzung der Sektionsgröße zur Folge. Bei der zweiten Messung wurden alle Sektionen im externen DDR2 SDRAM abgelegt. Hierbei erfolgt der Speicherzugriff entweder über den XCL oder den PLB Bus. Um die Auswirkung der Caches zu analysieren, wurde jede Messung sowohl bei aktivierten als auch bei deaktivierten Caches durchgeführt. Die Aktivierung der Caches erfolgt mit den Makros *XCACHE_ENABLE_ICACHE()* und *XCACHE_ENABLE_DCACHE()*. Bei aktivierten Caches und einer Speicherung im BRAM ist der Zugriff um ein zehnfaches schneller als bei inaktiven Caches (vgl. Tab. 5.5).

Des Weiteren wurden die Taktzyklen beim Speichervorgang von unterschiedlichen Datentypen gemessen. Hierfür wurden die Daten wiederum sowohl im BRAM als auch im DDR2 SDRAM gespeichert. Lokale Variablen werden erst beim Funktionsaufruf erstellt und der Speicherplatz wird nach Verlassen der Funktion wieder freigegeben. Sie werden auf dem Stack abgelegt. Hingegen sind globale Variablen von Beginn an verfügbar und überall sichtbar. Messungen ergaben, dass der Unterschied beim Schreiben auf lokale bzw. globale Variablen nahezu identisch ist. Aus diesem Grund sind die Messungen in Tabelle 5.6 sowohl auf lokale als auch auf globale Variablen bezogen.

Während den Messungen sind alle anderen IP Blöcke inaktiv und es findet kein zusätzlicher PLB Transfer, der die Messung verfälschen könnte, statt. Bei einer Taktfrequenz von 62,5 MHz berechnet sich die Ausführungszeit wie folgt:

$$Zeit(ms) = \frac{Takte}{Frequenz} = \frac{TimerTakte}{62.500kHz} = Millisekunden \tag{5.1}$$

Bei der Messung des MPMCs wurden drei 32 Bit breite Arrayfelder an eine vorgegebene Adresse geschrieben. Anschließend wird mit einer „Read“ Funktion die gleichen Adressen gelesen und in einem Array gespeichert. Eine Übereinstimmung der geschriebenen Daten zu den Gelesenen muss stattfinden. Dies wird mit einer UART Ausgabe überprüft.

```
0 #define WRITE_MEMORY(ADDR, DATA) (*(U32Ptr)(ADDR) = (DATA))          // MPMC Write an eine bestimmte Adresse
  #define READ_MEMORY(ADDR, DATA)  ((DATA) = *(U32Ptr)(ADDR))         // MPMC Read von einer bestimmten Adresse
```

Listing 5.2: Makrofunktionen zum Lesen und Schreiben über den MPMC

IP	.data & .bss Speicherort	Funktion	Mit Caches		Ohne Caches	
			Ø Takte	Dauer / μs	Ø Takte	Dauer / μs
FSL	BRAM	getfslx()	68	1,046 μs	758	11,661 μs
		putfslx()	58	0,892 μs	668	10,277 μs
	DDR2	getfslx()	245	3,769 μs	886	13,630 μs
		putfslx()	232	3,569 μs	794	12,215 μs
Mutex	BRAM	Lock()	115	1,769 μs	1120	17,231 μs
		Unlock()	117	1,800 μs	1210	18,616 μs
	DDR2	Lock()	294	4,523 μs	1300	20,000 μs
		Unlock()	298	4,585 μs	1410	21,692 μs
MPMC	BRAM	Write	71	1,092 μs	795	12,231 μs
		Read	192	2,954 μs	778	11,969 μs
	DDR2	Write	267	4,108 μs	940	14,461 μs
		Read	266	4,092 μs	922	14,184 μs

Tab. 5.5.: Messung von Funktions-Ausführungszeiten bei aktivierten und deaktivierten Caches. Speicherung der *.data* und der *.bss* Sektion im BRAM und DDR2.

Datentyp	.data & .bss Speicherort	Mit Caches		Ohne Caches	
		Ø Takte	Dauer / μs	Ø Takte	Dauer / μs
Xuint32	BRAM	58	0,892 μs	668	10,277 μs
	DDR2	234	3,600 μs	796	12,246 μs
Xuint16	BRAM	58	0,892 μs	670	10,308 μs
	DDR2	236	3,630 μs	796	12,246 μs
Xuint8	BRAM	76	1,169 μs	668	10,277 μs
	DDR2	234	3,600 μs	795	12,231 μs
Double	BRAM	61	0,938 μs	724	11,138 μs
	DDR2	242	3,723 μs	850	13,077 μs

Tab. 5.6.: Messung der Schreib-Zugriffszeiten bei aktiven und inaktiven Caches. Verschiedene Datentypen wurden per Linker Script in unterschiedliche Speicher abgelegt.

5.3. Shared Memory Architektur

Bei einer „Shared Memory“ Architektur greifen alle Prozessoren sowohl lesend als auch schreibend auf den gleichen Speicher zu. Der „Multi-Port Memory Controller“ koppelt die MicroBlazes und die PLB Bussysteme mit dem externen DDR2 Speicher (vgl. Abb. 5.1). Für den globalen Zugriff wird ein gemeinsamer „Direct-Mapped“ Adressraum zur Verfügung gestellt. Eine derartige „Uniform Memory Access (UMA)“ Architektur bietet einen identischen Speicherzugang, wobei die Latenz und die Zugriffsart auf den Speicher bei allen MicroBlazes identisch ist. Durch die vorgelagerten Caches ist eine Vorhersage des Durchsatzes aufgrund der unterschiedlichen Cache Inhalten nur schwer realisierbar [20]. Gemeinsame Speicher bieten den Vorteil der effizienten Speicherausnutzung und der schnellen Übertragung von großen Datenmengen. Jedoch muss zur Sicherstellung des wechselseitigen Ausschlusses kritische Speicherbereiche, die von beiden Prozessoren manipuliert werden, mit einem Mutex geschützt werden (vgl. Abb. 5.1). Ein weiteres Problem, dass bei UMA basierten MPSoC Architekturen auftritt, ist der „Speicher Flaschenhals“. Wenn beide MicroBlazes gleichzeitig Anfragen an den MPMC senden, werden diese nur sequentiell abgearbeitet und nur einem Prozessor wird der Zugriff auf den externen Speicher gewährt. Folgende Anforderungen gelten für ein „Shared Memory“ MPSoC [42]:

- **Speicher Kohärenz**: Da ein gemeinsamer Adressraum genutzt wird, muss jeder Prozessor stets ein aktuelles Abbild des Speichers haben. Dies erfordert die Trennung der Cache Bereiche oder die softwareseitige Implementierung eines Kohärenz-Protokolls. Der Nachteil hiervon ist der erhöhte Kommunikationsaufwand und die damit steigende Latenz.
- **Atomare Operationen**: Daten dürfen nur von einem Prozessor zur gleichen Zeit verändert werden. Für diesen exklusiven Schreib- und Lesezugriff stehen atomare Funktionen zur Verfügung, wie beispielsweise die Sperrfunktion des Mutexes. Eine Anfrage an ein bestimmtes Datum wird stets durch eine Antwort bestätigt. Erst dann wird dem zweitem Prozessor der Zugriff auf dieses Datum gewährt.
- **Software Modell**: Jedem MicroBlaze stehen bestimmte Funktionen zur Initialisierung, Kommunikation und Synchronisation der einzelnen Hardware Komponenten zur Verfügung. Diese Funktionen müssen in der gesamten Multiprozessorumgebung global erreichbar sein.

Da in einem MPSoC der Boot Bereich *.vectors* der Software nicht im gemeinsamen externen Speicher liegen darf, besitzt jeder MicroBlaze zusätzlich einen lokalen 16 KB großen BRAM Speicher. Der „Local Memory Bus (LMB)“ koppelt über die „ILMB- und DLMB BRAM Controller“ den On-Chip BRAM mit dem MicroBlaze (vgl. Abb. 5.1). Durch die Asymmetrie der Prozessorarchitektur hat jeder MicroBlaze ein eigenes ausführbares Programm, welches bei Systemstart aus dem lokalen BRAM gestartet wird. Sowohl der Heap als auch der Stack werden aufgrund des schnellen Zugriffs ebenfalls im lokalen BRAM abgelegt. Der ausführbare Programmcode *.text* und die initialisierten Konstanten *.data* werden per Linker Script im externen DDR2 Speicher angelegt (vgl. Abb. 5.2). Eine Partitionierung der Software hat den Vorteil, dass die lokalen BRAMs mit einer minimalen Größe von 16 Kilobyte implementiert werden. Die Speicherung der kompletten Software im BRAM hätte eine Begrenzung der Codegröße zur Folge.

5.3.1. Konfiguration des Multi-Port Memory Controllers

Der MPMC dient als DMA-Controller für den direkten Zugriff auf den gemeinsamen Adressbereich im externen DDR2 Speicher. Jeder MicroBlaze Cache ist über den IXCL und den DXCL mit dem MPMC verbunden. Über diese Verbindungen leitet der Cache Controller die Speicherzugriffe, deren angeforderte Adresse innerhalb des zu cachenden Bereichs liegt, an den MPMC weiter. Zusätzlich wird der MPMC zu Testzwecken mit beiden PLB Bussysteme gekoppelt (vgl. Abb. 5.1). Bei allen Adressen, die außerhalb der „Cache Area“ liegen, leitet der Cache Controller die Anfrage an das entsprechende PLB Interface weiter. Für das MPSoC wurde sowohl *C_ICACHE_ALWAYS_USED* als auch *C_DCACHE_ALWAYS_USED* gesetzt, was bedeutet, dass alle Speicherzugriffe über die XCL Busse stattfinden. Eine Kopplung der PLB Busse wird nicht mehr benötig. Dies ist vor allem bei Multiprozessorsystemen vorteilhaft, die mehr als 4 MicroBlazes besitzen., da die insgesamt acht zur Verfügung stehende „Personality Interface Modules (PIM)“ direkt mit den Caches verdrahtet werden können (vgl. Abb. 5.3).

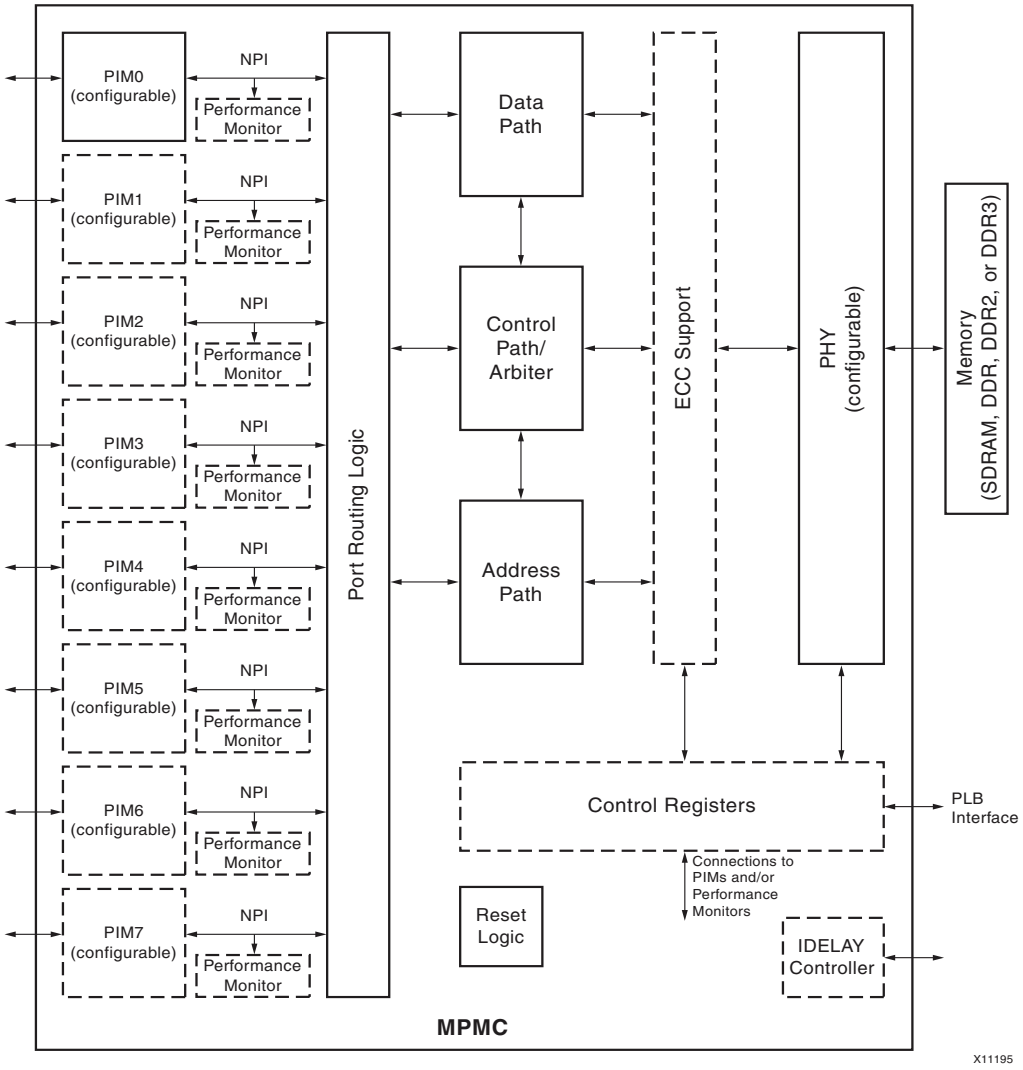


Abb. 5.3.: Multi-Port Memory Controller Blockdiagramm [92]

Zur weiteren Ressourceneinsparung und zur Analyse wurde bei der MPMC Konfiguration die Parameter *C_XCL0_B_IN_USE* und *C_XCL1_B_IN_USE* für Port 0 und Port 1 gesetzt. Hiermit wird für den entsprechende MPMC Port ein zweiter XCL Bus instantiiert und die MicroBlaze IXCL und DXCL Busse werden direkt auf einen gemeinsamen MPMC PIM geroutet (vgl. Abb. 5.1). Das MPMC interne XCL PIM arbitriert zwischen den beiden Bussen nach einem Round-Robin Verfahren [92]. Die Aktivierung der Parameter hat den Vorteil, dass bis zu 8 MicroBlazes mit insgesamt 16 XCL Busse an den MPMC angeschlossen werden können. Dadurch wird der langsamere und sequentiell arbeitende PLB Transfer vermieden. Die XCL Ports arbeiten nach einem Pipeline Verfahren, wobei die Anzahl der Stufen über den Parameter *C_XCL0_PIPE_STAGES* konfiguriert wird. Für das MPSoC wurde eine 2-stufige Pipeline implementiert, welche durch ein zusätzliches Synchronisationsregister eine Verbindung von unterschiedlichen „Clock Domains“ zulässt [92]. Alle acht PIM Ports werden individuell konfiguriert.

Sobald mehrere MicroBlazes sich einen gemeinsamen Speicher teilen, müssen die Zugriffe koordiniert werden. Der MPMC ist über einen Daten- und einen Adressbus mit dem externen Speicher verbunden. Die Reihenfolge der Zugriffe wird durch eine Slot-Arbitrierung festgelegt. Während des Idle-Zustandes werden bei jedem Takt die Slots inkrementiert. Sind jedoch aktive Anfragen in der FIFO, dann werden die Slots nach einem erfolgreichen Zugriff inkrementiert. Der Arbitrierungs-Algorithmus des MPMCs wird über *C_ARV0_ALGO* eingestellt und bietet drei Konfigurationen [92]:

- **Fixed**: Diese Arbitrierung hat ein festes Prioritätenscheduling und nutzt am wenigsten FPGA Ressourcen. Jeder PIM Port hat entsprechend seiner ID eine feste aufsteigende Priorität (Port0 = höchste Priorität). Der Nachteil einer festen Prioritätenvergabe ist, dass Anfragen über niedrigere Portnummern stets bearbeitet werden und andere Komponenten warten müssen.
- **Round Robin**: Der Standardalgorithmus vergibt die Zugriffe nach einem Rotationsverfahren, wobei jeder PIM Port gleichberechtigt ist. Ein Port der gerade den Zugriff auf den Speicher gewährt bekommen hat, wird in seiner Priorität abgestuft. Damit ist sichergestellt, dass die Anfragen jeder Komponente bearbeitet werden. Jedoch kann es durch die fehlende Bevorzugung vorkommen, dass ein Master Prozessor aufgrund eines langsamen Speicherzugriffes auf seine Antwort warten muss.
- **Custom**: Für echtzeitfähige Systeme ist ein Zeitmultiplexverfahren von Vorteil. Die „Custom“ Arbitrierung bietet eine anwenderspezifische Konfiguration der Zeitslots. Jedem Port wird für eine bestimmt Zeit der Zugang zum Speicher gewährt, wobei insgesamt 16 Timeslots zur freien Verfügung stehen. Der Vorteil dieser Konfiguration ist die Vorhersehbarkeit der Zugriffe.

In der für das SoC-Fahrzeug eingesetzten MPSoC Konfiguration nutzt der MPMC die Round-Robin Arbitrierung und hat einen Bedarf an 17 von 84 BRAM Blöcken, der hauptsächlich auf die im BRAM implementierten FIFOs zurückzuführen ist. Für jeden PIM Port wird eine Lese- und eine Schreib FIFO zum Puffern der Daten aus dem externen Speicher implementiert. Die FIFOs können getrennt voneinander deaktiviert oder wahlweise in LUTs implementiert werden. Jedoch wird aufgrund der langsameren Zugriffszeiten und dem daraus folgenden schlechteren Timing, die BRAM Implementierung bevorzugt. Zusätzlich zu den BRAM FIFOs nutzt der MPMC noch ein BRAM für die

Zustandsmaschine [92]. Weitere BRAMs werden genutzt, wenn eine angepasste Arbitrierung oder optionale MPMC Features eingesetzt werden, wie der „Error Correction Code Support“ oder dem „Performance Monitor“ (vgl. Abb. 5.3).

Die MicroBlazes können den Adressraum auf unterschiedliche Weise interpretieren. Im „Real Mode“ werden absolute Adressen für den direkten Zugang zum Speicher verwendet. Dies ist die Standardkonfiguration und wird für das implementierte MPSoC eingesetzt. Im „Virtual Mode“ werden die absoluten Adressen durch eine „Virtual-Memory Management Unit“ des Prozessors in eine physikalische Adresse übersetzt. Hierfür enthält jede Adressen eine „Page Number“ und einen „Offset“. Die softwareseitige „Page-Translation Table“ mapped die virtuellen Adresse auf eine Adresse im externen Speicher und häufig verwendeten Adressen werden im „Translation Lookaside Buffer“ gespeichert“ [91]. Da der „Virtual Mode“ den Einsatz der MMU erfordert, wird für das SoC-Fahrzeug der „Real Mode“ verwendet.

5.3.2. MPMC Implementierung mit Spartan-3A DSP Ressourcen

Für die Analyse und die Auswertung der genutzten Ressourcen des „Spartan 3A DSP“ FPGAs wurden verschiedene Xilinx Tools verwendet. Der vom EDK erstellte „Design Summary“ gibt einen textuellen Überblick über die Ausnutzung der Ressourcen. Das genaue Mapping des Systems auf die physikalisch implementierten Ressourcen wird im MAP Report erläutert. Für eine detailliertere hierarchische Analyse wird das „Plan Ahead“ Tool und der „Xilinx XPower Analyzer“ eingesetzt. Die berechnete Ausnutzung der einzelnen Reports und Tools sind teilweise unterschiedlich. Dies liegt daran, dass in einer hierarchischen Komponenten Ansicht die verwendeten Slices und LUTs je nach Hierarchieebene doppelt gezählt werden. „Ressource Sharing“ wird hierbei nicht berücksichtigt. Die Tabelle 5.7 zeigt, dass der MPMC als zentrale Komponente einen großen Einfluß auf den Gesamtbedarf an FPGA Ressourcen hat (vgl. Kap. 9.4). Durch die Implementierung der FIFOs in BRAMs, werden 20 % der zur Verfügung stehenden BRAM Blöcke verwendet. Eine Implementierung der FIFOs in LUT hätte zur Folge, dass aufgrund der schlechteren Speicherzugriffszeiten der Durchsatz sinkt.

MPMC Komponente	Slice FF	LUTs	BRAMs	DSP48As
2 x Dual XCL Ports	478	580	0	0
2 x PLB Ports	828	550	0	0
MPMC Adress Path	147	72	0	0
MPMC Control Path	32	45	1	0
Arbiter	74	71	0	0
Read FIFOs	540	248	8	0
Write FIFOs	606	62	8	0
Gesamtbedarf	3.515	2.221	17	0
Verfügbar	33.280	33.280	84	84
Ausnutzung	10 %	6 %	20 %	0 %

Tab. 5.7.: Hierarchischer MAP Report für den „Multi-Port Memory Controller“.

5.4. Interprozessorkommunikation mit FSL Bussen

Der unidirektionale „Fast Simplex Link (FSL)“ wird für die direkte Verbindung zwischen den MicroBlazes eingesetzt (vgl. Abb. 5.1). Er arbeitet nach einem FIFO-basierten Master-Slave Prinzip, wobei der Master Daten in die FIFO schreibt und der Slave die Daten liest. Die Breite des FSL Datenbusses beträgt 32 Bit und wird automatisch vom XPS berechnet. Aufgrund der Unidirektionalität werden für die wechselseitige Kommunikation zwei FSL Busse implementiert. Über den „XPS Core Configurator“ können für jeden MicroBlaze bis zu 16 „Stream Link Interface“ Paare aktiviert werden. Jedes Paar instantiiert automatisch ein Master- und ein Slave Interface, woraus folgt, dass bei einer MPSoC Implementierung bis zu 16 MicroBlazes direkt über zwei FSL Busse miteinander kommunizieren können. Im Vergleich zu der „XPS Mailbox“ bietet der FSL durch den direkten MicroBlaze Register Zugriff eine geringere Latenz. In vorherigen Arbeiten wurden die Zugriffszeiten und die wechselseitigen Ausführungszeiten der Mailbox gemessen. Die erarbeiteten Analysen ergaben, dass sich die Mailbox aufgrund der hohen Zugriffszeiten und des erhöhten Ressourcenbedarfs nicht für das im SoC-Fahrzeug eingesetzte Dual-MicroBlaze System eignet [78]. Gerade für die Interprozesskommunikation zwischen zwei Prozessoren eignet sich der Einsatz des FSLs aufgrund der nicht benötigten Arbitrierung und dem damit erhöhten Durchsatz.

Bei der Implementierung des FSL Busses wird über *C_ASYNC_CLKS* der Modus der interne FIFO entweder auf synchron oder asynchron festgelegt. Bei einer asynchronen Betriebsart werden für Master und Slave unterschiedliche Taktfrequenzen verwendet, was eine Belegung der Ports *FSL_M_CLK* und *FSL_S_CLK* erfordert. Für eine synchrone FIFO wird *FSL_CLK* sowohl für den Master als auch den Slave verwendet [87]. Das implementierte MPSoC verwendet synchrone FSL FIFOs und arbeitet bei einer Taktfrequenz von 62,5 MHz.

Die maximale Tiefe der FIFO ist abhängig von *C_ASYNC_CLKS* und dem Implementierungsschema *C_IMPL_STYLE*. Letzteres gibt an, ob die FIFO im BRAM oder in LUTs implementiert wird (vgl. Tab. 5.8). Das implementierte MPSoC arbeitet bei einer 32 Bit Wortbreite mit einer FIFO Tiefe *C_FSL_DEPTH* von 16 Wörtern. Die markierte Zeile kennzeichnet die gewählte Konfiguration:

C_ASYNC_CLKS	*C_IMPL_STYLE*	FIFO Minimum	FIFO Maximum
0	**0**	**1**	**8192**
0	1	1	8192
1	0	16	128
1	1	512	8192

Tab. 5.8.: Maximale und Minimale FSL FIFO Tiefe bei einer Abhängigkeit von der Betriebsart und dem Implementierungsschema [87].

Durch den Parameter *flags* in den vordefinierte Makrofunktionen *putfsl(val, slotid, flags)* und *getfsl(val, slotid, flags)* wird der durch die Slot-ID definierte FSL Link entweder blockierend oder nicht blockierend verwendet. Des Weiteren steht zur Validierung der Gültigkeit des empfangenen Datums die Funktion *fsl_isinvalid(invalid)* und *fsl_iserror(error)* zur Verfügung [88].

5.5. Synchronisation des gemeinsamen Speichers mit dem XPS Mutex

Durch den XPS Mutex IP wird der wechelseitige Ausschluss der Prozessoren beim Zugriff auf einen gemeinsamen Speicher gewährleistet. Er schützt kritische Speicherbereiche indem er bei nebenläufigen Prozesse die Zugriffe koordiniert. Jeder Mutex bietet eine maximal konfigurierbare Anzahl von 32 Memory Mapped Mutex Registern, wobei jedes Register 32 Bit groß ist und über ein LOCK Bit, acht CPUID Bits und 23 Reserved Bits verfügt [85][82]:

- Das LOCK Bit repräsentiert den Zustand des Mutexes (0 = frei und 1 = blockiert)
- Das CPUID Feld dient der Überprüfung welcher Prozessor welchen Mutex gerade gesperrt hat. Bei einem Sperrvorgang wird die entsprechende CPUID des MicroBlazes in das vorgesehen CPUID Feld geschrieben.
- Ein HWID Feld stellt einen zusätzlichen Sicherheitsmechanismus bereit, der vollkommen versteckt vom Anwender arbeitet und der sowohl die PLB Master ID als auch den PLB Port an dem der Prozessor angeschlossen ist, beinhaltet.

Zu jedem Mutex Register wird optional ein zugehöriges 32 Bit großes „User Configuration Register“ instantiiert. Dieses speichert einen bis zu 32 Bit großen Wert, der beispielsweise die Adresse des geschützten Speicherbereiches repräsentiert. Die Problematik bei der Aktivierung des Registers in Bezug auf den Ressourcenbedarf besteht darin, dass zu jedem Mutex Register ein zusätzliches 32 Bit Register erzeugt wird. Dies kann nicht explizit für jeden einzelnen Mutex ausgewählt werden, sondern nur global für alle aktiviert werden [85]. Zur Reduzierung der Spartan-3A DSP Ressourcen wird dieses Register in der aktuellen SoC-Fahrzeug Konfiguration deaktiviert.

Zum Sperren und Entsperren des Mutexes verwendet MicroBlaze_0 und MicroBlaze_1 die vordefinierten Makrofunktionen aus der *xmutex.h*:

- *XMutex_Lock (&mutex, MUTEX_NUM)*
- *status = XMutex_Unlock (&mutex, MUTEX_NUM)*

Die Software für die Laserscanner-basierte Objekterkennung auf MicroBlaze_0 verwendet den Mutex, um erkannte Segmente in den geteilten Speicherbereich abzulegen (vgl. Kap. 8). Der Controller MicroBlaze_1 liest diese Segmente nach Empfang der Statussignale über den FSL und bestimmt das von der Fahrspurerkennung abhängige nächst relevanteste Segment. Somit wird sichergestellt, dass ausschließlich aktuelle Segmente in die Bewertung mit einbezogen werden. Anderenfalls würde der Controller MicroBlaze_1 ein Ausweichmanöver einleiten, welches sich auf ein Segment in der Vergangenheit bezieht.

Der Mutex Core im MPSoC des SoC-Fahrzeugs koppelt den PLB_0 mit dem PLB_1 (vgl. Abb. 5.1). Insgesamt verfügt der IP Block über acht PLB Interfaces, die jeweils auf alle Mutex Register zugreifen, jedoch immer nur einer zur gleichen Zeit. Die restlichen Interfaces werden in dieser Zeit blockiert.

6. Umgebungserfassung mit dem Laserscanner URG-04LX

Für die Umgebungserfassung im Frontbereich des SoC-Fahrzeugs wird ein 2D-Laserscanner eingesetzt, dessen serielle Schnittstelle mit einem UART des MicroBlaze_0 gekoppelt wird (vgl. Abb. 5.1). Die gemessenen Abstandswerte werden zyklisch versendet und von einem Software Modul auf MicroBlaze_0 verarbeitet, dessen Funktionen in Abb. D dargestellt sind. Die Charakterisierung des Laserscanners URG-04LX und das eingesetzte SCIP 2.0 Kommunikationsprotokoll werden nachfolgend erläutert. Anschließend wird die Implementierung des Software-Interface auf MicroBlaze_0 zur Auswertung der Laserscandaten vorgestellt (vgl. Kap. 3.1.1).

Für die Laserscanner-basierte Objekterkennung bedeutet eine Scanfrequenz von 10 Hz, dass alle 100 ms eine Aktualisierung der Abstandswerte und somit eine Neu-Generierung der zusammenhängenden Segmente stattfindet. In Testfahrten wurde eine von der Fahrspurführung abhängige maximale Geschwindigkeit v_{max} von ca. 2 m/s als realistisch angesehen. Bei dieser Geschwindigkeit wird die Detektion der Fahrspur garantiert und ein Ausbrechen des SoC-Fahrzeugs durch zu hohe Kräfte verhindert. Die zurückgelegte Strecke s_{max} zwischen zwei Abtastperioden Δt berechnet sich wie folgt:

$$s_{max} = v_{max} * \Delta t \quad \rightarrow \quad s_{max} = 2\frac{m}{s} * 0.1s = 0,2m = 20cm \tag{6.1}$$

Die Erfassung von Abstandswerten erfolgt mit einer festen Schrittweite, was bedeutet, dass die Verteilung der Messwerte sowohl von der Distanz zwischen Laserscanner und Objekt abhängig ist als auch von der Ausrichtung des Objekts gegenüber dem Scanner. Für eine detaillierte Erkennung von kleinen Objekten, muss der Öffnungswinkel und damit die Schrittweite klein genug sein. Zur Berechnung der minimalen Abtastrate wird das Nyquist-Shannonschen Abtasttheorem verwendet, was aussagt, dass ein kontinuierliches Signal mit einer maximalen Frequenz f_{max} mit einer Frequenz f_a doppelt so groß wie f_{max} abgetastet werden muss, damit das Ursprungssignal ohne Informationsverlust in ein zeitdiskretes Signal transformiert wird [48].

$$f_a > 2 * f_{max} \quad \Rightarrow \quad f_{nyquist} = \frac{f_a}{2} \tag{6.2}$$

Basierend auf dem Abtasttheorem legt die Nyquist-Frequenz $f_{nyquist}$ eine Grenzfrequenz fest, die bei einer festen Abtastfrequenz kleine Objekte ausreichend erkennt. Nach diesem für die digitale Signalverarbeitung verwendeten Abtasttheorem, würde eine Abtastung von zwei Abstandswerten pro Objekt ausreichend sein. Jedoch ist dies im Falle von Laserscannern nicht der Fall, da ein Objekt im Schnitt durch vier bis fünf Messpunkte eindeutig identifiziert wird.

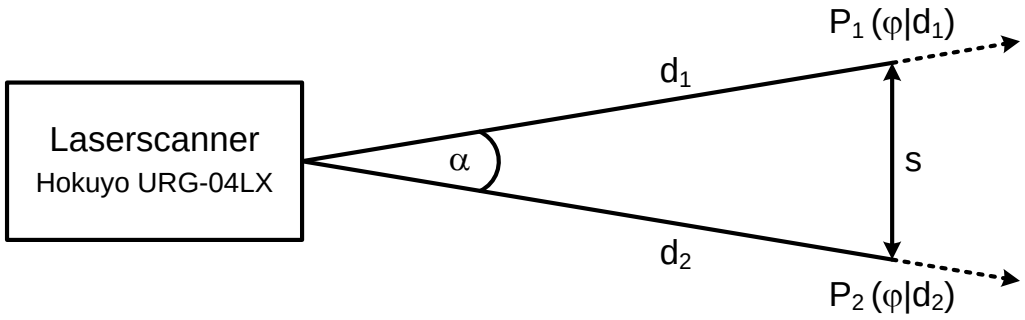


Abb. 6.1.: Der Abstand zwischen zwei benachbarten Messpunkten ist abhängig von der Distanz d und der Winkelauflösung α

Der Öffnungswinkel wird durch den Abstand von zwei nebeneinander ausgesendete Laserstrahlen bestimmt (vgl. Abb. 6.1). Abhängig von der Winkelauflösung α und der Distanz des Objektes zum Laserscanner d_1 und d_2 ändert sich der tangentiale Abstand s zwischen zwei benachbarten Messpunkten. Mit Hilfe des Kosinussatz wird dieser Abstand trigonometrisch bestimmt:

$$s = \sqrt{{d_1}^2 + {d_2}^2 - 2 * d_1 * d_2 * cos(\alpha)} \tag{6.3}$$

Bei einer minimalen Winkelauflösung von 0,35° und einem Abstand von 50 cm zum Objekt ergibt sich ein tangentialer Abstand von 0,31 cm. Dieser wächst bei gleichbleibender Winkelauflösung proportional der Entfernung. Mit der obengenannten Formel des Kosinussatzes wird ebenfalls die Breite eines Objekts bestimmt. Hierzu müssen die charakterisierenden Entfernungsmesswerte, die den linken und rechten Begrenzungspunkt festlegen, in Polar Koordinaten $P(\varphi|d)$ vorliegen.

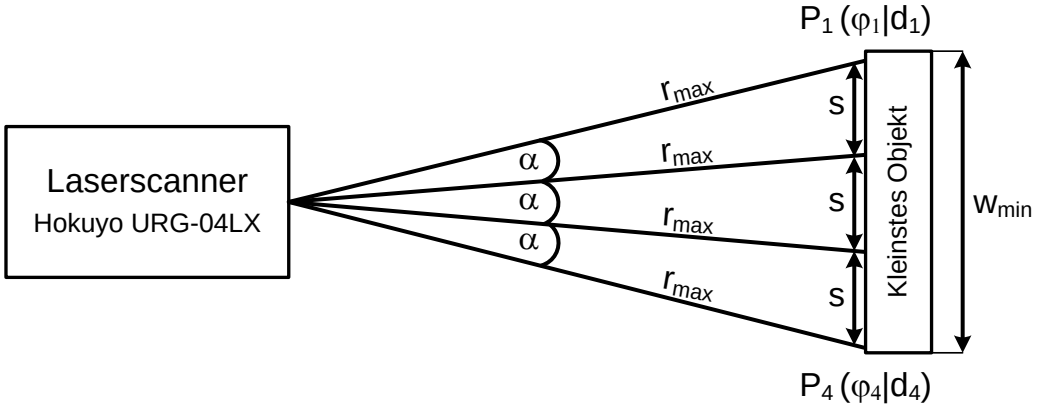


Abb. 6.2.: Die minimale Größe w_{min} eines erkennbaren Objektes in maximaler Reichweite r_{max} wird durch die Winkelauflösung α und den tangentiale Abstand s bestimmt.

Die aus der Nyquist-Frequenz resultierende Grenzfrequenz bestimmt die minimale Größe eines Objektes, welches bei einer festen Abtastfrequenz von 10 Hz erkannt wird. Nach dem obengenannten Nyquist-Shannonschen Abtasttheorem wird zur Bestimmung der minimale Größe w_{min} die Entfernungsmesswerte d_1 und d_2 auf die maximale Laserscanner Reichweite r_{max} von 4.095 mm gesetzt (vgl. Abb. 6.2). Nimmt man nun an, dass mindestens vier Punkte zur eindeutigen Identifizierung des Objektes benötigt werden, berechnet sich bei einer Winkelauflösung α von 0,35° die minimale Größe wie folgt:

$$\begin{aligned} w_{min} &= 3 * \sqrt{{r_{max}}^2 + {r_{max}}^2 - 2 * r_{max} * r_{max} * cos(\alpha)} \\ &= 3 * \sqrt{4.095mm^2 + 4.095mm^2 - 2 * 4.095mm * 4.095mm * cos(0,35)} \\ &= 7,5cm \end{aligned} \tag{6.4}$$

Bei der Berechnung der minimalen Größe w_{min} wurde der tangentiale Abstand mit drei multipliziert, da vier Messpunkte für eine Identifizierung eines Objektes benötigt werden (vgl. Abb. 6.2). Für die Laserscanner-basierte Objekterkennung wird die berechnete minimalen Objektgröße von 7,5 cm durch einen Toleranzbereich von 2,5 cm ergänzt. Dies ist notwendig, da abhängig von der Oberflächenstruktur des Objektes die Laserstrahlen unterschiedlich reflektiert werden (vgl. Kap. 6.3).

6.1. Serielles SCIP 2.0 Kommunikationsprotokoll

Der Laserscanner URG-04LX verwendet als Kommunikationsprotokoll standardmäßig das SCIP 1.1 Protkoll. Für das SoC-Fahrzeug wurde der Laserscanner durch ein Firmware Update mit der SCIP 2.0 Version ausgestattet, da durch das Update die bisherigen vier Kommandos mit neun zusätzlichen Kommandos ergänzt wurden [34]. Die Baudrate der RS232 Schnittstelle ist in einem Intervall von 9.600 - 750.000 Bits pro Sekunde frei konfigurierbar, wobei für das SoC-Fahrzeug eine Baudrate von 115.200 Bits pro Sekunde gewählt wurde.

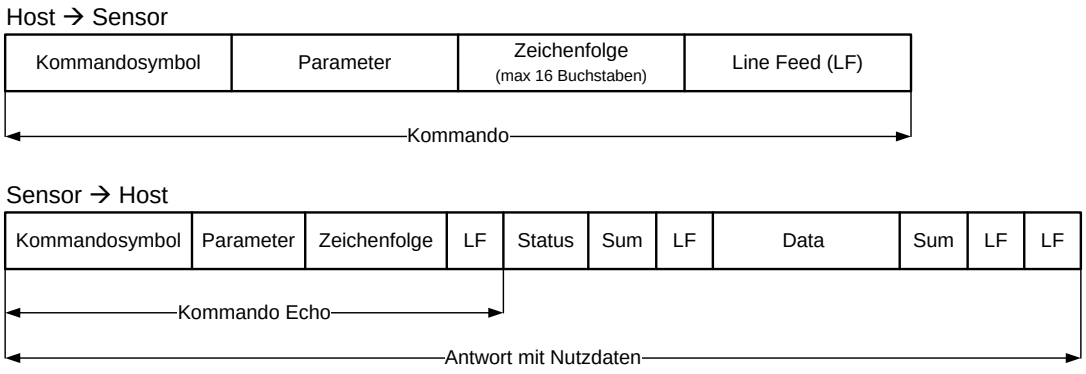


Abb. 6.3.: Aufbau des eingesetzten SCIP 2.0 Kommunikationsformats [34]

Der Laserscanner und MicroBlaze_0 tauschen vordefinierte Kommandos aus, die in einem festgelegten Kommunikationsformat übermittelt werden (vgl. Abb. 6.3). Die von MicroBlaze_0 initiierte Kommunikation beginnt immer mit einem spezifizierten zwei Byte großen Kommandosymbol gefolgt von weiteren Parametern, wobei ein oder zwei

abschließende Linefeeds das Kommunikationsende kennzeichnen [34]. Empfängt der Laserscanner ein Kommando, wird dieses stets durch ein Echo des gesendeten Pakets und durch die angeforderten Nutzdaten bestätigt. Die Nutzdaten werden stets in maximal 64 Byte großen Paketen übertragen, wobei die Anzahl der Pakete abhängig von der gewünschten Menge an Entfernungswerten und deren Kodierung ist. Durch einen internen Puffer im Laserscanner werden sequentiell gesendete Kommandos zwischengespeichert und durch ein FIFO-basierten Verfahren nacheinander verarbeitet.

Der Laserscanner URG-04LX überträgt die Entfernungswerte mit einer Auflösung von 1 mm, wofür zur Datenreduktion und zur Minimierung der Übertragungszeit eine Kodierung eingesetzt wird. Die zwei Byte Kodierung hat eine maximale Länge von 12 Bit und die drei Byte Kodierung eine maximale Länge von 18 Bit hat (vgl. Abb. 6.4). Die Kodierung erfolgt durch die Trennung der oberen und unteren 6 Bits und dem anschließenden Hinzufügen von 30H, was die kodierten Daten in ASCII konvertiert.

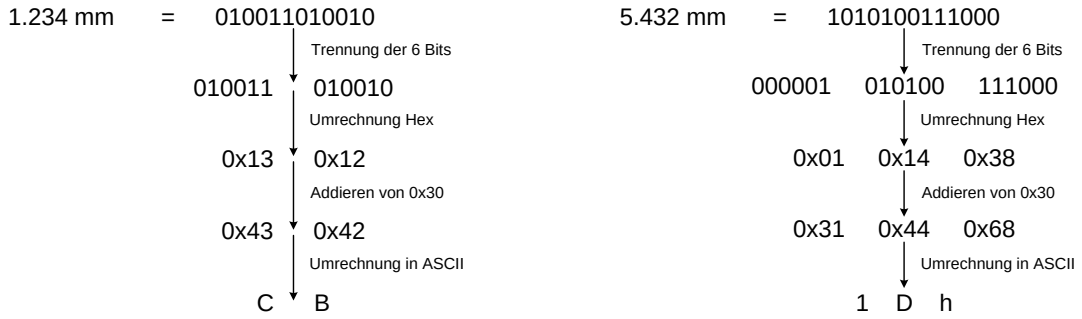


Abb. 6.4.: Kodierung der Entfernungswerte mit einer zwei Byte Kodierung (links) oder einer drei Byte Kodierung (rechts) [34]

Für die Laserscanner-basierte Objekterkennung werden die angeforderten Entfernungswerte stets mit zwei Byte kodiert, somit wird bei einer maximalen Länge von 12 Bit eine maximale Auflösung von 4.095 mm erreicht (vgl. Tab. 3.3).

Einschalten des Laserscanners durch das BM-Kommando

Zu Beginn der Entfernungsmessung sendet MicroBlaze_0 eine ASCII kodierte Zeichenkette, die mit den Buchstaben B und M eingeleitet wird (vgl. Abb. 6.5). Nach einem abschließenden Line Feed bestätigt der Laserscanner das Kommando mit einem Echo, welches das ursprüngliche Kommando sowie eine Checksumme und den Status enthält.

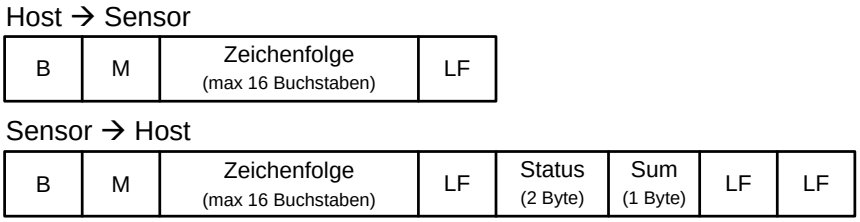


Abb. 6.5.: Aufbau des BM-Kommandos zum Einschalten des Laserscanners

MicroBlaze_0 empfängt das Echo Kommando und wertet zur Verifizierung das Statusfeld aus, wobei 01 eine Fehlfunktion kennzeichnet.

Ausschalten des Laserscanners durch das QT-Kommando

Empfängt MicroBlaze_0 das PB-2 Kommando von MicroBlaze_1, dann wird das QT-Kommando an den Laserscanner gesendet, welches analog zum BM-Kommando durch eine zwei Byte Zeichenkette eingeleitet wird (vgl. Kap. 8). Nach Empfang der ASCII kodierten Buchstaben Q und T bestätigt der Sensor das Kommando mit einem Echo (vgl. Abb. 6.6). Des Weiteren verfügt der Sensor über ein Reset Kommando, welches durch die Zeichenfolge RS eingeleitet wird und im Gegensatz zum QT-Kommando die internen Sensorkonfiguration zurückgesetzt, die nach dem Einschalten durch das BM-Kommando verändert wurde [34].

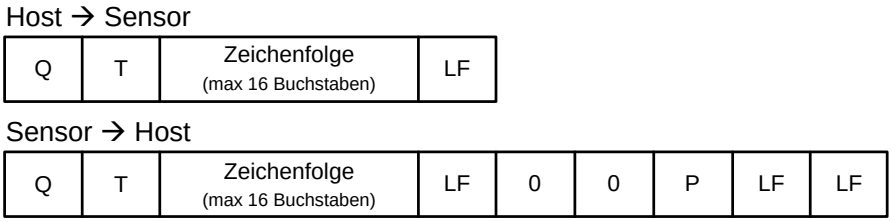


Abb. 6.6.: Aufbau des QT-Kommandos zum Ausschalten des Laserscanners

Beim QT-Kommando und beim RS-Kommando wird kein Status übermittelt, sondern stets der Wert 00 gefolgt von einem P. Bei der Laserscanner-basierten Objekterkennung wird zur Datenreduktion auf die Verwendung der Zeichenfolge verzichtet. Aus diesem Grund hat sowohl das BM-Kommando als auch das QT-Kommando eine feste Länge von drei Byte. Die Antwort des Laserscanners ist ebenfalls konstant und beträgt jeweils acht Byte, was den Vorteil hat, dass der Software die zu erwartende Anzahl an Bytes bekannt ist und somit eine schnelle Verarbeitung stattfindet.

Anfordern von Messwerten durch das GD/GS oder MD/MS Kommando

Das GD/GS Kommando überträgt die zuletzt gemessenen Entfernungswerte einmalig, hingegen werden beim MD/MS-Kommando die Messwerte kontinuierlich mit einer Frequenz von 10 Hz empfangen [34]. Bei beiden Kommandos wird durch das zweite Byte die Kodierung festgelegt, wobei ein S für eine zwei Byte Kodierung und ein D für eine drei Byte Kodierung steht (vgl. Abb. 6.7). Im Vergleich zum GD/GS Kommando verfügt das MD/MS über zwei zusätzliche Parameter, die die Anzahl der Scans und das Scan Interval festlegen.

- **Startschritt und Endschritt:** Liegen im Intervall von 44 und 725. Der Endschritt muss stets größer sein als der Starschritt. Es ist darauf zu achten, dass die natürlichen Zahlen ebenfalls als ASCII Äquivalent versendet werden, da sonst der Laserscanner diese nicht interpretieren kann.
- **Cluster Count:** Bestimmt die Winkelauflösung des Laserscanners, wobei bei einem Cluster Count von null die Messwerte mit einer Winkelauflösung von 0,35° übertragen werden. Bei der Implementierung wird ein Cluster Count von drei genutzt, was bedeutet, dass die Messwerte mit einer Winkelauflösung von 1,05° übermittelt werden. Dies hat den Vorteil, dass ein Drittel der Daten versendet wird und somit die Übertragungsrate steigt. Das Intervall des Cluster Counts liegt zwischen 0 und 99.

- **Status:** Liegen alle Parameter im definierten Bereich, dann wird durch den Status 00 oder 99 die erfolgreiche Übertragung signalisiert, andererseits wird eine Fehlfunktion gekennzeichnet
- **Scan Interval:** Bei einem kontinuierlichen Scan (MD/MS) gibt das Scan Interval die Anzahl der zu überspringenden Scans an. Bei einem Scan Interval von beispielsweise zwei, werden die durch Start- und Endschritt spezifizierten Messwerte jedes zweite Mal versendet. Dies bedeutet, dass sich die Scanperiode von 100 ms verdoppelt.
- **Anzahl Scans:** Durch diesen Parameter wird die gewünschte Anzahl an Scans beim MD/MS festgelegt. Wird die Anzahl an Scans auf null gesetzt, dann werden die Entfernungsmesswerte kontinuierlich alle 100 ms übertragen. Durch ein QT-Kommando wird die Übertragung gestoppt.
- **Zeitstempel:** Die Bestätigung des MD/MS und des GD/GS Kommandos erfolgt durch ein Echo Kommando, welches durch einen zusätzlicher Zeitstempel ergänzt wird. Ein 24 Bit interner Timer mit einer Auflösung von 1 ms erzeugt für jeden ersten Schritt eines Scans einen eindeutigen Zeitstempel, der mit vier Byte kodiert wird.
- **Zeichenfolge:** Ist ein freies 16 Byte großes Feld, welches beispielsweise zur Vergabe von Sequenznummern verwendet wird. Bei der Implementierung wird dieses Feld nicht genutzt, da die Kommandos sequentiell gesendet und somit auch geordnet verarbeitet werden.

Host → Sensor

G	D / S	Startschritt (4 Byte)	Endschritt (4 Byte)	Cluster (2 Byte)	Zeichenfolge (max 16 Byte)	LF

Sensor → Host

G	D / S	Startschritt (4 Byte)	Endschritt (4 Byte)	Cluster (2 Byte)	Zeichenfolge (max 16 Byte)	LF

0	0	P	LF	Zeitstempel (4 Byte)	Sum (1 Byte)	LF	
Nutzdaten 1 (64 Byte)					Sum (1 Byte)	LF	
----------					Sum (1 Byte)	LF	
Nutzdaten N-1 (64 Byte)					Sum (1 Byte)	LF	
Nutzdaten N (n Byte)					Sum (1 Byte)	LF	LF

Abb. 6.7.: Aufbau des GD/GS-Kommandos zum einmaligen Empfang von Messwerten

Sind die gewünschten Messwerte größer als 64 Byte, dann werden mehrere Pakete versendet, wobei das letzte durch zwei Line Feeds das Ende der Übertragung kennzeichnet. Die Anzahl an Bytes für einen Scan ist abhängig vom Cluster Count und der Kodierung.

Für die Laserscanner-basierte Objekterkennung wird das GS-Kommando periodisch an den Laserscanner gesendet, wobei 84 Abstandswerte mit 198 Byte an MicroBlaze_0 übertragen werden (vgl. Abb. 6.9).

6.2. Implementierung des Software-Interfaces auf MicroBlaze_0

Der Laserscanner wird kopfüber im Frontbereich des SoC-Fahrzeugs angebracht, sodass die Laserdiode einen Abstand von ca. 2 cm zur Fahrbahn hat und die Laserstrahlen mittig zum Fahrzeug nach vorne ausgesendet werden (vgl. Abb. 3.1). Die Objekterkennung wird im SoC-Fahrzeug zur Hinderniserkennung und zur Kollisionsvermeidung eingesetzt, wofür der Bereich neben und hinter dem Fahrzeug nicht relevant ist. Aus diesem Grund wird der Laserscanner so konfiguriert, dass ein 90° Scanbereich im Frontbereich abgedeckt wird (vgl. Abb. 6.8).

Die Software für die Laserscanner-basierte Objekterkennung wird auf MicroBlaze_0 ausgeführt. Ausgestattet mit einem Xilkernel RTOS wird das Software-Interface für den Laserscanner in das POSIX Thread Konzept eingebettet (vgl. Kap. 8). Dies hat den Vorteil, dass parallel zur Messdatenerfassung und Objekterkennung die FSL Kommunikation mit MicroBlaze_1 stattfinden kann. Ein Master Thread übernimmt die Koordinierung der Funktionsausführung und wartet auf Steuersignale vom MicroBlaze_1, der als Controller das SoC-Fahrzeug parametriert (vgl. Tab. 8.1). Die Messdatenerfassung läuft parallel dazu und kann durch den Master Thread unterbrochen werden.

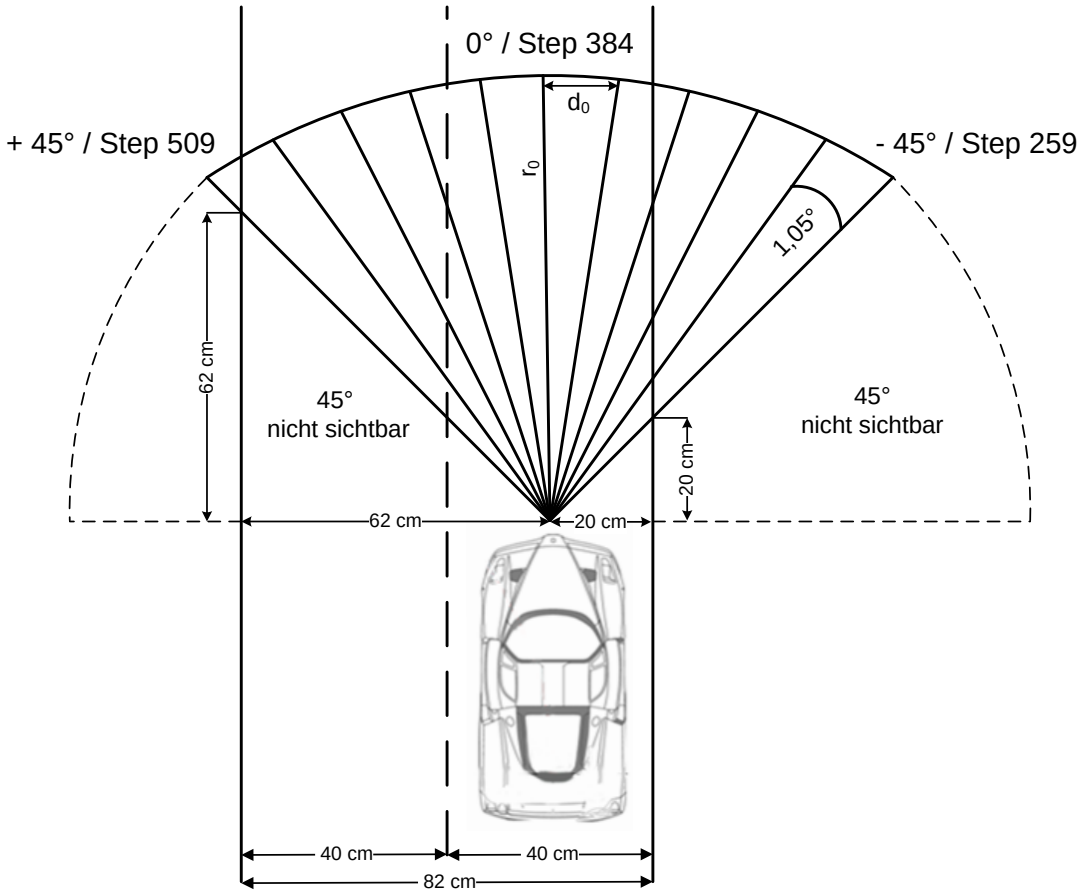


Abb. 6.8.: Laserscanbereich von 90° im Frontbereich des SoC-Fahrzeugs

6.2.1. Ermittlung der Abstandswerte in einem 90° Scanbereich

Für die Objekterkennung ist der Bereich innerhalb einer Reichweite von zwei Metern und einem Winkel von 90° relevant, was bedeutet, dass bei einer minimalen Winkelauflösung von 0,35° Objekte bis zu einer Größe von 3,7 cm erkannt werden (vgl. Abb. 6.8. Da die Hindernisse beim Carolo Cup mindestens 10 cm groß sind, wird für die Objekterkennung des SoC-Fahrzeugs eine Winkelauflösung von 1,05° verwendet (vgl. Tab. 6.1).

Scanbereich	90°
Startschritt	259
Endschritt	509
Cluster Count	3
Winkelauflösung	1,05°
Anzahl Abstandswerte	84

Tab. 6.1.: Konfigurationsparameter des Laserscanners für die SoC-Implementierung

Geht man davon aus, dass die gesamte Fahrbahn eine Breite von 82 cm hat, dann wird abhängig von der Entfernung die Anzahl an Messwerten, die sich auf der Fahrspur befinden, wie folgt berechnet:

Der tangentiale Abstand d_0 zwischen dem mittleren Laserstrahl bei 0° und dem rechts benachbarten Laserstrahl bei 1,05° wird durch das rechtwinklige Dreieck berechnet (vgl. Abb. 6.8). Hierbei ist die Winkelauflösung α gegeben und die Entfernung des mittleren Laserstrahls r_0 wird vorgegeben:

$$tan(\alpha) = \frac{d_0}{r_0} \quad \Rightarrow \quad d_0 = r_0 * tan(\alpha) \tag{6.5}$$

Fährt das SoC-Fahrzeug mittig auf der rechten Fahrspur, dann ist der Abstand a_r zur rechten Fahrspurmarkierung gleich 20 cm und der Abstand a_l zur linken Fahrspurmarkierung gleich 62 cm (vgl. Abb. 6.8). Durch den tangentialen Abstand d_0 und den Abständen zur Fahrspurmarkierungen a_r und a_l wird die Anzahl der Messwerte m_r und m_l, die sich zwischen der Fahrzeugmitte und den Fahrspurmarkierungen befinden, berechnet.

$$m_r = \frac{a_r}{d_0} \qquad m_l = \frac{a_l}{d_0} \tag{6.6}$$

Aus 6.5 und 6.6 folgt die Anzahl an Messwerten m, die sich auf der Fahrspur befinden. Hierbei ist zu beachten, dass 6.7 erst ab einer Reichweite von 62 cm korrekte Ergebnisse liefert, da der rechte äußere Laserstrahl bei 45° die rechte Fahrspurmarkierung erst in einer Entfernung von 62 cm schneidet. Bei -45° wird die linke Fahrspurmarkierung bei 20 cm geschnitten.

$$m = m_r + m_l = \frac{a_r}{d_0} + \frac{a_l}{d_0} = \frac{a_r + a_l}{d_0} = \frac{200mm + 620mm}{r_0 * tan(1,05)} = \frac{44740,27mm}{r_0} \tag{6.7}$$

Für eine maximale Reichweite von 2 m und einer Winkelauflösung von 1,05° ergeben sich ca. 22 Entfernungswerte, die auf der Fahrbahn liegen. Da durch die Fahrspurführung nicht sichergestellt ist, dass sich das SoC-Fahrzeug direkt in der Mitte der Fahrspur bewegt, sind die 22 Messpunkte ab einer Reichweite von 62 cm für die Hinderniserkennung ausreichend. Auf Grund des rechten Schnittpunktes des Laserstrahls mit der rechten Fahrspurmarkierung kann nicht garantiert werden, dass Hindernisse mit einer geringeren Entfernung von 20 cm erkannt werden (vgl. Tab. 3.3).

Da die Ausführungsintervalle der implementierten Funktionen zu hoch sind, erfolgt das Anfordern der 84 Messdaten nicht durch das MD/MS Kommando, sondern durch das GS-Kommando mit einer eigens erzeugten Abtastperiode (vgl. Kap. 8). Empfängt MicroBlaze_0 ein PB-4 Kommando über den FSL von MicroBlaze_1, wird das GS-Kommando über den UART an den Laserscanner gesendet (vgl. Abb. 6.9) . Sowohl der Startschritt von 259 als auch der Endschritt von 509 werden durch den 90° Scanbereich definiert (vgl. Abb. 6.8).

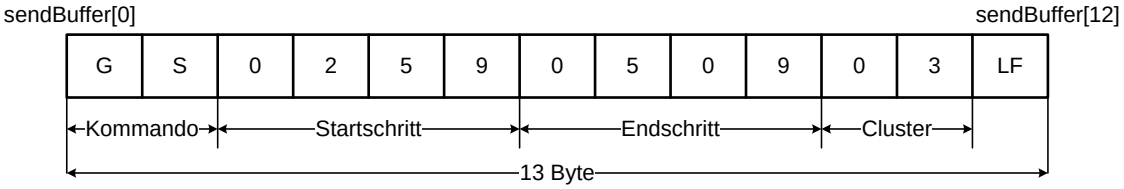


Abb. 6.9.: GS-Kommando zum Empfang von 84 Abstandswerten in einem Scanbereich von 90°

In Abb. 6.9 ist der Aufbau des 13 Byte großen Char Arrays gezeigt, welches zur Messdatenanforderung an den Laserscanner gesendet wird. Der als Integer definierte Start- und Endschritt wird als vier Byte Char Array angegeben, wofür die Konvertierfunktion *convertIntToChar()* entwickelt wurde (vgl. Code 6.1).

```
0  void convertIntToChar (int value, char convertResult[], int index){
       index--;
       while(value)
       {
           convertResult[index] = value % 10 ;
5          value /= 10;
           index--;
       }
   }
```

Listing 6.1: Konvertierung eines Integers in ein Char Array mit definierter Größe

Zum Senden und Empfangen werden die UART Makrofunktionen *XUartLite_Send()* und *XUartLite_RecvByte()* verwendet. Alle Prozessor spezifischen Makrofunktionen wurden in eigene Sende- und Empfangsfunktionen gekapselt, was den Vorteil hat, dass bei einer Portierung auf einen anderen Prozessor ausschließlich eine Funktion geändert werden muss.

- ***scip_send(char *send_buffer)*:** Diese Funktion wird von MicroBlaze_0 zum Senden von Kommandos an den Laserscanner verwendet, wobei der übergebene *send_buffer* die zu sendende Zeichenkette darstellt. Ein Integer, der die Anzahl der vom UART übertragenden Bytes enthält, wird als Rückgabeparameter genutzt.

- ***scip_recv(char * recv, int number)*:** Wird zum Empfang des Kommandoechos *recv* genutzt, wobei die Anzahl der zu empfangenen Bytes *number* bekannt ist.
- ***scip_recv_until_one_lf(char *recv_buffer)*:** Bis zum Empfang eines Line Feeds wird kontinuierlich aus der UARt FIFO gelesen und die empfangenen Daten in *recv_buffer* gespeichert.
- ***scip_recv_until_two_lf(char *recv_buffer)*:** Diese Funktion wird verwendet, um das letzte Nutzdatenpaket zu empfangen, wobei zwei Line Feeds das Ende der Übertragung signalisieren.

Jedes an den Laserscanner gesendete Kommando wird durch eine 1:1 Kopie des gesendeten Kommandos bestätigt. Empfängt MicroBlaze_0 das Echo, wird durch die *strncmp()* Funktion die ersten zwei Bytes mit dem ursprünglich gesendeten Kommando Symbol verglichen (vgl. Code 6.2). Das im Protokoll vorgesehene Zeichenfolge Feld zum Senden einer Sequenznummer wird nicht genutzt (vgl. Abb. 6.7).

Name	Echo Kommando					Status Header						3 Pakete Nutzdaten									
Bytes	2	4	4	2	1	2	1	1	4	1	1	64	1	1	64	1	1	40	1	1	1
Parameter Name	Kommando Symbol	Startschritt	Endschritt	Cluster Count	Line Feed LF	Status	Zeichen P	Line Feed LF	Zeitstempel	Check Summe	Line Feed LF	32 Messwerte (Nutzdaten)	Check Summe	Line Feed LF	32 Messwerte (Nutzdaten)	Check Summe	Line Feed LF	20 Messwerte (Nutzdaten)	Check Summe	Line Feed LF	Line Feed LF
Bytes	13					4			6			66			66			43			
Gesamt	198 Bytes																				

Abb. 6.10.: Datenmenge des GS-Kommandos bei Empfang der 84 Abstandswerte vom Laserscanner. Die durch Line Feed getrennten Daten werden zeilenweise eingelesen.

Nach dem Empfang des Echo Kommandos, wird das angehängte zwei Byte große Statusfeld und der Zeitstempel ausgelesen (vgl. Abb. 6.10). Ist das Status Array ungleich [0][0], wird die Kommunikation beendet, da der Laserscanner einen Fehler erkannt hat und keine Messdaten überträgt (vgl. Code 6.2).

```
0  XUartLite_Send(&uart, (u8*)sendBuffer, COMMAND_13_BYTE);      // sending GS command

   received_data = scip_recv_until_one_lf(header);               // receiving echo command
     if (strncmp(header, GDGS_COMMAND, 1)){
       return ERROR_WRONG_RESPONSE_COMMAND;
5    }
   received_data = scip_recv_until_one_lf(status);               // receiving status field
     if(status[0] != '0' || status[1] != '0'){
       char temp[1];
       scip_recv(temp,1);                                        // receiving last LF if status != 00
10     return ERROR_SEND_FAIL;
     }
```

Listing 6.2: Senden des GS-Kommandos und Empfang des Echos und des Status Felds

Die Nutzdaten werden in 64 Byte große Pakete mit einer angehängten Checksumme und einem Line Feed versendet. Für die Objekterkennung werden 84 Messwerte vom Laserscanner angefordert, die jeweils mit zwei Byte codiert werden. Hieraus ergibt sich eine Nutzdatengröße von 168 Byte und somit eine maximale Anzahl von drei Paketen. Auf Grund des zusätzlichen Overheads, der durch das Echo und den Header entsteht, werden pro Scan insgesamt 198 Bytes vom Laserscanner empfangen (vgl. Abb. 6.10). Bei einer Baudrate von 115.200 Bits/Sekunde und einer zu übertragenden Datenmenge n_{bits} von 198 Bytes berechnet sich die Übertragungszeit t_u wie folgt:

$$t_u = \frac{n_{bits}}{Baudrate} = \frac{198 * 8\ Bits}{115.200\ Bits} s = \frac{1.584}{115.200} s = 13,75ms \tag{6.8}$$

Zur Übertragung der 84 Abstandswerte vom Laserscanner zu MicroBlaze_0 wird eine Zeit von 13,75 ms benötigt. Das GS-Kommando zum Anfordern der Messdaten hat eine Größe von 13 Byte und wird in einer Zeit von 0,9 ms übertragen. Dies bedeutet, dass die ersten Nutzdaten nach einer Zeit von 2,5 ms bei MicroBlaze_0 ankommen (vgl. Abb. 6.11). Hierbei ist die Verarbeitungszeit des Laserscanners nicht berücksichtigt.

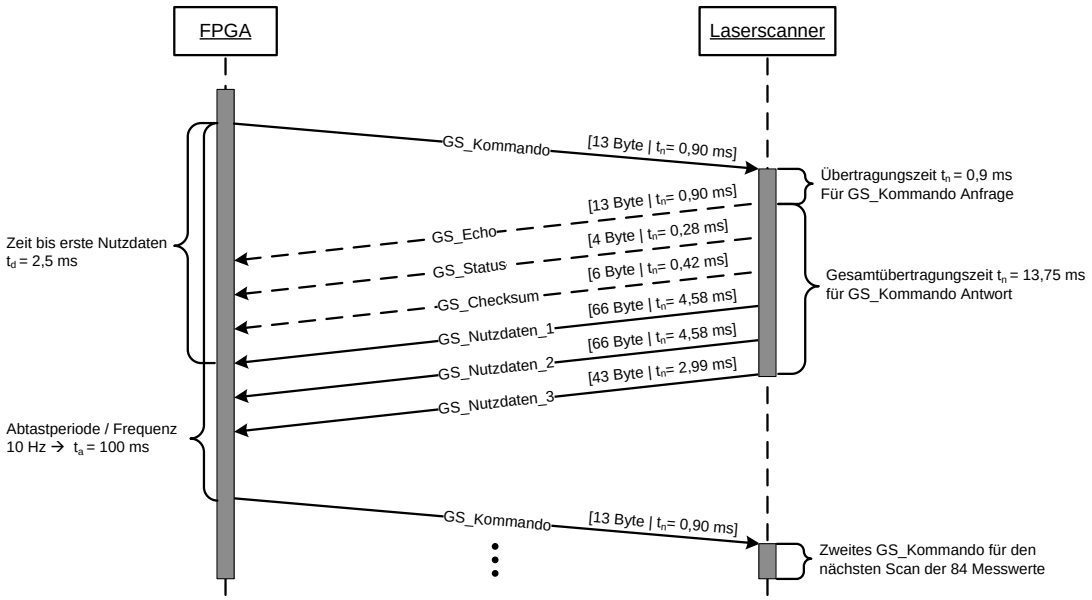


Abb. 6.11.: Sequenzdiagramm des GS-Kommandos mit Sende- und Empfangsdaten. Die Übertragungszeit t_n bei einer Baudrate von 115.200 Bits/s ist abhängig von der Datenmenge

Die Nutzdaten werden zeilenweise eingelesen und in einem globalen Array *measurementRecvBuffer[ARRAY_BYTE_LENGTH]* gespeichert, wobei die Checksumme und die Line Feeds entfernt werden. Auf Grund der zwei Byte Kodierung der Nutzdaten ist das Array 168 Byte groß. Jedoch wurde es dynamisch implementiert, sodass die Größe des Arrays abhängig von der zu empfangenen Datenmenge ist. Aus der Summe der Übertragungszeiten von Sende- und Empfangsdaten ergibt sich eine Zeit von 14,65 ms. Nach dieser Zeit und zusätzlich der Verarbeitungszeit sind alle Nutzdaten abgespeichert und die Dekodierung beginnt (vgl. Kap. 9.2).

Eine Erhöhung der Baudrate nicht vorteilhaft, da bei gleichbleibender Systemfrequenz der Baudraten Error steigt und außerhalb der zulässigen UART Toleranz liegt [86].

6.2.2. Dekodierung der empfangenen Abstandswerte

Die 84 empfangene Messwerte liegen im *measurementRecvBuffer[]* bereit, wobei durch die Kodierung jeder Abstandswert eine Größe von zwei Byte hat (vgl. Abb. 6.4). Sowohl die Checksummen als auch die Line Feeds wurden im vorherigen Schritt entfernt, sodass die Abstandswerte in der richtigen Reichenfolge vorliegen. Zur Dekodierung wurden die Funktionen *decodeByteBuffer90Degree()* und *decodeData()* implementiert:

- ***decodeData(laser_encoding, char * data)*:**
  Dekodiert das übergebene *data* Array abhängig von der Kodierung *laser_encoding*. Der Dekodier-Algorithmus konvertiert jedes ASCII kodierte Byte durch eine Subtraktion mit 0x30 in das zugehörige Hexadezimal Äquivalent (vgl. Abb. 6.4). Der zurückgegebene vorzeichenlose Integer repräsentiert den Abstandswert in einer Millimeter Darstellung (vgl. Code 6.3).

- ***decodeByteBuffer90Degree()*:**
  Dekodiert die 168 Byte der 84 Abstandswerte durch den Aufruf von *decodeData()* und speichert die Abstandswerten im *decoded_data_buffer[]*, wobei sowohl der Startschritt als auch die Anzahl der Schritte als globale Konstanten deklariert wurden. Des Weiteren wird zur Datenreduktion und zur Erhöhung des Programmausführungsintervalls die dekodierten Abstandswerte vorverarbeitet, indem bei Überschreitung eines definierten Schwellwerts *MAX_DISTANCE_RECOGNITION*, der Abstandswert auf null gesetzt wird.

Durch die Kapselung der *decodeData()* Funktion in *decodeByteBuffer90Degree()* wurde die Dekodierung nicht ausschließlich für die Verwendung im SoC-Fahrzeug konzeptioniert, sondern kann auch mit unterschiedlichen Scanbereichen verwendet werden. Da die Rohdaten geordnet im *measurementRecvBuffer[]* vorliegen, kann *decodeByteBuffer90Degree()* stets die zusammenhängenden Bytes, die einen Abstandswert kennzeichnen, an die Funktion *decodeData()* übergeben.

```
0  unsigned int decodeData (laser_encoding encoding, char *data){

       unsigned int decodedData = 0;
       int i = 0;

5      if(encoding == LASER_TWO_CHAR_ENCODING){

           for (i = 0; i < 2; ++i) {
               decodedData <<= 6;                  // Links Verschieben um 6 Bits (erstes Byte nach links verschieben)
               decodedData &= ~0x3f;               // Negierte (~) Und Verknüpfung - letzte 6 Bits auf Null setzen
10             decodedData |= data[i] - 0x30;      // Char Array byteweise verodern
           }
       }else if(encoding == LASER_THREE_CHAR_ENCODING){

           for (i = 0; i < 3; ++i) {
15             decodedData <<= 6;
               decodedData &= ~0x3f;
               decodedData |= data[i] - 0x30;
           }
       }else{
20          return ERROR_WRONG_PARAMETER;
           }
       return decodedData;
   }
```

Listing 6.3: Dekodierung der empfangenen Abstandswerte ahängig von der Kodierung

Für die Objekterkennung im SoC-Fahrzeug hat die global definierte Konstante *MAX_DISTANCE_RECOGNITION* einen Wert von 2.000 mm, was bedeutet, dass Objekte bis zu einer Reichweite von zwei Metern erkannt werden.

6.2.3. Glättung der Abstandswerte zur Störsignalunterdrückung

Nach der Spezifikation hat der Laserscanner bis zu einer Reichweite von einem Meter eine Abweichung von maximal 10 mm. Für Reichweiten ab einem Meter und bis zu vier Meter beträgt die Abweichung maximal 1 % von der gemessenen Distanz (vgl. Abb. 3.3). Dies bedeutet, dass bei einer Reichweite von zwei Metern die Messwerte eine maximale Abweichung von 20 mm besitzen. Jedoch kann es beim Laserscanning vorkommen, dass einzelne Strahlen falsch reflektiert werden und somit der gemessene Abstandswert nicht der Realität entspricht. Trifft ein Laserstrahl beispielsweise auf die Kante eines Objektes, dann wird dieser nicht zurück zum Empfänger reflektiert, sondern wird durch die Kante nach hinten abgelenkt. Das Messrauschen in der Menge der Abstandswerte wird durch sogenannte Peaks verursacht. Für die Objekterkennung würden Peaks zu einem falsch detektierten Hindernis führen. Testmessungen in verschiedenen Szenarien haben gezeigt, dass bei der für das SoC-Fahrzeug relevanten Reichweite von zwei Metern, kein relevantes Messrauschen auftritt. Die in Abb. 6.12 erkennbaren kleine Ausschläge befinden sich im unteren Millimeterbereich und sind auf die Ungenauigkeit des Laserscanners zurückzuführen. Für die Objekterkennung stellen derartige Abweichungen kein Problem dar, da durch das spätere Schwellwertverfahren die abweichenden Messwerte eindeutig zugeordnet werden. Des Weiteren werden ab einer Entfernung von zwei Metern die Abstandswerte durch die Dekodierung auf null gesetzt, sodass keine Peaks in die Berechnung mit eingehen (vgl. Kap. 6.2.2). Aus diesem Grund wird zur Erhöhung des Programmausführungsintervalls auf eine Glättung der Messwerte verzichtet.

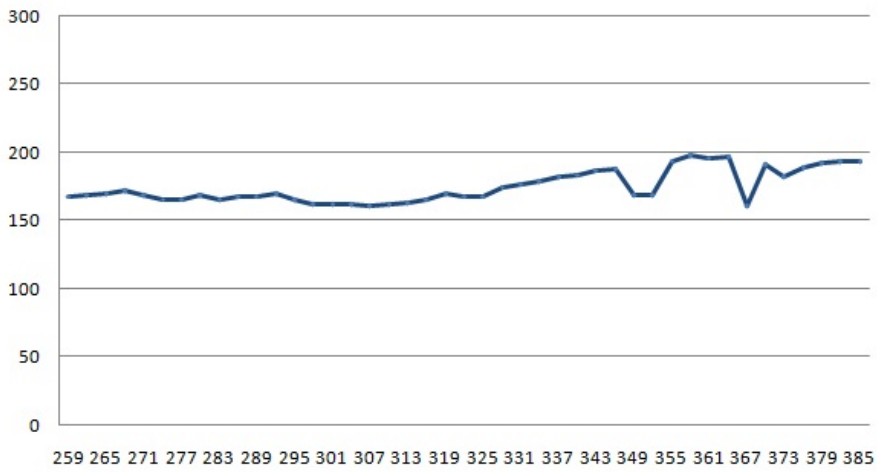


Abb. 6.12.: Messrauschen bei einer Testmessung von Schritt 259 bis 385

Wird der Laserscanner für Anwendungen eingesetzt, die keine Reichweitenbeschränkung einsetzen, wird für die Glättung der Abstandswerte ein gleitender Mittelwert eingesetzt. Dieser bildet den Mittelwert m über eine bestimmte Anzahl N an benachbarten Messwerten x_n und somit werden Peaks an den Mittelwert angepasst [62]. Das von *decodeByteBuffer90Degree()* zur Verfügung gestellte Integer Array enthält alle Abstandswerte für einen definierten Scanbereich. Dieses Array wird sukzessive durchlaufen und ein Mittelwert abhängig von N berechnet. Hierbei ist zu beachten, dass N nicht zu groß gewählt wird, da sonst zwei einzelne Objekte auf Grund des Schwellwerts zu einem Objekt zusammengefasst werden und somit ein falsches Abbild der Umwelt generiert wird.

6.3. Genauigkeit und Bewertung des Laserscanners

Bei der Erkennung von Objekten ohne Reflektoren ist die Intensität des zurückgesendeten Laserstrahls entscheidend für die Genauigkeit der Entfernungsmessung. Der Laserscanner URG-04LX berechnet aus der Phasendifferenz des ausgesendeten und empfangenen Laserstrahls direkt die Abstandswerte, wobei die Genauigkeit abhängig von den Reflexionseigenschaften des Objekts und des Auftreffwinkels des Laserstrahls ist. Anhand von Testmessungen mit unterschiedlichen Objekten in unterschiedlichen Abständen wurde die Genauigkeit des Laserscanners in Bezug auf die Entfernungsmessung analysiert. Da das Datenblatt hierzu keine Aussage trifft, wurden die Testergebnisse mit IEEE Veröffentlichungen verglichen. In den Arbeiten von [39] und [53] wurde der Hokuyo URG-04LX Laserscanner unter Laborbedingungen geprüft. Bis zum Erreichen eines Gleichgewichts zwischen Stromverbrauch und Wärmeableitung steigt die interne Temperatur des Laserscanners während der Aufwärmphase stetig an (vgl. Abb. 6.13). Die sogenannte „Warm-Up Time“ beträgt ca. 30 - 40 Minuten. Nach dieser Zeit wird der stationäre Zustand erreicht und die Entfernungswerte sind nahezu konstant [53].

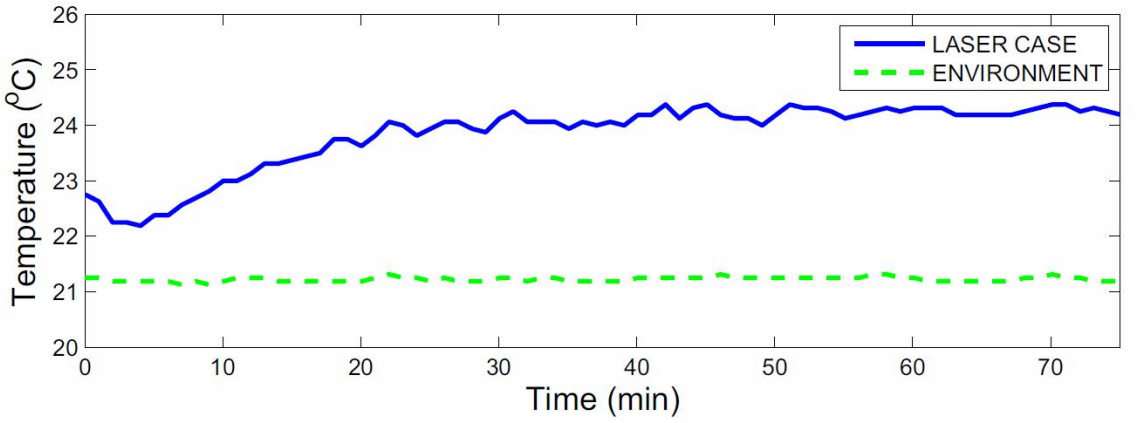


Abb. 6.13.: Anstieg der Laserscanner Temperatur während der Aufwärmphase [53]

Um die Reflexionseigenschaften von unterschiedlichen Materialen zu analysieren, wurden bei der ersten Messung unterschiedliche Objekte mit verschiedenen Farben in einer Entfernung von 500 mm platziert. In einem Intervall von ca. 30 Minuten wurde mit dem MS-Kommando jeweils 50 Testmessungen für den Scanschritt 384 (Winkel 0°) angefordert, wobei die empfangenen Messdaten dekodiert und über die RS232 Schnittstelle an den PC übertragen wurden. In Tabelle 6.2 wird der Mittelwert der 50 Abstandswerte und die Standardabweichung gezeigt. Der Mittelwert $\bar{x}$ berechnet sind wie folgt:

$$\bar{x} = \frac{1}{N} \sum_{i=1}^{N} x_i \tag{6.9}$$

Die Standardabweichung σ gibt an, wie weit sich die einzelnen Abstandswerte um den Mittelwert streuen und wird aus dem aktuellen Abstandswert x_i und dem Mittelwert $\bar{x}$ berechnet :

$$\sigma = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (x_i - \bar{x})^2} \tag{6.10}$$

Der Mittelwert $\bar{x}$ und die Standardabweichung σ werden aus 50 Einzelmessungen gebildet. Des Weiteren zeigt Tabelle 6.2 die Abweichung $v_{\bar{x}}$ des Mittelwerts von der tatsächlichen Entfernung von 500 mm. Da sich die Laserdiode in einem Abstand von 1 cm zur Außenwand befindet, wird dieser Versatz von jedem dekodierten Abstandswert subtrahiert.

Objekt		Entfernungsmessung bei 500 mm				
		Kaltstart	30 min	45 min	60 min	90 min
Weiß	$\bar{x}$	520,22	510,94	506,92	508,66	506,36
	$v_{\bar{x}}$	20,22	10,94	6,92	8,66	6,36
	σ	2,82	3,04	2,23	3,02	2,35
Schwarz	$\bar{x}$	453,36	455,86	454,72	453,50	452,40
	σ	-46,64	-44,14	-45,28	-46,50	-47,60
	$v_{\bar{x}}$	4,44	2,99	2,95	2,73	2,76
Grau	$\bar{x}$	519,10	510,16	505,24	502,90	503,70
	σ	19,10	10,16	5,24	2,90	3,70
	$v_{\bar{x}}$	2,38	2,47	2,87	2,79	2,51
Hellbraun	$\bar{x}$	513,40	511,30	502,36	499,60	500,08
	σ	13,40	11,30	2,36	-0,40	0,08
	$v_{\bar{x}}$	2,24	2,95	2,26	2,47	2,05

Tab. 6.2.: Testmessung von unterschiedlichen Objekten bei einer konstanten Entfernung von 500 mm. Berechnung des Mittelwerts $\bar{x}$ und der Standardabweichung σ aus 50 Messungen. Die Abweichung $v_{\bar{x}}$ berechnet sich aus der reellen und gemessenen Entfernung.

Während der Aufwärmphase von 30 Minuten sind die Abstandswerte stark schwankend, da die Abweichungen nicht im angegebenen Toleranzbereich des Laserscanners liegen (vgl. Tab. 3.3). Der Grund hierfür, ist auf den Anstieg der internen Sensor Temperatur zurückzuführen, welche aufgrund der Wärmeabgabe des Scankopfmotors steigt. Bei einer Entfernung von 50 cm beträgt die Abweichung für ein weißes Objekt ca. 2 cm, was sowohl an der nicht garantierten Abstandsmessung bis zu 60 cm als auch am Anstieg der Temperatur liegt. Bei einer Entfernung von 1,5 Metern wurde innerhalb von 30.000 Messungen eine konstante Abweichung von mindestens 1 cm während den ersten 30 Minuten festgestellt [39].

Die gemessenen Abstandswerte des Laserscanners URG-04LX sind abhängig von der Oberflächengegebenheit des reflektierten Objekts. Die Testergebnisse zeigen, dass die Abstandswerte bei einem weißen Hindernis geringere Abweichungen haben als bei einem schwarzen Objekt. Des Weiteren ist die gemessene Entfernung bei einem weißen Objekt größer als die tatsächliche Entfernung. Bei schwarzen Hindernissen ist dies nicht der Fall, sondern die gemessene Entfernung ist kleiner der tatsächlichen Größe, was darauf zurückzuführen ist, dass schwarze Objekte das Licht mehr absorbieren und somit die Licht Intensität sinkt (vgl. Tab. 6.2). Dieser Effekt ist unabhängig von der Entfernung.

Da für das SoC-Fahrzeug die Bereiche von einem bis zwei Meter relevant sind, wurde die Testmessung ebenfalls für die Reichweite von 1.000 mm vorgenommen. Unterschiedliche

Objekte wurden im Abstand von einem Meter frontal vor dem Laserscanner positioniert, wobei die Messergebnissen mit den Ergebnissen aus der 500 mm Messung vergleichbar sind (vgl. Tab. 6.3). Die unterschiedlichen Abweichungen sind zum Teil auf die verschiedenen Lichtverhältnisse zurückzuführen. Die konstante Standardabweichung von ca. 2 % zeigt, dass der angegebene Toleranzbereich von 1 % überschritten wird und sowohl in [39] als auch in [53] wurde gezeigt, dass die Abweichung relativ zur Entfernung ist.

Objekt		Entfernungsmessung bei 1000 mm				
		Kaltstart	30 min	45 min	60 min	90 min
Weiß	$\bar{x}$	1030,16	1022,82	1019,02	1017,68	1017,40
	$v_{\bar{x}}$	30,16	22,83	19,02	17,68	17,40
	σ	3,81	2,08	2,55	2,53	2,67
Schwarz	$\bar{x}$	983,34	983,22	984,50	984,56	983,84
	σ	-16,66	-16,78	-15,50	-15,44	-16,16
	$v_{\bar{x}}$	2,48	3,26	2,21	2,60	2,37
Grau	$\bar{x}$	961,50	976,66	975,22	974,88	973,78
	σ	-38,50	-23,34	-24,78	-25,12	-26,22
	$v_{\bar{x}}$	1,99	2,47	2,32	2,80	2,67
Hellbraun	$\bar{x}$	1011,14	1006,02	1003,34	1006,64	1005,16
	σ	11,14	6,02	3,34	6,64	5,16
	$v_{\bar{x}}$	2,67	2,94	3,26	2,59	2,65

Tab. 6.3.: Messung von unterschiedlichen Objekten bei einer konstanten Entfernung von 1.000 mm. Berechnung des Mittelwerts $\bar{x}$ und der Standardabweichung σ aus 50 Messungen.

Da die „Warm-Up Time“ des Laserscanners ca. 30 - 40 Minuten beträgt, wird der stationärer Zustand erst danach erreicht. Helle Objekte zeigen eine starken Einfluss auf die „Warm-Up Time“, da während dieser Zeit eine Annäherung an die tatsächliche Entfernung stattfindet. Hingegen sind die Abstände bei dunklen Objekte nahezu konstant (vgl. Abb. 6.14).

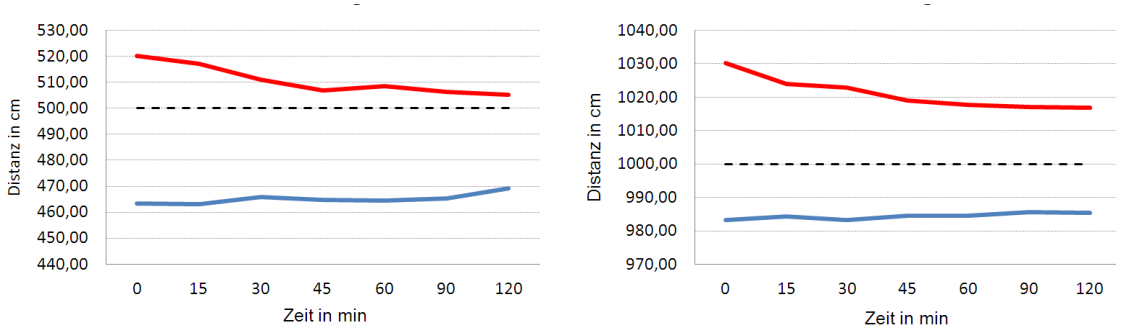


Abb. 6.14.: Unterschied bei weißen und schwarzen Objekten in der Abweichung der Messwerte. Rote Linie kennzeichnet den Verlauf bei weißen Objekten. Blaue Linie kennzeichnet die schwarzen Objekte

7. Segmentklassifizierung zur Objekterkennung

Die Objekterkennung generiert aus benachbarten Abstandswerten Segmente, welche als Objekte der Umwelt interpretiert werden. Für die Objekterkennung des SoC-Fahrzeugs wurden die Konzepte des QNX-basierten SCV-Fahrzeugs analysiert und als Grundlage für die Implementierung verwendet [54] [62]. Zur Datenreduktion wird jedes Segment durch die linken und rechten Begrenzungspunkte sowie dem Punkt mit der geringsten Entfernung zum Hindernis spezifiziert (vgl. Abb. 7.1).

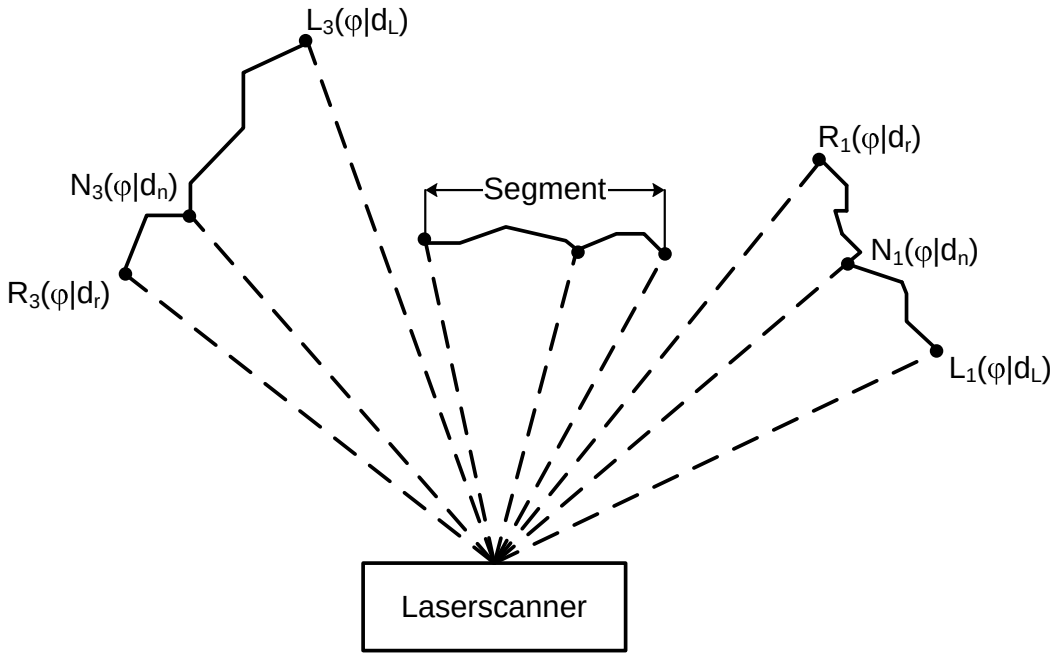


Abb. 7.1.: Spezifizierung eines Segmentes durch drei Punkte in Polarkoordinaten

Die dekodierten Abstandswerte werden als geordnete Punktwolke im *decoded_data_buffer[84]* bereitgestellt und schrittweise zu Segmenten verarbeitet:

- Die 84 Abstandswerte werden mit einem Schwellwertverfahren segmentiert, wobei die drei Punkte in Polarkoordinaten $P(\varphi, d)$ angegeben werden. Die Distanz d wird durch die Abstandswerte gewonnen und der korrespondierende Winkel φ ist durch die definierte Schrittweite des 90° Scanbereichs gegeben.
- Die Abstandswerte, die keinem Segment zugeordnet werden, werden zur Datenreduktion verworfen.
- Nach der Segmentierung werden die Polarkoordinaten in das kartesische Koordinatensystem transformiert, wobei unter Berücksichtigung der Position des Laserscanners alle Segmente im Fahrzeugkoordinatensystem angegeben werden (vgl. Abb. 3.13).

Die Objekterkennung wurde getrennt von der Laserscanner Verarbeitung in einer Source-Datei *ObjectRecognition.c* und einer Header-Datei *ObjectRecognition.h* auf MicroBlaze_0 implementiert. Dies hat den Vorteil, dass durch die Modularität eine Portierbarkeit und eine Erweiterbarkeit gewährleistet ist.

- ***ObjectRecognition.h*:**
  Dient als globale Schnittstelle für die Objekterkennung und enthält alle Funktionsprototypen und Deklarationen. Zur einheitlichen Darstellung werden die Segmenteigenschaften in der Struktur *Segment* gespeichert, welche sowohl die Polar- als auch die Kartesischen Koordinaten der Punkte, sowie die Breite eines Objektes und dessen ID enthält (vgl. Code 7.1).
- ***ObjectRecognition.c*:**
  Die Source-Datei enthält die Implementierung aller Funktionen zur Segmentierung von Abstandswerten anhand eines Schwellwertverfahrens. Funktionen zur Objektverfolgung aus dem SCV-Fahrzeug wurden ebenfalls implementiert(vgl. Abb. D).

```
typedef struct
{
        double  Distance;
        double  Angle;
}SegmentPointPolar;

typedef struct
{
        double  X;
        double  Y;
}SegmentPointCartesian;

typedef struct
{
        unsigned short          ID;                 // unique ID of the segment
        double                  Width;              // the width ob the segment - distance between left and right
        SegmentPointPolar       Left;               // polar coordinate of the left segment point
        SegmentPointPolar       Right;              // polar coordinate of the right segment point
        SegmentPointPolar       Nearby;             // polar coordinate of the nearest segment point
        SegmentPointCartesian   LeftCartesian;      // cartesian coordinate of the left segment point
        SegmentPointCartesian   RightCartesian;     // cartesian coordinate of the right segment point
        SegmentPointCartesian   NearbyCartesian;    // cartesian coordinate of the nearest segment point
}Segment;
```

Listing 7.1: Deklaration der Struktur zur Speicherung der Segmenteigenschaftem

Die Speichergröße eines Segments beträgt 106 Byte, was sich aus der Summe der einzelnen Attribute berechnet. Zur Reduzierung der Speichergröße kann ein Punkt ausschließlich durch eine Koordinate spezifiziert werden. Im Rahmen dieser Arbeit wurde darauf verzichtet, da einerseits genügend externer Speicher zur Verfügung steht anderseits die genaue Struktur des Gesamtsystems noch nicht bekannt ist.

In vergleichbaren Arbeiten wurden momentane Segmente mit nachfolgenden Scans verglichen und somit die Bewegung eines dynamisches Objektes berechnet [58] [30]. In dieser Arbeit wurde eine Implementierung für statische Segmente entwickelt, die Segmente nach jedem Scan neu generiert. Die Funktionen zur Objektverfolgung aus dem Konzept des SCV-Fahrzeugs wurden ebenfalls implementiert, sind jedoch nicht einsetzbar, da im SoC-Fahrzeug keine Rückkopplung des Lenkwinkelsensors stattfindet und somit die Bewegung nicht berechnet werden kann (vgl. Abb. D). Des Weiteren wird das SoC-Fahrzeug auf Rundkursen eingesetzt, wobei eine Gerade meist kürzer als zwei Meter ist und somit aufgrund des Scanbreichs eine Objektverfolgung nicht gewährleistet.

Nachfolgend wird die Implementierung zur Generierung von statischen Segmenten anhand eines Schwellwertverfahrens erläutert.

7.1. Segmentierung der Abstandswerte mit einem Schwellwertverfahren

Bei der Segmentierung werden zusammenhängende Abstandswerte in benachbarten Regionen zu einem Segment vereinigt. Für das SoC-Fahrzeug wurde die Objekterkennung nach dem „Successive Edge Following“ Algorithmus implementiert (vgl. Kap. 4.2). Ein Messwert gehört zu einem Segment, wenn die Differenz aus dem Abstand des rechten Punktes und dem Abstand des aktuellen Punktes kleiner als ein definierter Schwellwert *SEGMENT_THRESHOLD* ist [67]. Durch die 3-Punkte Spezifikation eines Segmentes wird stets der rechte Punkt eines Segmentes aktualisiert (vgl. Abb. 7.1). Die dekodierten Abstandswerte im *decoded_data_buffer[84]* werden der Funktion *generate_segments(unsigned int laser_scan_values[])* übergeben (vgl. Abb. 7.2):

- Das *decoded_data_buffer[84]* Array wird sukzessive durchlaufen und die Distanzen von benachbarten Abstandswerten verglichen.
- Da die 84 Abstandswerte in geordneter Reihenfolge im Array vorliegen, wird der Winkel ausgehend von der Array Position berechnet, wobei der erste Abstandswert an Position null einen Winkel von -45° hat.
- Der aktuelle Winkel *actual_angle* wird bei jeder Iteration mit der Winkelauflösung von 1,05° inkrementiert und der entsprechenden Array Position zugeordnet.
- Mit der Variable *bypass* werden Abstandswerte gezählt, die durch die Dekodierung auf null gesetzt wurden. Somit ist sichergestellt, dass durch die Inkrementierung von *actual_angle* die nachfolgenden Abstandswerte korrekt zu den korrespondierenden Winkeln zugeordnet werden.

Alle Segmente werden in einer verketteten Liste gespeichert, wobei ein next-Zeiger stets auf das nachfolgend Segment zeigt. Im Gegensatz zu einer Array Speicherung hat die verkettete Liste den Vorteil, dass das Einfügen von neuen Segmenten keine komplette Datensatz Verschiebung erfordert, da ausschließlich die next-Zeiger verändert werden. Zum Hinzufügen eines Segments in die globale Liste wurde die Funktion *add_segment_to_list(T_List* segment_list, Segment* segment)* implementiert, die prüft, ob der linke und der rechte Punkt des neuen Segments den gleichen Winkel besitzen und somit das Segment aus nur einem Abstandswert besteht (vgl. Code 7.2).

```
0  int add_segment_to_list(T_List* segment_list, Segment* segment){

       if(segment->Left.Angle != segment->Right.Angle)
       {
           segment->ID     = list_size(segment_list);
5          list_add(segment_list, segment, -1);
           return EXIT_SUCCESS;
       }
       else
       {
10         return EXIT_FAILURE;
       }
   }
```

Listing 7.2: Funktion zum Hinzufügen eines Segments in die verkettete Liste

Da zu Programmstart die Anzahl der Segmente nicht bekannt ist, wird eine dynamische Speicherallokation eingesetzt, die durch den Parameter *config_bufmalloc* im Xilkernel

aktiviert wird. Die Speicherbereiche werden durch *mem_tables* festgelegt, wobei jeder *mem_table* durch die Speicherblockgröße *mem_bsize* und der Blockanzahl *mem_nblks* definiert wird. Durch die Funktion *bufmalloc(membuf, size)* wird Speicher der Größe *size* im Speicherbereich des *membuf* reserviert. Bei der Segmentierung wird für jedes neue Segment Speicher angefordert, wofür eine *mem_table* mit einer Blockgröße von 120 Byte spezifiziert wurde. Ist die *mem_table* nicht bekannt, wird durch den Parameter *MEMBUF_ANY* dem Compiler die Wahl des Buffers überlassen. Jeder angeforderte Speicherbereich wird durch die Funktion *buffree(membuf, mem)* wieder freigegeben [81].

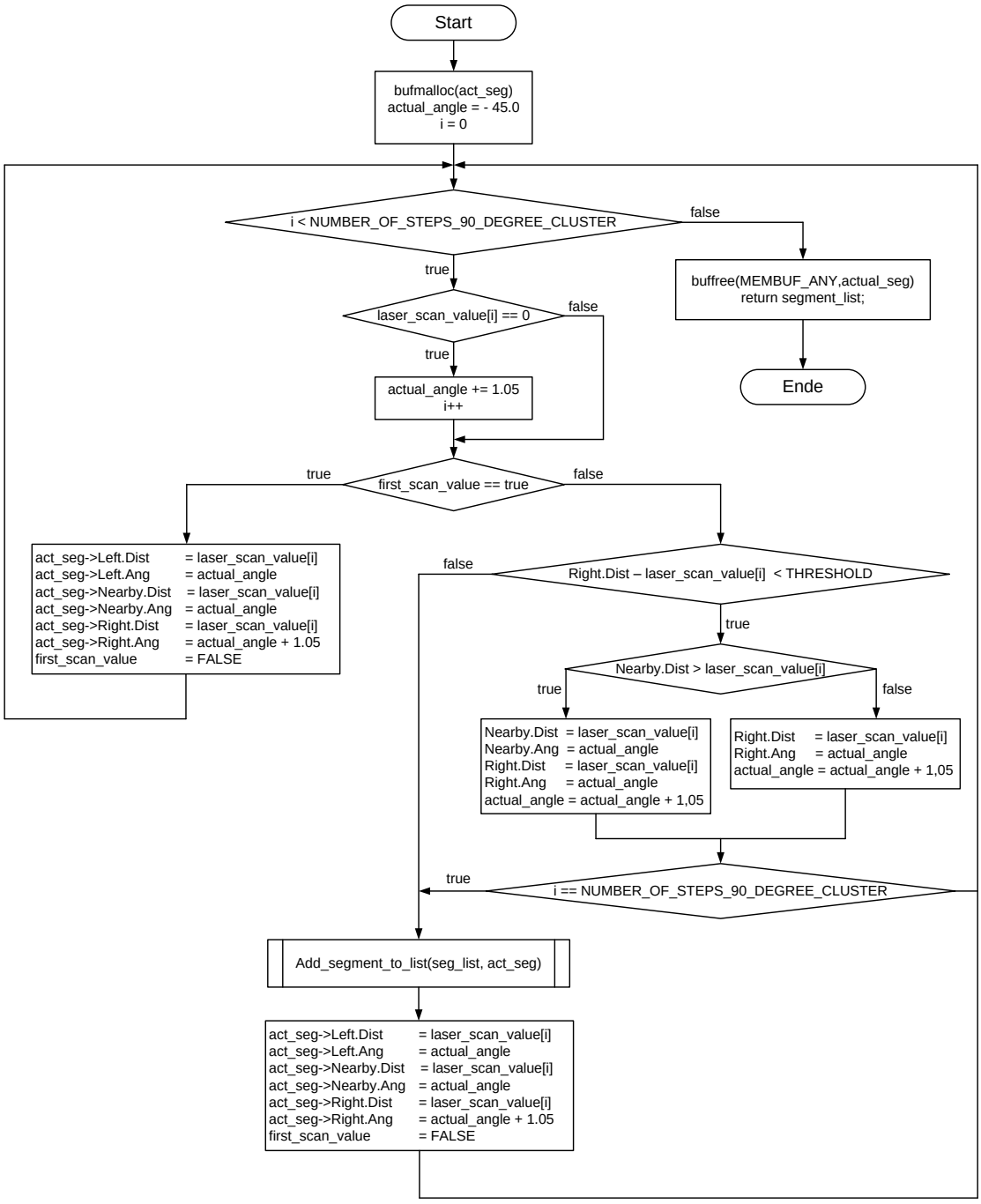


Abb. 7.2.: Programmablaufplan der Funktion generate_segments() zur Erzeugung von Segmenten

Bei Aufruf der Funktion *generate_segments()* wird die globale Segmentliste *segment_list* initialisiert, welche nach Beendigung zurückgegeben wird, wobei nur diejenigen Segmente enthalten sind, die aus mehr als zwei Abstandswerten bestehen und die maximale Reichweite nicht überschreiten. Die Zuordnung von benachbarten Abstandswerten erfolgt durch den Schwellwert *SEGMENT_THRESHOLD*.

- Bei der Iteration der 84 Abstandswerte wird stets das aktuelle Segment *actual_segment* zur Generierung der drei Punkte genutzt
- Im ersten Iterationsschritt ist die Variable *first_scan_value* gesetzt und signalisiert somit die Initialisierung des *actual_segments* (vgl. Abb. 7.2).
- In jedem Iterationsschritt wird die Differenz aus der aktuellen Distanz des rechten Segmentpunkts und dem aktuellen Abstandswert im *laser_scan_values[i]* Array mit dem Schwellwert verglichen
- Ist die Differenz größer als der Schwellwert, dann wird das *actual_segment* in die Liste eingereiht und ein neues Segment generiert.
- Die Aktualisierung des rechten und dichtesten Punktes erfolgt, wenn die Differenz kleiner als der Schwellwert ist
- In der ersten Implementierung der Objekterkennung für das SoC-Fahrzeug wird ein statischer Schwellwert von 70 mm verwendet (vgl. Tab. 7.1).

Die Wahl des Schwellwerts hat einen Einfluss auf die Anzahl der erkannten Segmenten. Ist der Schwellwert zu klein, wird ein Objekt in der Umwelt auf mehrere Segmente verteilt. Bei einem zu großen Schwellwert werden unterschiedliche Objekte in der Umwelt zu einem Segment zusammengefasst (vgl. Abb. 7.3).

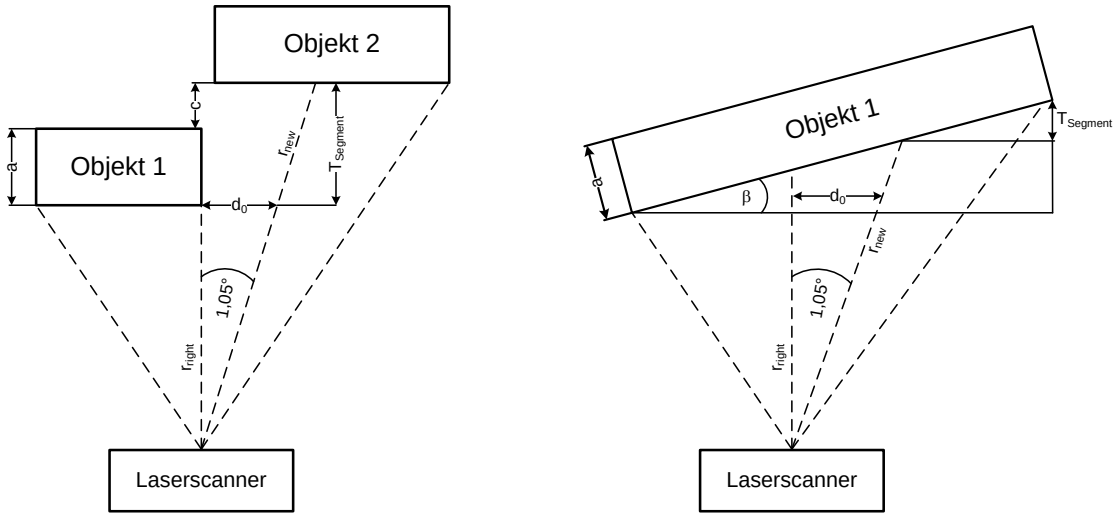


Abb. 7.3.: Vereinigung und Zerlegung von reellen Objekten bei einem zu großen Schwellwert.

Liegen zwei Objekte bei gleicher Entfernung direkt nebeneinander, dann bestimmt der tangentiale Abstand den horizontalen Mindestabstand von zwei Objekten (vgl. Gleichung 6.3). Des Weiteren ist der Schwellwert abhängig von dem maximalen Drehwinkel β des Objektes (vgl. Abb. 7.3 rechts). Steht ein Objekt nicht parallel zum Laserscanner, dann

ergibt sich der minimale Schwellwert aus den trigonometrischen Funktionen des rechtwinkligen Dreiecks. Hierbei ist die Ankathete AK durch den tangentialen Abstand d_0 gegeben und der Drehwinkel β wird experimentell vorgegeben. Durch die Berechnung der Gegenkathete GK wird der Schwellwert ermittelt, welcher die Differenz der beiden Laserstrahlentfernungen kennzeichnet.

$$tan(\beta) = \frac{GK}{AK} \quad \Rightarrow \quad GK = AK * tan(\beta) = d_0 * tan(\beta) \tag{7.1}$$

Nimmt man nun an, dass ein Objekt in einem 45° Winkel zum Laserscanner steht, dann muss der Schwellwert mindestens gleich dem tangentialen Abstand von 36,65 mm sein. Tab. 7.1 zeigt abhängig von der Entfernung die minimalen Schwellwerte für einen definierten Drehwinkel.

Entfernung [mm]	Tangentialer Abstand d_0	Drehwinkel β				
		15°	30°	45°	60°	75°
200	3,67 mm	0,98	2,11	3,67	6,36	13,69
500	9,16 mm	2,45	5,28	9,16	15,87	34,19
700	12,83 mm	3,44	7,41	12,83	22,22	47,88
1000	18,33 mm	4,91	10,58	18,33	31,75	68,41
1500	27,49 mm	7,37	15,87	27,49	47,61	102,59
2000	36,65 mm	9,82	21,16	36,65	63,48	136,78
2500	45,82 mm	12,28	26,45	45,82	79,36	171,00

Tab. 7.1.: Ermittlung des Schwellwerts anhand des tangentialen Abstands d_0 bei verschiedenen Entfernungen. Die Angabe erfolgt in Millimeter

Für die erste Konfiguration der Objekterkennung wurde ein Schwellwert von 70 mm gewählt, was bedeutet, dass Objekte bei einer Entfernung von zwei Metern mit einer maximalen Drehung von ca. 60° erkannt werden. Jedoch hat ein großer Schwellwert den Nachteil, dass bei geringen Entfernungen zusammenstehende Objekte als ein Segment identifiziert werden. In Kap. 9.3 werden Testmessungen zur Validierung des Schwellwertes vorgestellt.

Um die Nachteile eines statischen Schwellwerts zu minimieren, kann ein dynamischer Schwellwert eingesetzt werden [7]. Dieser wird zur Laufzeit abhängig vom aktuellen Abstandswert berechnet. Hierfür wurde auf MicroBlaze_0 eine Enumeration angelegt, bei der die maximale Entfernung von zwei Metern in vier Schwellwertkategorien eingeteilt wird (vgl. Code 7.3). Bei der sukzessiven Abarbeitung des *decoded_data_buffer[84]* wird mit jeder Iteration der Schwellwert dynamisch an die Entfernung angepasst.

```
0  enum DYNAMIC_THRESHOLD{
       THRESHOLD_50_CM      = 20,
       THRESHOLD_100_CM     = 35,
       THRESHOLD_150_CM     = 50,
       THRESHOLD_200_CM     = 70,
5  };
```

Listing 7.3: Enumeration Deklaration für einen dynamischen Schwellwert

7.1.1. Transformation der Koordinatensysteme

Nach der Segmentierung aller Abstandswerte liegen die Segmente in Polarkoordinaten vor. Die drei Punkte des Segmentes werden durch den gemessenen Abstand r und dem Winkel zur x-Achse φ definiert, wobei alle Punkte in der euklidischen Ebene in Bezug auf den Koordinatenursprung angegeben werden. Die x-Achse des Polarkoordinatensystems des Laserscanners verläuft horizontal zur aussendeten Laserdiode und schneidet die y-Achse im Mittelpunkt des Sensors. Durch die Einschränkung des Scanbereichs auf 90° ergibt sich im Polarkoordinatensystem ein Versatz von 45° (vgl. Abb. 7.4 links). Der Winkel φ liegt im Bereich von -45° bis 45°, wobei der Abstand r ein Maximum von 2.000 mm hat.

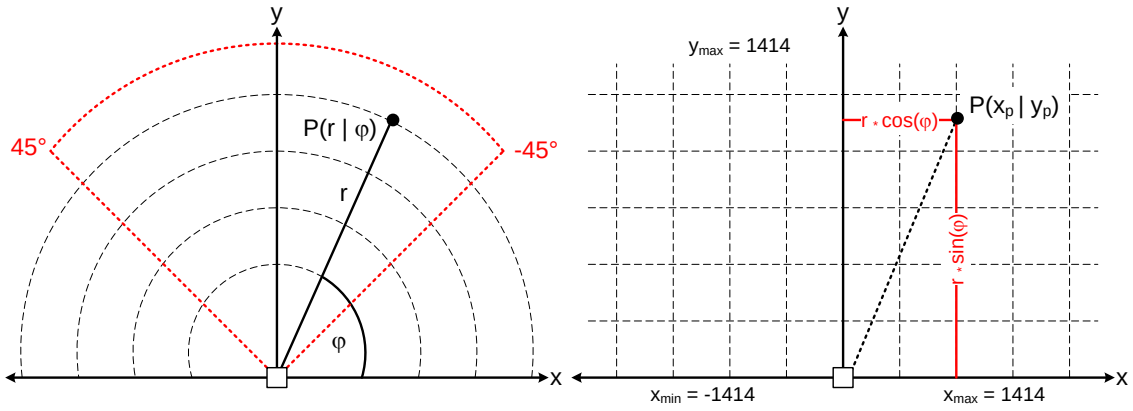


Abb. 7.4.: Darstellung der Segmentpunkte in Polarkoordinaten (links) und im kartesischen Koordinatensystem (rechts). Der Laserscanner dient als Koordinatenursprung.

Im ersten Schritt wird die Liste der Segmente sukzessive iteriert und alle Polarkoordinaten in kartesische Koordinaten transformiert. Die Struktur *Segment* enthält Attribute für beide Koordinaten (vgl. Code 7.1). Ein in Polarkoordinaten angegebener Punkt wird wie folgt in die kartesische Koordinaten x_p und y_p transformiert [52]:

$$x_p = r * cos(\varphi) \tag{7.2}$$

$$y_p = r * sin(\varphi) \tag{7.3}$$

Umgekehrt erfolgt die Transformation von kartesischen in Polarkoordinaten wie folgt. Der Abstand r wird anhand dem Satz des Pythagorases berechnet. Der Winkel φ ist abhängig vom Quadranten und wird durch Fallunterscheidungen berechnet.

$$r = \sqrt{x_p^2 + y_p^2} \tag{7.4}$$

$$\varphi = \begin{cases} \arctan \frac{y_p}{x_p} & wenn: \ x_p > 0 \\ \arctan \frac{y_p}{x_p} + \pi & wenn: \ x_p < 0, \ y_p \geq 0] \\ \arctan \frac{y_p}{x_p} - \pi & wenn: \ x_p < 0, \ y > 0 \end{cases} \tag{7.5}$$

Die Berechnung des Winkels φ kann für die Laserscanner-basierte Objekterkennung eingeschränkt werden, da die kartesische Koordinate y_p aufgrund des frontal ausgerichteten Laserscanners stets positiv ist. Durch die in *math.h* implementierte Funktion *atan2()* werden die Fallunterscheidungen automatisch berücksichtigt und es erfolgt keine manuelle Unterscheidung der Quadranten. Das Maximum und das Minimum für die kartesischen x-Koordinaten ergeben sich aus der maximalen Entfernung d_{max} von 2.000 mm und dem maximalen Winkel von 45°. Daraus folgt, dass die x-Koordinaten der Segmentpunkte im geschlossenen Intervall $[-1414; 1414]$ liegen. Die y-Koordinaten liegen im Intervall $[0; 1414]$, wobei kein Segmentpunkt den Abstand null besitzt. Für die Transformation der Koordinaten wurden vier Funktionen implementiert, die global in *ObjectRecognition.h* definiert sind:

- *double convertPolarToCartesianX (double angle, double distance)*
- *double convertPolarToCartesianY (double angle, double distance)*
- *double convertCartesianToPolarDistance (double x_polar, double y_polar)*
- *double convertCartesianToPolarAngle (double x_polar, double y_polar)*

Die Funktion *transform_segment_coordinates(T_List* segment_list)* wird nach der Segmentierung aufgerufen (vgl. Code 7.4). Hierbei wird über alle Segmente iteriert und die kartesischen Koordinaten für die drei Segmentpunkte berechnet. Da der Laserscanner mittig zum SoC-Fahrzeug angebracht ist, erfolgt die Transformation des Laserscanner Koordinatensystems in das Fahrzeugkoordinatensystem durch die Addition der y-Koordinate mit dem Abstand zur Hinterachse *DISTANCE_LASER_BACK_AXLE*. Eine Verschiebung der x-Achse muss nicht erfolgen, da beide Koordinatensysteme den Ursprung im Mittelpunkt des SoC-Fahrzeugs haben (vgl. Abb. 3.13). Für die Listen Iteration stehen die Funktionen *list_get_first(list, it)* und *list_get_next(it)* bereit. Mit dem Iterator *it* wird die Liste sukzessive durchlaufen. Zuvor wird durch den Aufruf von *bufmalloc()* Speicher für den Iterator reserviert.

```
0  void transform_segment_coordinates(T_List* segment_list){

       Segment* seg;
       T_List_iterator * seg_it;
       seg_it = (T_List_iterator*) bufmalloc(MEMBUF_ANY, sizeof (T_List_iterator));
5
       seg = (Segment*) list_get_first(segment_list,seg_it);
       while(seg != NULL)
       {
           seg->LeftCartesian.X = convertPolarToCartesianX(seg->Left.Angle, seg->Left.Distance);
10         seg->LeftCartesian.Y = convertPolarToCartesianY(seg->Left.Angle, seg->Left.Distance) +
               DISTANCE_LASER_BACK_AXLE;

           seg->RightCartesian.X = convertPolarToCartesianX(seg->Right.Angle, seg->Right.Distance);
           seg->RightCartesian.Y = convertPolarToCartesianY(seg->Right.Angle, seg->Right.Distance) +
               DISTANCE_LASER_BACK_AXLE;

15         seg->NearbyCartesian.X = convertPolarToCartesianX(seg->Nearby.Angle, seg->Nearby.Distance) ;
           seg->NearbyCartesian.Y = convertPolarToCartesianY(seg->Nearby.Angle, seg->Nearby.Distance) +
               DISTANCE_LASER_BACK_AXLE;

       seg = (Segment*) list_get_next(seg_it);
       }
20     buffree(MEMBUF_ANY, seg_it);
   }
```

Listing 7.4: Funktion zur Tranformation der Polarkoordinaten in kartesische Koordinaten

Bei der Transformation der Koordinaten wird von einem statischen Zustand ausgegangen, wobei das Fahrzeug sich direkt auf das Objekt zubewegt und somit keine Veränderung des Lenkwinkels stattfindet. In nachfolgenden Arbeiten muss zur Implementierung

des Ausweichmanövers die Rotation und Translation des Fahrzeugs berücksichtigt werden. Aktuell liegt über die Bewegung des SoC-Fahrzeugs lediglich Informationen zur Ist-Geschwindigkeit vor. Zur Berechnung der Rotation muss ebenfalls der aktuelle Lenkwinkel bekannt sein, was bedeutet, dass eine Rückkopplung des Lenkwinkelstellers erfolgen muss, sodass der aktuell eingeschlagene Winkel zur Verfügung steht. Für die Objekterkennung des Fahrerlosen Transportsystems SCV wurde die Transformation auf Basis der Rotation und Translation berechnet [54][12].

7.1.2. Filterung und Klassifikation der erkannten Segmente

Nach der Segmentierung liegen alle Segmente nach der ID geordnet in der *segment_list* bereit. Die ID wird entsprechend der Winkelposition gegen den Uhrzeigersinn vergeben. Ein Segment, welches am rechten äußeren Scanbereich erkannt wurde, hat eine niedrigere ID, als ein Segment am linken Rand. Zur Feststellung der Relevanz von Segmenten für das Fahrspurführungssystem, wurden Funktionen zur Berechnung der Objektbreite und zur Filterung implementiert. Im Labor vorgenommene Testmessungen haben gezeigt, dass in einem 90° Scanbereich im Schnitt bis zu 10 Segmente identifiziert werden. Dies liegt vor allem an den meist geringen Abständen der Gegenstände innerhalb der Laborumgebung. Jedoch ist für das SoC-Fahrzeug ebenfalls mit mehreren Objekten innerhalb einer Reichweite von zwei Metern zu rechnen, da neben der Fahrspur Menschen und Gegenstände stehen. Aus diesem Grund wurden die Funktionen *get_nearest_segment(T_List*)* und *get_segment_in_front(T_List*)* implementiert, welche beide ein einzelnes Segment aus der Liste zurückgeben.

- ***get_nearest_segment(T_List *)*:**
  Das Segment mit dem kürzesten Abstand zum Laserscanner wird in der Liste gesucht, wobei die gesamte Segmentliste sukzessive iteriert wird und eine temporäre Variable *smallest_distance* stets den kürzesten Abstand speichert. In jedem Iterationsschritt wird durch den Vergleich von *smallest_distance* mit dem *Nearby* Segmentpunkt geprüft, ob das aktuelle Segment näher am Laserscanner liegt als das vorherige.

- ***get_segment_in_front(T_List*)*:**
  Diese Funktion dient der Filterung von Segmenten, indem über die Liste der Segmente iteriert und nach dem Segment direkt vor dem Laserscanner gesucht wird. Der Scanschritt 384 kennzeichnet den Abstandswert bei Winkel 0°. Jedoch ist durch die Reduktion der Daten auf drei Punkte kein Rückschluss auf den eigentlichen Scanschritt mehr gegeben. Für die Suche nach dem Segment direkt vor dem Laserscanner, werden die Winkel der in Polarkoordinaten angegebenen linken und rechten Segmentpunkte verwendet. Hierbei ist zu beachten, dass der Winkel des linken Punktes stets kleiner ist als der Winkel des rechten Segmentpunktes. Das gesuchte Segment wird durch das Produkt der Winkel aus dem linken und rechten Segmentpunkt berechnet. Befindet sich das Objekt direkt vor dem Nullpunkt des Sensors, dann ist der Winkel des linken Segmentpunkts negativ und der Winkel des rechten Punktes positiv. Durch die Variable *smallest_angle* wird stets das aktuelle Ergebnis der Multiplikation mit der vorherigen verglichen. Das kleinste Produkt kennzeichnet das Segment direkt vor dem Laserscanner.

Bei einer detailreichen Umgebung mit vielen Objekten werden die 84 Messwerte in mehreren Segmente gekapselt, wobei jedes Segment unterschiedlich viele Abstandswerte umfasst. Die Breite eines Objektes nimmt Einfluss auf die Entscheidungsfindung für das SoC-Fahrzeug. Die Funktion *calculate_segment_width(..)* berechnet anhand des linken und rechten Punktes die Breite eines Segments und gibt diese zurück, wobei das Ergebnis in der Struktur *Segment* gespeichert wird. Aufgrund der zweidimensionalen Sichtweise des Laserscanners kann keine exakte Breitenberechnung stattfinden. Die Berechnung basiert auf dem Abstand zwischen zwei Punkten, wobei eine Approximation an die tatsächliche Breite stattfindet. Die Funktion *calculate_segment_width(..)* ist sowohl für Polar- als auch für Kartesische Koordinaten ausgelegt. Im Polarkoordinatensystem berechnet sich der Abstand d des linken Punkts $L(r_l, \phi_l)$ zum rechten Punkt $R(r_r, \phi_r)$ wie folgt [52]:

$$d = \overline{LR} = \sqrt{r_l^2 + r_r^2 - 2 * r_l * r_r * \cos(\phi_l - \phi_r)} \tag{7.6}$$

Sind die Segmentpunkte $L(x_l, y_l)$ und $R(x_r, y_r)$ in kartesischer Form angegeben, dann wird d wie folgt bestimmt:

$$d = \overline{LR} = \sqrt{(x_r - x_l)^2 + (y_r - y_l)^2} \tag{7.7}$$

Testmessungen ergaben, dass sich die Breitenberechnung durch die obengenannten Formeln nur für parallel zum Laserscanner positionierte Objekte eignet. Der Grund hierfür ist auf die Annahme zurückzuführen, dass die Kante des Objektes auf der durch die Punkte gehende Gerade liegt (vgl. Abb. 7.5 links). Jedoch ist dies bei schrägen oder nicht geradlinigen Objekten nicht der Fall, da der dichteste Punkt $N(r_n, \phi_n)$ nicht auf der Geraden liegt (vgl. Abb. 7.5 rechts).

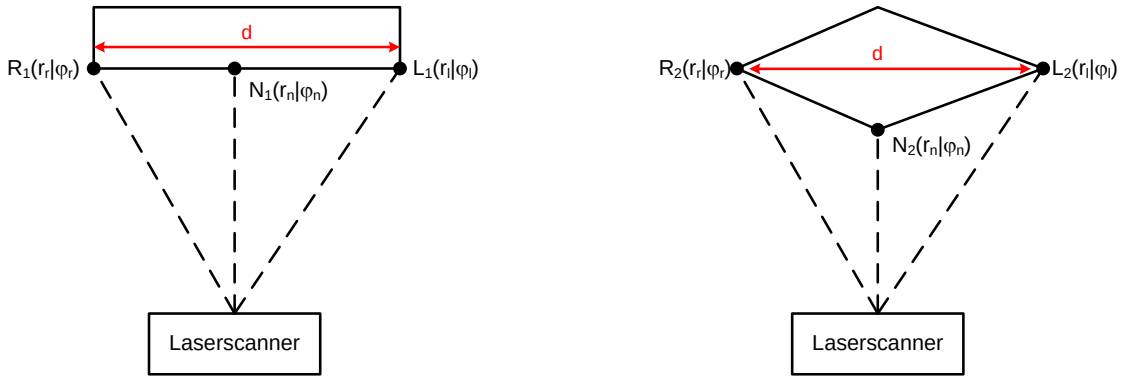


Abb. 7.5.: Approximation der Objektbreite basierend auf dem Abstand vom linken und rechten Segmentpunkt. Ungenaues Ergebnis bei schräg positionierten Objekten (rechts)

Die Anzahl der Abstandswerte pro Segment wird durch die Winkelwerte der linken und rechten Punktkoordinaten berechnet. Ist die Anzahl der Messpunkte und die Entfernung des Segmentes bekannt, kann die Objektbreite ebenfalls mit dem tangentialen Abstand berechnet werden (vgl. Formel 6.5). Hierbei wird der tangentiale Abstand zweier Laserstrahlen mit der Anzahl an Abstandswerten multipliziert und somit eine Gerade approximiert. Die Breitenberechnung ist stets abhängig von der Position des Objekts. Bei überlappenden Objekten gibt die Breite eine Information über den sichtbaren Bereich des Objekts, aber nicht über die tatsächliche Größe.

7.2. Objektverfolgung von dynamischen Objekten

Die aktuelle Konfiguration der Objekterkennung basiert auf statischen Segmenten, welche bei jedem Laserscan neu generiert werden. Zur Verfolgung von dynamischen Objekten wurden die Konzepte aus dem SCV-Fahrzeug analysiert und implementiert [54][62]. Hierbei sind Segmente das Ergebnis der statischen Segmentierung und die Struktur *Object* kennzeichnet real existierende Objekte der Umwelt. Die Struktur *Object* ist vergleichbar mit dem *Segment*, wurde jedoch mit den Geschwindigkeitsvektoren in x und y Richtung erweitert.

Nach jedem Scan werden die aktuellen Segmente mit den im Speicher vorhandenen Segmenten verglichen. Hierbei wird geprüft, ob zwei Segmente aus unterschiedlichen Scans das gleiche reale Objekt beschreiben und die entsprechende Wahrscheinlichkeit in der Variable *match* gespeichert. Das Segment mit dem höchsten *match* Wert wird ausgewählt und das zugehörige Objekte mit der gleichen ID gesucht und anschließend aktualisiert. Ist kein entsprechendes Objekt in der Liste vorhanden, wird ein neues Objekt mit der Funktion *create_object()* erzeugt. Bei der Initialisierung sind alle Geschwindigkeitsvektoren null, da erst beim nachfolgenden Scan eine Aussage über die Bewegung getroffen wird. Wird ein Objekt beim nächsten Scan nicht mehr erkannt, wird das Objekt aufgrund der Eigenbewegung des Fahrzeugs aktualisiert. Dies ist vor allem dann der Fall, wenn durch eine Kurve, Objekte nicht mehr im Sichtbereich des Laserscanners liegen.

Für jeden Objekt- und Segmentpunkt wurde ein Bereich definiert, in dem sich der Punkt während zwei Scans bewegen darf (vgl. Abb. 7.6). Dies ist notwendig, da beim Vergleich von zwei Segmenten aus nachfolgenden Scans, die Gültigkeit des Objektes überprüft wird. Für die Wiedererkennung von Objekten wird der *Nearby* Punkt als Erkennungsmarke verwendet. Die Wahrscheinlichkeit, dass sowohl der rechte als auch der linke Punkt im nachfolgenden Scan nicht wiedererkannt wird, ist sehr hoch. Die mathematischen Zusammenhänge zur Berechnung des Gültigkeitsbereichs ist [54] zu Entnehmen.

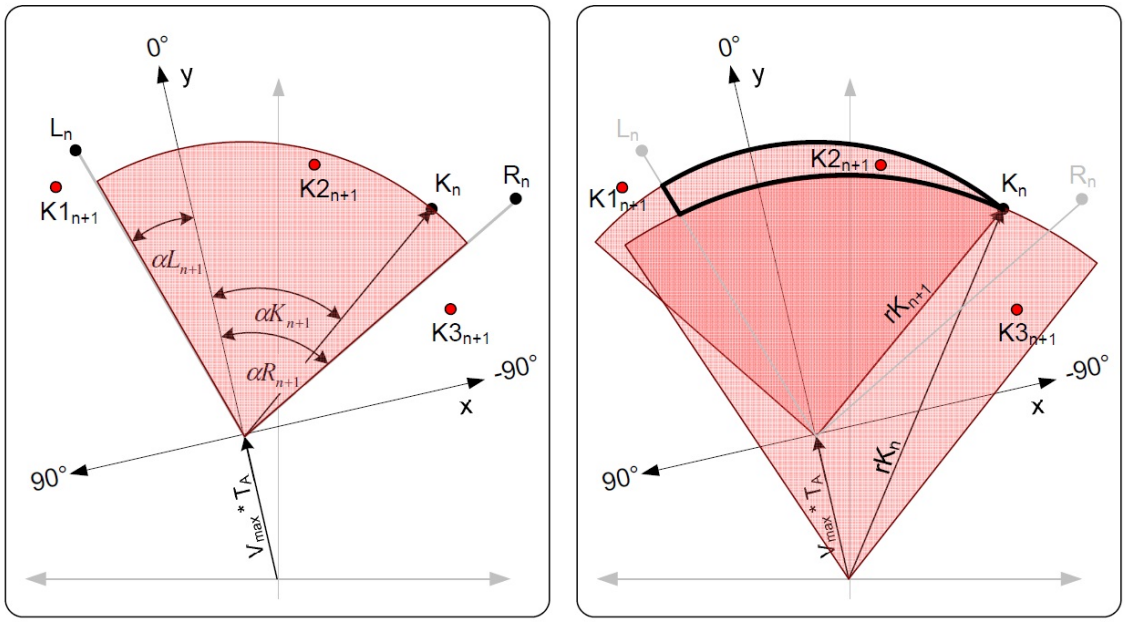


Abb. 7.6.: Gültigkeitsbereich des dichtesten Punktes K in zwei aufeinanderfolgenden Scans [54]

7.3. Bewertung und Test der Objekterkennung

Zum Testen der Laserscanner-basierten Objekterkennung wurden nacheinander verschiedene Objekte im Abstand von 500 mm und 1.000 mm vor dem Sensor positioniert. Wie in Kap. 6.3 erläutert, ist die Entfernungsmessung des Laserscanners abhängig von der Oberflächenstruktur und der Farbe des reflektierten Objekts. Aus diesem Grund wurden Objekte aus unterschiedlichen Materialen und in verschiedenen Farben verwendet (vgl. Tab. 7.2). Die Ergebnisse zeigen, dass bei einer Entfernung von 1.000 mm die Abweichung kleiner ist als bei 500 mm.

Farbe	Material	Breite	Abstand	P_{max}	L_P	R_P	K_P
Weiß	Karton	24,5 cm	500 mm	26	-12,45 530	13,80 526	0,15 510
			1.000 mm	14	-6,15° 1027	7,50° 1035	-3,00° 1023
Hellbraun	Karton	11,5 cm	500 mm	12	-5,10° 506	6,50° 500	5,40° 500
			1.000 mm	7	-3,00° 1021	3,30° 1015	-1,95° 1010
Grau	Kunstoff	16 cm	500 mm	17	-7,20° 493	9,60° 505	-0,90° 488
			1.000 mm	9	-4,05° 986	4,35° 993	0,90° 983
Orange	Schaumstoff	23,5 cm	500 mm	24	-11,40° 529	12,75° 527	-3,00° 519
			1.000 mm	13	-6,15° 1022	6,45° 1027	-5,10° 1018
Schwarz	Kunstoff	31,5 cm	500 mm	32	-16,65° 484	15,90° 459	10,65° 443
			1.000 mm	17	-8,25° 1007	8,55° 1010	2,25° 974
Braun	Karton rund	⊘ 8,3 cm	500 mm	8	-3,00° 506	4,35° 504	2,25° 504
			1.000 mm	5	-1,95° 1011	2,25° 1000	-0,90° 999
Farblos	Glasflasche	⊘ 6 cm	Laserstrahl durchdringt das Objekt				

Tab. 7.2.: Test der Objekterkennung mit unterschiedlichen Objekten im Abstand von 500 mm und 1.000 mm. Die Anzahl der Messpunkte P_{max} nimmt abhängig von der Entfernung ab

Die Abstände fast aller Objekte liegen im Toleranzbereich des Laserscanners. Bei farblosen Materialien, wie Glas oder Kunstoff, durchdringt der Laserstrahl das Objekt und reflektiert von einem weiter entfernten Gegenstand. Dies hat zur Folge, dass derartige Objekte nicht erkannt werden. Für das SoC-Fahrzeug ist dieser Aspekt hinfällig, da die Hindernisse auf der Fahrbahn stets die Farbe weiß haben. Dunklere Objekte werden ebenfalls erkannt, jedoch sind die Abstandswerte mit einer größeren Abweichung behaftet (vgl. Kap. 6.3).

Tests haben gezeigt, dass die Berechnung der Objektbreite durch die Funktion *calculate_segment_width(..)* ebenfalls mit einer Abweichung behaftet ist. Die berechnete Breite ist meist kleiner als die tatsächliche Objektbreite. Je größer die Entfernung desto größer ist die Abweichung. Der Grund hierfür, ist auf den exponentiell steigenden Öffnungswinkel bei größer werdender Entfernung zurückzuführen. Treffen die Laserstrahlen direkt auf die linke und rechte Kante des Objektes, dann kann die korrespondierende Breite exakt bestimmt werden und die Abweichung geht Richtung null. In den meisten Situation treffen die Laserstrahlen jedoch nicht exakt auf die Kante und es ergibt sich eine Abweichung zwischen Reflexionspunkt und Eckpunkt. Die Breitenberechnung hat somit falsche Eckkoordinaten und das Ergebnis ist kleiner als die tatsächliche Breite. Die maximale Abweichung ergibt sich aus dem von der Entfernung abhängigen tangentialen Abstand.

Für das SoC-Fahrzeug wird durch die Filterung und die Klassifizierung der Segmente, die Entscheidungsfindung auf Seiten des MicroBlazes_0 getroffen. Beim Controller MicroBlaze_1 entsteht kein zusätzlicher Overhead für die Suche nach dem nächsten relevanten Objekt. Da der Laserscanner fest an der Vorderseite des SoC-Fahrzeugs angebracht ist, geht der eingeschlagene Lenkwinkel direkt auf den Laserscanner über und es entsteht kein Versatz.

Die Funktion *get_segment_in_front(T_List*)* liefert abhängig von der Laserscannerfrequenz stets das Segment, welches sich mittig zum Fahrzeug befindet. Das Segment mit dem kürzesten Abstand ist ebenfalls von der 10 Hz Frequenz des Sensors abhängig. Bei Aufruf von *get_nearest_segment(T_List *)* wird auf die Messwerte des letzten Scans zurückgegriffen. Jedoch hat sich das Fahrzeug während der Messdatenerfassung und der Segmentierung mit einer konstanten Geschwindigkeit bewegt. Da die Filtermethoden erst nach der Erfassung und der Segmentierung aufgerufen wird, ergibt sich eine Abweichung der Abstandswerte aufgrund der Bewegung des Fahrzeugs. Wird davon ausgegangen, dass sich das SoC-Fahrzeug mit einer konstanten Geschwindigkeit von 2 m/s bewegt, dann ist die zurückgelegte Strecke bis zur Segmentierung abhängig von der Verarbeitungsgeschwindigkeit. Dieser Toleranzbereich von unter einem Millimeter wird für den Aufruf der Filtermethoden akzeptiert.

7.4. Visualisierung von Objekten mit einem JAVA Applet

Die in einer Liste gespeicherten Segmente repräsentieren reelle Objekte der Umwelt. Zum Testen der Objekterkennung und zur Visualisierung der Objekte wurde ein Java Applet entwickelt, welches die Segmente in einem kartesischen Koordinatensystem aufzeigt (vgl. Abb. 7.7). Das Programm *VisualizeLaserObjects* enthält eine Klasse *VisualizeMain*, die mit einer Einfachvererbung durch die Klasse *Applet* erweitert wird. Durch die in hierarchischen Relationen angeordneten Klassen werden der Unterklasse die Funktionen der Oberklasse zur Verfügung gestellt.

Für die Visualisierung eines Objekts werden die kartesischen Koordinaten des rechten und linken Segmentpunkts in einer Textdatei gespeichert. Diese enthält alle zu visualisierenden Segmente, wobei die einzelnen Segmente durch eine Leerzeile getrennt werden. Über den *ActionListener()* des Buttons *Open File* kann der Anwender einen „Datei Öffnen“ Dialog starten. Hierfür wird ein Java spezifischer *JFileChooser* genutzt, der zur Laufzeit die Dateiendung auf **.txt* überprüft und gegebenenfalls eine Fehlermeldung anzeigt.

Mit dem Button *Visualize* werden die Objekte im kartesischen Koordinatensystem angezeigt. Durch das Überschreiben der *paint()* Methode wird bei jedem Klick das Koordinatensystem neu gezeichnet, wofür im zugehörigen *ActionListener()* die Methode *repaint()* aufgerufen wird. Die neu definierte *paint()* Methode zeichnet zu Beginn stets ein leeres Koordinatensystem durch die Funktion *paint_coordinate_system()*. Anschließend wird die Funktion *paint_point_line()* aufgerufen, welche die Zeilen der ausgewählte Datei mit einem *BufferedReader* einliest und in einem Array speichert. Jedes Segment wird durch zwei x-Koordinaten und zwei y-Koordinaten beschrieben. Zur Darstellung eines Objekts im Koordinatensystem wird eine Linie durch den rechten und linken Segmentpunkt gezeichnet. Das *Graphics2D* Koordinatensystem stellt hierfür die Methode *drawLine* zur Verfügung, welche die vier Koordinaten als Übergabeparameter bekommt. Zusätzlich werden die angezeigten Objekte textuell in einer *TextArea* dargestellt.

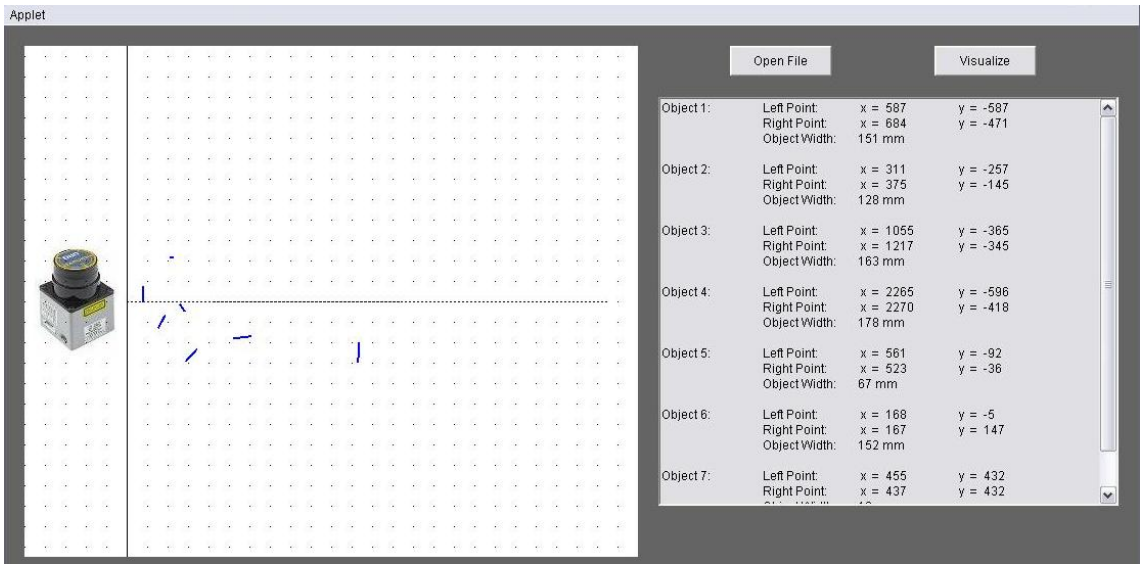


Abb. 7.7.: Java Applet zur Visualisierung von erkannten Objekten

8. Integration der Objekterkennung in das Multiprozessor System

Zur Funktionsanalyse der MPSoC Plattform wurde die Laserscanner-basierte Objekterkennung in das Dual-MicroBlaze System integriert. MicroBlaze_0 und MicroBlaze_1 kommunizieren über eine direkte FSL Kopplung und sind mit einem Xilkernel RTOS ausgestattet (vgl. Abb. 5.1). Der gemeinsame Speicher ist für beide Prozessoren über separate XCL Busse zugänglich, derren Zugriffe mit dem XPS Mutex synchronisiert werden. Die Umgebungserfassung und die Objekterkennung werden auf MicroBlaze_0 ausgeführt. Der MicroBlaze_1 dient als Steuerungsprozessor, welcher über die FSL Verbindung Kommandos an MicroBlaze_0 sendet (vgl. Abb. 8.1). Die Integration der Laserscanner-basierten Objekterkennung in das POSIX Thread Modell und die Funktionsweise der Objekterkennung im Kontext des Multiprozessorsystems wird nachfolgend erläutert.

8.1. Software Partitionierung auf zwei MicroBlazes

Beide MicroBlazes nutzen das Xilkernel RTOS zum Scheduling von POSIX Threads, wobei durch die asymmetrische Prozessorarchitektur zwei unterschiedliche Software Programme ausgeführt werden. Die Laserscanner-basierte Umgebungserfassung und die Objekterkennung wird auf MicroBlaze_0 in ein POSIX Thread Modell mit zwei Threads integriert. Der als Controller arbeitende MicroBlaze_1 nutzt drei Threads zur Kommunikation mit MicroBlaze_0:

- **MicroBlaze_0**
  - **master_thread:** Empfängt Steuersignale von MicroBlaze_1 und startet die Laserscanner-basierte Objekterkennung
  - **laser_measurement:** Sendet GS-Kommando zum Empfang von 84 Abstandswerten an Laserscanner. Nach der Dekodierung und Koordinatentransformation wird eine Statusnachricht an MicroBlaze_1 gesendet
- **MicroBlaze_1**
  - **master_thread:** Startet die Threads für den FSL Empfang und Sendevorgang
  - **thread_fsl_send:** Sendet Steuersignale abhängig von den Push Buttons an MicroBlaze_0
  - **thread_fsl_recv:** Empfängt Statussignale von MicroBlaze_1

Die nachfolgende Tabelle 8.1 zeigt die detaillierte Aufgabenverteilung auf beiden MicroBlazes sowie die zugehörige Thread Integration.

Prozessor	Thread	Aufgabenbeschreibung	Funktion	Peripherie
MB_0	master_thread	Steuersignale von MB_1 empfangen	getfslx()	FSL
		Objekterkennung initialisieren	scip_laser_initialize()	UART Laser
		Plattfrom initialisieren	init_platform()	UART, Mutex, LED
		Thread erzeugen	pthread_create()	keine
		Objekterkennung starten	scip_laser_get_data_90degree()	keine
	laser_measurement	Laserscandaten anfordern	scip_laser_get_current_data()	UART Laser
		198 Byte Rohdaten empfangen	scip_laser_get_current_data()	UART Laser
		Dekodierung von 84 Abstandswerte	decode_byte_buffer()	keine
		Generierung von Segmenten	generate_segments()	Mutex
		Transformation von Koordinaten	transform_segment_coordinates()	keine
		Filterung und Klassifikation	calculate_segment_width()	keine
		Statussignale an MB_1 senden	putfslx()	FSL, LED
MB_1	master_thread	Erzeugung von zwei Threads	pthread_create()	keine
		Plattform initialisieren	init_platform()	Mutex, Push Buttons
		Warten auf Thread Beendigung	pthread_join()	keine
	thread_fsl_send	Auswertung der Push Buttons	XGpio_DiscreteRead()	Push Butons
		Steuersignale an MB_0 senden	putfslx()	FSL
	thread_fsl_recv	Statussignale von MB_0 empfangen	getfslx()	FSL
		Ausgabe auf RS232 Terminal	xil_printf()	JTAG UART

Tab. 8.1.: Aufgabenverteilung und Software Partitionierung auf MicroBlaze_0 und MicroBlaze_1 mit zugehörigen Threads und Funktionen sowie der verwendeten Peripherie (vgl. Abb. 8.1)

Auf MicroBlaze_0 wird die Laserscanner-basierte Objekterkennung in einem Zwei-Thread Modell mit einem prioritätenbasierten Scheduling ausgeführt (vgl. Abb. 8.1). Der Master Thread wird vor dem Start des Xilkernels durch den Aufruf von *xmk_add_static_thread(master_thread, 1)* mit einer Priorität von eins erzeugt. Die Aufgabe des Master Threads auf MicroBlaze_0 besteht darin, die von MicroBlaze_1 durch den *fsl_send* Thread gesendeten Kommandos zu empfangen und den Laserscanner zu steuern. Der Thread *thread_fsl_send* auf MicroBlaze_1 wertet die Eingabe des Anwenders durch ein kontinuierliches Polling auf die Push Button GPIO aus und sendet das entsprechende Kommando über den FSL an MicroBlaze_0. Die Nummer des Push Buttons ist gleich dem Kommando, welches als Integer Wert über den FSL übertragen wird (vgl. Kap. 6.1):

- PB-1: Anschalten des Laserscanners durch das BM-Kommando
- PB-2: Ausschalten des Laserscanners durch das QT-Kommando
- PB-3: Einmaliger Empfang von Messdaten und für Test Szenarien
- PB-4: Beginn der periodischen Messdatenerfassung und der Objekterkennung

Empfängt der Master Thread auf MicroBlaze_0 das Kommando PB-4, dann wird der Thread *thread_laser_measurement* gestartet, welcher durch Senden des GS-Kommandos die 198 Byte an Messdaten vom Laserscanner empfängt (vgl. Kap. 6.1). Nach der Dekodierung der 84 Abstandswerte erfolgt die Segmentierung zur Objekterkennung. Bei Beendigung des *thread_laser_measurement* Threads bekommt der Master Thread die CPU zugeteilt und empfängt das nächste Kommando vom MicroBlaze_1.

Die erkannten Objekte werden im geteilten Speicher abgelegt, wofür im Linker Script beider MicroBlazes eine neue „Shared Memory Region“ mit einer definierten Größe angelegt wurde (vgl. Code 8.1). Diese hat auf beiden Prozessoren die gleiche Startadresse *_sharedmem_start* und die gleiche Endadresse *_sharedmem_end*. Damit Daten im geteilten Speicher abgelegt werden können, wird in den Header Dateien der MicroBlazes die Variable *_sharedmem_start* als globale Startadresse mit dem Keyword *extern* deklariert. Somit haben beide MicroBlazes die Startadresse des geteilten Speichers und die gemeinsame Variablen können als Pointer auf die Startadresse und einen Offset deklariert werden. Der Zugriff wird über den XPS Mutex synchronisiert, sodass beide MicroBlazes stets auf aktuellen Daten arbeiten. Nach der Segmentierung durch *generate_segments()* auf MicroBlaze_0 wird der Mutex gesperrt und die Daten im gemeinsamen Speicher abgelegt. Anschließend sendet MicroBlaze_0 eine Statusnachricht an MicroBlaze_1, der bei Empfang den Mutex sperrt und dem Anwender die ausgelesen Daten über den JTAG UART ausgibt (vgl. Abb. 8.1).

```
0  MEMORY
   {
      ilmb_cntlr_mb_1_new_dlmb_cntlr_mb_1_new : ORIGIN = 0x00000050, LENGTH = 0x00001FB0
      DDR2_SDRAM_MPMC_BASEADDR : ORIGIN = 0x8A000000, LENGTH = 0x02000000
      DDR2_SDRAM_SHARED_MEM_BASEADDR : ORIGIN = 0x8C000000, LENGTH = 0x04000000
5  }

   .sharedmem : {
      __sharedmem_start = .;
      *(.sharedmem)
10    __sharedmem_end = .;
   } > DDR2_SDRAM_SHARED_MEM_BASEADDR
```

Listing 8.1: „Shared Memory Region“ im Linker Script beider MicroBlazes (vgl. Anhang C)

Die Laserscanner-basierte Objekterkennung wird durch ein über den FSL gesendetes Kommando vom Controller MicroBlaze_1 gestartet (vgl. Abb. 8.2). Dieser arbeitet mit einem Drei-Thread Modell, wobei der Master Thread die Threads *thread_fsl_send* und *thread_fsl_recv* startet (vgl. Abb. 8.1). Durch den Aufruf der *pthread_join(thread)* Methode wartet der Master Thread auf die Beendigung der zwei Sub-Threads. Da diese in einer Endlosschleife die FSL Kommunikation mit MicroBlaze_0 steuern, wird der Master Thread ausschließlich durch einen Reset neu initialisiert.

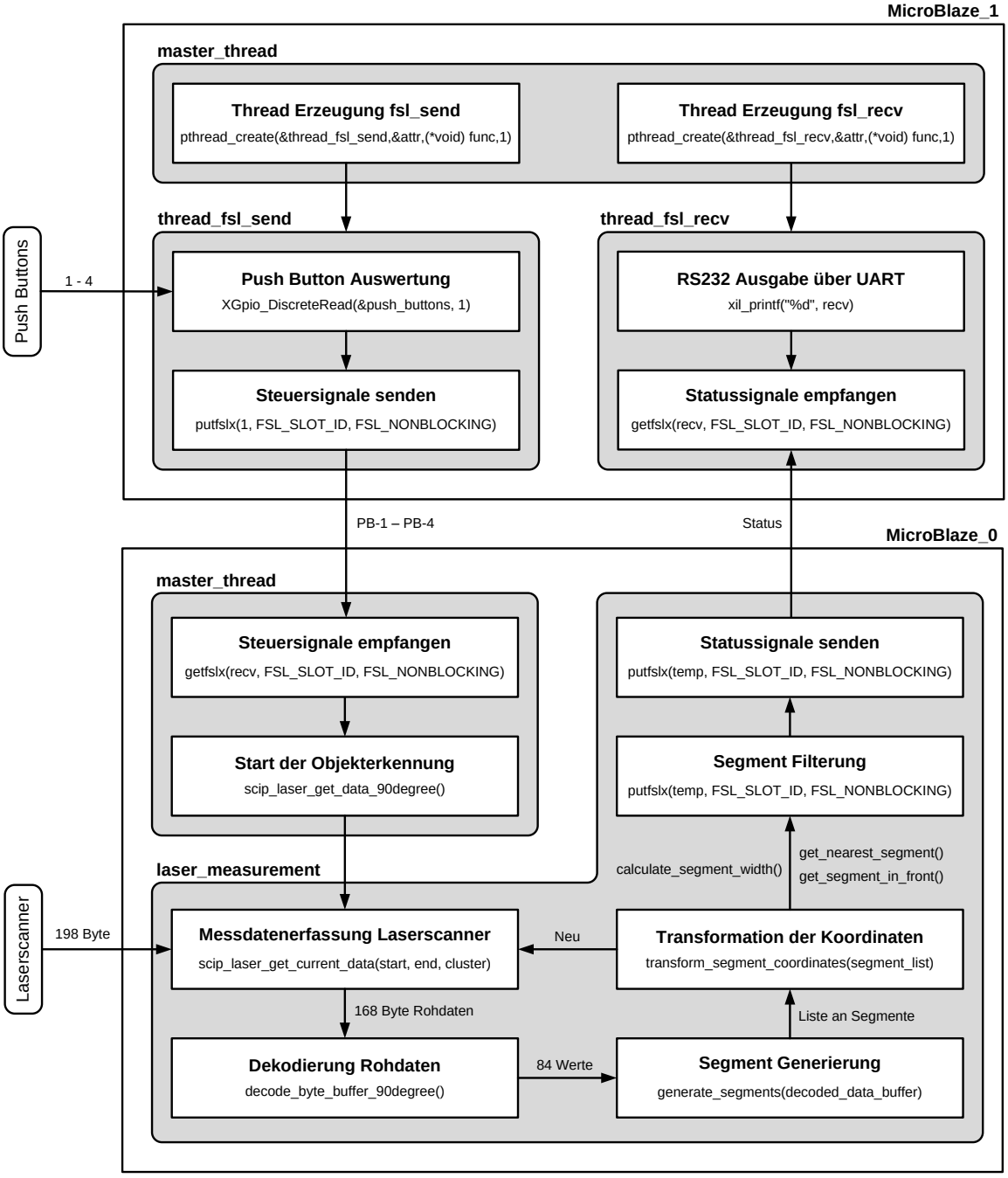


Abb. 8.1.: Threadkonzept der Laserscanner-basierten Objekterkennung. Controller MicroBlaze_1 nutzt drei Threads und MicroBlaze_0 nutzt zwei Threads

Das Senden der Kommandos durch den Thread *thread_fsl_send* auf MicroBlaze_1 erfolgt stets nicht blockierend. Hierfür wird der *putfslx()* und der *getfslx()* Funktion des FSL Busses der Parameter *FSL_NONBLOCKING* übergeben. Tests haben ergeben, dass bei einer blockierenden Funktionsweise des FSL Busses, ein Reset Signal nicht ordnungsgemäß den Prozessor neustartet. Des Weiteren wird durch eine blockierende FSL Funktion der Threadwechsel verhindert, da der Scheduler einem blockierenden Thread die zugeteilte CPU nur entziehen kann, wenn in der „Ready Queue“ ein höher priorisierter Thread wartet. Da auf MicroBlaze_0 der *thread_laser_measurement* höher priorisiert ist als der Master Thread, kann bei einem blockierenden FSL-Empfang die CPU nicht entzogen werden. Aus diesem Grund arbeiten alle FSL Funktionen auf beiden MicroBlazes nach dem nicht blockierenden Kommunikationsprinzip.

Der zweite Thread *thread_fsl_recv* auf dem Controller MicroBlaze_1 dient dem Empfang von Statussignalen, die nach der Segmentierung von MicroBlaze_0 gesendet werden, sowie zur seriellen Ausgabe von Informationen über den JTAG Uart(vgl. Abb. 8.2). Der Thread wird parallel zum *thread_fsl_send* vom Master Thread gestartet. Im Vergleich zu MicroBlaze_0 verwendet der MicroBlaze_1 ein Round-Robin Scheduling, da kein Thread bevorzugt behandelt werden muss, sondern ein Threadwechsel gleichberechtigt jede Millisekunde erfolgen kann. Empfängt der *thread_fsl_recv* Thread Daten vom FSL, dann wird eine definierte Adresse im gemeinsamen Speicher ausgelesen und auf der Konsole des JTAG Uarts die von MicroBlaze_0 erkannten Objektparameter ausgegeben. Zuvor wird der initialisierte Mutex zum Schutz der „Shared Memory Region“ gesperrt und nach einem erfolgreichem Lesevorgang wieder freigegeben (vgl. Kap. 5.5).

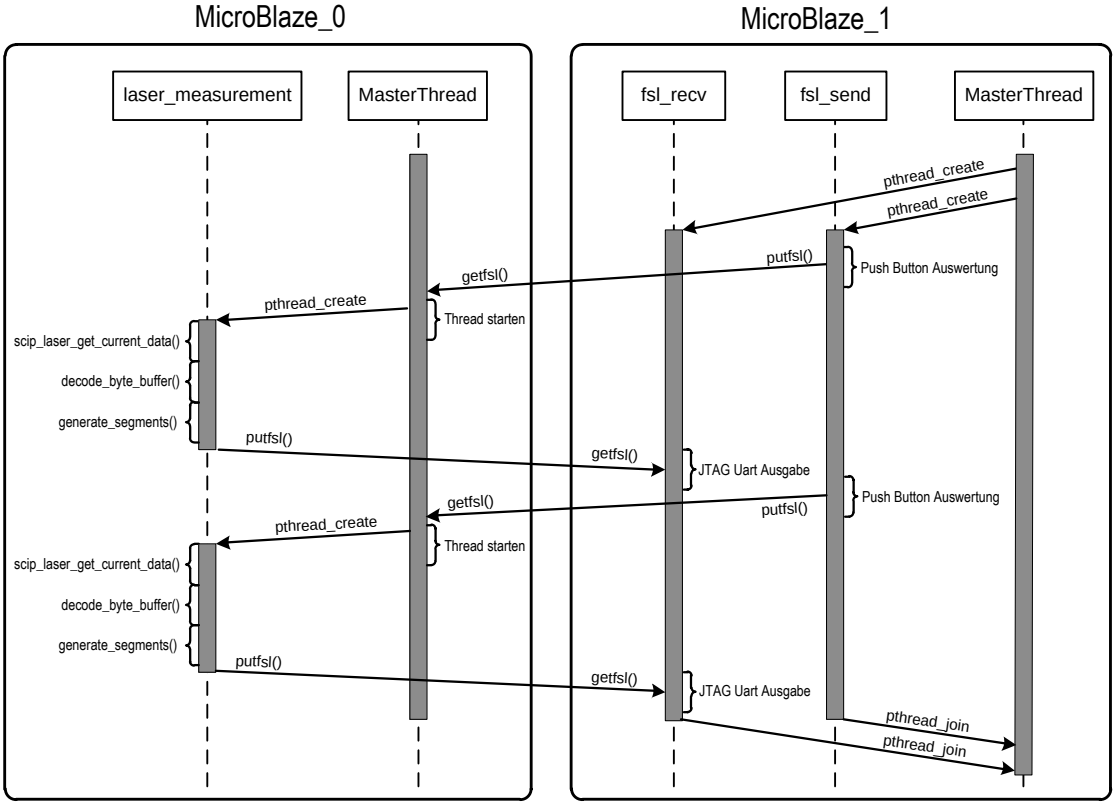


Abb. 8.2.: Sequenzdiagramm des Dual-MicroBlaze Systems zur Messdatenerfassung und zur Objekterkennung beim PB-4 Kommando (vgl. Abb. 8.1)

8.2. Scheduling des POSIX Thread Modells

Das Xilkernel RTOS verfügt über zwei statische Schedulingstrategien, die über das „Board Support Package (BSP)“ festgelegt werden (vgl. Kap. 4.4). Das Round-Robin Verfahren *SCHED_RR* arbeitet mit einer einzigen FIFO-basierten „Ready-Queue“, wobei alle Threads die gleiche Priorität besitzen und somit gleichberechtigt die CPU zugeteilt bekommen [81]. Die Abarbeitung der Threads erfolgt mit einem definierten Zeitintervall (Time-Slice), wobei nach Ablauf des Bearbeitungszeitintervalls ein neuer Thread nach dem FIFO-Prinzip aus der „Ready Queue“ gestartet wird. Das Xilkernel RTOS zur Objekterkennung auf MicroBlaze_0 wird mit einem prioritätenbasierten Scheduling *SCHED_PRIO* konfiguriert (vgl. Abb. 5.1). Hierbei hat ein Thread, der die höchste Priorität null besitzt, stets das Vorzugsrecht vor niedrig priorisierten Threads (vgl. Abb. 8.3). Threads mit gleicher Priorität werden nach dem Round-Robin Verfahren abgearbeitet. Der Parameter *SYSTMR_INTERVAL*, der die Größe der „Time-Slices“ und somit das Threadwechsel Intervall angibt, wurde für das SoC-Fahrzeug auf 1 ms festgelegt. Dies hat bei einem prioritätenbasierten Scheduling den Vorteil, dass ein höher priorisierter Thread beim Zustandswechsel von wartend auf rechenbereit, sofort die CPU zugeteilt bekommt und somit niedrig priorisierte Threads verdrängt. Sind zwei Threads mit der gleichen Priorität rechenbereit, dann erfolgt jede Millisekunde ein Threadwechsel.

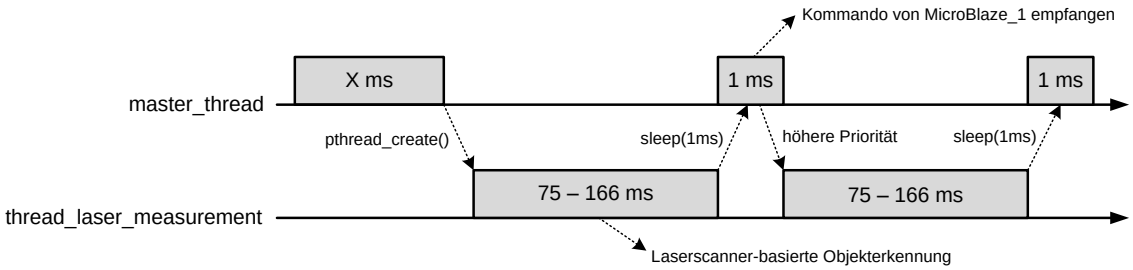


Abb. 8.3.: Prioritätenbasiertes Scheduling auf MicroBlaze_0. Master Thread die CPU nur zugeteilt, wenn ein sleep() aufgerufen wird

Auf MicroBlaze_0 wird mit *xmk_add_static_thread(master_thread, 1)* vor dem Start des Xilkernels ein Master Thread mit der Priorität eins angelegt (vgl. Abb. 8.1). Dieser empfängt das PB-4 Kommando vom Controller MicroBlaze_1 über die FSL Kopplung und erzeugt für die Laserscanner-basierten Objekterkennung den mit der Priorität null konfigurierten *thread_laser_measurement* Thread. Da durch das prioritätenbasierte Scheduling stets der höchst priorisierte Thread abgearbeitet wird, kann der *thread_laser_measurement* Thread nicht vom Master Thread unterbrochen werden. Da somit der Master Thread keine weiteren FSL-Kommandos vom Controller MicroBlaze_1 empfangen kann und somit blockiert ist, wird dies durch eine *sleep()* Anweisung am Ende einer Messdatenerfassung kompensiert. Dem *thread_laser_measurement* Thread wird die CPU für eine Millisekunde entzogen, indem er in die „Ready-Queue“ eingereiht wird und der Master Thread bekommt die Ressource zugeteilt (vgl. Abb. 8.3). Ein von MicroBlaze_1 gesendetes Steuersignal wird aus der FSL FIFO gelesen und verarbeitet. Mit dem aktuellen Thread Modell hat der Thread zur Messdatenerfassung und Objekterkennung stets Vorrang vor dem Master Thread.

Auf MicroBlaze_1 wird das Round-Robin Verfahren mit einem „Time-Slice“ von einer Millisekunde zum Scheduling eingesetzt. Bei Start des Xilkernels auf MicroBlaze_1 erzeugt der Master Thread die Threads *thread_fsl_send* und *thread_fsl_recv*. Durch die *pthread_join()* Anweisung wartet der Master Thread auf die Beendigung der Sub-Threads, da diese jedoch in einer Endlosschleife die FSL-Kommunikation mit MicroBlaze_0 steuern, bekommt der Master Thread erst bei einem Reset die CPU zugeteilt. Der *thread_fsl_send* Thread wertet die Push Buttons aus und sendet die Steuersignale an MicroBlaze_0 (vgl. Tab. 8.1), wobei jede Millisekunde ein Threadwechsel zu *thread_fsl_recv* stattfindet, der auf Statusinformationen von MicroBlaze_0 wartet (vgl. Abb. 8.4).

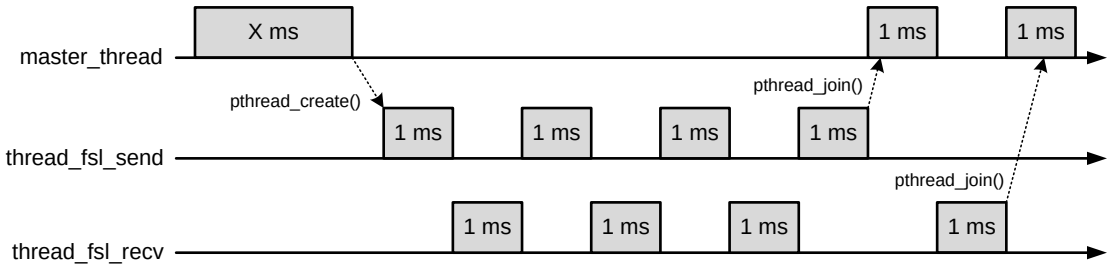


Abb. 8.4.: Round-Robin Scheduling auf MicroBlaze_1. Der Master Thread warten auf die Beendigung beider Threads

Auf Microblaze_0 wird die Laserscanner-basierte Objekterkennung im *thread_laser_measurement* Thread periodisch ausgeführt, indem innerhalb einer Schleife die Funktion *scip_laser_get_data_90degree()* mehrfach aufgerufen wird. Dies wird nicht wie gewöhnlich durch eine Interrupt generierte Abtastperiode innerhalb einer ISR realisiert, sondern durch die *sleep()* Anweisung am Ende der Funktion (vgl. Abb. 8.3). Die Gründe, dass keine konstante Abtastperiode verwendet wird, sind:

- Eine messtechnische Analyse ergab, dass das Ausführungsintervall für die Funktion *scip_laser_get_data_90degree()* stark schwankend zwischen 75 ms - 166 ms ist (vgl. Tab. 9.1)
- Das Ausführungsintervall für die Segment Generierung ist nicht konstant, da die Anzahl an Objekten in der Umwelt variabel ist und somit die Zeit für die Segmentierung und Transformation proportional der Anzahl ist.
- Da aufgrund der Laserscanner Frequenz von 10 Hz die minimale Abtastperiode 100 ms beträgt, würde eine zu groß gewählte Abtastperiode dazu führen, dass MicroBlaze_0 die CPU Zeit nicht ausnutzt.
- Eine Interrupt generierte Abtastperiode würde den Einsatz von einem zusätzlichen XPS Timer erfordern, der über einen Interrupt Controller mit den MicroBlazes gekoppelt wird. Da der Interrupt Eingang des MicroBlazes bereits durch den Xilkernel Timer belegt ist, kann keine direkte Kopplung erfolgen.

Für die Objekterkennung wird keine feste Abtastperiode genutzt, da eine Berechnung der optimalen Abtastperiode aufgrund der sehr schwankenden Ausfühungsintervalle nicht realisier ist.

9. Messtechnische Analyse und Ergebnisse der Objekterkennung

Der Laserscanner URG-04LX arbeitet mit einer Frequenz von 10 Hz, wobei die Abstandswerte entweder einmalig mit dem GS-Kommando oder kontinuierlich mit dem MS-Kommando angefordert werden. Bei letzterem werden die angeforderten Messdaten alle 100 ms übertragen (vgl. Abb. 9.1). Für die Objekterkennung wird das GS-Kommando, welches die 84 Abstandswerte einmalig sendet, periodisch von MicroBlaze_0 versendet (vgl. Kap. 8.2). Das GS-Kommando wurde gewählt, da das MS-Kommando eine höhere Fehlerrate aufweist und aufgrund den zu hohen Verarbeitungszeiten (vgl. Kap. 9.2).

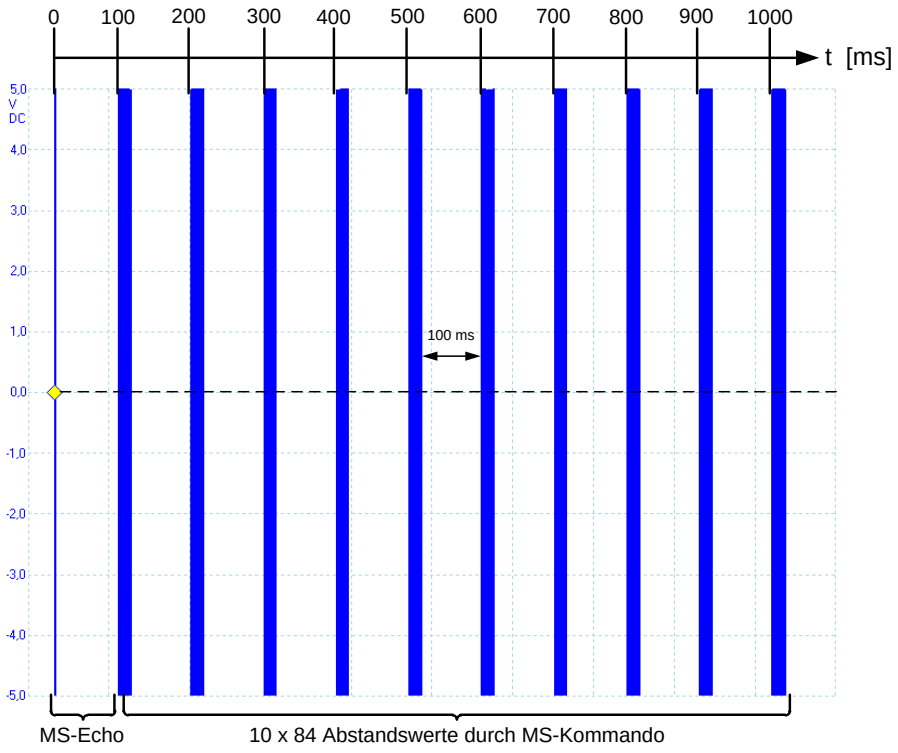


Abb. 9.1.: Tx-Kanal der RS232 Schnittstelle des Laserscanners zur Messung der 10 Hz Frequenz bei einem MS-Kommando.

Beim MS-Kommando werden die 84 Abstandswerte jeweils in einem Block mit einer konstanten Periode von 100 ms übertragen (vgl. Abb. 9.1). Die Signalleitungen der RS232 Schnittstelle des Laserscanners werden im Spannungsbereich von -6 Volt bis 6 Volt betrieben. Da das „Xtreme Spartan-3A DSP Board“ mit Pegeln von 3,3 - 5 Volt arbeitet, werden die RS232 Signale beim Anschluss an die Pins des „Extension Boards“ durch einen Pegelwandler auf TTL Pegel gewandelt.

9.1. Serielle Übertragungszeit der SCIP 2.0 Kommandos

MicroBlaze_0 ruft zur Messdatenerfassung die Funktion *scip_laser_get_data_90degree()* auf, die zu Beginn den Laserscanner durch das BM-Kommando anschaltet (vgl. Tab. 8.1). Nach 75 ms hat MicroBlaze_0 die Bestätigung des BM-Kommandos empfangen und sendet zur Anforderung der 84 Abstandswerte das GS-Kommando (vgl. Abb. 9.2). Die Zeit bis MicroBlaze_0 das GS-Kommando sendet, ist nicht konstant, da die Bestätigung des BM-Kommandos abgewartet werden muss (vgl. Kap. 9.2).

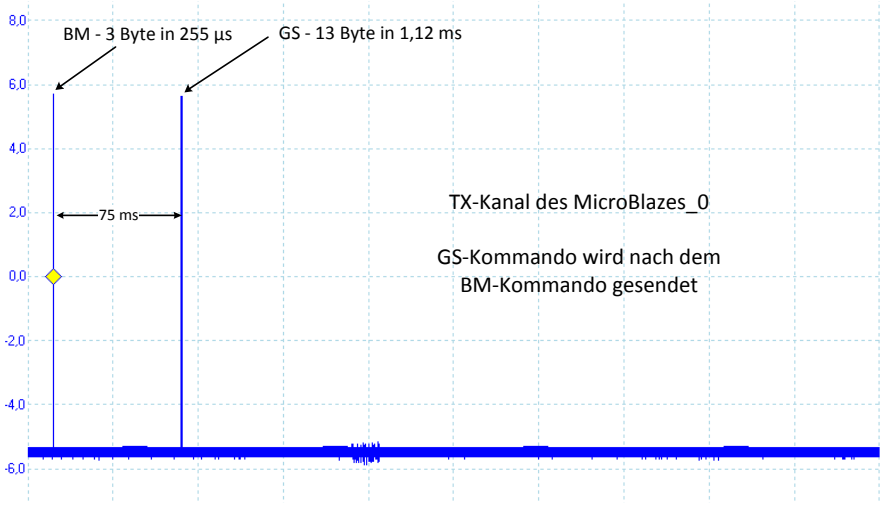


Abb. 9.2.: Tx-Kanal des MicroBlaze_0 beim Senden des BM-Kommandos zum Einschalten des Laserscanners und des GS-Kommandos zur Messdaten Anforderung (vgl. Kap. 6.1)

Empfängt der Laserscanner ein GS-Kommando, dann werden die 84 kodierten Abstandswerte nacheinander übertragen. Aus den 198 Byte zu übertragenden Daten und der Baudrate von 115.200 Bits pro Sekunde folgt eine Übertragungszeit von 13,75 ms (vgl. Kap. 6.8). Jedoch liegt die tatsächlich gemessene Zeit bei 17,09 ms (vgl. Abb. 9.3).

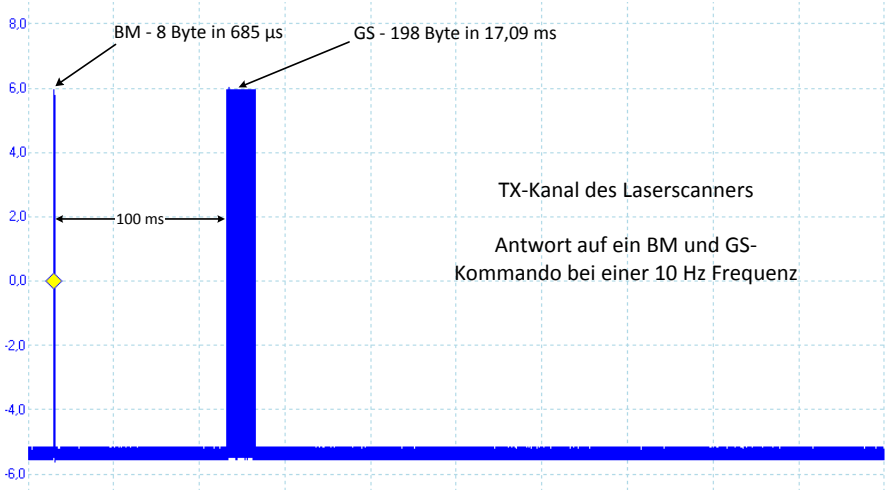


Abb. 9.3.: Tx-Kanal des Laserscanners beim Senden der Antwort auf ein BM und GS-Kommando. Aufgrund der 10 Hz Frequenz beträgt die Zeit zwischen den Bestätigungen von zwei aufeinanderfolgenden Kommandos 100 ms.

9.2. Bearbeitungsintervalle der Objekterkennung

Die Hauptfunktion zur Erfassung der Abstandswerte und der darauf folgenden Segmentierung ist *scip_laser_get_data_90degree* (vgl. Tab. 8.1). Mit dieser Funktion wird die Objekterkennung durch den Thread *thread_laser_measurement* gestartet und die vier Sub-Funktionen sequentiell aufgerufen (vgl. Abb. 9.4). Die Messung der Bearbeitungsintervalle gibt einen Aufschluss über die maximale Laserscanner Frequenz, die für eine optimale Objekterkennung genutzt werden kann.

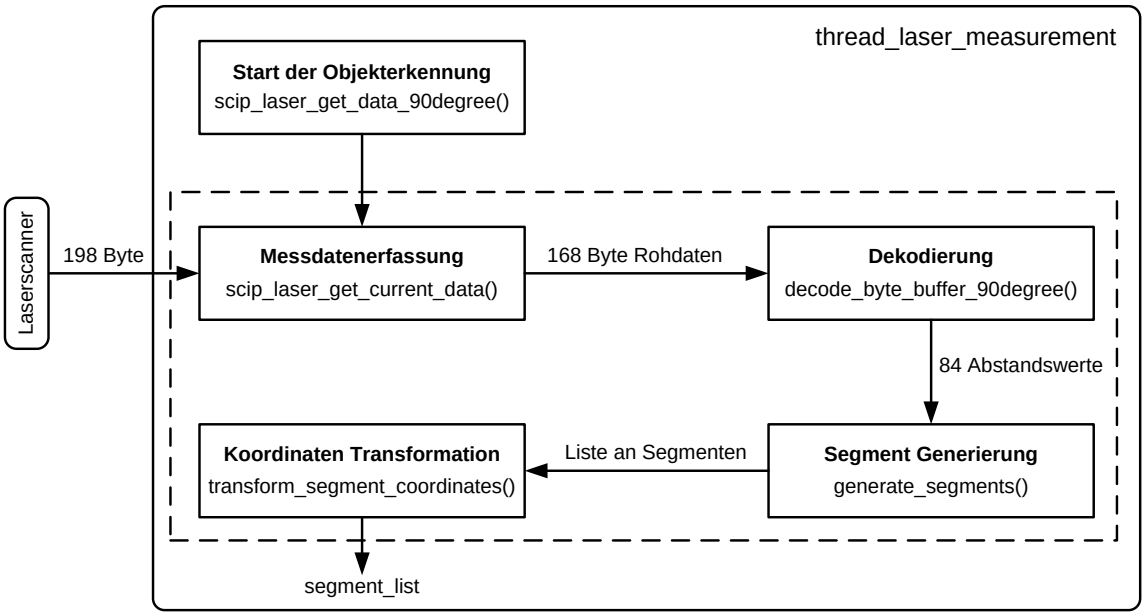


Abb. 9.4.: Aufbau der Funktion *scip_laser_get_data_90degree* zur Erfassung von Abstandswerten und zur Segmentierung. Gekapselt im Thread *laser_measurement* (vgl. Kap. 8)

Zur Messung der Bearbeitungsintervalle bietet das Xilkernel RTOS die Funktion *xget_clock_ticks()* [81]. Bei Aufruf dieser Funktion wird die aktuelle Anzahl an „Xilkernel Ticks“ seit dem Start des RTOS in einer Integer Variable gespeichert. Über den Parameter *SYSTMR_INTERVAL* wurde die Größe der „Time-Slices“ für das Scheduling auf 1 ms festgelegt, wobei ein „Xilkernel Tick“ einem „Time-Slice“ entspricht (vgl. Kap. 8). Aufgrund der 1 ms großen „Time-Slices“ hat *xget_clock_ticks()* eine Auflösung von einer Millisekunde. Da dies für die meisten Funktionen ein zu großes Zeitfenster darstellt, wurden die Bearbeitungsintervalle durch Oszillosgraphierung von Pegelwechseln an externen Pins gemessen, wobei die Pegelwechsel über GPIO-Schreibfunktionen erzeugt wurden. Die daraus resultierende Differenzzeit zwischen zwei Flankenwechel ergibt ein höher aufgelösted Bearbeitungsintervall als *xget_clock_ticks()*.

Um ein möglichst genaues Abbild der realen Bearbeitungsintervalle zu bekommen, wurden die Messungen stets in der gleichen Hardware / Software Konfiguration bei einer Systemfrequenz von 62,5 MHz vorgenommen. Jede Messung wurde in das in Kap. 8 vorgestellte Thread-Modell integriert, sodass durch das prioritätenbasierte Scheduling auf MicroBlaze_0 keine Latenz durch einen Threadwechsel entsteht (vgl. Kap. 8.2). Die Messergebnisse zeigen ein durchschnittliches Bearbeitungsintervall, da alle Funktionen bis auf die Dekodierung keine konstante Zeit aufweisen (vgl. Tab. 9.1). Dies ist vor allem

auf die nicht vorhersehbare Anzahl an Segmenten und auf die Verarbeitungsgeschwindigkeit des Laserscanners zurückzuführen. Damit die Ergebnisse für die Funktionen *generate_segments()* und *transform_segment_coordinates* als nahezu konstant angenommen werden können, wurde der Laserscanner so positioniert, dass stets nur ein Segment in Sichtweite erkannt wird. Die Messung für die Funktion *scip_laser_get_current_data()* liefert stark schwankende Zeiten (vgl. Tab. 9.1). Zur Ermittlung des durchschnittlichen Bearbeitungsintervalls wurde jede Messung 20 mal ausgeführt.

Funktion	Bearbeitungsintervall
scip_laser_get_current_data()	42 ms - 110 ms
decode_byte_buffer_90degree()	1,3 ms
generate_segments()	30,88 ms
transform_segment_coordinates	2,55 ms
scip_laser_get_data_90degree	75 ms - 166 ms

Tab. 9.1.: Durchschnittliche Bearbeitungsintervalle für die vier Funktionen zur Messdatenerfassung und zur Segmentierung. Die Funktion *scip_laser_get_data_90degree* kapselt alle vier Sub-Funktionen.

In einer Messreihe von 20 Messungen wurde ein minimales Bearbeitungsintervall von 40,12 ms und ein maximales von 109,40 ms für die Funktion *scip_laser_get_current_data()* gemessen. Für eine detaillierte Analyse dieser stark schwankenden Zeiten, wurden die einzelnen Funktionsblöcke der Methode *scip_laser_get_current_data()* einzelnd vermessen. Die Ergebnisse zeigen, dass für den Empfang des 13 Byte großen Headers der längste Zeitabschnitt benötigt wird (vgl. Tab. 9.2).

scip_laser_get_current_data()	
Beschreibung	Bearbeitungsintervall
Initialisierung und Zusammenstellen des Send Pakets	129 μs
GS-Kommando senden mit *XUartLite_Send()*	36 μs
13 Byte Header empfangen mit *scip_recv_until_one_lf()*	26 ms - 96 ms
4 Byte Status empfangen mit *scip_recv_until_one_lf()*	350 μs
6 Byte Zeitstempel empfangen mit *scip_recv_until_one_lf()*	520 μs
168 Byte Rohdaten empfangen mit *scip_recv_until_two_lf()*	15,1 ms
scip_laser_get_current_data()	42 ms - 110 ms

Tab. 9.2.: Bearbeitungsintervalle für die einzelnen Funktionsblöcke der Funktion *scip_laser_get_current_data()*. Die Zeit bis zum Empfang des 13 Byte großen Headers ist aufgrund der Laserscanner Verarbeitungsgeschwindigkeit stark schwankend.

Die Funktion *scip_laser_get_data_90degree* benötigt zur Erfassung, zur Dekodierung und zur Generierung von Segmenten zwischen 75 ms und 166 ms (vgl. Tab. 9.1. Nach diesem Bearbeitungsintervall liegen die erkannten Segmente in einer globalen Liste bereit. Den Hauptanteil des Bearbeitungsintervalls wird durch die Funktion *scip_laser_get_current_data()* bestimmt, bei der sowohl das GS-Kommando an den Laserscanner gesendet wird als auch die 198 Byte Daten empfangen. Wie in Tab. 9.2 erkennbar, wird für den Empfang des 13 Byte großen Headers die meiste Zeit benötigt. Die gemessene Zeit zwischen dem Senden des Kommandos und dem Empfang des ersten Header-Bytes beträgt zwischen 26 ms und 96 ms, wobei die schwankende Zeit auf die Verarbeitungsgeschwindigkeit des Laserscanners zurückzuführen ist. Tests haben gezeigt, dass der Laserscanner nach dem Senden eines Kommandos nicht umgehend die Antwort sendet. Durch die parallele Messung an Sende- und Empfangskanal wurde die Zeit zwischen dem gesendeten Kommando und der Laserscanner Antwort gemessen, wobei die Differenzzeit bis zu 80 ms beträgt (vgl. Abb. 9.5).

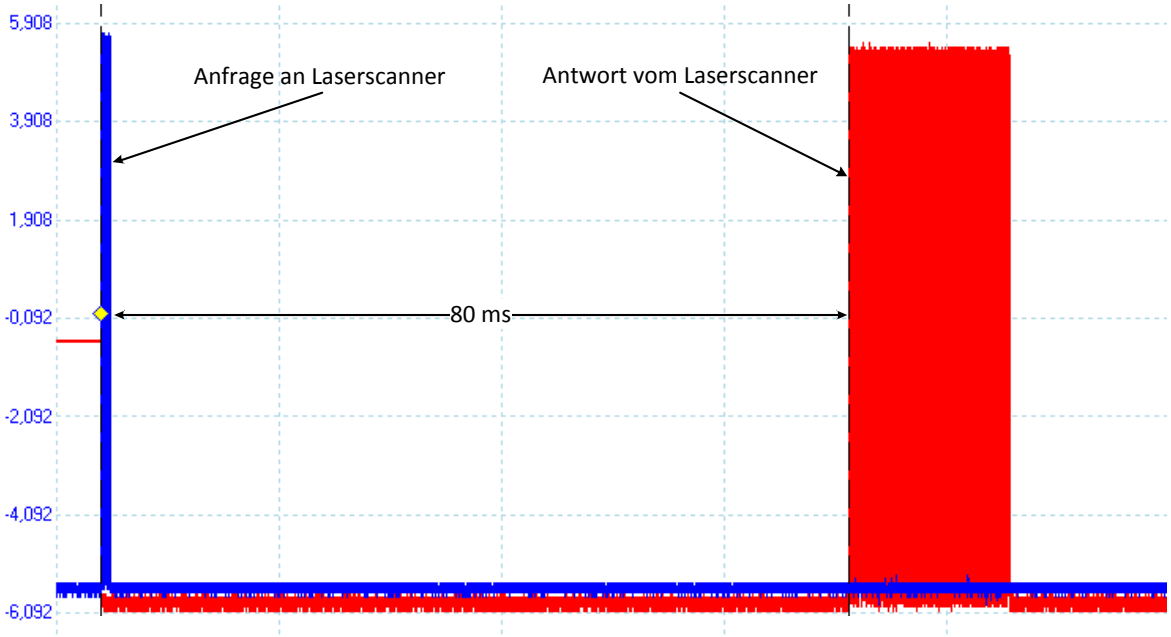


Abb. 9.5.: Parallele Messung des Sende- und Empfangskanals zur Bestimmung der Differenzzeit zwischen Anfrage und Antwort vom Laserscanner

Anhand den Bearbeitungsintervallen zum Empfang des Status Feldes und des Zeitstempels mit der Funktion *scip_recv_until_one_lf()* wurde gezeigt, dass das durchschnittliche Intervall zum Empfang eines Bytes ca. 86 μs beträgt. In dieser Zeit wird das empfangene Byte aus der UART FIFO gelesen und die Gültigkeit überprüft. Da zum Empfang die Makrofunktion *XUartLite_RecvByte()* verwendet wird, ist eine Reduzierung der Empfangszeit nicht möglich.

Durch die Verwendung des MS-Kommandos zum kontinuierlichen Empfang von Messdaten kann die Differenzzeit zwischen Anfrage und Antwort minimiert werden, da das Kommando nur einmalig gesendet wird. Sinkt die Zeit zum Empfang des ersten Header-Bytes, dann ergibt sich aus der Summe der einzelnen Intervalle ein Gesamtbearbeitungsintervall von ca. 75 ms. Jedoch steigt die Zeit zur Generierung und Transformation von Segmenten proportional zur Anzahl. Bei einer detailreichen Umgebung mit vielen Objekten wird

das Gesamtbearbeitungsintervall größer als 100 ms und übersteigt somit die Periode der Laserscannerfrequenz. Tests haben gezeigt, dass dies nach der ersten Erfassung zu einer falschen Auswertung der nachfolgenden Rohdaten führt, da die zu schnell ankommenden Messdaten nicht mehr geordnet empfangen werden und somit die Dekodierung zu falschen Abstandswerten führt. ´

9.2.1. Auswirkung von trigonometrischen Funktionen

In der Funktion *transform_segment_coordinates* werden zur Transformation der Polarkoordinaten in kartesische Koordinaten Sinus und Kosinus Funktionen eingesetzt, die in der Headerdatei *math.h* der Standard C Library deklariert sind (vgl. Kap. 7.1.1). Da die Approximation der Gleitkommazahlen mit der Integerarithmetik der MicroBlaze CPU einen hohen Rechenbedarf bedeutet, wurde zur Feststellung der Auswirkung die einzelnen Bearbeitungsintervalle gemessen (vgl. Tab. 9.3). Die Funktionen zur Berechnung der kartesischen x- und y-Koordinaten nutzen den *sin(radiant)* und den *cos(radiant)* aus der *math.h*. Des Weiteren wurden Funktionen zur rückwirkenden Transformation der kartesischen Koordinaten in Polarkoordinaten implementiert, die zur Berechnung des Polarkoordinatenwinkels die *atan2(y,x)* Funktion einsetzen (vgl. Kap. 7.1.1). Der Polarkoordinatenabstand berechnet sich aus dem Pythagoras.

Funktion aus *math.h*	Anzahl Messungen	Bearbeitungsintervall	
		ohne FPU	mit FPU
*10 * 10*	1	2,73 μs	2,8 μs
	10	4,77 μs	4,75 μs
*cos(10 * M_PI / 180)*	1	506 μs	400 μs
	10	4,95 ms	4,95 ms
*sin(10 * M_PI / 180)*	1	461 μs	400 μs
	10	4,06 ms	4,10 ms
*atan2(10,10) * M_PI / 180)*	1	711 μs	670 μs
	10	6,29 ms	6,42 ms
sqrt(pow(10,2) + pow(10,2))	1	1,6 μs	2,1 μs
	10	15,9 μs	17,2 μs

Tab. 9.3.: Bearbeitungsintervalle für die trigonometrischen Funktionen mit Umrechnung in Grad bei aktivierter und deaktivierter „Floating Point Unit“.

Die Ergebnisse in Tab. 9.3 zeigen, dass das Aktivieren der „Floating Point Unit“ im MicroBlaze keine bedeutende Erhöhung des Bearbeitungsintervalls bewirkt. Durch die Aktivierung der FPU werden zusätzliche Operationen freigeschaltet, die der Compiler bei der Übersetzung verwendet. Jedoch ist dies mit einem erhöhten Ressourcenbedarf verbunden. In der aktuellen Konfiguration ist zur Reduzierung der Hardware Ressourcen die „Floating Point Unit“ in beiden MicroBlazes deaktiviert.

9.3. Validierung des Schwellwertes zur Segmentierung

Bei der Laserscanner-basierten Objekterkennung wird für die Segmentierung ein Schwellwertverfahren eingesetzt. Die Genauigkeit der Abbildung von realen Objekten auf Segmente ist abhängig vom Schwellwert. Des weiteren hat die Wahl des Schwellwerts Einfluss auf die Anzahl der erkannten Segmenten. Ist der Schwellwert zu klein, wird ein Objekt in der Umwelt auf mehrere Segmente verteilt. Bei einem zu großen Schwellwert werden unterschiedliche Objekte in der Umwelt zu einem Segment zusammengefasst (vgl. Kap. 7.1). Um den optimalen Schwellwert zu bestimmen, wurden Testmessungen in zwei verschiedenen Szenarien durchgeführt.

1. Beim ersten Test wurden zwei Objekte mit einem Abstand von einem Meter direkt nebeneinander parallel zum Laserscanner positioniert (vgl. Abb. 9.6). Der Abstand d wurde bei verschiedenen Schwellwerten variiert, sodasss beide reale Objekte stets als zwei Segmente erkannt werden.
2. Bei der zweiten Messung wurde ein Objekt um seine eigene Achse rotiert, sodass die Laserstrahlen quer auf das Objekt treffen. Hierbei wurde der Rotationswinkel α bei 45° und bei 60° analysiert (vgl. Abb. 9.7). Das Objekt wurde nacheinander bei einem Abstand von 0,5 m und 1 m positioniert.

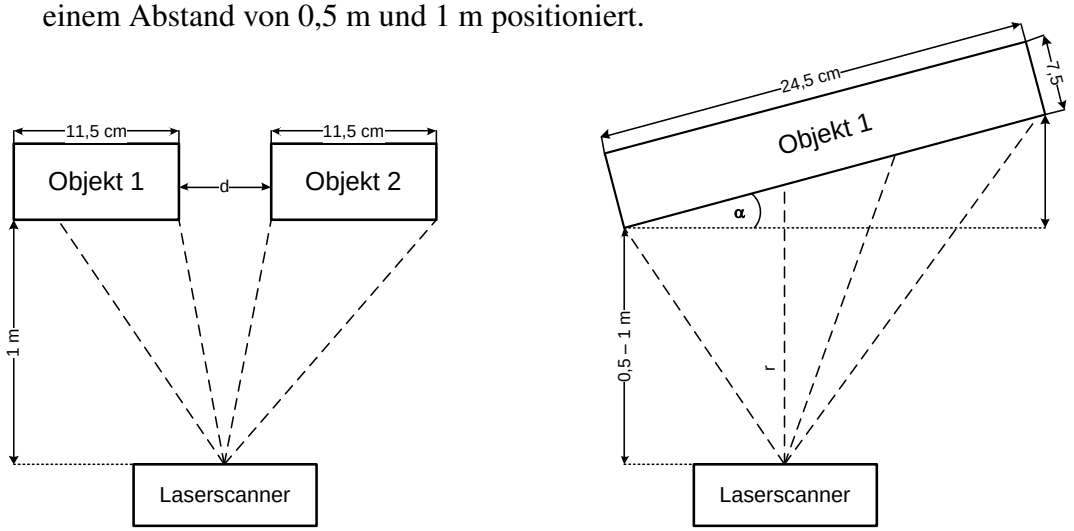


Abb. 9.6.: Validierung des Schwellwerts durch Testmessungen in zwei Szenarien.

Zum Vergleichen der einzelnen Messungen wurde stets das Array mit allen dekodierten Abstandswerten und die Liste der erkannten Segmente ausgegeben. Da die Anzahl an Abstandswerte für ein Segment Aufschluss über die Genauigkeit der Objekterkennung gibt, wurde diese mit der tatsächlichen Breite des Objektes verglichen und so eine Aussage über den Schwellwert getroffen.

Da das Objekt eine konstante Breite von 24 cm hat, wird die Anzahl an Abstandswerten durch den tangentialen Abstand bestimmt (vgl. Gleichung 6.3). Für die Distanz von 500 mm beträgt die Anzahl an Abstandswerten ca. 26, wobei durch eine Verdopplung der Distanz die Anzahl halbiert wird. Die Messungen ergaben, dass bei einem Schwellwert von 20 mm, die Objekte mit zu wenigen Abstandswerten beschrieben werden (vgl. Tab. 9.4). Dies ist darauf zurückzuführen, dass bei schräg zum Sensor positionierten Objekten

die Abstandswerte von benachbarten Laserstrahlen stark variieren und somit durch den kleinen Schwellwert neue Segmente generiert werden. Bei einem Schwellwert von 50 mm und 70 mm wird das Objekt nahezu mit der tatsächlichen Anzahl an Abstandswerten beschrieben (vgl. Tab. 9.4). Abweichungen treten auf, wenn ein Laserstrahl direkt auf die Kante eines Objektes trifft und somit das Licht abgelenkt und falsch reflektiert wird. Aus der Phasendifferenzmessung folgt demnach ein falsches Ergebnis (vgl. Kap. 2) und die Anzahl der Abstandswerte zur Beschreibung des Objektes sinkt.

Distanz	Winkel α	Anzahl an Abstandswerten		
		Schwellwert 20 mm	Schwellwert 50 mm	Schwellwert 70 mm
500 mm	45°	23	24	23
	60°	14	20	20
1.000 mm	45°	13	13	13
	60°	6	10	11

Tab. 9.4.: Validierung des Schwellwerts bei Objekten mit einem Rotationswinkel α. Das Objekt mit einer Breite von 24 cm wird bei einer Distanz von 500 mm positioniert.

Die Ergebnisse der Schwellwert Validierung zeigen, dass bei nebeneinander positionierten Objekte ein Schwellwert von unter 20 mm zu klein für die Erkennung beider Objekte ist, da zwei reale Objekte der Umwelt zu einem Segment vereinigt werden (vgl. Abb. 9.7). Für die Laserscanner-basierte Objekterkennung wurde ein Schwellwert von 50 mm und 70 mm getestet. Beide Konfigurationen führten zu einem akzeptablen Ergebnis für Objekte in einer Reiweite bis zu zwei Metern.

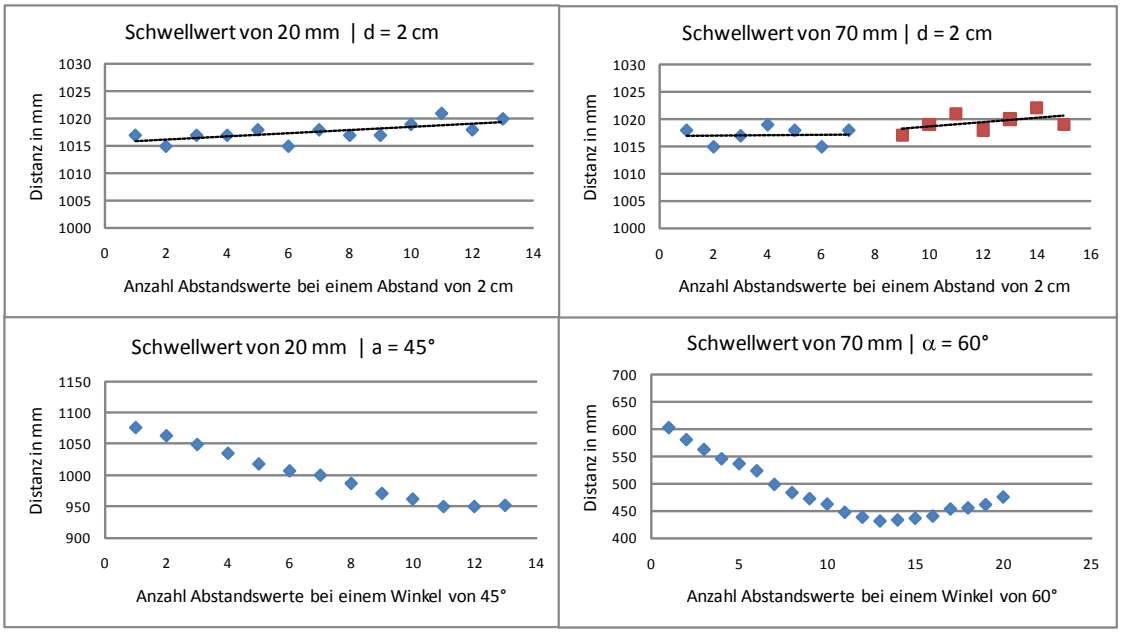


Abb. 9.7.: Auswertung der Schwellwert Validierung bei nebeneinander (oben) und schräg (unten) positionierten Objekten.

9.4. Ressourcenbedarf des Gesamtsystems

Für die Analyse und Auswertung der genutzten Ressourcen des „Spartan-3A DSP“ FPGAs wurden verschiedene Xilinx Tools verwendet. Der vom EDK erstellte „Design Summary“ gibt einen textuellen Überblick über die Ausnutzung der Ressourcen. Das genaue Mapping des Systems auf die physikalisch implementierten Ressourcen wird im MAP Report erläutert. Für eine detailliertere hierarchische Analyse wird das „Plan Ahead“ Tool und der „Xilinx XPower Analyzer“ eingesetzt. Die berechnete Ausnutzung der einzelnen Reports und Tools sind teilweise unterschiedlich. Dies liegt daran, dass in einer hierarchischen Komponenten Ansicht die verwendeten Slices und LUTs je nach Hierarchieebene doppelt gezählt werden. „Ressource Sharing“ wird hierbei nicht berücksichtigt. Die Ressourcenauslastung ist hauptsächlich von der implementierten MicroBlaze- und MPMC Konfiguration abhängig. In Tab. 5.7 wurde die Ressourcen Ausnutzung des MPMCs erläutert. Die nachfolgenden Tabellen zeigen die detaillierte Nutzung der „Spartan-3A DSP“ Ressourcen für einen einzelnen MicroBlaze und das Dual-MicroBlaze MPSoC. Hierbei ist zu beachten, dass in Tab. 9.5 nicht alle Komponenten aufgelistet sind, jedoch entspricht der Gesamtbedarf der tatsächlich Ausnutzung.

MB_0 Komponente	Slice FF	LUTs	BRAMs	DSP48As
Barrel Shifter	35	153	0	0
Integer Multiplier	17	0	0	3
Exception Register	32	33	0	0
Register File	0	384	0	0
MSR Register	19	55	0	0
Instruction Cache	125	100	5	0
Data Cache	113	290	5	0
Prefetch Buffer	50	139	0	0
FSL Interface	37	48	0	0
PLB Interface	104	8	0	0
Debug Logic	212	144	0	0
Gesamtbedarf	1.342	2.830	10	3
Verfügbar	33.280	33.280	84	84
Ausnutzung MB_0	4 %	9 %	12 %	4 %

Tab. 9.5.: Hierarchischer MAP Report für die MicroBlaze_0 Komponente auf einem Spartan-3A DSP FPGA (vgl. Kap. 3.1.1).

Die Ausnutzung in Tab. 9.5 zeigt, dass bei einem MicroBlaze der Bedarf an BRAMs alleinig von den Caches abhängig ist. Die aktuelle MPSoC Konfiguration hat sowohl einen 8 kByte großen Instruktionscache als auch einen 8 kByte großen Datencache. Da MicroBlaze_0 zwei FSL-Interfaces nutzt, ist der Ressourcenbedarf im Vergleich zu MicroBlaze_1 um 35 LUTs erhöht. In Tab. 9.6 ist der Gesamtbedarf an Ressourcen des MPSoCs für die aktuelle Konfiguration des SoC-Fahrzeugs gezeigt. Der *SAV IP* kapselt die Fahrspurführung und die Geschwindigkeitsregelung in einem Beschleuniger Modul, welches über den *SAV Connector* an den PLB angeschlossen wird. Für das gesamte MPSoC werden für die lokalen 16 kByte Speicher, die Caches und den MPMC insgesamt 80 BRAMs

genutzt. Durch eine Deaktivierung der Caches und eine MPMC FIFO Implementierung in LUTs können die BRAMs minimiert werden. Dies hätte zur Folge, dass aufgrund der schlechteren Speicherzugriffszeiten der Durchsatz sinkt. Die Ausnutzung der einzelnen MPSoC IPs gliedert sich wie folgt:

MPSoC Komponente	Slice FF	LUTs	BRAMs	DSP48As
MicroBlaze 0	1.342	2.830	10	3
MicroBlaze 1	1.344	2.795	10	3
MPMC DDR2 SDRAM	3.515	2.221	17	0
Boot BRAM MB0	0	0	16	0
Boot BRAM MB1	0	0	16	0
XPS Mutex	99	91	0	0
Debug Module	127	148	0	0
PLB MB0	89	309	0	0
PLB MB1	83	248	0	0
4 x FSL	146	179	0	0
4 x LMB	4 x 1	4 x 1	0	0
2 x ILMB Controller	2 x 2	0	0	0
2 x DLMB Controller	2 x 2	2 x 5	0	0
2 x XPS Timer	2 x 298	2 x 349	0	0
Interrupt Controller	105	78	0	0
LED 8 Bit	106	65	0	0
Push Buttons	82	52	0	0
Vehicle LED	223	114	0	0
UART RS232	124	117	0	0
CAM UART	133	127	0	0
Frame Grabber	28	9	0	0
PMOD Bluetooth	129	89	0	0
PMOD Uart	369	493	0	0
Clock Generator	4	0	0	0
Reset Module	35	22	0	0
SAV Connector	246	187	0	0
SAV IP	4.184	9.282	10	20
Gesamtverbrauch	13.121	20.447	80	26
Verfügbar	33.280	33.280	84	84
Ausnutzung	39%	61%	95%	31%

Tab. 9.6.: Bedarf an Spartan-3A DSP Ressourcen für das Gesamtsystem.

Der längste Laufzeitpfad in einem digitalen System identifiziert den längsten in der Schaltung existierenden Logikpfad aller Übergangslogikstufen. Der längste Laufzeitpfad des Fahrspurführungs-SoC betrifft das Beschleuniger Modul *SAV IP*, wobei die minimale Taktperiode 37,987 ns und somit die maximale Frequenz 26,325 MHz beträgt.

9.5. Speicherbedarf der Software

Die Software von MicroBlaze_0 und MicroBlaze_1 wird per Linker Script im externen DDR2 SDRAM Speicher abgelegt, wobei zum Starten der Bootsektor *.vectors* im lokalen BRAM gespeichert wird. Die komplette Software für beide MicroBlazes wurde mit dem „Software Development Kit (SDK)“ entwickelt, wobei das implementierte Hardware-Design direkt aus dem EDK in das SDK exportiert wurde [90]. Der FPGA Bistream (*system.bit*) und das „Block Memory Map File (BMM)“ werden automatisch in eine „Xilinx Hardware Platform Specification“ importiert, worauf beide MicroBlazes zugreifen.

Die Software auf MicroBlaze_0 beinhaltet sowohl die Messdatenerfassung als auch die Segmentierung zur Objekterkennung (vgl. Abb. D). Durch die zusätzlichen Funktionen zur Initialisierung, zur Speicherverwaltung und zur FSL-Kommunikation ist der Speicherbedarf um 82 kByte höher als bei MicroBlaze_1 (vgl. Tab. 9.7). In der aktuellen Konfiguration sind die Heap- und Stack Speicher auf beiden MicroBlazes 8 kByte groß, wobei eine Erhöhung einen Anstieg des Speicherbedarfs in der *.bss* Sektion zur Folge hätte.

Sektion	MicroBlaze_0	MicroBlaze_1
.text	83.808 Byte	21.038 Byte
.data	1.684 Byte	468 Byte
.bss	49.200 Byte	30.898 Byte
Gesamt	134.692 Byte	52.404 Byte

Tab. 9.7.: Speicherbedarf der Software auf MicroBlaze_0 und MicroBlaze_1. Die Optimierung durch den Compiler ist deaktiviert.

Zur Reduzierung des Speicherbedarfs kann im GCC-Compiler der „Optimierungs Level“ konfiguriert werden. Die oben gezeigte Tabelle erläutert den Speicherbedarf für beide MicroBlazes wenn keine Optimierung *o0* aktiviert ist (vgl. Tab. 9.7). Der GCC-Compiler bietet vier Optimierungs Level, wobei mit aufsteigender Nummer der Grad der Optimierung steigt. Bei der ersten Optimierung *o1* werden alle lokalen Variablen auf dem Stack abgelegt und nicht in Registern gehalten. Das Optimierungs Level *oS* hat die Reduzierung der Codegröße zum Ziel, wobei die Performance nicht beachtet wird [24]. Tabelle 9.8 zeigt den unterschiedliche Speicherbedarf abhängig von der Optimierung durch den Compiler.

Sektion	Optimierungs Level				
	o0	o1	o2	o3	oS
.text	83.808	71.888	71.612	80.108	70.864
.data	1.684	1.580	1.580	1.580	1.580
.bss	49.200	45.096	45.100	45.100	45.096
Gesamt	134.692	118.564	118.292	126.788	117.540

Tab. 9.8.: Vergleich des Speicherbedarf der MicroBlaze_0 Software bei verschiedenen Compiler Optimierungen. Die Angabe erfolgt in Byte.

10. Zusammenfassung & Ausblick

Bei dem vorgestellten speichergekoppelten Dual-MicroBlaze System greifen beide Prozessoren über den MPMC auf den gemeinsamen externen DDR2 Speicher zu. Damit keine Überlappung der Speicherbereiche stattfindet, wurden die Adressbereiche für die Software Sektionierung der einzelnen MicroBlazes mit den Linker Skripten aufgeteilt. Der Zugriff auf einen gemeinsam verwendeten Speicherbereich wird für den wechselseitigen Ausschluss mit einem XPS Mutex synchronisiert und koordiniert. Die durchschnittliche Zugriffszeit bei einem Sperr- oder Entsperrvorgang liegt bei 5 Mikrosekunden. Vor allem bei der Entwicklung von Echtzeitsystemen muss die Nutzung des gemeinsamen Speichers gut koordiniert werden, da eine gegenseitige Blockierung der Prozessoren beim Sperren des Mutexes zu einer erhöhten Wartezeit führt. Aus diesem Grund wird für die Kommunikation zwischen den Prozessoren eine FSl-basierte Interprozessorkommunikation eingesetzt. Die MicroBlazes werden direkt mit dem unidirektionalen FSL Bus miteinander verbunden und die zu übertragenden Daten werden ähnlich wie bei der XPS Mailbox durch eine interne FSL FIFO gepuffert. Der Vorteil durch den Einsatz des FSL Busses ergibt sich aus der Kopplung der Registerfiles der Prozessoren und dem damit verbesserten Durchsatz.

Jeder MicroBlaze hat für den schnellen Datenzugriff einen lokalen On-Chip BRAM Speicher. In diesem wird der Bootsektor *.vectors* der jeweiligen Software gespeichert und bei Systemstart direkt gestartet. Die ausführbaren Programmcodes der Prozessoren werden im gemeinsamen DDR2 SDRAM Speicher abgelegt. Dies hat den Nachteil, dass durch den sequentiellen Zugriff des MPMCs auf den externen Speicher eine Verzögerung bei der Ausführung der Software auftritt. Eine Speicherung der Software im BRAM hat eine Begrenzung der Codegröße zur Folge, da die maximale Größe eines BRAM Blocks 512 Kilobyte beträgt.

Zur Erhöhung des Datendurchsatzes bei einer Shared-Memory Multiprozessorarchitektur wird für jeden Prozessor ein 8 Kilobyte großer Instruktions- und Datencache instantiiert. Diese minimieren den Speicherflaschenhals, der beim Zugriff auf den externen Speicher entsteht. Über den XCL Bus werden die MicroBlaze Caches mit dem MPMC gekoppelt und bieten im Vergleich zum PLB Bus einen schnelleren Zugriff auf den Speicher. Der Datencache verwendet die „Write-Through“ Strategie, um Daten in den externen Speicher zurückzuschreiben. Durch die Trennung der Speicherbereiche für jeden Prozessor ist die Kohärenz der Datencaches gewährleistet. Eine im Linker Script neu definierte „Shared Memory Section“ ist für den Austausch von gemeinsamen Daten zuständig.

Damit das SoC-Fahrzeug autonom und kollisionsfrei einer Fahrspur folgen kann, muss ein Modell der Umgebung mit verschiedenen Sensoren aufgenommen und analysiert werden. Für die Lokalisierung und die Wiedererkennung von Hindernissen wird der Laserscanner URG-04LX eingesetzt, der einen 90° Scanbereich mit einer Winkelauflösung von 1,05° zyklisch scannt. Die 84 über die RS232-Schnittstelle übertragenen Abstandswerte werden

durch das Software-Interface auf MicroBlaze_0 empfangen und dekodiert. Die messtechnische Analyse des Laserscanners ergab, dass die Entfernungsmessung erst nach einer „Warm-Up Time“ von ca. 30-40 Minuten den stationären Zustand erreicht. Dies hat zur Folge, dass die Genauigkeit der Abstandswerte erst nach der „Warm-Up Time“ im angegebenen Toleranzbereich von $\pm 1\%$ liegt. Das SCIP 2.0 Kommunikationsprotokoll definiert Kommandos zur Steuerung des Laserscanners und wurde als Software Bibliothek auf MicroBlaze_0 implementiert.

Die dekodierten Abstandswerte werden zum Clustering einer nachgelagerte Segmentierung bereitgestellt, die einen Schwellwert von 70 mm zur Klassifizierung nutzt. Jedes Segment repräsentiert ein Objekt der Umgebung und wird durch drei Segmentpunkte sowohl in Polarkoordinaten als auch in kartesichen Koordinaten beschrieben. Der Segmentierungsalgorithmus iteriert über die Punktwolke der Abstandswerte und sucht nach zusammenhängenden Messwerten, die abhängig vom Schwellwert entweder zum gleichen Segment oder zu einem neuen Segment gehören. Jedes Segment wird durch den linken und rechten Begrezungspunkt sowie dem dichtesten Punkt beschrieben, wobei für jeden Punkt sowohl die Distanz als auch der korrespondierende Winkel gespeichert wird. Da für das SoC-Fahrzeug nur Objekte in einer Reichweite von zwei Metern von Interesse sind, werden die Abstandswerte vor der Segmentierung begrenzt. Dies hat den Vorteil, dass die Anzahl der generierten Segmente reduziert wird und das Messrauschen in den Abstandswerten nicht geglättet werden muss, da dies ausschließlich bei weiteren Entfernungen auftritt. Nach der erfolgreichen Segmentierung werden die Polarkoordinaten der in einer verketteten Liste gespeicherten Segmente in kartesische Koordinaten transformiert.

Eine Laufzeitanalyse ergab, dass die Verwendung von trigonometrische Funktionen aus der Standard-C Bibliothek *math.h* aufgrund der Approximaion des Ergebnisses den Hauptanteil des Ausführungsintervalls bestimmt. Des Weiteren wurde durch die Laufzeitanalyse festgestellt, dass die Laufzeit der Funktionen zur Messdatenerfassung und Segmentierung größer der Laserscanner Taktperiode von 100 ms ist. Dies ist vor allem auf die nicht konstante Zeit zwischen Anfrage an den Laserscanner und Antwort sowie auf die nicht konstante Anzahl an Segmente zurückzuführen. Aus diesem Grund wird für den kontinuierlichen Empfang von Abstandswerten eine Abtastperiode von ca. 150-200 ms genutzt, was bedeutet, dass das SoC-Fahrzeug in dieser Zeit bei einer Geschwindigkeit von 2 m/s eine Strecke von 30-40 cm zurücklegt.

Der zweite MicroBlaze_1 ist für die Steuerung der Laserscanner-basierten Objekterkennung zuständig, indem er die über die Push Buttons empfangenen Kommandos über die FSL Verbindung an MicroBlaze_0 sendet. Der für die Segmentierung zuständige MicroBlaze_0 wartet auf die Initiierung der Objekterkennung durch MicroBlaze_1 und legt die angeforderten Segmente im gemeinsamen Speicher ab. Nach dem Entsperren des Mutexes sendet MicroBlaze_0 die Bestätigung, dass neue Daten im Speicher liegen, über den FSL an MicroBlaze_1. Dieser sperrt den Mutex und sendet die ausgelesenen Daten über den JTAG Uart an das Terminal des Anwenders.

Auf beiden MicroBlazes läuft ein preemptives Xilkernel RTOS, welches die Verwendung des POSIX Interfaces ermöglicht. Auf MicroBlaze_0 werden zwei Threads nebenläufig ausgeführt, wobei der Master Thread den Empfang und die Verarbeitung der Kommandos durch MicroBlaze_1 koordiniert. Der zweite Thread auf MicroBlaze_0 ist für die Erfassung der Abstandswerte und derren Segmentierung zuständig und ist höher priorisiert als

der Master Thread. Durch ein prioritätenbasiertes Scheduling auf MicroBlaze_0 hat der Thread zur Objekterkennung stets den Vorrang vor dem Master Thread und kann während einer Segmentierung nicht unterbrochen werden. Die ankommenden Kommandos von MicroBlaze_1 werden von der FSL FIFO zwischengespeichert und bei Zuteilung der CPU an den Master Thread, wertet dieser die Kommandos aus. Auf MicroBlaze_1 werden ebenfalls zwei Threads zum Empfangen und Senden von FSL Kommandos genutzt, sowie ein Master Thread, der die Sub-Thread erzeugt. Das Round-Robin Scheduling auf MicroBlaze_1 behandelt alle Threads gleichberechtigt und verteilt die CPU nach einem fest konfiguierten „Time Slice“, welcher auf beiden MicroBlazes eine Millisekunde beträgt. Für das Timing des Xilkernels wurden die MicroBlazes mit einem XPS Timer gekoppelt, welcher bei einer Systemfrequenz von 62,5 MHz jede Millisekunde einen Interrupt generiert.

10.1. Ausblick

In dieser Arbeit wurde die Laserscanner-basierte Objekterkennung zur ersten Funktionsanalyse als Software-Interface implementiert. Es wurde gezeigt, dass die Segmenklassifizierung durch ein Schwellwertverfahren ausreichend genaue Ergebnisse für den Einsatz im SoC-Fahrzeug liefert. Jedoch ergab die messtechnische Analyse, dass die Ausführungsintervalle der Funktionen höher als die Taktperiode des Laserscanners sind und somit ein kontinuierlicher Empfang von Abstandswerten innerhalb der Laserscannerfrequenz nicht realisierbar ist. Die Auslagerung der Messdatenerfassung in ein Hardware-Beschleunigermodul ist ein erster Ansatz für nachfolgende Arbeiten. Des Weiteren kann das MPSoC mit einem weiteren MicroBlaze erweitert werden, sodass die Erfassung und die Segmentierung auf zwei unterschiedlichen Prozessoren parallel ausgeführt wird. Da die Laserscanner-basierte Objekterkennung im SoC-Fahrzeug für ein automatisches Ausweichmannöver eingesetzt werden soll, muss eine Anpassung der aktuellen Fahrspurerkennung erfolgen. Die folgenden Erweiterungen sind am MPSoC und an der Objekterkennung vorzunehmen, um eine Integration zu gewährleisten:

- Damit die Objekte der Fahrspur zugeordnet werden können, müssen sich die Koordinaten des Kamera ROIs und der Segmente in einem einheitlichen Koordinatensystem befinden. Somit kann der Controller MicroBlaze_1 bestimmen, ob sich die bereitgestellten Segmente auf der Fahrspur befinden und ein Ausweichmannöver eingeleitet werden muss. Die aktuelle Konfiguration der Fahrspurerkennung nutzt einen ROI, der die rechte Fahrspurmarkierung erkennt. Eine Verdopplung des ROIs würde bei einem ausreichendem Blickwinkel der Kamera beide Fahrspurmarkierungen erkennen und somit kann sich das SoC-Fahrzeug bei einem Ausweichmannöver an der gegenüberliegenden Markierung orientieren.
- In der aktuellen Konfiguration erkennt der Laserscanner Objekte in einem frontalen 90° Scanbereich. Für das Ausweichmannöver muss das Ende eines Hindernisses durch die seitlich angebrachten Infrarotsensoren erkannt werden. Hierfür ist sowohl eine Integration der Sensoren in das MPSoC als auch die Implementierung einer Steuerungssoftware auf MicroBlaze_1 notwendig.

Literaturverzeichnis

[1] AMDAHL, Gene M.: Validity of the single processor approach to achieving large scale computing capabilities. In: *AFIPS spring joint computer conference* (1967), Nr. AFIPS Conferences 30

[2] AMELING, C.: *Steigerung der aktiven Sicherheit von Kraftfahrzeugen durch ein Kollisionsvermeidungssystem*. VDI Verlag, 2002. – ISBN 978-3-183-51012-2

[3] A.YAMAWAKI ; M.IWANE: An FPGA Implementation of a Snoop Cache with Synchronization for a Multiprocessor System-On-Chip. In: *IEEE Parallel and Distributed Systems* (2007), Nr. 978-1-4244-1889-3

[4] BENRA, J. ; HALANG, W.A.: *Software-Entwicklung für Echtzeitsysteme*. Springer, 2009. – ISBN 978-3-642-01595-3

[5] BLACKMAN, S.S. ; POPOLI, R.: *Design and Analysis of Modern Tracking Systems*. Artech House, 1999 (Artech House radar library). – ISBN 978-1-580-53006-4

[6] BORDASCH, H.: *VHDL-Modellierung einer Geschwindigkeitsregelung für ein autonomes Fahrzeug implementiert auf einer SoC-Plattform*, HAW Hamburg, Bachelorarbeit, 2009

[7] BORGES, G.A. ; ALDON, M.: Line Extraction in 2D Range Images for Mobile Robotics. In: *Journal of Intelligent & Robotic Systems* 40 (2004), S. 267–297

[8] BRESHEARS, Clay: *The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications*. O'Reilly Media, 2009. – ISBN 978-0-5965-2153-0

[9] BROOKS, R.: A robust layered control system for a mobile robot. In: *IEEE Journal of Robotics and Automation* 2 (1986), Nr. 1, S. 14 – 23

[10] BURKHARDT, T ; FEINÄUGLE, A ; FERICEAN, S ; FORKL, A: *Lineare Weg- und Abstandssensoren: Berührungslose Messsysteme für den industriellen Einsatz*. Süddeutscher Verlag, 2004. – ISBN 978-3-93788-907-8

[11] CAROLO CUP: *Studenten entwickeln autonome Modellfahrzeuge*. 2011. – URL http://www.carolo-cup.de/. – abgerufen 20.11.2011

[12] CORDES, S.: *Automatischer Bremsassistent auf Basis einer Laserscanner-Abstandserfassung für ein fahrerloses Transportsystem*, HAW Hamburg, Masterarbeit, 2006

[13] DAIMLER CHRYSLER: Systembeschreibung ETC Deutschland. In: *DCSMM/ GE* (2003). – URL http://www.dcl.hpi.uni-potsdam.de/teaching/mobilitySem03/slides/tollcollect.pdf. – aufgerufen am 18.11.2011

[14] DIN 70000: *Straßenfahrzeuge; Fahrzeugdynamik und Fahrverhalten; Begriffe (ISO 8855:1991, modifiziert)*. 1994

[15] DIN VDE 0848 - TEIL 2: *Sicherheit in elektromagnetischen Feldern - Schutz von Personen im Frequenzbereich 30 kHz bis 300 GHz*. 1991

[16] EICHLER, J. ; EICHLER, H.J.: *Laser: Bauformen, Strahlführung, Anwendungen*. Springer, 2010. – ISBN 978-3-64-210461-9

[17] FRAUNHOFER ILT: *Presseinformation - Femtosekundenlaser*. 2010. – URL http://www.ultrakurzpulslaser.de/pdf/PM_fs-Laser.pdf

[18] FU, Guoqiang ; CORRADI, P. ; MENCIASSI, A. ; DARIO, P.: An Integrated Triangulation Laser Scanner for Obstacle Detection of Miniature Mobile Robots in Indoor Environment. In: *Mechatronics, IEEE/ASME Transactions on* 16 (2011), Nr. 4, S. 778 –783

[19] FUERSTENBERG, K ; SCHULZ, R: Laserscanner for Driver Assistance. In: *3rd International Workshop on Intelligent Transportation, Hamburg*, Ibeo Automotive Systems GmbH, 2006

[20] G.BENGEL ; C.BAUN ; M.KUNZE ; K.U.STUCKY: *Masterkurs Parallele und Verteilte Systeme*. Vieweg+Teubner, 2008. – ISBN 978-3-8348-0394-8

[21] GERONIMO, D. ; LOPEZ, A.M. ; SAPPA, A.D. ; GRAF, T.: Survey of Pedestrian Detection for Advanced Driver Assistance Systems. In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 32 (2010), Nr. 7, S. 1239 –1258

[22] GOHRINGER, D. ; BIRK, M. ; DASSE-TIYO, Y. ; RUITER, N. ; HUBNER, M. ; BECKER, J.: Reconfigurable MPSoC versus GPU: Performance, power and energy evaluation. In: *Industrial Informatics (INDIN), 2011 9th IEEE International Conference on*, 2011, S. 848 –853

[23] GONZALEZ, R.C. ; WOODS, R.E.: *Digital Image Processing*. Pearson/Prentice Hall, 2008. – ISBN 978-0-131-68728-8

[24] GOUGH, B. ; STALLMAN, R.M.: *An introduction to GCC: for the GNU compilers gcc and g++*. Network Theory, 2004. – ISBN 978-0-954-16179-8

[25] GOULD, R. G.: The LASER, Light Amplification by Stimulated Emission of Radiation. In: *The Ann Arbor Conference on Optical Pumping, the University of Michigan,*, Franken, P.A. and Sands, R.H., 1959, S. 128

[26] GSCHWANDTNER, M. ; KWITT, R. ; UHL, A. ; PREE, W.: BlenSor: Blender Sensor Simulation Toolbox. In: *7th international conference on Advances in Visual Computing - Volume Part II*, Springer-Verlag, 2011 (ISVC'11), S. 199–208. – ISBN 978-3-642-24030-0

[27] GUIVANT, J. ; NEBOT, E. ; BAIKER, S.: Autonomous navigation and map building using laser range sensors in outdoor applications. In: *Journal of Robotic Systems* 17 (2000), S. 3817–3822

[28] HAUCK, S. ; DEHON, A.: *Reconfigurable Computing: the theory and practice of FPGA-based computation*. Morgan Kaufmann, 2008 (Systems on Silicon). – ISBN 978-0-123-70522-8

[29] HAW HAMBURG: *FAUST Fahrerassistenz- und Autonome Systeme*. 2012. – URL http://www.informatik.haw-hamburg.de/faust.html. – abgerufen 18.01.2012

[30] HE, Fujun ; DU, Zhijiang ; LIU, Xiaolei ; TA, Yueyue: Laser Range Finder based Moving Object Tracking and Avoidance in Dynamic Environment. In: *Information and Automation (ICIA), 2010 IEEE International Conference on*, 2010, S. 2357 – 2362

[31] HELLA KGAA HUECK & CO: Technical Information Electronics - Driver Assistance Systems. (2007). – URL http://www.hella.com/produktion/HellaUSA/WebSite/MiscContent/Download/AutoIndustry/Electronics/TI_ADAS_GB_TT_07.pdf. – aufgerufen am 16.11.2011

[32] HENNESSY, John L. ; PATTERSON, David A.: *Computer Architecture: A Quantitative Approach - 4th Edition*. Morgan Kaufmann, 2006. – ISBN 978-0-12-370490-0

[33] HOKUYO AUTOMATIC CO.LTD: *Scanning Laser Range Finder URG-04LX - Specification*. 2005

[34] HOKUYO AUTOMATIC CO.LTD: *Communication Protocol Specification For SCIP2.0 Standard*. 2006

[35] IBEO AUTOMOBILE SENSOR GMBH: *ALASCA User Manual*. 2006

[36] INSTRUMENTS, National: *NI Developer Zone*. 2008. – URL http://zone.ni.com/devzone/cda/tut/p/id/6461. – abgerufen 19.01.2012

[37] JERRAYA, Ahmed A. ; WOLF, Wayne: *Multiprocessor Systems-on-Chips*. Morgan Kaufmann, 2005. – ISBN 978-0-123-85251-9

[38] JÄHNE, B.: *Digitale Bildverarbeitung*. Springer, 2005. – ISBN 978-3-540-24999-3

[39] KNEIP, L. ; TACHE, F. ; CAPRARI, G. ; SIEGWART, R.: Characterization of the compact Hokuyo URG-04LX 2D laser range scanner. In: *Robotics and Automation. ICRA '09. IEEE International Conference*, 2009, S. 1447 –1454

[40] KNEUBÜHL, F.K. ; SIGRIST, M.W.: *Laser*. Vieweg + Teubner, 2008 (Teubner Studienbücher). – ISBN 978-3-83-510145-6

[41] KNIERIEMEN, T.: *Autonome mobile Roboter: Sensordateninterpretation und Weltmodellierung zur Navigation in unbekannter Umgebung*. BI Wissenschaftsverlag, 1991 (Reihe Informatik). – ISBN 978-3-411-15031-1

[42] KOWALCZYK, Jeremy: Multiprocessor Systems. In: *Xilinx White Paper* (2003), Nr. WP162 (v1.1)

[43] LRP ELECTRONIC GMBH: *Crawler Brushless 21.5 Turns und A.I. Brushless Pro Reverse Digital*. 2011. – URL http://www.lrp.cc/

[44] MAHR, T. ; GESSLER, R.: *Hardware-Software-Codesign: Entwicklung flexibler Mikroprozessor FPGA-Hochleistungssysteme*. Vieweg+Teubner Verlag, 2007. – ISBN 978-3-834-80048-0

[45] MASTORAKIS, N. ; MLADENOV, V. ; KONTARGYRI, V.T.: *Proceedings of the European Computing Conference*. Springer, 2009 (Lecture Notes in Electrical Engineering Band. 1). – ISBN 978-0-387-84813-6

[46] MELLERT, C.: *Modellierung einer Video-basierten Fahrspurerkennung mit dem Xilinx System Generator für eine SoC-Plattform*, HAW Hamburg, Bachelorarbeit, 2010

[47] MENK, Christoffer: Einführung in Echtzeitbetriebssysteme. In: *Echtzeit-Betriebssysteme und -Bussysteme*, Prochnow, S.H. and Hanxleden, R.v., 2007 (Seminar im Wintersemester 2006/07), S. 3–12

[48] MEYER, M.: *Signalverarbeitung: Analoge und digitale Signale, Systeme und Filter*. Vieweg+Teubner Verlag, 2008 (Informations- und Kommunikationstechnik). – ISBN 978-3-834-80494-5

[49] M.LOGHI ; M.PONCINO ; L.BENINI: Cache Coherence Tradeoffs in Shared-Memory MPSoCs. In: *ACM Transactions on Embedded Computing Systems* (2006), Nr. Volume 5 Nr. 2

[50] MOORE, Gordon E.: Cramming more components onto integrated circuits. In: *Electronics* (1965), Nr. Volume 38 Number 8

[51] MOTORNEWS: *Hella - Die automatische Abstandsregelung (ACC = Adaptive Cruise Control) erstmals in Serie*. 2007. – URL http://www.motornews.eu/cms/front_content.php?idcatart=4306. – abgerufen 17.11.2011

[52] PAPULA, L.: *Mathematik für Ingenieure und Naturwissenschaftler 1*. Vieweg+Teubner Verlag, 2009 (Viewegs Fachbücher der Technik Band. 1). – ISBN 978-3-834-80545-4

[53] PASCOAL, J. ; MARQUES, L. ; ALMEIDA, A.T. de: Assessment of Laser Range Finders in risky environments. In: *Intelligent Robots and Systems. IROS 2008. IEEE/RSJ International Conference*, 2008, S. 3533 –3538

[54] PRÖHL, A.: *Automatischer Ausweichassistent auf Basis einer Laserscanner-Abstandserfassung für ein fahrerloses Transportsystem*, HAW Hamburg, Masterarbeit, 2006

[55] PÜSKÜL, Ö. N.: *SoC-basierte Bildverarbeitungspipeline für eine Fahrspurerkennung und -visualisierung*, HAW Hamburg, Bachelorarbeit, 2011

[56] RANGANATHAN, Aanjhan: Experimental Analysis of Snoop Filters for MPSoC Embedded Systems / École Polytechnique Fédérale de Lausanne, Switzerland. 2010. – Master Project

[57] RAUBER, Thomas ; RÜNGER, Gudula: *Multicore: Parallele Programmierung*. Springer Verlag, 2008. – ISBN 978-3-540-73113-9

[58] REBAI, K. ; BENABDERRAHMANE, A. ; AZOUAOUI, O. ; OUADAH, N.: Moving Obstacles Detection and Tracking with Laser Range Finder. In: *Advanced Robotics, 2009. ICAR 2009. International Conference on*, 2009, S. 1 –6

[59] REIF, K: *Sensoren im Kraftfahrzeug*. Vieweg+Teubner Verlag, 2010. – ISBN 978-3-83481-315-2

[60] RULL, A.: *Sensorbasierte Umgebungskartierung mit lokaler Positionskorrektur für autonome Fahrzeuge*, HAW Hamburg, Bachelorarbeit, 2008

[61] SCHAUMONT, P.R.: *A Practical Introduction to Hardware/Software Codesign*. Springer, 2010. – ISBN 978-1-441-95999-7

[62] SCHETLER, D.: *Automatischer Ausweichassistent mit einer Laserscanner-basierten Abstandsregelung für ein fahrerloses Transportsystem*, HAW Hamburg, Masterarbeit, 2007

[63] SCHNEIDER, C.: *Ein System-on-Chip-basiertes Fahrspurführungssystem*, HAW Hamburg, Bachelorarbeit, 2011

[64] SCHRÖDER, J.: *Adaptive Verhaltensentscheidung und Bahnplanung für kognitive Automobile*. Univ.-Verl., 2009. – ISBN 978-3-866-44406-5

[65] SEUSS, Dipl.-Ing. J.: Sensoren für Fahrerassistenzsysteme - Von Komfort zu Sicherheitssystemen. In: *15. Aachener Kolloquium Fahrzeug- und Motorentechnik* (2006). – URL http://www.aachen-colloquium.com/pdf/Vortr_Nachger/2006/Seuss.pdf. – aufgerufen am 16.11.2011

[66] SHARP: *GP2D12 Optoelectronic Device - Data Sheet*. 2005

[67] SIADAT, A. ; KASKE, A. ; KLAUSMANN, S. ; DUFAUT, M. ; HUSSON, R.: *An Optimized Segmentation Method for a 2D Laser-Scanner Applied to Mobile Robot Navigation*. In Proceedings of the 3rd IFAC Symposium on Intelligent Components and Instruments for Control Applications. 1997

[68] SONY: *Color Camera Module - Technical Manual FCB-PV10*. 2006

[69] SPETH, Dr.-Ing. J.: *Videobasierte modellgestützte Objekterkennung für Fahrerassistenzsysteme*. Cuvillier Verlag Göttingen, 2010. – ISBN 978-3-86955-317-7

[70] SPIES, M. ; SPIES, H.: Automobile Lidar Sensorik: Stand, Trends und zukünftige Herausforderungen. In: *Advances in Radio Science* 4 (2006), S. 99–104

[71] STEINMÜLLER, J.: *Bildanalyse: Von der Bildverarbeitung zur räumlichen Interpretation von Bildern*. Springer, 2008. – ISBN 978-3-540-79742-5

[72] TANENBAUM, A.S.: *Moderne Betriebssysteme*. Pearson Studium, 2009. – ISBN 978-3-827-37342-7

[73] T.DORTA ; J.JIMENEZ ; J.L.MARTIN ; U.BIDARTE ; A.ASTARLOA: Overview of FPGA-Based Multiprocessor Systems. In: *2009 International Conference on Reconfigurable Computing and FPGAs* (2009), Nr. IEEE Conferences

[74] TIMOSCHENKO, A.: *Implementierung einer Geschwindigkeitsregelung als Prozessor-Element auf einer SoC-Plattform für ein autonomes Fahrzeug*, HAW Hamburg, Bachelorarbeit, 2011

[75] UNGERER, Theo ; BRINKSCHULTE, Uwe: *Mikrocontroller und Mikroprozessoren*. Springer-Verlag Gmbh, 2010. – ISBN 978-3-54-046801-1

[76] WEHR, A ; LOHR, U: Airborne laser scanning - an introduction and overview. In: *ISPRS Journal of Photogrammetry and Remote Sensing* 54 (1999), Nr. 2-3, S. 68–82

[77] WENDER, S.: *Multisensorsystem zur erweiterten Fahrzeugumfelderfassung*, Universität Ulm, Dissertation, 2008

[78] WILKEN, Heiko: Multiprocessor System-on-Chip - Dual MicroBlaze Implementierung auf einem Spartan-3A / HAW Hamburg. 2011. – Projekt-2 Ausarbeitung

[79] WINNER, H: Mit aktiven Sensoren das Kfz-Umfeld erfassen: Funktion und Leistungsfähigkeit von Radar & Co. In: *Information Technology* 49 (2007), Nr. 1, S. 17–24

[80] WINNER, H. ; HAKULI, S. ; WOLF, G.: *Handbuch Fahrerassistenzsysteme: Grundlagen, Komponenten und Systeme für aktive Sicherheit und Komfort*. Vieweg + Teubner, 2009 (ATZ-MTZ Fachbuch). – ISBN 978-3-83-480287-3

[81] XILINX: Xilkernel. In: *Data Sheet* (2006), Nr. v3.00a

[82] XILINX: Designing Multiprocessor Systems in Platform Studio. In: *White Paper: Xilinx Platform Studio* (2007), Nr. WP262(v2.0)

[83] XILINX: Local Memory Bus (LMB) v1.0. In: *Product Specification* (2009), Nr. v1.00a

[84] XILINX: Spartan-3A DSP Starter Platform. In: *User Guide* (2009), Nr. UG454 v1.1

[85] XILINX: XPS Mutex. In: *Product Specification* (2009), Nr. v1.00c

[86] XILINX: XPS UART Lite. In: *Product Specification* (2009), Nr. v1.01a

[87] XILINX: LogiCore IP Fast Simplex Link (FSL) V20 Bus. In: *Product Specification* (2010), Nr. DS449 v2.11c

[88] XILINX: OS and Libraries Document Collection. In: *Reference Guide* (2010), Nr. UG 643

[89] XILINX: Spartan-3A DSP FPGA Family Data Sheet. In: *Product Specification* (2010), Nr. DS610

[90] XILINX: EDK 13.1 Concepts, Tools, and Techniques. In: *A Hand-On Guide to Effective Embedded System Design* (2011), Nr. UG683

[91] XILINX: MicroBlaze Processor Reference Guide. In: *Embedded Development Kit EDK 13.1* (2011), Nr. UG081 (v12.0)

[92] XILINX: Multi-Port Memory Controller (MPMC). In: *Product Specification* (2011), Nr. DS643 v6.03a

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

A. Lasertriangulation

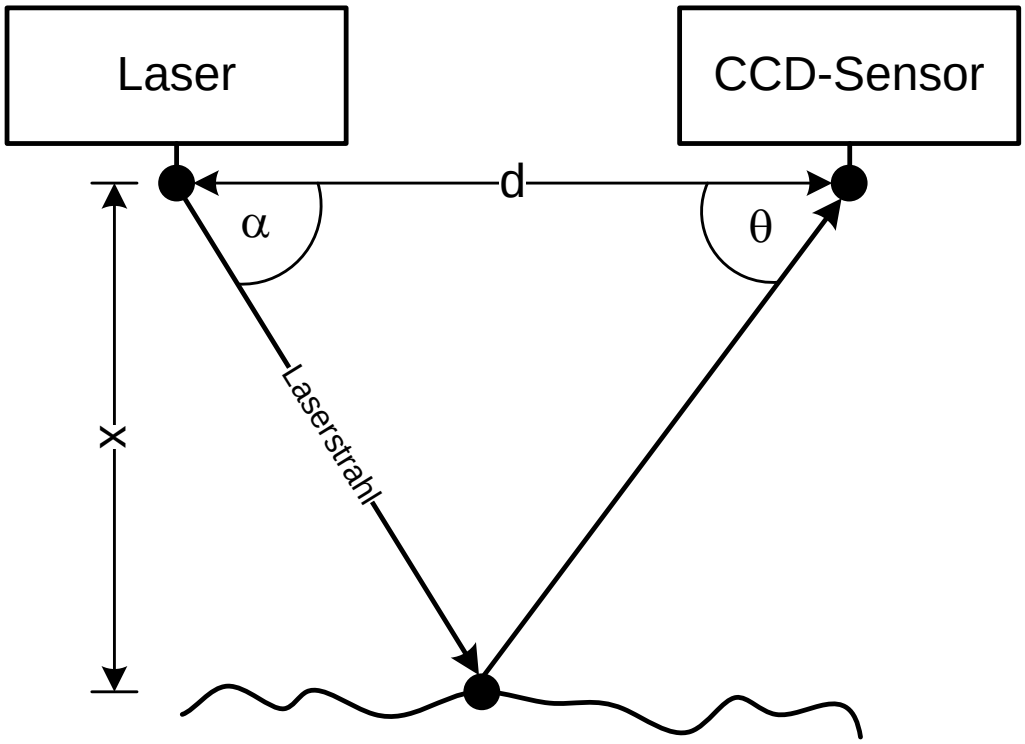


Abb. A.1.: Abstandsmessung mit Lasertriangulation. Der Abstand d zwischen Lasersensor und CCD-Sensor sowie die Winkel sind bekannt. Berechnung der Entfernung x erfolgt anhand trigonometrischen Funktionen [45]

$$x = \frac{d * \tan(\alpha)}{1 + \tan(\alpha)/\tan(\theta)} \tag{A.1}$$

B. MHS-File für MicroBlaze und MPMC

```
BEGIN microblaze_0
 PARAMETER INSTANCE = microblaze_0
 PARAMETER C_AREA_OPTIMIZED = 0
 PARAMETER C_USE_BARREL = 1
 PARAMETER C_DEBUG_ENABLED = 1
 PARAMETER C_ICACHE_BASEADDR = 0x88000000
 PARAMETER C_ICACHE_HIGHADDR = 0x89FFFFFF
 PARAMETER C_CACHE_BYTE_SIZE = 8192
 PARAMETER C_ICACHE_ALWAYS_USED = 1
 PARAMETER C_DCACHE_BASEADDR = 0x88000000
 PARAMETER C_DCACHE_HIGHADDR = 0x89FFFFFF
 PARAMETER C_DCACHE_BYTE_SIZE = 8192
 PARAMETER C_DCACHE_ALWAYS_USED = 1
 PARAMETER HW_VER = 8.10.a
 PARAMETER C_USE_ICACHE = 1
 PARAMETER C_USE_DCACHE = 1
 PARAMETER C_FSL_LINKS = 1
 PARAMETER C_ICACHE_STREAMS = 0
 PARAMETER C_ICACHE_VICTIMS = 0
 PARAMETER C_DCACHE_USE_WRITEBACK = 0
 BUS_INTERFACE DPLB = plb_0
 BUS_INTERFACE IPLB = plb_0
 BUS_INTERFACE DXCL = microblaze_0_DXCL
 BUS_INTERFACE IXCL = microblaze_0_IXCL
 BUS_INTERFACE DEBUG = microblaze_0_mdm_bus
 BUS_INTERFACE SFSL0 = fsl_mb1_master
 BUS_INTERFACE DLMB = dlmb_mb_0_new
 BUS_INTERFACE ILMB = ilmb_mb_0_new
 BUS_INTERFACE MFSL0 = fsl_mb0_master
 PORT MB_RESET = mb_reset
 PORT INTERRUPT = interrupt_controller_Irq
END
```

```
BEGIN microblaze_1
 PARAMETER INSTANCE = microblaze_1
 PARAMETER C_AREA_OPTIMIZED = 0
 PARAMETER C_USE_BARREL = 1
 PARAMETER C_DEBUG_ENABLED = 1
 PARAMETER C_ICACHE_BASEADDR = 0x8A000000
 PARAMETER C_ICACHE_HIGHADDR = 0x8BFFFFFF
 PARAMETER C_CACHE_BYTE_SIZE = 8192
 PARAMETER C_ICACHE_ALWAYS_USED = 1
 PARAMETER C_DCACHE_BASEADDR = 0x8A000000
 PARAMETER C_DCACHE_HIGHADDR = 0x8BFFFFFF
 PARAMETER C_DCACHE_BYTE_SIZE = 8192
 PARAMETER C_DCACHE_ALWAYS_USED = 1
 PARAMETER HW_VER = 8.10.a
 PARAMETER C_USE_ICACHE = 1
 PARAMETER C_USE_DCACHE = 1
 PARAMETER C_FSL_LINKS = 1
 PARAMETER C_ICACHE_STREAMS = 0
 PARAMETER C_ICACHE_VICTIMS = 0
 PARAMETER C_DCACHE_USE_WRITEBACK = 0
 BUS_INTERFACE DPLB = plb_1
 BUS_INTERFACE IPLB = plb_1
 BUS_INTERFACE DXCL = microblaze_1_DXCL
 BUS_INTERFACE IXCL = microblaze_1_IXCL
 BUS_INTERFACE DEBUG = microblaze_1_mdm_bus
 BUS_INTERFACE SFSL0 = fsl_mb0_master
 BUS_INTERFACE DLMB = dlmb_mb_1_new
 BUS_INTERFACE ILMB = ilmb_mb_1_new
 BUS_INTERFACE MFSL0 = fsl_mb1_master
 PORT INTERRUPT = xps_timer_plb_1_Interrupt
 PORT MB_RESET = mb_reset
END
```

```
BEGIN mpmc
 PARAMETER INSTANCE = DDR2_SDRAM
 PARAMETER C_NUM_PORTS = 4
 PARAMETER C_MEM_PARTNO = MT47H32M16-5E
 PARAMETER C_MEM_CLK_WIDTH = 2
 PARAMETER C_MEM_DATA_WIDTH = 32
 PARAMETER C_DDR2_DQSN_ENABLE = 1
 PARAMETER C_PIM0_BASETYPE = 1
 PARAMETER C_XCL0_B_IN_USE = 1
 PARAMETER C_PIM1_BASETYPE = 1
 PARAMETER C_XCL1_B_IN_USE = 1
 PARAMETER HW_VER = 6.03.a
 PARAMETER C_PIM2_BASETYPE = 2
 PARAMETER C_PIM3_BASETYPE = 2
 PARAMETER C_MPMC_BASEADDR = 0x88000000
 PARAMETER C_MPMC_HIGHADDR = 0x8fffffff
 BUS_INTERFACE XCL0 = microblaze_0_IXCL
 BUS_INTERFACE XCL0_B = microblaze_0_DXCL
 BUS_INTERFACE XCL1 = microblaze_1_IXCL
 BUS_INTERFACE XCL1_B = microblaze_1_DXCL
 BUS_INTERFACE SPLB2 = plb_0
 BUS_INTERFACE SPLB3 = plb_1
 PORT MPMC_Clk0 = clk_125_0000MHzDCM0
 PORT MPMC_Rst = sys_periph_reset PORT MPMC_Clk90 = clk_125_0000MHz90DCM0
END
```

C. Auszug aus Linker Script für MicroBlaze_0

```
MEMORY
{
  ilmb_cntlr_mb_0_new_dlmb_cntlr_mb_0_new : ORIGIN = 0x00000050, LENGTH = 0x00001FB0
  DDR2_SDRAM_MPMC_BASEADDR : ORIGIN = 0x88000000, LENGTH = 0x02000000

  DDR2_SDRAM_SHARED_MEM_BASEADDR : ORIGIN = 0x8C000000, LENGTH = 0x04000000
}
```

Definition der Start- und Endadressen für die „Shared Memory Region“
Die Adresse ist in den Linker Scripts von MicroBlaze_0 und MicroBlaze_1 identisch

DDR2_SDRAM_MPMC_BASEADDR : ORIGIN = 0x8A000000, LENGTH = 0x02000000
DDR2_SDRAM_SHARED_MEM_BASEADDR : ORIGIN = 0x8C000000, LENGTH = 0x04000000

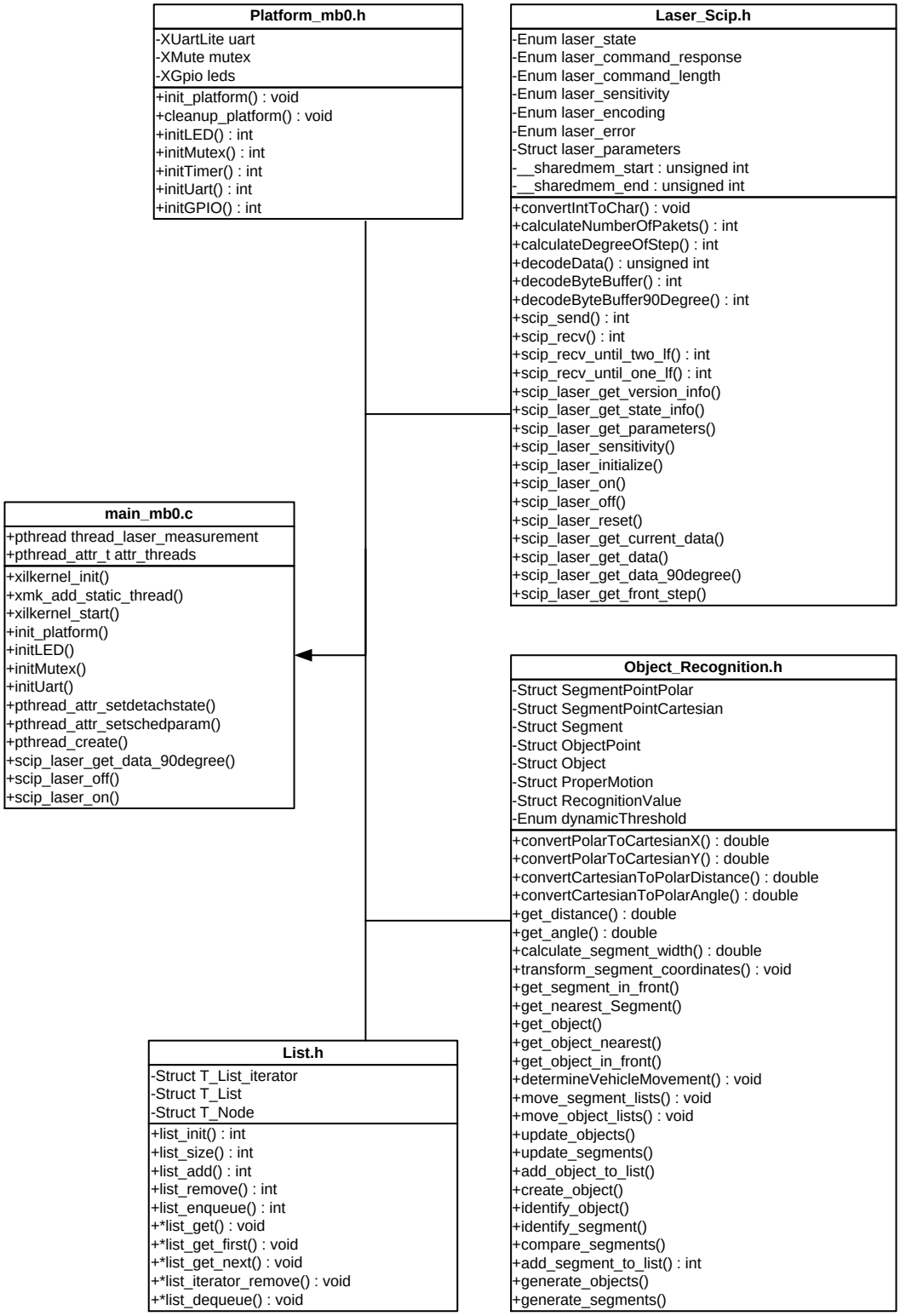
```
.text : {
  *(.text)
  *(.text.*)
  *(.gnu.linkonce.t.*)
} > DDR2_SDRAM_MPMC_BASEADDR

.data : {
  . = ALIGN(4);
  __data_start = .;
  *(.data)
  *(.data.*)
  *(.gnu.linkonce.d.*)
  __data_end = .;
} > DDR2_SDRAM_MPMC_BASEADDR

.bss : {
  . = ALIGN(4);
  __bss_start = .;
  *(.bss)
  *(.bss.*)
  *(.gnu.linkonce.b.*)
  *(COMMON)
  . = ALIGN(4);
  __bss_end = .;
} > DDR2_SDRAM_MPMC_BASEADDR

.sharedmem : {
  __sharedmem_start = .;
  *(.sharedmem)
  __sharedmem_end = .;
} > DDR2_SDRAM_SHARED_MEM_BASEADDR

.heap : {
  . = ALIGN(8);
  _heap = .;
  _heap_start = .;
  . += _HEAP_SIZE;
  _heap_end = .;
} > ilmb_cntlr_mb_0_new_dlmb_cntlr_mb_0_new

.stack : {
  _stack_end = .;
  . += _STACK_SIZE;
  . = ALIGN(8);
  _stack = .;
  __stack = _stack;
} > ilmb_cntlr_mb_0_new_dlmb_cntlr_mb_0_new
```

Der ausführbare Programmcode und die vor-initialisierten Konstanten werden im DDR abgelegt

Die „Shared Memory Region“ liegt im DDR Speicher und ist bei beiden MicroBlazes identisch

Heap und Stack werden bei beiden MicroBlazes im lokalen BRAM Speicher abgelegt

D. Klassendiagramm von MicroBlaze_0



E. Implementierte SCIP 2.0 Kommandos

Das Software-Interface für den Laserscanner wurde sowohl für den Einsatz im SoC-Fahrzeug als auch für den Einsatz in anderen Anwendungen konzeptioniert. Aus diesem Grund wurden alle Kommandos des SCIP 2.0 Protokolls als Funktionen in einer Bibliothek implementiert [34]. Hierzu zählen Funktionen zum Auslesen von Status- und Zustandsinformationen des Laserscanners als auch Funktionen zur Konfiguration von statischen Parametern, wie beispielsweise der Baudrate. Die Kodierung und Konvertierung der Kommandos wurde durch modulare Funktionen entkapselt, sodass der Anwender den Laserscanner mit einfachen Funktionsaufrufen ansteuern kann. Zu den bereits erwähnten Funktionen wurden noch folgende Funktionen implementiert:

SCIP 2.0	Funktionsname	Beschreibung
BM	*scip_laser_on()*	Schaltet den Laserscanner ein
QT	*scip_laser_on()*	Schaltet den Laserscanner aus
GS/GD	*scip_laser_get_current_data()*	Einmaliger Empfang von Abstandswerten
MS/MD	*scip_laser_get_data()*	Kontinuierlicher Empfang von Abstandswerten
VV	*scip_laser_get_version_info()*	Empfängt Versionsdetails wie Seriennummer und Firmware
II	*scip_laser_get_state_info()*	Empfängt aktuelle Zustandsinformationen wie Zeitstempel und Baudrate
PP	*scip_laser_get_parameters()*	Empfängt die Parameter wie Anzahl der Schritte und Reichweite
HS	*scip_laser_sensitivity(laser_sensitivity)*	Stellt die Sensitivität des Sensors ein. Es wird zwischen hoch und niedrig unterschieden
RS	*scip_laser_reset()*	Setzt den Laser auf den Zustand nach dem Einschalten zurück

Tab. E.1.: Implementierte Funktionen des SCIP 2.0 Kommunikationsprotokolls

Die globale Funktion *scip_laser_initialize()* initialisiert die Integer- und Char Arrays zur Speicherung der Abstandswerte. Des Weiteren werden die in einer Struktur gespeicherten Laserscanner Parameter festgelegt. Mit einer globalen Enumeration Deklaration wird ein einheitliches „Error Handling“ erreicht, welches durch definierte Konstanten einen Fehler eindeutig identifiziert. Die Laserscanner Bibliothek besteht aus der *laser_scip.c* Source Datei und einer *laser_scip.h* Header Datei. Letztere enthält alle global definierten Variablen und alle Funktionsprototypen.

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §24(5) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, den 27. Januar 2012 | _____

Ort, Datum | Unterschrift