Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Philipp Teske

Entwicklung einer Bibliothek als Basis für die Programmierung und Steuerung des c't-Bots

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Philipp Teske

Entwicklung einer Bibliothek als Basis für die Programmierung und Steuerung des c't-Bot

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. G. Klemke
Zweitgutachter: Prof. Dr.-Ing. A. Meisel

Abgegeben am 07. Juli 2009

Thema der Bachelorarbeit

Entwicklung einer Bibliothek als Basis für die Programmierung und Steuerung des c't-Bot

Stichworte

c't-Bot, Multitasking, Embedded, Mikrocontroller, Aksen

Zusammenfassung

Gegenstand der Arbeit ist die Entwicklung einer Programmbibliothek, die grundlegende Funktionen zur Ansteuerung der Sensoren und Aktoren eines c't-Bots zur Verfügung stellt. Mit ihrer Hilfe soll es möglich sein, sowohl einfache Programme, wie auch komplexe Systeme mit mehreren Prozessen entwickeln zu können und über das vorhandene Zigbee-Modul Daten mit anderen Geräten auszutauschen. Hierbei sollen jedoch alle systemrelevanten Funktionen, wie z.b. Interrupthandling und Prozessverwaltung vor dem Benutzer verborgen bleiben, damit sich dieser bei der Entwicklung eigener Programme nicht mit diesen Funktionen zusätzlich auseinandersetzen muss. Kern der Bibliothek ist ein Multitasking-System, welches die Verwaltung und Steuerung einer beliebigen Anzahl von Prozessen erlaubt.

Title of the paper

Development of a library as a base for the programming and control of the c't-Bot

Keywords

c't-Bot, Multitasking, Embedded, Microcontroller, Aksen

Abstract

This thesis documents the development of a function library which provides basic functions for controlling of sensors and actuators of a c't-Bot. Development of simple programs as well as of complicated systems with several processes and the ability to exchange data with other devices by means of the available Zigbee-module should be possible by using the function library. Nevertheless all system relevant functions for example interrupt handling and process administration, should remain concealed so that the user must not give attention to this. Core of the library is a multitasking system which permits the administration and control of any number of processes.

Inhaltsverzeichnis

1. Einleitung

1.1 Motivation

Bereits seit mehr als 10 Jahren wird an der Hochschule für Angewandte Wissenschaften (HAW) in Hamburg im Bereich der Robotik und autonomen Systeme geforscht. Der ursprünglich mit Lego-Technik und dem 6.270-Board vom Massachusetts Institute of Technology (MIT) begonnene Forschungszweig teilte sich im Laufe der Zeit in mehrere Bereiche auf. So führt u.a. das Projekt FAUST (Fahrerassistenz- und Autonome Systeme) bereits in den Bereich der professionellen Entwicklung entsprechender Systeme, während andere Projekte wie die Arbeiten mit Lego Mindstorms, dem 6.270-Board und einigen Eigenentwicklungen eher Lehr- und Forschungszwecken dienen. Seit einigen Semestern stehen zudem mehrere c't-Bots des Heise Verlags [26] für Versuche zur Verfügung. Auf Grund mangelnder Alternativen dient als Softwaregrundlage gegenwärtig jedoch noch das originale Standard-Software-Framework.
Dieses ist für weitaus mehr Funktionen ausgelegt, als in den an der Hochschule laufenden Projekten benötigt werden. In der Praxis erweist sich das unübersichtliche und mit Funktionen überladene Framework als ungeeignet für ein effektives Programmieren. Die Einarbeitung in die Software ist mit einem erheblichen Zeitaufwand verbunden und erschwert die Projektarbeit, da relevante Funktionen vorher gefiltert werden müssen.
Daraus folgernd stellt sich die Frage nach einer speziell an die Bedürfnisse der HAW angepassten Software, die ein effektiveres Lernen und zügigeres Umsetzung der Projekte ermöglicht. Mit der Entwicklung einer hochschuleigenen Software soll diese Problematik gelöst werden.

1.2 Zielsetzung

Ziel dieser Bachelorarbeit ist es, eine optimierte, anwenderfreundliche und leicht erlernbare Software für den c't-Bot zu entwickeln, die genau auf die Bedürfnisse der HAW zugeschnitten ist. Diese Software soll von ihrer Form her dem Framework des an der Fachhochschule Brandenburg entwickelten AKSEN-Boards angelehnt sein und als eine Bibliothek realisiert werden, die Basisfunktionen zur Programmierung des c't-Bots zur Verfügung stellt. Mit ihrer Hilfe wird es möglich sein, sowohl einfache Programme, wie auch komplexe Systeme mit mehreren Prozessen entwickeln zu können und über das vorhandene Zigbee-Modul Daten mit anderen Geräten auszutauschen. Gleichzeitig sollen systemrelevanten Funktionen, wie z.b. Interrupthandling und Prozessverwaltung vor dem Benutzer verborgen bleiben, damit sich dieser bei der Entwicklung eigener Programme nicht mit diesen auseinandersetzen muss.
Den Kern der Bibliothek wird ein Multitasking-System darstellen, welches einen preemptiven Scheduler beinhaltet und Prozesse im Round-Robin Verfahren bearbeitet.

1.3 Historie

Seit frühester Zeit schon sind die Menschen bestrebt, sich das Leben einfacher und leichter zu gestalten. Bereits um 320 v. Chr. schrieb Aristoteles
„ …, w*enn jedes Werkzeug auf einen erhaltenen oder erratenen Befehl hin sein Werk zu erzielen vermöchte, wie es von den Bildsäulen des Daidalos und den Dreifüßen des Hephaistos heißt, von welchen der Dichter sagt, dass sie von selbst eingingen in die Schar der Götter, wenn so die Weberschiffe selber webten und die Zitherschlägel von selbst die Zitter schlügen, dann freilich bedürfe es für die Meister keine Gehilfen und für die Herren keine Sklaven.“* [3],
In Erfüllung gegangen ist dieser Traum jedoch erst in den letzten Jahrzehnten, als die Maschinen durch Aufkommen der Computer erstmals in der Lage waren, komplexere Arbeitsvorgänge selbstständig auszuführen.
Der Begriff „Roboter“ wurde erst durch die Autoren Karel Capek und Isaac Asimov populär. Capek erwähnte 1920/21 erstmals den Begriff Roboter in seinem Roman „Rossum's Universal Robots“. Asimov war Science-Fiction-Autor und schrieb in den 40er Jahren Geschichten, in denen er das Thema verarbeitet. In dieser Zeit begann auch die eigentliche Entwicklung und Forschung von Industrie- und Nutzrobotern. 1954 meldete der amerikanische Erfinder George C. Devol ein Patent für einen „programmierbaren Manipulator zum Transport von Gegenständen“ an. Basierend auf dieser Vorarbeit entwickelte die Firma Unimation in den 60er Jahren den ersten kommerziellen und hydraulisch gesteuerten Industrieroboter (Abb. 1). Dieser hatte die Aufgabe, in einem Werk von General Motors Gussteile aus einer Maschine zu nehmen. In den nächsten zehn Jahren wurden die Roboter für Einsatzfelder wie Punktschweißen und zur Bearbeitung von Werkstücken weiterentwickelt.

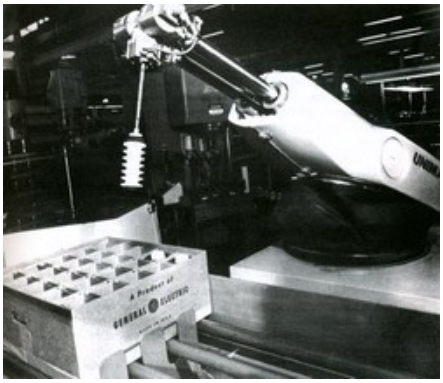


Abbildung 1: Unimate
(Quelle: mmvr.burg-halle.de/typo3/3706.html)

1966 präsentierte das Stanford Research Institute (heute SRI Technology) den ersten mobilen Roboter namens Shakey. Er war in der Lage, sich mit Hilfe einer Kamera, Laserentfernungsmessern und Stoßsensoren selbständig in Räumen zu bewegen. Ebenfalls am SRI entwickelte der Student Victor Scheimann 1969 den sogenannten „Stanford-Arm“ (Abb. 2) [47]. Dieses Greifwerkzeug stellt den Grundstein für die später entwickelten Industrieroboter dar. Knapp 5 Jahre später stellte die Firma ASEA den ersten „Kleinstrechner-gesteuerten, komplett elektrischen Industrieroboter“ vor [31]. Dieser war in der Lage, stufenlose Bahnbewegungen zu vollziehen, welche Voraussetzung für Lichtbogenschweißen und die Zerspannungstechnik waren. Weitere 5 Jahre vergingen bis SCARA (Selective Compliance Assembly Robot Arm) der Öffentlichkeit vorgestellt wurde. Er wurde in Japan von Hiroshi Makino an der Yamanashi University entwickelt. Im Gegensatz zum Stanford-Arm besaß SCARA nur 4 Achsen. Durch dieses Design war er perfekt geeignet für die Montage von

Abbildung 2: Stanford-Arm
(Quelle: infolab.stanford.edu/pub/voy/museum/pictures/display/1-Robot.htm)

Kleinteilen [31],[30]. Zur selben Zeit begann die Entwicklung der ersten integrierten Schaltkreise und Mikrocontroller. Dies hatte zur Folge, dass sich immer leistungsfähigere und platzsparendere Systeme entwickeln ließen. Es war somit nahe liegend, dass sich die Robotik auch in andere Bereiche ausdehnte.
Besonders die Rüstungsindustrie zeigte Interesse an technischen Neuheiten.
Seit 1995 setzen besonders die USA bei Aufklärungs- und Angriffsmission in Krisengebieten auf unbemannte Luftfahrzeuge, sogenannte Drohnen. 1998 entwickelte eine Unterabteilung von Northrop Grumman die in Abb. 3 dargestellte „Global Hawk“, einen unbemannten und autonom agierenden Langstreckenaufklärer. Die Global Hawk ist das zur Zeit größte UAV (Unmanned/Uninhabited Aerial Vehicle) der Welt und das erste, welches in den USA eine Zulassung für die Teilnahme am zivilen Luftverkehr bekommen hat [34].

Abbildung 3: RQ-4 Global Hawk
(Quelle: www.symscape.com/node/626)

Aber die Forschung hat sich auch auf weniger spektakuläre Bereiche des Lebens ausgedehnt. So findet man gegenwärtig in Kinderzimmern immer seltener Puppen und Spielzeugautos. Stattdessen wurden sie durch Furby, Aibo (Abb. 4) oder andere sogenannte „intelligente Spielzeuge“ ersetzt. Ältere Kinder begeistern sich für den Umgang mit komplexerer Technik. Hierfür sorgt Lego mit seiner *Mindstorms* Produktreihe.
Seit Mikrocontroller und Sensoren zu Massenprodukten wurden, werden kleine, autonome Roboter auch in höheren Schulen und Universitäten zu Forschungs- und Ausbildungszwecken eingesetzt.

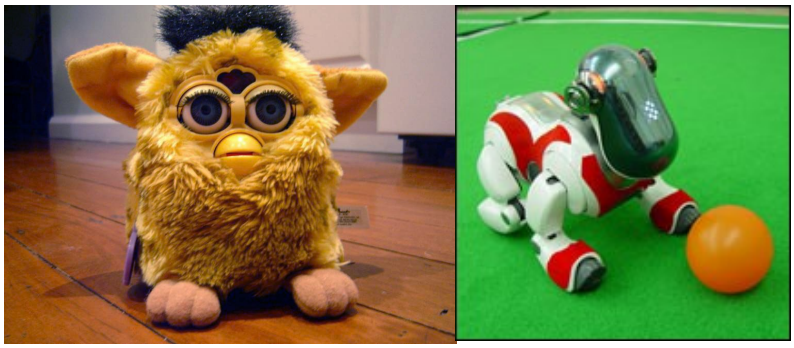


Abbildung 4: Furby und Aibo

1.4 Handliche Roboter

Eine sowohl in Europa wie auch in den USA weit verbreitete Steuereinheit für reaktive Roboter ist das 6.270-Board und dessen Nachfolger, das „Handy Board“. Beide wurden am MIT entwickelt und dienten ursprünglich zur Steuerung von Robotern, welche aus Legosteinen zusammengesetzt waren [17],[50]. Die Fachhochschule Brandenburg erweiterte das 6.270-Board um einen leistungsstärkeren Mikrocontroller sowie mehr Anschlussmöglichkeiten. Dieses neue Board wurde unter dem Namen „AKSEN-Board“ veröffentlicht und tritt in mehreren Deutschen Universitäten die Nachfolge des 6.270-Boards an [20].

Ein weiterer, in Deutschland als Heimroboter weit verbreiteter Roboter ist der ASURO (Abb. 5). Dieser wurde am Institut für Robotik und Mechatronik in Oberpfaffenhofen entwickelt und besteht zum größten Teil aus handelsüblichen Bauteilen. Der ASURO wurde ursprünglich vom Deutschen Zentrum für Luft- und Raumfahrt (DLR) für ein Experiment des DLR School Lab konzipiert, in dessen Verlauf er von Schülern zusammengebaut und programmiert werden sollte. Aufgrund seines Erfolges und der großen Nachfrage, die er nicht zuletzt seinem niedrigen Preis verdankt, ist er seit einiger Zeit auch im Handel erhältlich. Er besitzt zwei Lichtsensoren, 6 Tastsensoren und zwei Lichtschranken zur Orientierung sowie 2 Antriebsmotoren. Die Software für den 8-Bit Mikrocontroller des Roboters kann wahlweise in den Programmiersprachen Assembler oder C geschrieben werden [21].

Abbildung 5: Asuro
(Quelle: www.rn-wissen.de/index.php/Asuro)

Für jüngere Bastler bietet Lego die passende Alternative.
Das Mindstorms System besteht aus einer zentralen Steuereinheit, die mit Hilfe der bei Lego üblichen Steckverbindungen, mit bis zu 3 Motoren und 4 verschiedenen Sensoren verbunden werden kann. Die Entwicklungsumgebung erlaubt die Entwicklung von Software wahlweise über eine graphische Oberfläche oder mit Hilfe einer speziell entwickelten Programmiersprache [42].
Seit Mitte der 90er bietet die Firma Conrad Electronics GmbH ein dem AKSEN-Board ähnliches Mikrocontroller-System namens „C-Control“ an. Mit Hilfe des C-Control Systems lassen sich einfache Automatisierungsvorgänge realisieren. Ihren Erfolg verdankt die C-Control u.a. einer großen Auswahl an deutschsprachiger Literatur und zahlreichen Internetseiten zu diesem Thema. Um aus der C-Control eine autonom agierende Maschine zu machen, werden verschiedene Bausätze angeboten. Die Programmierung der C-Control und ihrer Nachfolger erfolgt in einer speziellen Form der Programmiersprachen Basic, Assembler oder C [44],[24].
Ein weiterer Hersteller von Robotern ist die Firma Merlin Systems aus Großbritannien. Sie vertreibt Miniroboter mit dem Namen „Miabot". Diese kleinen Roboter in Form eines Würfels sind seit einigen Jahren fester Bestandteil des Internationalen FIRA (Federation of International Robot-soccer Association) – Cup [46],[43].

1.5 Software Architekturen

Im Zusammenhang mit der Erstellung von Steuerungsprogrammen für autonome Roboter lassen sich die zahlreich existierenden Architekturen in zwei große Gruppen unterteilen. Auf der einen Seite stehen die klassischen Ansätze, welche die Bewegung eines Roboters immer nach dem selben Ablauf der Bearbeitungsschritte planen, auf der anderen Seite die verhaltensbasierten Systeme.
Die traditionelle Herangehensweise für den Aufbau eines Systems eines autonomen Roboters besteht aus der Unterteilung in funktionale Module, wie z.B. Sensorinterpretation, Umgebungsmodellierung, Aktionsplanung und Aktionsausführung (Abb. 6) [15],[7],[4].

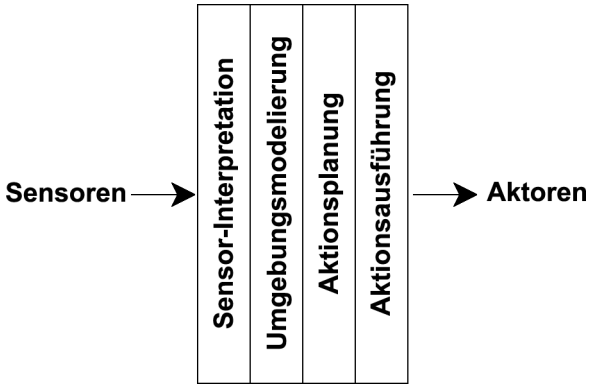


Abbildung 6: Traditionelle Architektur

Bei diesem Ansatz werden zu Beginn die Daten aller Sensoren gesammelt, gefiltert und ausgewertet. Hierbei werden auch Störungen oder widersprüchliche Daten beseitigt, so dass im zweiten Schritt ein konsistentes Modell erstellt werden kann, welches die aktuelle Umgebung des Roboters repräsentiert. Hier liegt jedoch eine der großen Herausforderungen dieser Art von Architekturen. Selbst mit den besten sensorischen Ausstattungen ist es nur möglich die Umgebung bis zu einem bestimmten Grad zu erfassen. Ein Modell der Umgebung wird somit nie der realen Umwelt entsprechen können. Je nach Sensor-Ausstattung, Geschwindigkeit des Roboters und evtl. vorhandenen, beweglichen Objekten, weicht das Modell von der wirklichen Umgebung ab. Hier zeigt sich einer der Nachteile dieser Systeme. Durch die sequentielle Abarbeitung der einzelnen Schritte können Veränderungen der Umwelt erst bei einem erneuten Durchlauf erkannt werden. Sollte sich, nachdem die Sensoren die Umwelt erfasst haben, die Umgebung ändern, so wird die geplante Aktion unbrauchbar und kann zu einer falschen und möglicherweise fatalen Fehlentscheidung führen.
Die Probleme, welche durch die Abbildung der Umwelt entstehen, können durch die Architekturen der zweiten großen Gruppe, den verhaltensbasierten Systemen, vermieden werden [12]. Bei diesen besteht dir Bewegung des Roboters nicht aus der Umsetzung eines vorgefertigten Plans, sondern durch direkte Koppelung zwischen Sensoren und Aktoren. Darüber hinaus wird die Steuerung auf viele parallel arbeitende Module verteilt, aus deren Zusammenspiel sich wiederum das Verhalten des Roboters zusammensetzt. Jedoch bekommt zu jedem Zeitpunkt nur ein Modul den Zugriff auf die Aktoren des Roboters. Es lässt sich also sagen, dass die einzelnen Module um den Zugriff konkurrieren, daher wird diese Art der Architekturansätze auch als „Kompetitive Ansätze“ bezeichnet [12].

Ein Beispiel für eine verhaltensbasierte Architektur ist Brooks berühmte „subsumption architecture“. Diese wurde 1986 am MIT Artificial Intelligence Laboratory von Professor Rodney A. Brooks und seinen Mitarbeitern entwickelt [9],[8],[11],[10]. Anstatt die Steuerung des Systems in die vorher üblichen funktionellen Module zu unterteilen, entschied man sich, das System in mehrere unabhängige Schichten, sogenannte Behaviours, aufzuteilen und diese mit unterschiedlichen Prioritäten zu versehen. Zusätzlich sollten Sensoren nicht mehr vorab ausgewertet werden, sondern lediglich dazu dienen, die Abarbeitung einzelner Behaviours anzustoßen.
Die Subsumption Architektur verfolgt dabei den Ansatz, dass Behaviours höherer Schichten in der Lage sind, die In- und Outputs niedrigerer Schichten zu unterdrücken. Somit kann eine Schicht die darunter liegende hindern, auf bestimmte Sensorwerte zu reagieren oder deren Ansteuerung der Aktoren unterbinden.
Dies lässt sich an einem einfachen Beispiel verdeutlichen: Abb. 7 zeigt die Steuerung eines autonomem Roboters. Sie besteht aus einer einfachen Subsumption Architektur mit zwei Schichten. Die untere Schicht dient dazu, den Roboter zu einem bestimmten Ziel zu bewegen und wird durch Infrarotsensoren und einem GPS-System gesteuert. Die obere Schicht wird durch Ultraschallsensoren gesteuert und sorgt dafür, dass der Roboter Hindernissen ausweicht. Sobald das obere Verhalten durch Annäherung an ein Hindernis aktiv wird, bekommt es exklusiven Zugriff auf die Motoren des Roboters um eine Kollision zu verhindern und unterdrückt dabei den Output der unteren Schicht. Diese Art von Architektur erfordert durch die parallelen Module jedoch die Implementation der gleichen Fähigkeiten in unterschiedlichen Modulen. Das Programm enthält somit eine gewisse Redundanz, wodurch es schwerer zu warten wird [9],[5].

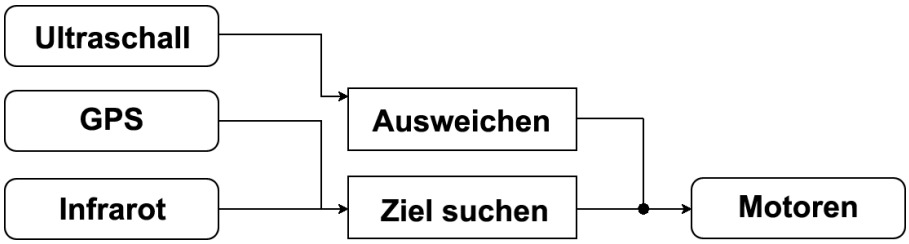


Abbildung 7: Subsumption Beispiel

Eine weitere Art von Architekturen sind die hybriden und konnektionistischen Ansätze. Bei hybriden Systemen handelt es sich um eine Symbiose der klassischen sequentiellen und der parallelen Ansätze. Diese Art der Architektur gliedert sich hierbei in mehrere Schichten. In den höheren Schichten befindet sich eine sequentielle Struktur, welche ein Modell der Umgebung enthält und aus diesem die weiteren Aktionen plant. Niedrigere Schichten spalten den erstellten Plan in Abhängigkeit der aktuellen Situation in eine zeitlich und räumlich beschränkte Sequenz [2]. Die niedrigste Schicht schließlich besteht aus parallel arbeitenden Modulen, die von der erzeugten Sequenz modifiziert oder zur Ausführung gebracht werden. Eine der ersten hybriden Architekturen war das von Arkins entwickelte AuRA [13],[14].
Einen komplett anderen Weg zur Erstellung eines Steuerungsprogrammes hingegen beschreiben die konnektionistischen Architekturen. Das Steuerungsprogramm besteht bei diesen Systemen aus einem neuronalen Netzwerk, was insbesondere das Erlernen von Verhaltensweisen ermöglicht. Ein Beispiel für diese Art von Architektur bildet die „Dynamical Recurent Associative Memory Architecture“ von Billards und Hayes [1].

1.6 Das c't-Bot Projekt

Der c't-Bot entstand aus einem Projekt der Zeitschrift c't des Heise Verlags. Erstmals vorgestellt worden ist er in der Ausgabe vom 2/2006.
Er ist mit zwei Motoren ausgestattet, die es ihm ermöglichen auf engstem Raum zu agieren. Um seine seine Umgebung wahrzunehmen stehen 13 Sensoren zur Verfügung.
In den 3 Jahren seit der Entstehung hat sich eine große Fangemeinde um den kleinen Roboter gebildet. Er wurde im Laufe der Zeit sowohl hardwaremäßig als auch softwaremäßig ständig weiterentwickelt. Hardwareseitige Erweiterungen werden per „Erweiterungsplatinen“ (Extension-Boards) einfach über der Hauptplatine platziert. So gibt es vom Heise Verlag bisher zwei Erweiterungen für den c't-Bot. Zum einen eine Platine, die es ermöglicht, mit dem Roboter über Wireless-Lan zu kommunizieren und Speicherkarten auszulesen, zum anderen eine Klappe für das Transportfach, damit einmal eingefangene Gegenstände nicht wieder verloren gehen. Beides wurde in der Ausgabe vom 2/2007 veröffentlicht.

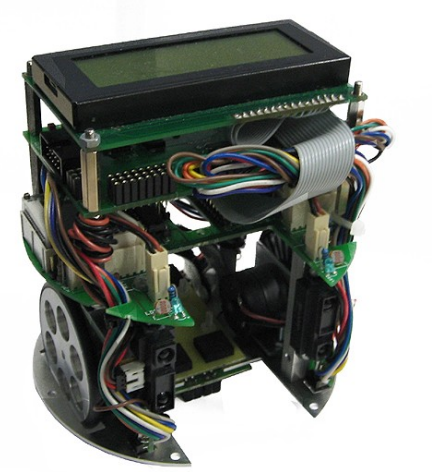


Abbildung 8: c't-Bot mit LCD-Erweiterung

Findige Bastler haben jedoch noch weit mehr an Erweiterungen parat. So findet man bereits c't-Bots mit Ultraschallsensoren, Temperatursensoren oder einem elektronischen Kompass.
Die zentrale Steuereinheit des c't-Bots besteht aus einem 8-Bit Mikrocontroller der Firma Atmel. Dieser liefert die nötige Rechenleistung, um die Motoren anzusteuern, Sensoren auszulesen und mit der Außenwelt zu kommunizieren. Um diese ganzen Aufgaben in einer angemessen kurzen Zeit zu bewältigen, wird er mit einem Takt von 16Mhz betrieben.
Für diejenigen, die nicht im Besitz eines c't-Bots sind, steht ein extra Simulator zur Verfügung. Dieser nennt sich „c't-Sim“ und wurde in der Programmiersprache Java verfasst. Er ermöglicht es, seine entwickelten Programme in einer virtuellen Umgebung zu testen, bevor man sie auf den Roboter überträgt. Die Software ist inzwischen soweit fortgeschritten, dass man den Bot per seriellem- oder USB-Kabel mit der Simulation Daten austauschen zu lassen kann und somit die Möglichkeit hat, seine Programme noch realitätsnaher zu testen. [27][26]

2. Anforderungsanalyse

Ziel dieser Arbeit ist die Entwicklung einer optimierten, anwenderfreundlichen und leicht erlernbaren Software für den c't-Bot. Diese Software soll in Form einer Bibliothek realisiert werden, die Basisfunktionen zur Programmierung des Bots zur Verfügung stellt. Gleichzeitig sollen systemrelevanten Funktionen, wie z.b. Interrupthandling und Prozessverwaltung vor dem Benutzer verborgen bleiben.
Viele Studenten der HAW nutzen als alternatives Betriebssystem neben Windows sowohl MacOS wie auch eines der zahlreichen Linux Derivate. Die zu entwickelnde Softwarebibliothek muss sich somit aufgrund der unterschiedlichen Plattformen und Vorlieben der Nutzer möglichst einfach in ein Projekt einer beliebigen Entwicklungsumgebung einbinden lassen. Dies bedeutet, dass keine Betriebssystem und Entwicklungsumgebung abhängigen Eigenschaften verwendet werden dürfen.
Weiterhin soll die Bibliothek die Freiheit bieten, Programme erstellen zu können, die von einer einfachen linearen Funktion bis hin zu einem System mit parallel arbeitenden Prozessen reichen. Resultierend aus diesem breiten Anwendungsspektrum ergibt sich für die Bibliothek somit die Notwendigkeit nach einem Multitasking-System, welches sich automatisch den jeweiligen Bedingungen anpasst. Zudem soll der Benutzer jedoch in der Lage sein, beliebige Prozesse manuell anzuhalten, fortzusetzen oder zwischen ihnen zu wechseln.
Des weiteren müssen alle in dem Roboter verbauten Sensoren sowie berechnete Sensorwerte ohne größeren Aufwand abgefragt werden können. Sollte es erforderlich sein, werden die Sensorwerte von der Bibliothek vorverarbeitet.
Selbiges gilt für Aktoren wie Servos oder Motoren. Hier ist gewünscht, dem Benutzer ein möglichst einfaches Interface zur Verfügung zu stellen. Komplexere Aufgaben der Ansteuerung werden vollständig von der Bibliothek übernommen.
Hinzukommend verfügen die c't-Bots der HAW über ein von Bruno Carstensen und Oliver Neumann entwickeltes, hochschuleigenes Erweiterungsboard, welches ebenfalls genutzt werden soll. Auf diesem befinden sich neben einigen Sensoren auch ein Kommunikationsmodul für Datenübertragungen per Funk. Somit muss zum einen die fehlerfreie Kommunikation mit dem Erweiterungsboard gewährleistet werden, zum anderen müssen die Sensoren des Boards mit in die Bibliothek integriert werden. Als wünschenswert wird zudem noch die Ansteuerung der Zigbee-Funkmodule auf den Erweiterungsboards angesehen.
Neben sämtlichen funktionalen Anforderungen darf die Bibliothek jedoch nur wenig Programm- und Datenspeicher belegen, um genügend Ressourcen für die Software des Anwenders zur frei zu halten. Gleiches gilt für die Ausnutzung der Rechenzeit.
Zusammenfassend lässt sich somit sagen, dass die Software folgende Eigenschaften aufweisen soll:

- Einfache Einbindung in beliebige Entwicklungsumgebungen und bestehende Projekte
- Multitasking-System mit Funktionen zum Anlegen, Löschen, Anhalten und Fortsetzen, sowie zum manuellen Wechseln von Prozessen
- Einfachen Zugriff auf digitale und analoge Sensoren
- Einfache Ansteuerung aller Aktoren, wie Servos und Motoren
- Kommunikation mit dem Erweiterungsboard und dem darauf befindlichen Zigbee-Modul
- Wenig Verbrauch an Systemressourcen, speziell Daten- und Programmspeicher

3. Hardwareanalyse

3.1 c't-Bot Übersicht

Der c't-Bot ist ein kleiner autonomer Roboter, dem zwei Getriebemotoren als differentiellen Antrieb dienen. Das Herzstück bildet ein 8-Bit Mikroprozessor vom Typ ATmega32 der Firma Atmel, welcher über einen Quarz mit einem Takt von 16Mhz betrieben wird [22]. Zur Wahrnehmung seiner Umgebung dienen dem Roboter insgesamt 12 Sensoren. Wände und andere Objekte, die sich vor ihm befinden, werden mit Hilfe von zwei Distanzsensoren der Firma Sharp erkannt. Der Untergrund wird von insgesamt 4 Reflex Optokopplern abgetastet. Dieser Sensortyp dient ebenfalls als Radencoder. Auf der Rückseite der beiden Räder sind jeweils streifenförmig schwarze Linien aufgebracht, mit deren Hilfe erkannt werden kann, wie schnell es sich diese drehen. Ein weiterer Sensor befindet sich ebenfalls unter dem Roboter und dient zur Messung der zurückgelegten Wegstrecke. Bei diesem Sensor handelt es sich um einen optischen Bewegungssensor wie er auch in handelsüblichen optischen Mäusen eingesetzt wird.
Zur Kommunikation stehen dem c't-Bot neben 8 individuell ansteuerbaren LEDs noch ein Dot-Matrix LC-Display mit 4 Zeilen je 20 Zeichen und ein Infrarot-Empfänger zur Verfügung. Letzterer dient zum Empfangen von Befehlen durch Fernbedienungen oder anderen Infrarotquellen. Der Antrieb wird mit zwei Getriebemotoren der Firma Faulhaber realisiert. Jeder Motor ist für eine maximale Drehzahl von 151 UpM ausgelegt. Da diese einen höheren Strom benötigen als vom Mikrocontroller geliefert werden kann, ist ihnen noch ein Motortreiber vorgeschaltet. Mit dessen Hilfe wird zudem eine einfachere Steuerung von Drehrichtung und -geschwindigkeit erzielt. Da der Controller nur über eine verhältnismäßig geringe Anzahl an Anschlüssen verfügt, wurde die Peripherie, die nur zur Ausgabe von Daten dient, über Seriell-Parallel Wandler mit dem Mikrocontroller verbunden. Konkret handelt es sich hierbei um das Display, die LEDs sowie 8 Steuersignale, mit denen einige Sensoren aktiviert und deaktiviert werden können. Damit der c't-Bot aufrüstbar ist, wurde zusätzlich eine Schnittstelle für Erweiterungsboards mit auf der Platine untergebracht. Diese besteht neben der Spannungsversorgung aus einer seriellen RS232 Schnittstelle [27],[26]. Abbildung 9 zeigt den Aufbau eines c't-Bots mit allen Komponenten.

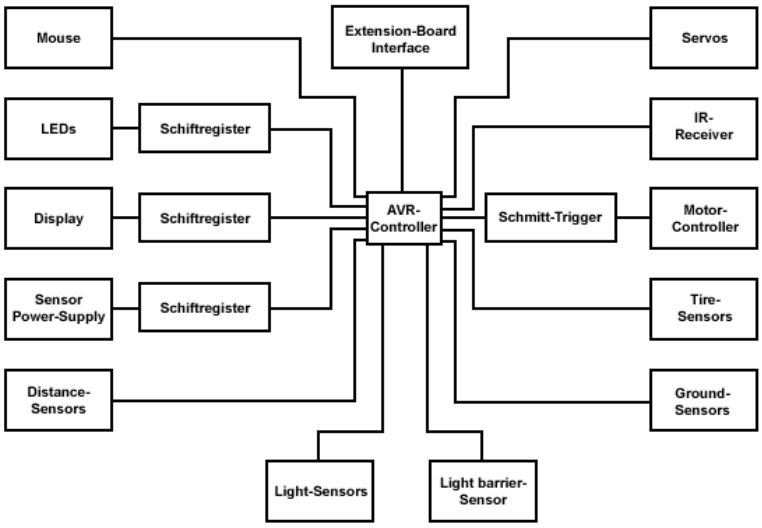


Abbildung 9: Schematik des c't-Bots

3.2 Das AKSEN-Board

Da die zu entwickelnde Software mit der des AKSEN-Boards vergleichbar soll, ist es vorteilhaft sich zunächst einmal mit der Hardware des Systems auseinander zu setzen. Dies ermöglicht später eine leichtere Analyse der Softwarebibliothek.

Das AKSEN-Board ist, anders als der c't-Bot, für den Aufbau von unterschiedlichen Arten von Robotern vorgesehen. Es besitzt, wie in Abb. 10 dargestellt, separate Anschlüsse für Aktoren wie Servos oder Motoren sowie drei Encoder-Eingänge und einen spezieller IR-Port. Die meisten Komponenten werden jedoch über die zahlreich vorhanden analogen Eingänge oder über digitale I/O-Anschlüsse mit dem System verbunden.
Neben diesen besitzt das Board eine eigene serielle Schnittstelle sowie einen CAN-Port. Ähnlich wie beim c't-Bot stehen zur Ausgabe von Informationen 4 Anschlüsse für Lampen oder LEDs sowie ein aufsteckbares Display bereit. Für statische Funktionen, wie z.B. das Einstellen eines Programm-Modus steht dem Benutzer des AKSEN-Board zusätzlich ein 4-poliger DIP-Schalter zur Verfügung. Tabelle 1 veranschaulicht die Unterschiede zwischen der Hardware des AKSEN-Boards und des c't-Bots.

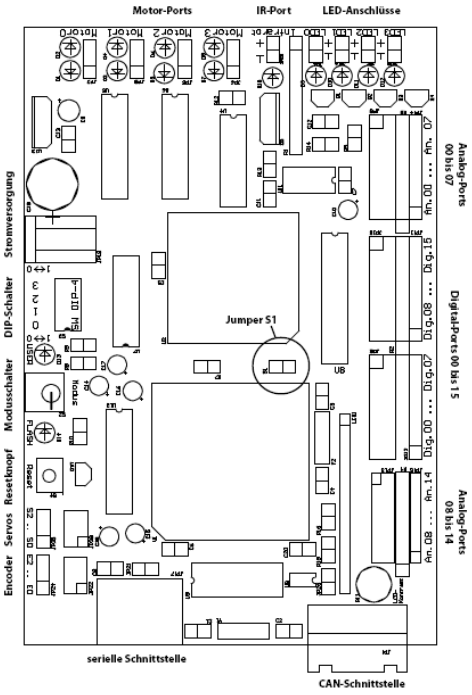


Abbildung 10: AKSEN-Board Layout (Quelle: AKSEN-Board Handbuch)

		AKSEN-Board	c't-Bot
Optik	**LED**	4	4
	LCD	1 Anschluss	1 Anschluss
Aktoren	**Motor**	max. 4	max. 2
	Servo	max. 11	max. 2
Sensoren	**Distanzsensor**	Insgesamt 15 analoge u. digitale Eingänge für beliebige Sensoren	2
	Lichtstärke Sensor		2
	Berührungssensor		x
	Encoder	3	2
	Infrarotsensor	1	1
	Sonstige Sensoren	x	4 Reflex-Optokoppler 1 optischer Bewegungssensor
Schnittstellen	**CAN**	1	x
	Seriell	1	1
Sonstiges	**Sonstiges**	4-pol. DIP-Schalter 1 Infrarotsender	Anschluss für Erweiterungsboards

Tabelle 1: Vergleich AKSEN-Board & c't-Bot

3.3 Mikrocontroller

Das Herzstück des c't-Bots, der Atmega32, ist ein 8-Bit Mikrocontroller der AVR-Serie der Firma Atmel (Abb. 11). Er verfügt über 32KByte Programmspeicher, 2kByte statischen Ram-Speicher (SRAM) sowie ein 1KByte EEPROM um langfristige Daten zu speichern. Sein RISC-Befehlssatz umfasst insgesamt 131 Befehle, von denen die meisten innerhalb eines Taktes abgearbeitet werden können [22].

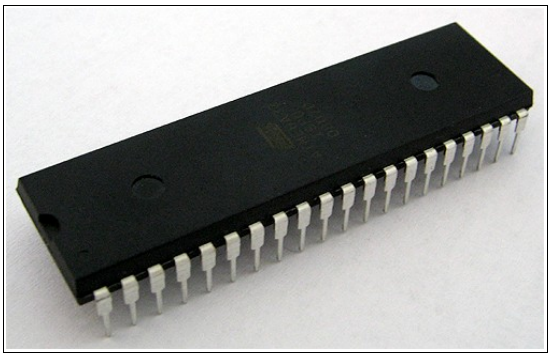


Abbildung 11: Mikrocontroller Atmega32

Unterstützt wird dies unter anderem durch die Harvard-Architektur. Bei diesem Prozessor-Design sind die Speicher von Software und Nutzdaten physikalisch voneinander getrennt. Somit können Befehle und Daten zeitgleich aus den jeweiligen Speichern geladen werden. Für weitere Performance sorgen die 32 „General Purpose Register“, auf welche innerhalb von einem Taktzyklus zugegriffen werden kann.
Seinen Arbeitstakt bekommt der Prozessor entweder aus einem internen Oszillator oder durch eine extern angeschlossene Taktquelle. Hierbei kann es sich um einen Quarz oder einen Oszillator mit bis zu 16Mhz handeln.
Neben dem eigentlichen Prozessor befinden sich auch 3 Timer in dem Chip, mit denen es unter anderem möglich ist, die Zeit zwischen zwei Ereignissen zu messen oder bis zu 4 PWM-Signale zu erzeugen.
Zur Kommunikation stehen zusätzlich zu den 32 I/O-Leitungen auch eine serielle Schnittstelle (UART), eine SPI-Schnittstelle und ein TWO-Wire Interface, welches kompatibel zu dem von Philips entwickelten I²C-Bus Protokoll ist, zur Verfügung.
Damit keine Ereignisse verpasst werden, ist noch ein Interrupt-Controller in dem Gehäuse untergebracht. Der Interrupt-Controller erlaubt es, den Programmablauf beim Auftreten von verschiedenen Ereignissen zu unterbrechen und eine vom Benutzer definierte Funktion auszuführen. Als mögliche Auslöser stehen unter anderem Timerüberläufe, Ereignisse an einem der oben genannten Schnittstellen oder Flanken- und Pegelwechsel an einem von drei speziellen I/O-Pins zur Verfügung.
Des weiteren befindet sich auf dem Chip noch ein 10-Bit Analog-Digital-Wandler mit dessen Hilfe 8 analoge Signale digitalisiert werden können. Abbildung 12 veranschaulicht den internen Aufbau des Prozessors und seiner Komponenten.
Wichtige und dauerhafte Einstellungen, wie z.B. das Einstellen der Taktquelle oder festlegen des Bootloaders werden über so genannte „Fuse Bits“ vorgenommen.
In diesen Bits lässt sich auch der sogenannte „Watchdog“ aktivieren. Hierbei handelt es sich um einen speziellen Timer, der, sobald er abläuft, einen Reset auslöst. Der Watchdog-Timer wird in einem normal laufenden Programm regelmäßig zurückgesetzt. Sollte das Programm aufgrund eines unerwarteten Fehler einmal hängen bleiben, wird auch der Timer nicht zurückgesetzt und löst nach dem Ablaufen einen Reset aus, der den Mikrocontroller neu startet [22].

Für die Softwareentwicklung bietet der Hersteller mit dem „AVR Studio“ eine kostenlose Entwicklungsumgebung an. Mit ihrer Hilfe lassen sich für alle Mikrocontroller der AVR-Serie Programme entwickeln. Allerdings unterstützt das AVR Studio standardmäßig nur Assembler-Code. Um Software in einer Hochsprache wie C zu entwickeln, wird ein zusätzlicher Cross-Compiler benötigt.
Eine Besonderheit der AVR-Mikrocontroller ist ihre ISP (In-System-Programming)-Fähigkeit. Diese ermöglicht es, den Mikrocontroller zum Aufspielen neuer Programme in der Schaltung verbaut zu lassen. Die Programmierung findet über die SPI (Serial Peripheral Interface) oder JTAG Schnittstelle statt.
Zusätzlich kann neben dem eigentlichen Programm noch ein Bootloader im Programmspeicher untergebracht werden. Dieser wird, je nach Einstellungen in den „Fuse-Bits“ nach einem Reset vor dem eigentlichen Programm ausgeführt und erlaubt es, Programmcode aus einer anderen Quelle als den bereits genannten, nachzuladen [22].
Für den Fall, das die Kapazität oder Leistung des ATmega32 nicht mehr ausreichen sollte, bietet der Hersteller unter der Bezeichnung ATmega644 eine schnellere Version mit größerem Speicher an [23].

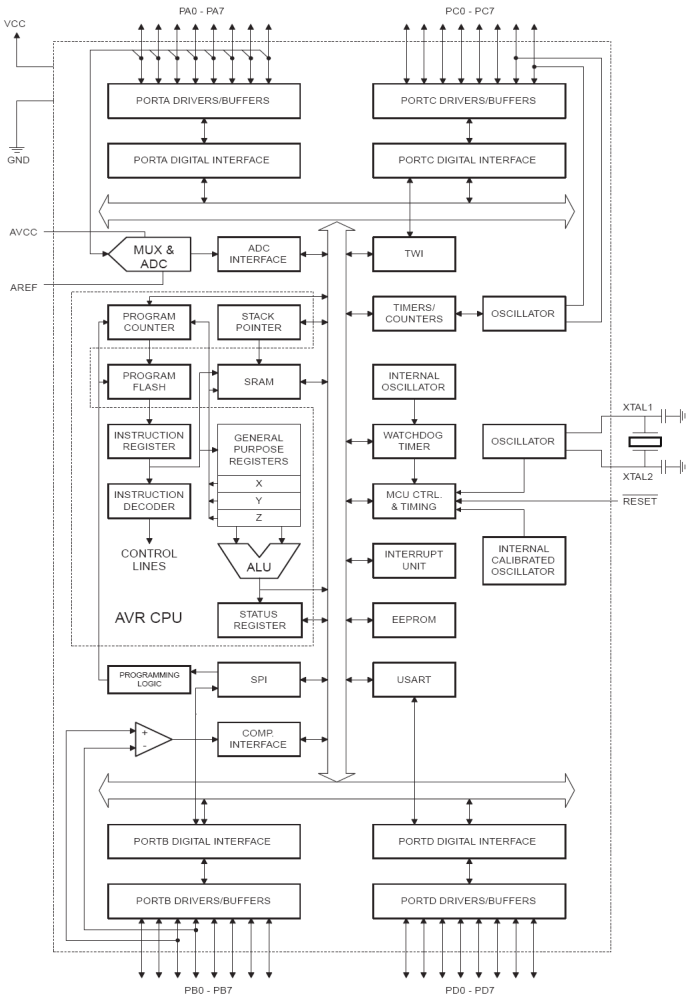


Abbildung 12: ATmega32 Blockdiagramm

3.4 Motortreiber

Zwischen Mikrocontroller und den beiden Antriebsmotoren sitzt ein IC mit der Bezeichnung L293D. Bei diesem handelt es sich um einen „Quadruple Half-H“-Treiber. Er dient zur dazu, die Motoren direkt von der Batterie mit Spannung zu versorgen. Dies ist nötig, da der Mikrocontroller nicht in der Lage ist, an seinen Ausgängen den für die Motoren benötigten Strom zu Verfügung zu stellen. Wie in Abb. 13 zu sehen ist, beinhaltet das IC 4 Halbbrücken, von denen jeweils zwei mit einem Motor verbunden sind. Durch diese Anordnung ist es möglich, Richtung und Geschwindigkeit beider Motoren unabhängig voneinander zu kontrollieren [40].

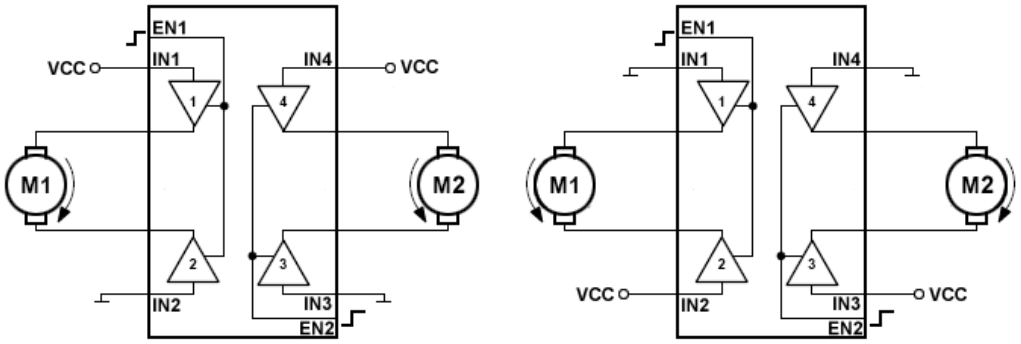


Abbildung 13: L293D Aufbau

Um die Drehrichtung der Motoren festzulegen, muss jeweils ein Pin der Paare IN1/IN2 und IN3/IN3 gegen Masse und einer gegen VCC geschaltet werden. Damit hierfür möglichst wenig Pins am Mikrocontroller belegen werden, wurde vor den Eingängen IN2 und IN4 ein invertierender Schmitt-Trigger geschaltet (Abb. 14). Dieser bewirkt, dass die besagten Pins immer den entgegengesetzten Zustand haben, wie ihre Partner IN1 und IN3, die jeweils mit einem Anschluss des Mikrocontrollers verbunden sind. Somit lässt sich mit nur 2 Pins die Drehrichtung beider Motoren regeln. Gleichzeitig wird sichergestellt, dass nie an beiden Pins einer der Treiberhälften die selbe Spannung anliegt. Durch das An- und Abschalten der Treiberhälften über die Pins EN1 und EN2 mit einer bestimmten Frequenz lässt sich anschließend die Geschwindigkeit der Motoren regeln. Dies geschieht durch ein vom Mikrocontroller generiertes PWM-Signal [18], [25].

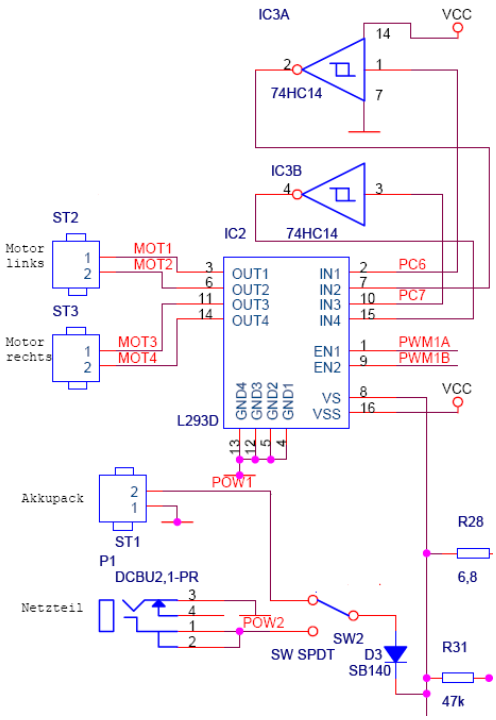


Abbildung 14: Motortreiber-Schaltung

3.5 Shiftregister

Zwischen dem Mikrocontroller und einigen Peripherie-Komponenten sind sogenannte Seriell-Parallel Wandler mit der Bezeichnung 74HC595 zwischengeschaltet.
Das 74HC595 ist ein 8-Bit Shiftregister mit parallelem Output. Das IC besitzt einen seriellen Eingang, welcher den anliegenden Wert bei jeder steigenden Flanke des SHCLK Pins übernimmt. Auf diese Weise können bis zu 8-Bit in dem IC zwischengespeichert werden. Sobald die gewünschten Daten in das Shiftregister geladen wurden, können sie über den RCLK Pin in das Outputregister und an die parallelen Ausgänge übernommen werden (Abb. 15) [18].

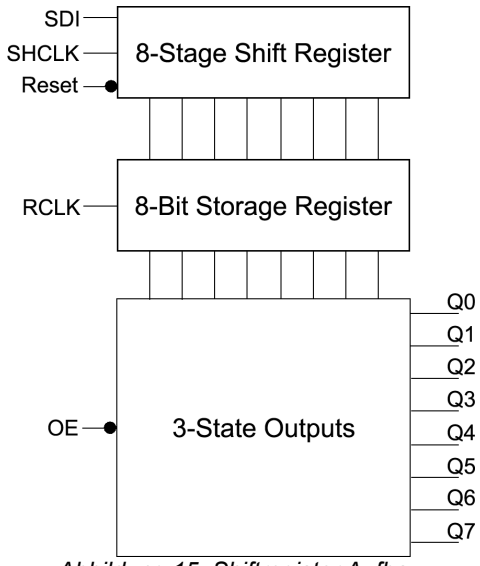


Abbildung 15: Shiftregister Aufbau

Da für die Steuerung aller externen Peripherie nicht genügend Pins am Controller zu Verfügung stehen würden, wurden einige der Komponenten, von denen keine Daten empfangen werden, an Shiftregister gehängt. Dies sind die 8 LEDs, das Display und die Signalleitungen mit denen die Sensoren an- und abgeschaltet werden können. Alle 3 Shiftregister werden über einen einzigen Pin (PC0) des Controllers mit Daten versorgt. Welches Register die Daten annimmt, wird mit Hilfe des SHCLK-Pins bestimmt. Dieses ist bei jedem IC mit einem anderen Pin am Controller verbunden (PC1, PC3, PC4). Eine Ausnahme bildet der Anschluss des Displays. Bei diesem werden zusätzlich zu den 8 Datenleitungen, die über das Shiftregister laufen noch 3 weitere Leitungen für Steuersignale benötigt [25] .
Abb. 16 zeigt einen Ausschnitt des Schaltplans, welcher die 3 Shiftregister und deren Peripherie enthält.

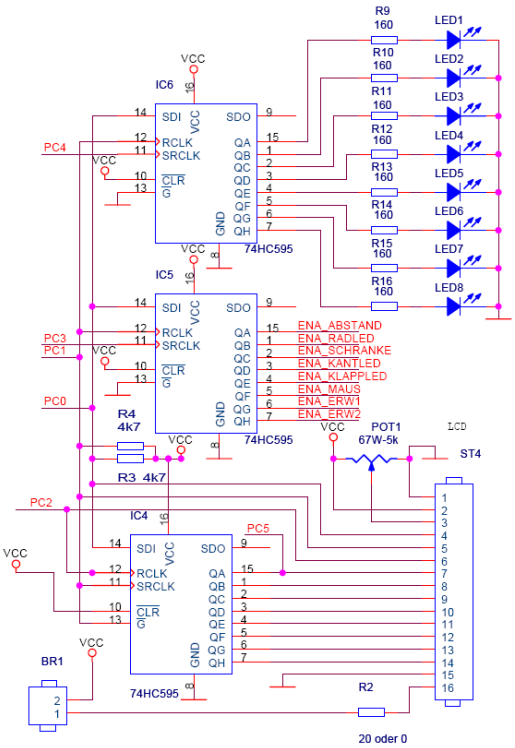


Abbildung 16: Shiftregister-Schaltung

3.6 LCD

Im c't-Bot kommt ein Punktmatrix-Display zum Einsatz. Diese Art von LCDs (Liquid Crystal Display) dienen zum Anzeigen von alphanumerischen Zeichen und besitzen meist einen fest vorgegebenen Zeichensatz. Der Name Punktmatrix-Display beschreibt zugleich die Art, in der Zeichen dargestellt werden. Jedes Zeichen wird auf dem Display in einem gleich großen Pixelfeld dargestellt, meistens 5x8 Pixel. Auf dem Großteil aller Punktmatrix-Displays findet sich als Controller ein IC, welches zu dem weit verbreiteten LCD-Controller HD44780 von Hitachi kompatibel ist [36]. Somit können fast alle Punktmatrix-LCDs mit gleicher Displaygröße untereinander ausgetauscht werden ohne aufwändige Änderungen an Hard- und Software vornehmen zu müssen. Neben den 8 Datenleitungen DB0-DB7 besitzen die Displays zur Steuerung die Anschlüsse E, R/W und RS. Dabei dient der Eingang E zum Aktivieren des Displays, R/W zur Festlegung der Übertragungsrichtung und RS signalisiert, ob es sich bei den anliegenden Daten um Befehle oder auszugebende Zeichen handelt. (Abb. 17)

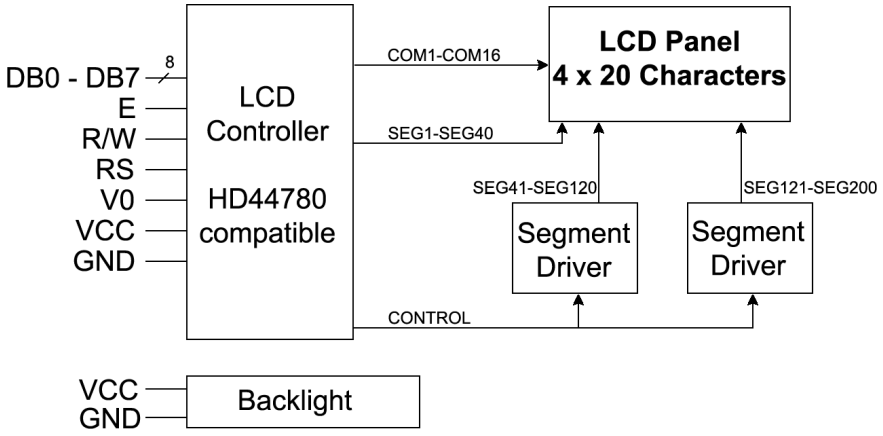


Abbildung 17: LCD-Aufbau

Im c't-Bot wird ein Display mit 4 Zeilen je 20 Zeichen eingesetzt. Die Datenleitungen des Displays sind über eines der bereits erwähnten Shiftregister mit dem Mikrocontroller verbunden. Da das Shiftregister allerdings nur in eine Richtung funktioniert, können nur Daten an das Display gesendet werden, nicht gelesen [41].

3.7 Sensoren

3.7.1 Distanzsensor

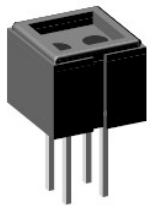


Abbildung 18: CNY70
(Quelle: www.vishay.com/doc?83751)

Im c't-Bot kommen zwei Distanzsensoren der Firma Sharp zum Einsatz. Bei diesen handelt es sich um Entfernungssensoren des Typs GP2D12 [35]. Diese sind Aufgrund ihrer einfachen Ansteuerung und der Tatsache, dass sie ohne zusätzliche externe Bauteile auskommen, besonders bei Hobby-Modellbauern sehr beliebt.
Ein GP2D12-Sensor besteht im wesentlichen aus einem Infrarot-Sender und -Empfänger, sowie ein Minimum an Elektronik zum Auswerten des Signals. Die Infrarot-LED des GP2D12 sendet einen modulierten Infrarotstrahl aus, der von Objekten reflektiert und von dem PSD-Chip (Position-Sensitive-Device) im Sensor empfangen wird. Abhängig von der Position, auf der das Licht auf dem PSD eintrifft, wird über die Elektronik ein analoges Ausgangssignal im Bereich von 2,5 bis 0,4V ausgegeben.
Der optimale Messbereich des Sensors liegt zwischen 10cm und 80cm. Wie in den Diagrammen 20 und 18 ersichtlich ist, sinkt die Spannung bei einer Distanz von weniger als 10cm stark ab.

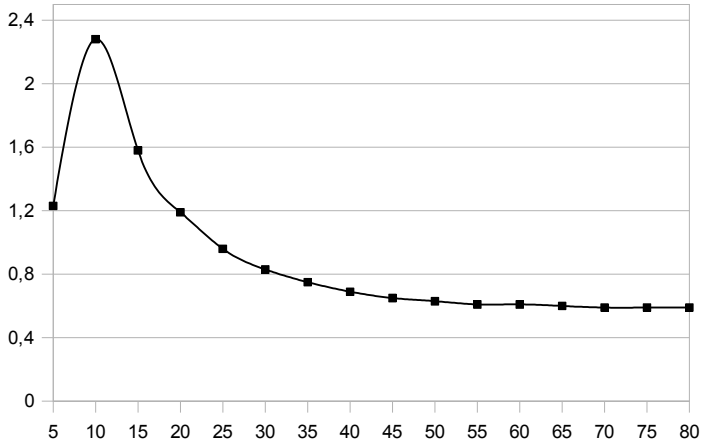


Abbildung 20: Gemessener Spannungsverlauf

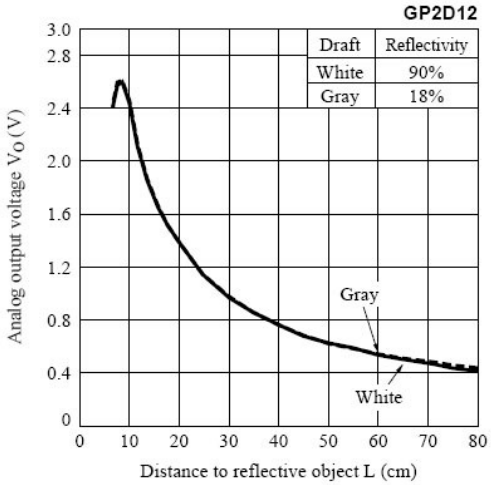


Abbildung 19: Optimaler Spannungsverlauf
(Quelle: GP2D12 Datenblatt)

Der GP2D12 ist zwar relativ zuverlässig und liefert selbst bei unterschiedlichen Materialien und Farben ein nahezu identisches Ausgangssignal. Da er allerdings im Infrarotbereich arbeitet, ist er extrem störanfällig gegenüber anderen Infrarotquellen wie z.b Sonneneinstrahlung oder starke Scheinwerfer [35],[38].

Im c't-Bot dienen dieses Sensoren zur Erkennung von Hindernissen, die sich vor dem Roboter befinden. Hierfür sind die Ausgänge der Sensoren direkt mit den Eingängen des Analog-Digital-Wandlers des Mikrocontrollers verbunden.

3.7.2 Reflexkoppler

Neben den Distanzsensoren besitzt der c't-Bot 7 weitere Sensoren, die nach einem ähnlichen Prinzip arbeiten.
Hierbei handelt es sich um sogenannte Reflexkoppler, oder Reflex Optokopplern des Typs CNY70. Sie bestehen aus einer Infrarot LED als Sender und einem Infrarot Fototransistor als Empfänger. Mit ihnen kann auf einer Distanz von wenigen Millimetern das reflektierte Licht der Infrarot LED gemessen werden (Abb. 21). Je dichter ein Objekt sich am Sensor befindet oder je besser seine Oberfläche reflektiert, desto mehr Strom fließt durch den Transistor [28].

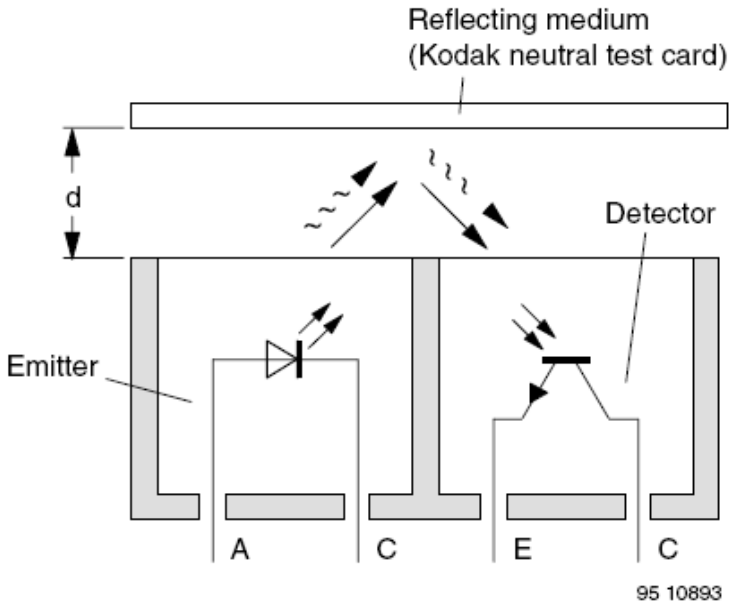


Abbildung 21: CNY70 Testaufbau
(Quelle: www.vishay.com/doc?83751)

Einige von den im c't-Bot verbauten Sensoren dienen als berührungslose Schalter, andere zur Erkennung des Untergrunds. Bei letzteren ist der Kollektor C des Transistors über einen Widerstand mit der Betriebsspannung VCC des Roboters verbunden. Zwischen Widerstand und dem Sensor wird mit Hilfe des Analog-Digital Wandlers des Mikrocontrollers die Spannung gemessen. Nährt sich der Sensor einer hellen bzw. gut reflektierenden Oberfläche, kann durch den Transistor ein höherer Strom fließen, was gleichzusetzen ist mit einem Absinken des Widerstandes des Transistors. Dies wiederum führt zur Messung einer niedrigeren Spannung.
Des weiteren wird diese Art der Sensoren als Radencoder, also zur Erkennung der Umdrehungszahl der Räder und der Messung der zurückgelegten Wegstrecke verwendet. An beiden Rädern befindet sich hierzu jeweils eine schwarz/weiße Indexscheibe. Diese können von den beiden CNY70 abgetastet werden sobald die Räder in Bewegung sind. Um ein sauberes Digitalsignal für den Microcontroller zu erhalten, wird der Kollektor Eingang des Transistors über einen Spannungsteiler mit einem invertierenden Schmitt-Trigger verbunden. Dieser schaltet ab einer Eingangsspannung von ca. 2,3V seinen Ausgang auf „Low“ und bei Unterschreiten von ca. 1,4V auf „High“.

3.7.3 Maussensor

Ein weiterer Sensor zur Abtastung des Untergrundes befindet sich unter dem Roboter. Der ADNS2610 der Firma Agilent, bzw. Avago, ist ein optischer Sensor zur Registrierung von 2D-Bewegungen [19]. Er besteht aus einem Bilderfassungssystem, sowie einem digitalen Signalprozessor (DSP) zur Verarbeitung der Bilddaten (Abb. 22). Das Bilderfassungssystem hat eine Auflösung von 18x18 Pixel, erkennt 63 unterschiedliche Graustufen und ist zusammen mit dem DSP in der Lage 1512 Bilder pro Sekunde zu verarbeiten und Geschwindigkeiten bis zu 30,48cm pro Sekunde zu erkennen.

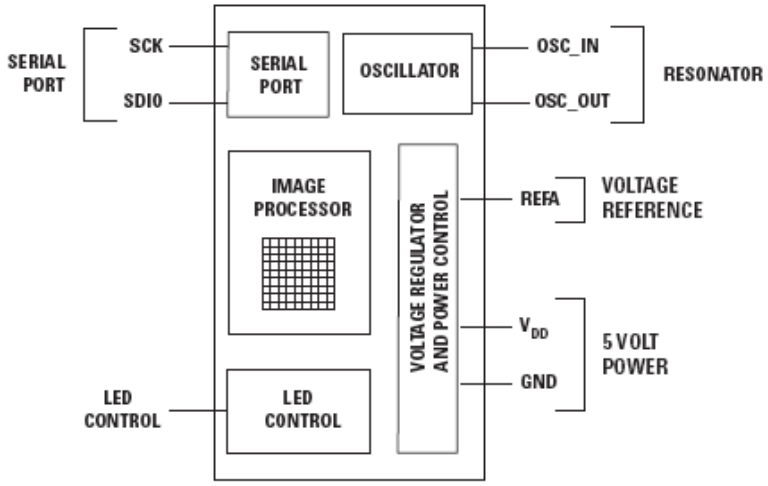


Abbildung 22: ADNS2610 Blockdiagramm
(Quelle: cp.literature.agilent.com/litweb/pdf/5988-9774EN.pdf)

Der Sensor kommuniziert über einen seriellen Zweidraht-Bus mit dem Mikrocontroller. Der Sensor übernimmt die Rolle eines passiven Clients und reagiert nur, wenn ein Befehl vom Mikrocontroller eintrifft. Dieser sendet Befehle immer nach dem selben Schema: Um Sensordaten auszulesen wird eine 0, gefolgt von einer 7-Bit langen Register-Adresse an den Sensor gesendet (Abb. 23). Nach dem letzten übertragenen Bit muss auf der Takt-Leitung für min. 100µS ein „High“-Pegel anliegen. Diese Zeit benötigt der Sensor um die Daten vorzubereiten. Anschließend wird bei jedem weiteren Takt vom Sensor ein Bit auf der SDIO Leitung ausgegeben, bis genau ein Byte übertragen wurde [19].

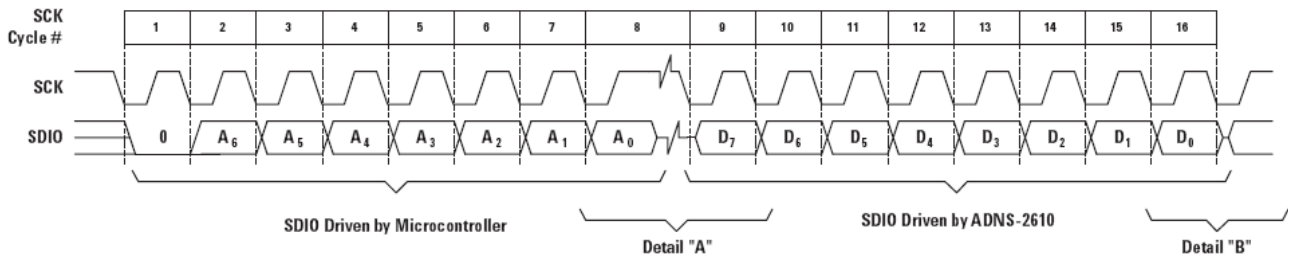


Abbildung 23: ADNS2610 Lese-Befehl
(Quelle: cp.literature.agilent.com/litweb/pdf/5988-9774EN.pdf)

Sollen Daten an den Sensor gesendet werden (z.b. zur Konfiguration) wird die 0 bei Beginn einer Übertragung durch eine 1 ersetzt. Anschließend folgt wieder die 7-Bit große Adresse und eine Pause von 100µS. Erst dann können die eigentlichen Daten übertragen werden (Abb. 24).

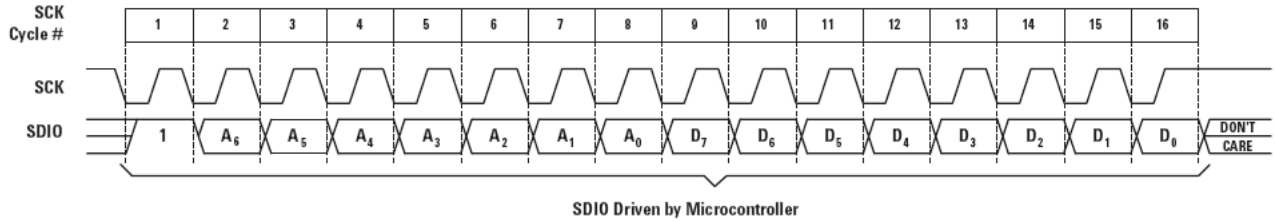


Abbildung 24: ADNS2160 Schreib-Befehl
(Quelle: cp.literature.agilent.com/litweb/pdf/5988-9774EN.pdf)

Folgende Adressen können im Sensor angesprochen werden:

Register	**Adresse**
Configuration	0x00
Status	0x01
Delta_X	0x02
Delta_Y	0x03
SQUAL	0x04
Maximum_Pixel	0x05
Minimum_Pixel	0x06
Pixel_Sum	0x07
Pixel Data	0x08
Shutter_Upper	0x09
Shutter_Lower	0x0A
Inverse_Product	0x11

Tabelle 2: ADNS2160 Register

Allerdings haben Schreibbefehle nur bei zwei der 12 Adressen eine Wirkung. Zum einen kann bei Adresse 0x00 die Konfiguration des Sensors eingestellt werden, zum anderen wird durch ein Schreiben an Adresse 0x08 das Übertragen eines aufgenommen Bildes von vorne begonnen [19].

Configuration
Dies ist eines der beiden Register, bei denen eine Schreiboperation möglichen ist.
Mit Hilfe des Configuration Registers kann der Sensor neu gestartet oder in den Standby Betrieb geschaltet werden.

Status
Das Status-Register liefert Informationen über die Produkt-ID (000beim ADNS-2610), sowie den aktuellen Betriebszustand (Standby oder Aktiv)

Delta_X und Delta_Y
Diese Register enthalten die zurückgelegte Anzahl an Pixeln in X-, bzw. Y-Richtung, seit dem letzten Lesevorgang. Die Daten liegen im 8-Bit 2er Komplement vor, d.h sie umfassen einen Bereich von -128 bis +127 Pixel.

SQUAL
Squal steht für „Surface Quality“ und beschreibt die Anzahl der Erkennungsmerkmale im aktuellen Bild, anhand derer eine Bewegung registriert wird. Die reale Anzahl der Merkmale ergibt sich aus dem Registerwert mal 2.

Maximum_Pixel und Minimum_Pixel
In diesen Registern ist der maximale bzw. minimale Grauwert enthalten, den ein Pixel auf dem aktuellen Bild hat. Da der Bildsensor nur 64 Graustufen wahrnehmen kann, enthalten diese Register jeweils nur ein 6-Bit Wert.

Pixel_Sum
Mit Hilfe dieses Registers ist es möglich den Durchschnittsgrauwert des aktuellen Bildes zu errechnen. Hierzu enthält es die oberen 8 Bit eines 15-Bit Wertes, der aus der Summe aller 324 Pixel gebildet wurde. Der Durchschnittswert kann mit Hilfe folgender Formel gebildet werden:

*Average Pixel = Register value * 128 / 324 = Register value * 0.395*

Pixel Data
Jedes mal wenn dieses Register ausgelesen wird, liefert es den Grauwert eines Pixels. Die oberen beiden Bits enthalten Informationen über den Beginn eines neuen Bildes (SOF-Bit), sowie die Gültigkeit der Daten (Data_Valid-Bit). Ein Schreiben in dieses Register bewirkt, dass wieder bei Pixel 1 begonnen und das SOF-Bit gesetzt wird. Mit jedem weiteren Lesebefehl für dieses Register wird das nächst höhere Pixel ausgelesen. Zu beachten ist hierbei, das die Pixel des Bildsensors unten links beginnen und senkrecht verlaufen.

Shutter_Upper und Shutter_Lower
Der Sensor regelt die Belichtungszeit nach jedem Frame nach um die aktuellen Pixelwerte in einem normalen Bereich zu halten. Diese beiden Register enthalten zusammen einen 16-Bit Wert, der die Belichtungszeit in Takten angibt. Pro Frame ändert sich der Wert um max. 1/16.

Inverse_Product
In diesem Register ist die Produkt ID als invertierter 4-Bit Wert enthalten.

3.7.4 Infrarotempfänger

Der c't-Bot besitzt einen Infrarotempfänger des Typs TSOP34836 [51]. Mit diesem ist es möglich, Infrarotsignale von Fernbedienungen zu empfangen, die im RC-5 Format dekodiert sind.
Der RC5 Code wurde ursprünglich von Philips entwickelt.
Er besteht aus insgesamt 14 Bit. 2 Startbits gefolgt von einem Togglebit das sich bei jedem Tastendruck ändert, 5 Adressbits und 6 Kommandobits (Abb. 25).

Die beiden Startbits kennzeichnen den Beginn einer gültigen Übertragung. Da durch die 6 Kommandobits nur max. 64 Codes möglich sind, wird bei neueren Geräten teilweise das zweite Startbit als invertiertes 7. Kommandobit verwendet. Dadurch erweitert sich der mögliche Befehlsbereich auf 127 Codes. Die Invertierung des Bits ist nötig, damit eine Kompatibilität mit älteren Geräten gewährleistet ist, da diese als zweites Startbit ebenfalls eine „1“ erwarten. Für die Kommandos 0-63 ist das zweite Startbit somit „1“, für die erweiterten Kommandos 64-127 hingegen ist es „0“.
Das Togglebit wechselt bei jeden Tastendruck zwischen 0 und 1. Hierdurch kann unterschieden werden, ob eine Taste mehrfach hintereinander oder länger gedrückt wird.
Da mit einer Fernbedienung ggf. mehrere Geräte angesteuert werden können, wird noch eine Geräteadresse mitgesendet. Über die 5 Adressbits können insgesamt 32 unterschiedliche Geräte angesteuert werden.

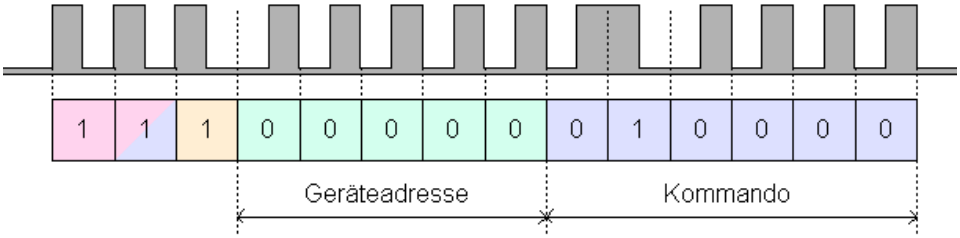


Abbildung 25: RC5-Kommando mit entsprechendem Bitverlauf
(Quelle: www.roboternetz.de/wissen/index.php/RC5-Code)

Ein Bit besteht immer aus zwei Teilen, einem das “High” ist und einem das “Low” ist. Ist der erste Teil des Bits „High“ und der zweite „Low“ wird dies als 1 interpretiert, im anderen Fall – erst „Low“, dann „High“ - als 0. Diese Codierung wird als 2-Phasen-Codierung, oder Manchester-Codierung bezeichnet (Abb. 26).
Beim Übertragen der Daten wird das Signal zusätzlich noch mit 36khz moduliert. Dies dient zur weiteren Übertragungssicherheit und zum Unterdrücken von Störungen.

Ein Halbbit besteht nach der Modulation aus 32 Impulsen und hat eine Dauer von 888,9µs. Ein ganzes Bit also 1,778ms. Für die komplette Übertragung eines Befehls werden somit 24,889ms benötigt. Wird eine Taste länger gedrückt gehalten, wiederholt sich der Befehl alle 114ms [39],[29].

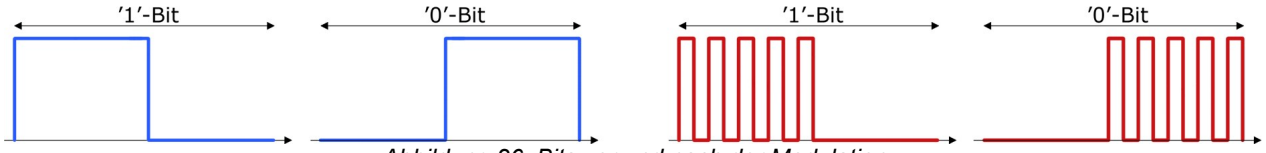


Abbildung 26: Bits vor und nach der Modulation

Der TSOP34836 besteht neben der Empfangseinheit aus einem Demodulator. Dieser wandelt das noch modulierte Signal in ein, für den Mikrocontroller, kompatibles Signal um (Abb. 27).

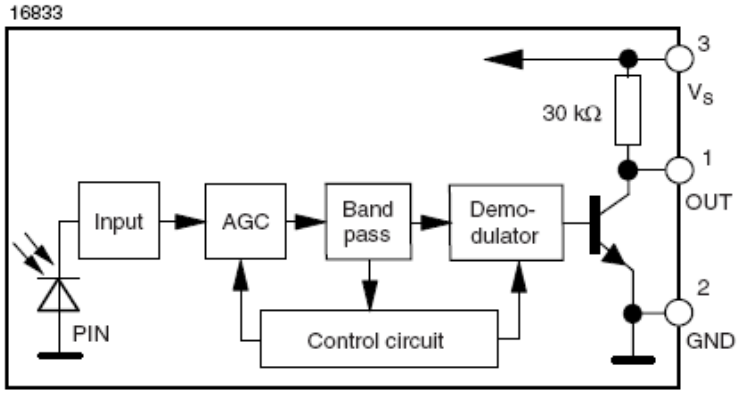


Abbildung 27: Blockschaltbild TSOP 34836
(Quelle: www.vishay.com/docs/81732/tsop348.pdf)

Da der Demodulator, wie im oberen Bild zu sehen ist, bei eintreffenden Infrarotdaten, das Ausgangssignal auf „Low“ zieht, liegen die Daten am OUT-Pin invertiert an. D.h. aus dem „High“-Teil eines Bits, wird ein „Low“-Teil und umgekehrt. Ebenso muss bei der Implementierung beachtet werden, das am OUT-Pin ein „High“-Pegel anliegt, wenn keine Daten empfangen werden [51].

4. Softwareanalyse

4.1 Entwicklungsumgebung

Zur Entwicklung der Software für den Mikrocontroller ist ein „Cross-Compiler“ nötig, welcher aus dem Quellcode einen, auf dem Mikrocontroller lauffähigen, Maschinencode generiert. Für Programme die in Assembler geschrieben wurden, stehen für Windows neben der „All-In-One“-Lösung von Atmel, die neben einem Compiler gleich eine Entwicklungsumgebung und einen Simulator mitbringt, auch diverse kostenlose Projekte zur Verfügung. Für die Hochsprache C existieren neben einigen kommerziellen Compilern unter anderem der Open Source Compiler gnu-gcc, welcher für Windows unter dem Namen WinAVR bekannt ist. Damit allerdings effektiv gearbeitet werden kann, wird zusätzlich noch eine passende Entwicklungsumgebung benötigt. Für Windows bietet sich hier wieder Atmels AVR-Studio an. Sollte auf dem PC der WinAVR-Compiler installiert sein, wird dieser automatisch mit ins AVR-Studio eingebunden und erlaubt die Kompilierung von C-Code direkt aus dem Programm. Für Entwickler, welche auch andere Programmiersprachen benötigen, eignet sich das Programm Eclispe. Hierbei handelt es sich um eine Entwicklungsumgebung, die ursprünglich für die Entwicklung von Java-Programmen vorgesehen war. Durch zahlreiche Plugins, wie z.B. das CDT-Plugin (C/C++ Development Tools), lässt es sich jedoch für eine Vielzahl von Programmiersprachen erweitern. Durch diese Flexibilität und der Tatsache, dass es sowohl auf Windows wie auch auf Linux und MacOS läuft, ist es eines der am weit verbreitetsten Entwicklungsumgebungen für Mikrocontroller. Die nötige Konfiguration des avr-gcc Compilers kann wahlweise per Hand oder mit Hilfe eines weiteren Plugins wie „CDT AVR Plugin“ oder „CDT AVRGCC“ vorgenommen werden.
Da das AVR-Studio ausschließlich für Windows-Systeme entwickelt wird, stehen für die Softwareentwicklung unter Linux nur kommerzielle oder Open Source Projekte zur Verfügung. Am häufigsten trifft man hier Eclipse mit CDT-Plugin an. Ein weitere, allerdings kostenpflichtige, Umgebung ist „CrossWorks“ von der Firma Rowley Associates [48]. Es enthält neben einem eigenen C-Compiler einen Simulator und die nötige Software, um die Programme auf den Mikrocontroller zu laden und zu Debuggen.
Neben den Compilern für Assembler und C gibt es noch einige, die es erlauben Programme in den Sprachen Basic, Pascal und Java zu schreiben.

4.2 Die AKSEN-Bibliothek

Passend zu dem AKSEN-Board (Abb. 28) stellt die FH-Brandenburg eine Softwarebibliothek zur Verfügung, welche die Arbeit mit dem Board erleichtern soll. Da die Bibliothek des c't-Bots in Bezug auf ihren Aufbau und Funktionsumfang, an die des AKSEN-Boards angelehnt sein soll, ist es nötig, diese genau zu untersuchen. Da die Bibliothek nur in vorkompilierter Form veröffentlicht wird und somit kein Quellcode zur Verfügung steht, kann die Analyse nur mit Hilfe des Handbuchs stattfinden [20].

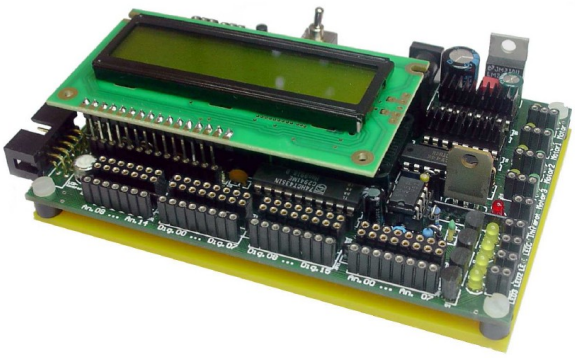


Abbildung 28: AKSEN-Board

Die Bibliothek abstrahiert den Zugriff auf Sensoren, Aktoren und I/O-Pins des Controllers soweit, dass man ohne größeres Wissen über die Hardware mit dieser arbeiten kann.
Für den Zugriff auf die digitalen Ein- und Ausgänge des Boards stehen Funktionen wie `digital_bytein`, `digital_byteout`, `digital_in` und `digital_out` zur Verfügung. Mit den Funktionen `digital_in` und `digital_out` ist es möglich, einzelne Pins eines Ports zu abzufragen und zu setzen. Zusätzlich lassen sich durch die Funktionen `digital_bytein` und `digital_byteout` jeweils 8-Bit eines Ports zugleich auslesen, bzw. ändern. Über einen im Mikrocontroller integrierten Analog-Digital-Wandler lassen sich 15 unterschiedliche Spannungen zwischen 0V und 5V erfassen. Die hierfür zuständige Funktion `analog` übersetzt die anliegende Spannung der übergebenen Portadresse in einen Wert zwischen 0 (0V) und 255 (5V).
Des weiteren befinden sich 4 Anschlüsse mit LEDs auf dem Board. Diese lassen sich ähnlich wie die digitalen Ausgänge ansteuern. Über die Funktion `led` können die Lampen beliebig aus- und einschaltet werden. Als Parameter wird die Nummer der LED und deren neuer Zustand übergeben.
Eine Funktionalität, welche nicht in die c't-Bot Bibliothek übernommen werden braucht, ist die Ansteuerung einer Infrarotdiode. Der c't-Bot ist hardwareseitig nur für das Empfangen von Infrarotsignalen ausgelegt, nicht jedoch für das Senden.

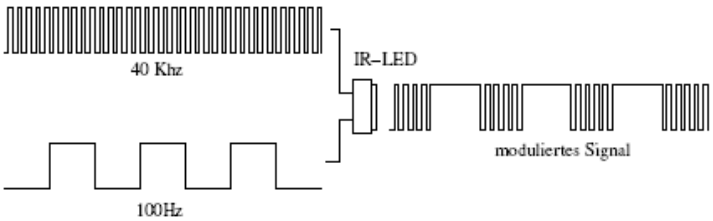


Abbildung 29: IR-Modulierung beim AKSEN-Board

Die Infrarotdiode des AKSEN-Boards kann auf zwei unterschiedliche Arten angesteuert werden. Zum einen ist es möglich, ein einfaches Signal mit einer Frequenz von 40khz zu versenden. Dieses wird durch die Funktion `mod_ir_an` gestartet und `mod_ir_aus` gestoppt. Zum anderen lässt sich das Signal noch mit einer Rechteckschwingung überlagern, wie in Abb. 29 dargestellt. Hierzu wird der Funktion `setze_ir_senden` ein Wert übergeben, welcher der halben Periodendauer

der gewünschten Frequenz entspricht.
Der Empfang von Infrarotsignalen erfolgt über das Zählen der Perioden.
Mit Hilfe der Funktionen `mod_irX_takt` und `mod_irX_maxfehler`, bei denen das X im Namen für die Nummer des Empfängers steht, lässt sich für jeden der vier Eingänge die erwartete Frequenz und die maximale Abweichung pro Periode festlegen.
Eine weitere Komponente des AKSEN-Boards, welche nicht im c't-Bot zu finden ist, ist ein vier-poliger DIP-Schalter, dessen Stellung mit Hilfe von zwei Funktionen abgefragt werden kann. Durch seine bis zu 16 verschiedenen Stellungen, kann er u.a. zum Einstellen von Betriebsarten verwenden werden.

Zur Fortbewegung oder anderen mechanischen Aktionen lassen sich bis zu vier Gleichstrommotoren ansteuern. Mit dem Befehl `motor_richtung` lässt sich die Drehrichtung eines Motors festlegen. Wird der Funktion als zweiter Parameter eine 0 übergeben, läuft der Motor rückwärts, bei einer 1 vorwärts. Die Geschwindigkeit ist durch `motor_pwm` regelbar. Die Abstufung erfolgt in 11 Schritten von 0(aus) bis 10 (max. Geschwindigkeit). Bei beiden Befehlen wird als erster Parameter die Portadresse des Motors angegeben.
Sowohl das AKSEN-Board, als auch der c't-Bot besitzen Anschlüsse für Servos. Sind es beim c't-Bot jedoch nur maximal 2, so können am AKSEN-Board bis zu 11 Servos angeschlossen werden. Allerdings stehen nur 3 dedizierte Servoausgänge zur Verfügung. Um alle 11 nutzen zu können, müssen Teile der digitalen Ein- und Ausgänge umgeschaltet werden. Dies geschieht über die Funktion `servobank_start`, die die Ports mit Hilfe der, für die Servos zuständigen, ISR ansteuert. Allerdings ist keine Nutzung der Anschlüsse als digitale Ein- oder Ausgänge möglich, so lange die Servobank aktiviert ist.
Um die Anschlüsse normal nutzen zu können, kann die Servobank jederzeit durch den Befehl `servobank_stop` deaktiviert werden. Für die Ansteuerung von Servos sind die Befehle `servo` und `servobank` verantwortlich. Letzterer ist jedoch ausschließlich für die Steuerung der Servos zuständig, welche sich auf der Servobank befinden. Als Parameter erwarten beide Funktionen die Nummer des Servos und eine Winkelangabe in Mikrosekunden. Zusätzlich steht für die drei Standard-Servoanschlüsse noch `servo_arc` zur Verfügung. Dieser Funktion wird neben der Portnummer des Servos noch der Winkel übergeben, auf den sich der Servo einstellen soll. Es lässt sich auf diese Weise jedoch nur ein Winkel von 0° bis 90° anfahren.
Das AKSEN-Board wie auch der c't-Bot besitzt Anschlüsse für spezielle Encoder.
Anders als beim c't-Bot dienen diese am AKSEN-Board allerdings nicht zum Registrieren von Radumdrehungen, sondern zählen negative Flanken, also Pegelübergänge von „High“ nach „Low“, von angeschlossen Geräten. Es lassen sich so bis zu 3 unterschiedliche Signale abtasten. Über die Befehle `encoder0` - `encoder2` wird die jeweilige Anzahl an negativen Flanken seit dem letzten Aufruf der Funktion abgefragt.
Des weiteren existiert ein CAN-Interface, welcher für den c't-Bot jedoch nicht relevant ist, da keine passende Hardware vorhanden ist. Zum Senden und Empfangen stehen die Funktionen `CanEmpfang` und `CanSenden` zur Verfügung. Beide erwarten als einzigen Parameter einen Zeiger auf eine Datenstruktur vom Typ CAN_MSG.

Ebenfalls auf beiden Boards ist ein LC-Display (LCD) vorhanden. Auf dem Display des AKSEN-Boards kann an jeder beliebigen Stelle geschrieben werden, hierfür wird der Cursor mit der Funktion `lcd_setxy` an die gewünschte Position verschoben. Anschließend wird mit Hilfe von `lcd_puts` eine Zeichenkette ausgegeben. Selbstverständlich stehen auch Funktionen zur Konfiguration sowie zum Löschen des Displays zur Verfügung.

Die serielle Schnittstelle des AKSEN-Boards ist fest auf 9600Baud, 8 Datenbits, kein Paritätsbit und ein Stoppbit eingestellt. Hier besteht nur die Möglichkeit ein Zeichen (`serielle_putchar`) oder String (`serielle_puts`) zu senden, bzw. zu empfangen. Die beiden Funktionen `serielle_gets` und `serielle_getchar` zum Lesen arbeiten zudem blockierend, d.h. sie liefern erst ein Ergebnis, wenn ein Zeichen oder String empfangen wurde. Das Ende eines Strings wird durch die Zeichen 0x00, 0x0A (LF) oder 0x0D (CR) gekennzeichnet.
Die Besonderheit an der AKSEN-Bibliothek ist, dass sie einen simplen Multitasking Kernel besitzt. Dieser arbeitet mit einem Round-Robin Algorithmus, jedoch ohne Prioritäten. Somit wird jeder neue Prozess am Ende der Prozess-Liste eingereiht. Das System ist in der Lage insgesamt bis zu 20 Prozesse zu verwalten. Beim Erzeugen eines Prozesses wird dem System sowohl der Name der aufzurufenden Funktion als auch die Zeit, die der Prozess die CPU an einem Stück benötigt, mitgeteilt. Nach dem Anlegen liefert die Funktion eine Prozess-ID zurück. Diese ist notwendig um einzelne Prozesse zur Laufzeit identifizieren und wieder aus der Liste entfernen zu können. Die Prozesszeit kann nachträglich mit Hilfe der Funktion `process_set_ticks` neu festgelegt werden. Des weiteren ist es möglich, dass einzelne Prozesse ihre Zeitscheibe vorzeitig abgeben bzw. die Abgabe verzögern.
Um exklusiven Zugriff auf Hardwareressourcen zu gewährleisten, stehen in der Bibliothek keine Funktionen zur Verfügung. Jedoch wird in einem Beispiel gezeigt, wie man ohne viel Aufwand eigene Semaphoren erstellt kann.
Zusätzlich verfügt die Bibliothek über einen separaten Timer welcher im Hundertstelsekundentakt zählt und zur Erstellung einer einfachen Zeitsteuerung verwendet werden kann. Er lässt sich über die Funktionen `akt_time` abfragen und durch `clear_time` zurücksetzen. Ob dieses Feature jedoch in der Bibliothek des c't-Bots realisiert werden kann, muss sich erst noch herausstellen. Es ist möglich, das bereits alle Timer des Mikrocontrollers von notwendigeren Funktionen belegt werden.

4.3 Das c't-Bot Framework

Mit Hilfe des offiziellen c't-Bot Frameworks von Heise bekommt man eine Vorlage in die Hand, die es ermöglicht, verschiedene Verhalten für den Roboter entwickeln. Alle erstellten Verhalten werden zusammen mit einigen bereits vordefinierten in eine Liste eingetragen. Eine Prioritätszahl zwischen 0 und 255 legt hierbei die Position in der Liste fest. Diese Liste wird anschließend der Reihe nach abgearbeitet.
Ein Verhalten besteht aus einer Funktion, die nicht unterbrochen wird. Allerdings kann man von einem Verhalten aus ein anderes aufrufen, welches das vorherige deaktiviert. Der Grundgedanke hinter diesem System ist die Subsumption-Architektur mit der Eigenschaft, komplexere Verhalten in mehrere einfachere aufzusplitten.
Seit im Februar 2008 das Stable-Release Nr. 14 veröffentlicht wurde, enthält das Framework zudem ein einfaches Multitasking-System, dass es ermöglicht, zusätzlich zu den Verhalten parallel ablaufende Prozesse zu erstellen.
Es ist eine Anzahl bereits vordefinierter Verhalten vorgegeben, die den Einstieg und die Entwicklung eigener Abläufe erleichtern sollen. So gibt es u.a. folgende Verhalten: Einer Linie folgen, Gegenstände einsammeln, an einer Wand entlang, oder einfach nur geradeaus fahren. Somit muss sich nicht mehr um grundlegende Dinge wie die Ansteuerung der einzelnen Aktoren und Sensoren gekümmert werden. Wenn man z.B. möchte, dass sich der Bot dreht oder eine gewisse Strecke geradeaus fährt, ruft man einfach das entsprechende Verhalten auf. Alle Sensoren werden periodisch abgefragt und die Werte in globalen Variablen gespeichert. Diese können vom Benutzer einfach ausgelesen werden. Ein simples Verhalten könnte wie folgt aussehen:

```
1  static uint8 simple_state = 0;
2  void bot simple behaviour(Behaviour t *data) {
3      switch (simple_state) {
4          case 0:
5              bot_drive_distance(data, 0, BOT_SPEED_MAX, 14);
6              simple_state = 1;
7          break;
8          case 1:
9              bot_turn(data, 90);
10             simple_state = 0;
11         break;
12         default:
13             return_from_behaviour(data);
14         break;
15     }
16 }
```

Tabelle 3: Einfaches Verhalten aus behaviour_simple.c

Mit diesem Verhalten fährt der Roboter mit voller Geschwindigkeit 14cm vorwärts, dreht sich anschließend um 90° und beginnt wieder von vorne.

4.4 Mögliche Realtime-Kernel

Um auf dem Prozessor mehrere Prozesse parallel bearbeiten zu können ist es nötig, ein Multitasking-System zu implementieren [6],[16].
Da sich aufgrund der geringen Programm- und Speichergröße die Implementierung eines Multitasking-Systems für den AVR als aufwändig erweisen könnte, empfiehlt es sich, bereits vorhandene Realtime-Kernel auf ihre Verwendbarkeit hin zu überprüfen.
Nach einer Vorauswahl stehen die Systeme „FreeRTOS“, „femto OS“ und „picoOS“ zur Wahl. Alle diese Systeme arbeiten nach einem ähnlichen Prinzip [49].
Für jeden Prozess, den der Benutzer anlegt, wird eine bestimmte Menge Speicher im RAM reserviert. Dieser dient dazu, den Stack und einige Informationen über den Prozess zu speichern. Ein Prozesswechsel erfolgt nach Ablauf der dem Prozess zugewiesenen Zeit oder wenn dieser selbständig einen Wechsel veranlasst. Die dem Prozess zur Verfügung stehenden Zeit beträgt normalerweise einige Millisekunden und ist in den meisten Systemen fest eingestellt. Bei einigen Systemen kann sie jedoch auch individuell für jeden Prozess vergeben werden. Der eigentliche Wechsel findet in der Interruptroutine eines Timers statt.

FreeRTOS ist eines der am meist genutzten und verbreitetsten Realtime-Kernel. Es ist für verschiedenste Mikrocontroller verfügbar. So läuft dieses, laut Entwickler-Website, mit entsprechender Portierung auf 8-Bit AVRs genau so wie auf einem 32-Bit Prozessor der AT91 Serie von Atmel oder der PIC Serie von Microchip. Ebenfalls unterstützt werden in FPGAs eingebettete Soft- und Hardcores wie Virtex4 FPGAs mit Microblaze Kern oder PowerPCs. FreeRTOS ist als Open Source unter einer abgeänderten GPL-Lizenz sowie mit einer kommerzielle Lizenz zu erhalten.
Der Kernel bietet in der aktuellen Version 5.1.1 neben einer unbegrenzten Anzahl an Prozessen noch die Möglichkeit sogenannte „Co-Routinen“ zu erstellen. Diese Routinen benötigen im Gegensatz zu einzelnen Prozessen keinen eigenen Stack sondern teilen sich einen, wodurch sie allerdings in Bezug auf Ihren Funktionsumfang sehr eingeschränkt sind. Des weiteren besitzt das System eine „Stack overflow detection“, mit der es möglich ist einen Stacküberlauf zu erkennen. Bedauerlicherweise ist eine Überprüfung nur bei einem Prozesswechsel möglich, mit der Folge, dass ein Überlauf bereits stattgefunden haben kann. Jedem Prozess kann eine individuelle Priorität zu geordnet werden. Prozesse mit der selben Priorität werden anschließen der Reihe nach abgearbeitet.
Um einzelne Programmbereiche zu schützen besteht die Möglichkeit, Mutexe oder Semaphoren einzusetzen [33].
In der Praxis erweist sich die Anwendung des FreeRTOS Kernels jedoch als nicht besonders benutzerfreundlich, was unter anderem daran liegt, dass der vom c't-Bot verwendete ATMega32 noch nicht offiziell unterstützt wird. Ein weiteres Hindernis auf das man stößt, ist die Zeit zwischen zwei Timer-Interruptaufrufen. Hier lässt das System Abstände von unter 1ms nicht ohne weiteres zu. Ein deutlicher Nachteil dieses Systems ist der benötigte Speicher. Es verbraucht mit 4 Task Prioritäten, 2 Co-Routine Prioritäten, einer minimalen Stacksize von 85Byte, Funktionen zum Verzögern und Löschen von Prozessen, sowie 1kB Heap-Size bereits knapp 30% Programmspeicher und 61% RAM, wobei der größte Teil hiervon allerdings bereits für den Heap vorgesehen ist.

Femto OS ist ein Realtime-Kernel, welcher überwiegend für Controller mit wenig Programm- und Datenspeicher ausgelegt ist. Die aktuelle Version 0.86 besitzt ein prioritätenbasiertes Round-Robin Scheduling-Verfahren, bei dem für jeden Prozess festgelegt werden kann, ob dieser kooperativ oder preemptiv behandelt werden soll. Zudem bietet es die Möglichkeit, für jeden Prozess festzulegen, welche Register bei einem Prozesswechsel gesichert werden sollen. Da die zu sichernden Register jedoch bei Programmstart bereits festgelegt sein müssen, setzt dies voraus, dass die Software entweder in Assembler geschrieben wurde, oder der Entwickler sich den Compiler-Output ansieht, um die Register erfahren. Darüber hinaus lässt dieser Kernel einen bevorstehenden Stack-Overflow erkennen und ermöglicht, diesen rechtzeitig zu vermeiden. Da die komplette Konfiguration per *defines* vorgenommen wird und auch die einzelnen Prozesse vor dem Programmstart bekannt sein müssen, eignet es sich nur bedingt für den Einsatz in der Bibliothek, obwohl es durch den Verbrauch von nur 9.1 % Programmspeicher und knapp 4 % Datenspeicher mit angenehm wenig Ressourcen auskommt [32].

PicoOS überrascht zu Beginn durch seine geringe Anzahl an Dateien. Es kommt mit lediglich 9 Dateien aus. Dennoch kann es sich sehen lassen. Zur Auswahl stehen zwei Scheduling-Algorithmen, zum Einen ein Standard Prioritäten basiertes und zum Anderen ein Round-Robin mit Prioritäten. Es kann auf 8-Bit Mikrocontroller bis zu 64 Prozesse verwalten, die in maximal 8 Prioritäten unterteilt werden können.
Zudem bietet es die Möglichkeit Mutexe, Semaphoren und eine unbegrenzte Anzahl an Events zu erstellen. Mit Hilfe dieser Events ist es möglich, virtuelle Interrupts auszulösen. Des weiteren bietet dieses System atomare Funktionen zum Zugriff auf Variablen an.
Von Nachteil ist jedoch, dass der Speicher, wie bei fast allen anderen Systemen auch, bereits beim Programmstart belegt wird. Zudem wird bei der Standard-Konfiguration für max. 8 Prozesse und 8 Prioritäten bereits 15.6 % Programmspeicher und knapp 60% Datenspeicher belegt. Somit eignet sich auch dieses System nur begrenzt als Kernel in der Bibliothek [45].

Die Hauptnachteil fast aller Systeme ist ihr statischer Ressourcenverbrauch, der von Programmstart bis Programmende gleich bleibt. Zudem werden viele Funktionen, die diese Systeme bieten, nicht benötigt und wären unnötiger Ballast. Um die Bibliothek möglichst effektiv zu halten, scheint es somit unumgänglich einen eigenen Realtime-Kernel zu entwickeln, welcher genau auf die Bedürfnisse angepasst ist. Hierzu zählt neben der Möglichkeit, beliebig viele Prozesse zur Laufzeit des Programms anlegen zu können, auch, einzelne Prozesse zu unterbrechen oder für eine bestimmte Zeit anzuhalten. Um das Multitasking-System übersichtlich zu halten, sollte es mit einem einfachen Scheduler ausgestattet sein, welcher die Prozesse nach dem Round-Robin Verfahren und ohne Beachtung von Prioritäten bearbeitet. Des weiteren sollte die Möglichkeit der Interaktion zwischen Prozessen gewährleistet werden, so dass diese in der Lage sind, sich gegenseitig zu manipulieren oder untereinander Daten auszutauschen.

5. Implementierung

5.1 Grundaufbau

5.1.1 Entwicklungsumgebung

Als Entwicklungsumgebung wurde Eclipse mit den bereits in Kapitel 4.1 beschriebenen Erweiterungen gewählt. Da bereits in anderen Bereichen des Studiums mit dieser Entwicklungsumgebung gearbeitet wurde, ist sie bekannt und es wird kaum Zeit für die weitere Einarbeitung benötigt. Zusammen mit der CDT AVRGCC Erweiterung lässt sich der c't-Bot nach dem Kompilieren direkt aus der Umgebung heraus programmieren, ein extra Tool wird somit nicht benötigt.

5.1.2 Systemaufbau

Während der Entwicklung werden die einzelnen Komponenten des Systems soweit wie möglich unabhängig von einander gehalten. Dies dient zum einen einer bessere Übersicht, zum anderen jedoch erleichtert es ein eventuelles späteres Austauschen einer Komponente erheblich. Das System lässt sich grob in folgende Komponenten aufteilen: Systemtimer, Multitasking-System, Analog und Digitalsensoren, Maussensor, RC5-Dekodierung, LCD, Motoren, Servos, UART und Erweiterungsboard-Kommunikation.
Der Kern besteht aus einer Funktion, welche in regelmäßigen Abständen aufgerufen wird. In dieser werden alle Aktionen ausgeführt, welche nötig sind, um das System am Laufen zu halten. Neben dem Scheduler für das Multitasking sind es vor allem einige zeitkritische Sensoren, deren Abfrage hier untergebracht wird.
Für eine Funktion, die in gleichmäßigen Abständen aufgerufen werden soll, kommt in einem Fall wie diesem nur die Interrupt-Service-Routine (ISR) einer der Timer des Mikrocontrollers in Frage. Jeder der drei Timer bietet zwei verschiedene Arten von Interrupts an. Wahlweise kann bei einem Timer-Overflow oder beim Erreichen eines bestimmten Timer-Wertes, einem „Compare-Match“, ein Interrupt ausgelöst werden. Da die Funktion höchst wahrscheinlich in geraden Zeitabständen wie 100µS, 500µS oder 1mS aufgerufen werden soll, muss ein Timer gewählt werden, der sich so einstellen lässt, dass mit einem seiner Zählerwerte der gewünschte Zeitabstand erreicht werden kann. Würden die Timer mit voller Geschwindigkeit laufen, also 16Mhz, würde er sich alle 62,5ns um Eins erhöhen. Um nun auf eine Zeit von 100µS zu kommen müsste der Zähler in der Lage sein, einen Wert von 1600 zu erreichen. Dies ist jedoch nur bei einem der drei Timer möglich. Da jedoch noch nicht feststeht, ob dieser für andere Aufgaben benötigt wird, muss eine alternative Lösung gefunden werden. Um die Zählrate eines Timers zu verlangsamen, kann er mit einem Prescaler versehen werden, welcher den Systemtakt teilt bevor er in den Timer eingespeist wird. Der niedrigste Prescaler mit dem ein Timer versehen werden kann, ist 8x. Anstatt der 16Mhz würde der entsprechende Timer also nur mit 2Mhz betrieben werden, was einer Verachtfachung der Schrittdauer entspricht. Durch die nun enthaltene Schrittweite von 500ns lässt sich auch mit den 8-Bit Timern ein ausreichend großer Bereich abdecken. Für eine Zeit von 100µS wird nun lediglich ein Timerwert von 200 benötigt. Für größere Zeitabstände würde ein entsprechend höherer Prescaler, wie beispielsweise 64x, verwendet werden müssen.

5.2 Multitasking

5.2.1 Aufbau

Der Aufbau des Multitasking Systems ist ähnlich der im Kapitel „Mögliche Realtime-Kernel“ beschriebenen Systeme. Jedem Prozess wird beim Anlegen ein Bereich im RAM zugewiesen welcher als Stack dienen wird. Bei einem Wechsel zwischen zwei Prozessen wird der Stackpointer des Mikrocontrollers so abgeändert, dass er auf den Speicher des jeweils aktuellen Prozesses zeigt. Zudem wird jedem Prozess beim erstellen eine eindeutige ID zugewiesen. Mit ihrer Hilfe ist es dem System möglich den Prozess zu identifizieren. Alle erstellten Prozesse werden in einer einfach verketteten Liste gespeichert (Abb. 30).

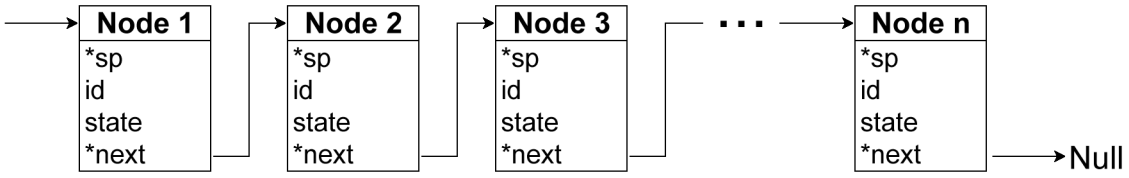


Abbildung 30: Prozess-Liste

Jedes Element enthält neben dem Zeiger auf den zugehörigen Stack, der ID und eines Zeiger auf das nächste Element der Liste noch ein Feld, welches den aktuellen Status des Prozesses beschreibt.

```
1  typedef struct _task_t {
2      void *sp;
3      uint16_t state; // 0: active, >0: sleep, UINT16_MAX: suspended
4      uint8_t id;
5      struct _task_t *next;
6  } task_t;
```

Tabelle 4: Element der Prozessliste

Das Statusfeld in den Listenelementen dient zum Kennzeichnen, ob der jeweilige Prozess bei einem Wechsel berücksichtigt werden soll, oder nicht. Wenn der Status 0 ist, bedeutet dies, dass der Prozess aktiv ist und darauf wartet, abgearbeitet zu werden. Wenn der Status größer als 0 ist, bedeutet dies, das er schläft. Bei jedem Systemtick wird der Status aller Prozesse, sofern über 0, um 1 verringert. Sollte der Status jedoch 65535 betragen, kennzeichnet dies, dass der Prozess unterbrochen wurde und erst weitergeführt werden kann, wenn er aufgeweckt wird. Statusfelder, die diesen Wert haben werden nicht verringert.
Sobald mehrere Prozesse vorhanden sind, werden diese nach dem Round-Robin Verfahren abgearbeitet, d.h. die Liste wird der Reihe nach durchgegangen. Allen Prozessen steht die gleiche Anzahl an Systemzeit zur Verfügung.

5.2.2 Prozesse anlegen

Damit ein Multitasking-System normal arbeiten kann, müssen mindestens zwei lauffähige Prozesse vorhanden sein. Da die Prozesse vom Benutzer angelegt werden können, muss hierfür eine möglichst einfach zu bedienende Funktion erstellt werden. Am benutzerfreundlichsten wäre es, wenn der zuständigen Funktion lediglich die Größe des Stackspeichers für den Prozess, die maximale Laufzeit, sowie ein Zeiger auf die Prozess-Funktion übergeben werden müsste. Da die Laufzeit eines Prozesses bereits fest im Code vorgegeben ist, kann die Angabe der Laufzeit jedoch entfallen. Mit Hilfe der anderen beiden Parameter lässt sich eine Funktion implementieren, welche einen neuen Prozess anlegt, den Speicher für seinen Stack reserviert und ihn mit in die globale Prozessliste einträgt.
Bevor ein neuer Prozess angelegt werden kann, muss erst einmal das letzte Element der Prozessliste gefunden werden. Anschließend wird ein neues Prozesselement (Tabelle 4) angelegt und der `next`-Zeiger des letzten Listenelements so abgeändert, das dieser auf das neu erzeugte Element zeigt (Tabelle 5).

```
1  // Find last element
2  last = g_tasks; // Pointer to list header
3  while(last->next != 0)
4      last = last->next;
5
6  // Create new element
7  newNode = (task t*) malloc(sizeof(task t));
8  newNode->next = 0;
9  newNode->id = g taskcount++;
10 newNode->state = 0;
11 last->next = newNode;
```

Tabelle 5: Prozess erstellen

Anschließend kann der Speicher für den Stack reserviert werden.
Sobald Daten auf dem Stack abgelegt werden, vermindert sich der Stackpointer. Entsprechend liegen die Daten somit an Speicheradressen, die größer sind als die, auf die der Stackpointer zeigt. Man kann also sagen, der Stack wächst „nach unten“. Damit die Daten in dem eben angelegten Speicherbereich abgelegt werden, muss der Zeiger auf diesen Bereich noch soweit abgeändert werden, dass er auf die größte Adresse des Stackspeichers zeigt.

Damit die Funktion, die als Prozess dienen soll, später auch ausgeführt wird, ist es nötig, die entsprechende Adresse als allererstes auf dem Stack abzulegen. Somit kann bei dem späteren Prozesswechsel diese als Rücksprungadresse einer Funktion verwendet werden. Es muss jedoch beachtet werden, dass die Daten auf dem Stack im Big-Endian Format gespeichert werden, d.h. die beiden Bytes der Adresse müssen vertauscht auf dem Stack gespeichert werden.

```
1  // Allocate stack
2  uint8 t *sp = 0;
3  sp = (uint8_t*) malloc(sizeof(uint8_t)*stacksize);
4  sp = sp+stacksize-1; // pointer to highest stack element
5
6  // Put function on stack
7  *sp = (uint8_t)((uint16_t)func&0x00FF);
8  *(sp-1) = (uint8_t)((uint16_t)func>>8);
9  sp -= 2;
10 newNode->sp = sp;
```

Tabelle 6: Stack anlegen

Abbildung 31 zeigt einen neu angelegten Prozessstack mit seiner Rücksprungadresse (0xDA07).

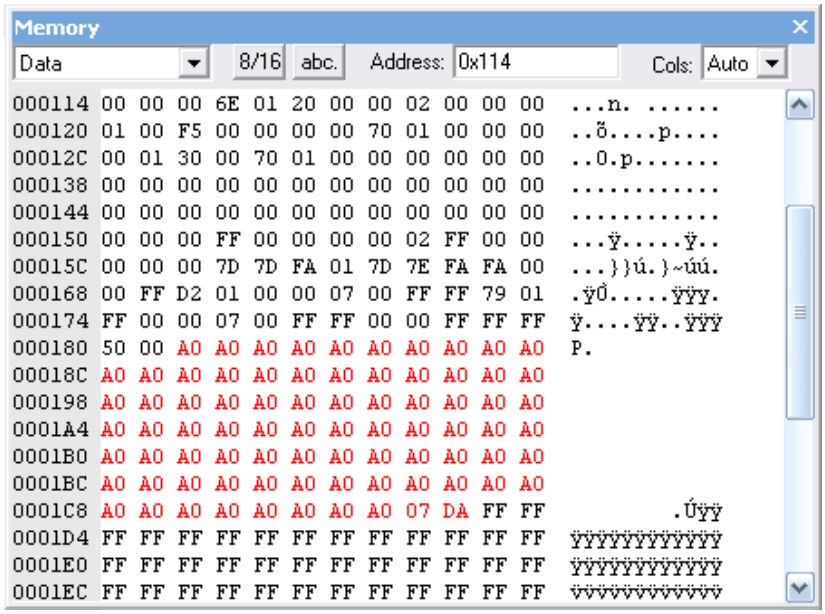


Abbildung 31: Prozessstack mit Rücksprungadresse

5.2.3 Prozesse wechseln

Soll zwischen zwei Prozessen gewechselt werden, müssen alle 32 Register sowie das Statusregister des Mikrocontrollers gesichert werden. Um nicht unnötig kostbaren Speicherplatz zu verbrauchen, bietet es sich an, die Daten auf dem jeweils aktuellen Stack abzulegen.
Nachdem alle Register gesichert wurden, kann der aktuelle Stackpointer gespeichert und durch den des nächsten Prozesses ersetzt werden.
Um ein bessere Kontrolle über die Register zu bekommen, ist es sinnvoll diese Prozedur in Assembler zu schreiben.
Die entsprechende Funktion würde, wie Abb. 32 verdeutlicht, in drei Schritten ablaufen:
1. Alle Register auf dem aktuellen Stack speichern
2. Den aktuellen Stackpointer speichern und den des neuen Prozesses laden
3. Alle Register des neuen aktuellen Prozesses vom Stack laden

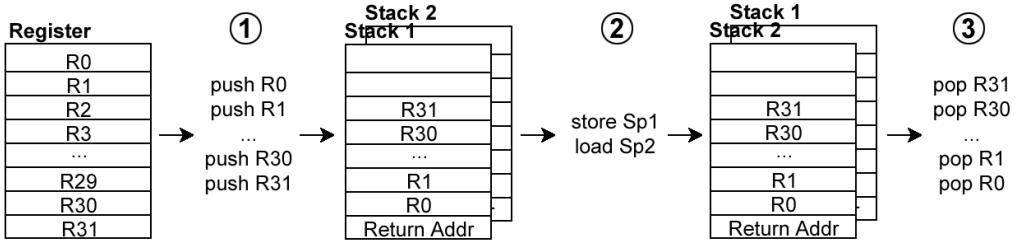


Abbildung 32: Stackwechsel

Es treten allerdings Probleme bei einem Wechsel zu Funktionen auf, die neu angelegt und bisher nie ausgeführt wurden. Der Stack dieser Prozesse ist, bis auf die Rücksprungadresse, leer. Somit würde Schritt 3, das Laden der Register vom Stack dazu führen, dass mit Werten außerhalb des gültigen Stackbereichs gearbeitet wird und die Software hängen bleibt.
Dieses Problem kann umgangen werden, indem der Wechsel des Stacks in einer sparaten Funktion stattfindet. Dadurch springt die Funktion nach dem Wechsel nicht zurück in die Funktion, von der sie aufgerufen wurde, sondern zu der, die in dem neuen Stack steht (Abb. 33).

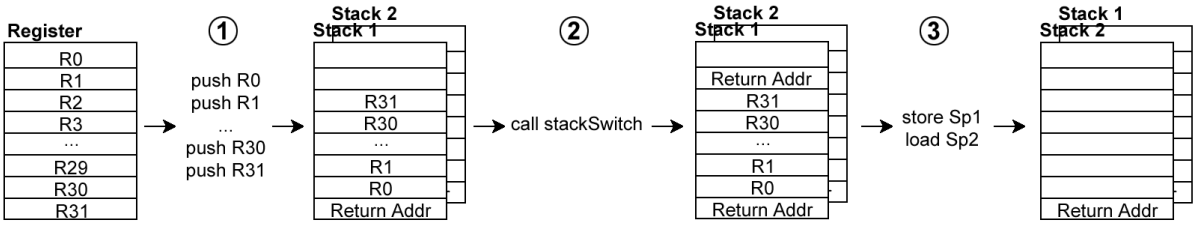


Abbildung 33: Stackwechsel mit Extrafunktion

5.2.4 Prozesse steuern

Da ein Multitasking-System, bei dem nur Prozesse angelegt und zwischen ihnen gewechselt werden kann zu starr für einen produktiven Einsatz ist, müssen noch einige zusätzliche Methoden implementiert werden, mit denen sich das Verhalten der Prozesse besser steuern lässt. Es ist u.a. sehr nützlich, einzelne Prozesse unterbrechen und fortsetzen zu können. Oder aber den aktuellen Prozess für eine bestimmte Zeit „schlafen“ zu legen. Durch die bereits geleistete Vorarbeit, lässt sich dieses innerhalb weniger Codezeilen realisieren.

Damit ein Prozess unterbrochen wird, d.h. bei einem Prozesswechsel so lange nicht berücksichtigt wird, bis er aufgeweckt wurde, muss lediglich sein Status auf den maximalen Wert gesetzt werden. Konkret bedeutet dies, das die Variable `state` in dem entsprechenden Listeneintrag den Wert 65.536 zugewiesen bekommt. Dies signalisiert der Funktion, die für den Prozesswechsel zuständig ist, das dieser Prozess bei der Auswahl nicht berücksichtigt werden soll. Sollte jedoch der aktuell aktivierte Prozess unterbrochen werden, muss ein Wechsel zu einem anderen Prozess erzwungen werden. Ansonsten würde der Prozess so lange weiterlaufen, bis seine Zeit abgelaufen ist und er regulär unterbrochen wird. Dies könnte jedoch besonders dann zu unerwünschten Nebeneffekten führen, wenn der weitere Prozessverlauf von einem anderen Prozess abhängig ist.

```
1  void suspendTask(uint8_t id) {
2      task_t *last = g_tasks;
3      // Find task by id
4      while(last != 0) {
5          if(last->id == id) { // Task found
6              last->state = UINT16_MAX; // suspend task
7              break;
8          }
9      }
10     // Current task was suspended, force task switch
11     if(id == this)
12         schedule();
13 }
```

Tabelle 7: suspendTask-Funktion

Damit ein Prozess auch wieder fortgesetzt werden kann, muss er aufgeweckt werden. Hierfür reicht es, in dem entsprechenden Listenelement den Wert der `state` Variablen wieder auf 0 zu setzen. Somit wird der Prozess beim nächsten Wechsel wieder mit berücksichtigt.

Wenn nun jedoch ein Prozess nur für eine gewisse Zeit angehalten werden soll, ist es umständlich, diesen erst abzuschalten und anschließend von einem anderen Prozess wieder aufwecken zu lassen. Dies ist sogar unmöglich, sollte nur ein Prozess laufen. Es wird also noch eine weitere Funktion benötigt, mit welcher sich die Prozess für eine bestimmte Zeit unterbrechen lassen.

```
1  void sleep(uint16_t ticks) {
2      // Scheduler aktive?
3      if(g_taskcount>0) {
4          g_runningTask->state = ticks;
5          Schedule(); // Force task switch
6      }
7      else {
8          delay(ticks); // No scheduler, delay main task
9      }
10 }
```

Tabelle 8: sleep-Funktion

Diese, in Tabelle 8 dargestellte, Funktion nennt sich `sleep` und dient als Ersatz für die in der avr-libc vorkommenden Funktionen `_delay_ms` und `_delay_us`. Während die beiden Funktionen jedoch das komplette System verzögern, hält `sleep` nur den aktuellen Prozess für eine bestimmte Anzahl von Systemticks an.
Da der Wert der `state` Variablen eines Prozesselements bei jedem Systemtick verringert wird, kann hiermit ein Prozess um eine gewisse Anzahl an Systemticks vom Prozesswechsel ausgeschlossen werden. Da der Prozess jedoch einfach weiterlaufen würde nachdem der Wert der Variablen geändert wurde, muss zusätzlich noch ein Prozesswechsel erzwungen werden. Dies setzt jedoch voraus, dass das Scheduling-System aktiv ist.
Sollte hingegen nur der Hauptprozess laufen, muss eine andere Möglichkeit gefunden werden, diesen für eine gewisse Zeit anzuhalten. Es bietet sich an, auf den Systemtimer zuzugreifen (Tabelle 9). Dieser speichert, wie bereits erwähnt, die Anzahl der verstrichenen Systemticks in einer globalen Variable.

```
1  void delay(uint16_t ms) {
2      uint32_t ticks = getSystemTicks();
3      while(getSystemTicks() < ticks+ms);
4  }
```

Tabelle 9: delay-Funktion

5.3 Analoge Sensoren

Zum Auslesen der analogen Sensoren wird der interne ADC (Analog-Digital-Converter) des Mikrocontrollers verwendet. Dieser bietet eine Auflösung von 10 Bit, womit im besten Fall bis zu 1024 unterschiedliche Werte gemessen werden können. Allerdings besitzt er bauartbedingt nur eine absolute Genauigkeit von 8 Bit, die letzten 2 Bit können schwanken [22].
Eine Wandlung dauert je nach Einstellung zwischen 13 und 25 Takten. Da der ADC nur bei einem Takt von 50 bis 200kHz zuverlässig und mit voller Auflösung arbeiten kann, wird der Systemtakt des Mikrocontrollers durch einen Prescaler geleitet. Dieser teilt den eingehenden Takt wahlweise durch 2, 4, 8, 16, 32, 64 oder 128, um so den optimalen Takt des ADCs zu erreichen. Eine komplette Wandlung dauert bei maximaler Geschwindigkeit somit 1/200kHz*13Takte = 65µS. Da es allerdings nicht möglich ist, die 16Mhz mit denen der Mikrocontroller getaktet ist, so zu teilen, dass der ADC mit genau 200kHz betrieben wird, sollte die nächst langsamere Frequenz gewählt werden, was in diesem Fall einem Teilungsfaktor von 128 und somit einer Arbeitsfrequenz von 125kHz entspricht.
Bei diesem Takt benötigt eine Wandlung 104µS. Zusätzlich muss beachtet werden, dass die erste Umwandlung nach Aktivieren des ADCs 25 Takte, also 200µS, benötigt, da der Analogteil des Wandlers initialisiert werden muss.
Für die Software bedeutet dies, dass bei der Initialisierung des ADCs zusätzlich einige Wandlungen durchgeführt werden sollten. Die vollständige Initialisierung sieht wie folgt aus:

```
1  // ADC
2  ADMUX = 0x40; // VCC as Vref
3  ADCSRA = (1<<ADEN)|(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0); // 128x Prescaler
4  for(i=0;i<2;i++) {
5      ADCSRA |= (1<<ADSC); // Start Conversion
6      while( (ADCSRA&(1<<ADSC)) != 0); // Wait for finish
7  }
```

Tabelle 10: Analog-Digital-Wandler Initialisierung

Durch setzen des Bits ADSC im Register ADCSRA wird dem ADC mitgeteilt, dass eine Wandlung gestartet werden soll (Zeile 5). Sobald die Wandlung abgeschlossen ist, wird das Bit gelöscht. Durch ständiges Auslesen des ADCSRA-Registers kann somit festgestellt werden, ob die Wandlung abgeschlossen ist oder noch läuft (Zeile 6).

Die Funktion zum Abfragen der analogen Sensoren sieht wie folgt aus:

```
1  uint16_t getAnalog(SENSOR_ANALOG id) {
2      uint16_t ret=0;
3      ADMUX = 0x40 | id;
4      ADCSRA |= (1<<ADSC);
5      while( (ADCSRA&(1<<ADSC)) != 0);
6      ret = ADCL | (ADCH<<8);
7      return ret;
8  }
```

Tabelle 11: getAnalog-Funktion

Übergeben wird dieser Funktion die ID des Sensors, der abgefragt werden soll. Hinter dem Variablentyp „SENSOR_ANALOG“ verbirgt sich ein „enum“, also eine Aufzählung von Variablen. Hierdurch kann anstatt einer Zahl auch die Bezeichnung des Sensors angegeben werden, wodurch die Lesbarkeit des Quellcodes erleichtert wird.

```
1  typedef enum  { ABSTL=0,ABSTR,LINEL,LINER,LDRL,LDRR,KANTEL,KANTER }
   SENSOR_ANALOG;
```

Die ID ist gleichzeitig der entsprechende Pin an dem der Sensor am ADC angeschlossen ist (Zeile 3). Somit wird eine aufwendige Dekodierung von der ID zum Anschlusspin vermieden. Nachdem der ADC mit der Wandlung begonnen hat, wird das Programm so lange angehalten, bis diese abgeschlossen ist (Zeile 5). Eine Alternative hierzu wäre, nach der Beendigung einer Wandlung einen Interrupt auszulösen. Allerdings würde hierbei die Frage aufkommen, welchen Aufgaben die Software nachkommen soll, während die Wandlung noch andauert, da ein Programm normalerweise für seinen weiteren Ablauf auf den jeweils aktuellen Wert angewiesen ist und somit warten müsste, bis die Wandlung abgeschlossen ist.
Eine weitere Option wäre es, die angeschlossen Sensoren periodisch abzufragen. Hierbei würden jedoch zeitliche Probleme Auftreten. Da eine Wandlung 104µS andauert, kann das Ergebnis des ADCs nicht in der ISR (Interrupt Service Routine) des Systemtimers ausgewertet werden, sondern benötigt eine eigene ISR.
Somit würde einerseits der Interrupt des ADCs unweigerlich mit dem des Systemtimers in Konflikt geraten, was einen verzögerten Aufruf der Systemtimer ISR hätte zur Folge haben könnte und somit ggf. Teilen der zeitkritischen Systeme Probleme bereiten würde. Andererseits ist es möglich, dass nicht alle Sensoren benötigt werden und somit unnötig Abgefragt werden würden.

Zum Abschluss der `getAnalog`-Funktion wird das Ergebnis der beiden Register ausgelesen und zurückgegeben. Durch die 10 Bit Auflösung des ADCs benötigt es zwei Register. In ADCL sind die unteren 8-Bit gespeichert und in ADCH die zwei oberen.

5.4 Digitale Sensoren

5.4.1 Allgemein

Von den digitalen Sensoren gibt es zwei Sorten. Zum einen die, welche nur dann abgefragt werden, wenn die Daten benötigt werden und zum anderen die Sensoren, deren Daten ständig aktualisiert und zwischengespeichert werden müssen. Zu den letzteren zählen z.B. die Radencoder oder der Infrarotempfänger. Die Sensoren des Transportfachs hingegen, dazu zählen die Lichtschranke und der Sensor der Klappe, brauchen nur abgefragt werden, wenn die Daten benötigt werden, da diese beiden Sensoren ihre Daten nicht innerhalb weniger Millisekunden ändern können.
Entgegen der analogen Sensoren muss bei den digitalen nichts initialisiert werden.
Auch die Funktion zum Abfragen der Sensoren ist entsprechend einfach aufgebaut:

```
1  uint8 t getDigital(SENSOR DIGITAL id) {
2      if(id == KLAPPE || id == RADR)
3          return (PIND&(1<<id)>>id);
4      else
5          return (PINB&(1<<id))>>id;
6  }
```

Tabelle 12: getDigital-Funktion

Sie ist von den Parametern her fast identisch mit der Funktion `getAnalog`. Bei dem Typ „SENSOR_DIGITAL“ handelt es sich allerdings nicht um eine Aufzählung sondern lediglich um eine 8-Bit vorzeichenlose Integerzahl, die per „`#define SENSOR_DIGITAL uint8_t`“ umbenannt wurde, um ein einheitliches Aussehen zu erreichen. Da die Eingänge der digitalen Sensoren anders als bei den analogen alle verstreut angeschlossen sind, ist es nicht möglich eine Aufzählung zu verwenden, ohne diese in der Funktion wieder zeitaufwendig in die Pinnummer auflösen zu müssen. Da es sich lediglich um 4 Sensoren handelt, die auf diese Art abgefragt werden, ist es einfacher die Namen per `#define` festzulegen.

```
1  #define SCHRANKE PB0
2  #define RADL PB4
3  #define KLAPPE PD6
4  #define RADR PD3
5  #define SENSOR DIGITAL uint8 t
```

Tabelle 13: Definition der digitalen Signale

Eine weitere Besonderheit ist, dass der Schrankensensor und der linke Radsensor an Port B des Mikrocontrollers angeschlossen sind, der Sensor der Klappe sowie der rechte Radsensor hingegen an Port D. Dies muss zusätzlich in der Funktion berücksichtigt werden.
Um dem Benutzer einen größtmöglichen Freiraum zu lassen, können die beiden Radsensoren mit dieser Funktion auf ihren aktuellen Status hin abgefragt werden.

5.4.2 Radencoder

Die Radencoder zählen die Wechsel der Streifen auf den Rädern von Beginn Programmstarts an.
Da nicht für beide Pins, an denen die beiden Sensoren angeschlossen sind, die Möglichkeit besteht bei einem Flankenwechsel ein Interrupt zu generieren, werden beide Eingänge kontinuierlich abgefragt und der jeweils aktuelle Status in einer Variablen zwischengespeichert. So lassen sich mögliche Änderungen effektiv erkennen.
Allerdings müssen die Eingänge, um keinen Pegelwechsel zu verpassen, mindestens doppelt so schnell abgefragt werden, wie sich das Signal ändern kann. Bei maximaler Geschwindigkeit drehen die Räder mit 151 U/m. Durch die 30 Markierungen auf den Rädern ergeben sich 60 Pegelwechsel pro Umdrehung an den Pins. Dies bedeutet, dass sich alle ~6,6ms der Pegel eines der Pins ändert. Damit kein Wechsel ausgelassen wird, müssen die Eingänge also mindestens alle 3,3ms abgefragt werden.
Es bietet sich an, dies mit in die ISR des Systemtimers zu implementieren, denn diese wird alle 0,1ms ausgeführt. Da der Code hierfür relativ kompakt ist, ergibt sich zudem nur eine minimale zeitliche Verzögerung.

```
1  if( ((PINB>>4)&0x01) != g tmpLeft) {
2      g encLeft++;
3      g tmpLeft = (PORTB>>4)&0x01;
4  }
5  if( ((PIND>>3)&0x01) != g tmpRight) {
6      g encRight++;
7      g tmpRight = (PORTD>>4)&0x01;
8  }
```

Tabelle 14: Radencoder

Wie in Tabelle 14 zu sehen ist, besteht die Abfrage lediglich aus zwei „if“-Anweisungen, eine für den Sensor des linken Rades und eine für den rechten. Es wird überprüft, ob der aktuelle Stand des Pins nicht mehr dem gespeicherten entspricht (Zeile 1 u. 5). Ist dies der Fall, hat sich das Rad um einen Streifen weiter gedreht und die Anzahl der Encoderschritte wird um 1 erhöht. Anschließend muss noch der aktuelle Pinstand gespeichert werden.

5.4.3 Maussensor

Eine Ausnahme bei der Ansteuerung bildet der Maussensor. Bei diesem wäre es angebracht, ihn periodisch abzufragen, um z.B. eine zurückgelegte Wegstrecke ermitteln zu können. Allerdings ist dies aufgrund der seriellen Übertragung zwischen Mikrocontroller und Sensor-IC sowie der Wartezeit, die zwischen zwei Befehlen eingehalten werden muss, nicht performant zu realisieren. Es wäre zwar durchaus möglich, das Abfragen des Sensors in der ISR des Systemtimer unterzubringen, jedoch würde diese hierdurch erheblich mehr Zeit in Anspruch nehmen, welche anschließend dem Benutzer bei der Ausführung seines Programms fehlt. Ein weiterer Grund, der gegen ein periodisches Abfragen des Maussensors spricht, ist die Erfahrung, die bisher mit diesem gesammelt wurde. Es hat sich durch mehrere Versuche und auch in der Praxis gezeigt, dass dieser Sensor teilweise sehr unzuverlässig arbeitet, besonders auf weißen Oberflächen. Daher wird er kaum mehr eingesetzt und sein periodisches Abfragen wäre eine unnötige Belastung des Systems. Für den Fall, dass er dennoch einmal benötigt werden sollte, wird die Bibliothek Funktionen enthalten, die ein manuelles Abfragen erlauben.

Bevor der Maussensor initialisiert werden kann, sollten mindestens 100mS gewartet werden. So hat die Spannungsversorgung Zeit, um sich zu stabilisieren. Anschließend wird der Sensor per Software neu gestartet und danach eingestellt. Hierfür wird, wie bereits in Kapitel 3.7.3 beschrieben, zuerst das zu verändernde Register an den Maussensor übertragen und anschließend der neue Wert. Zwischen dem Ende des ersten Befehls und dem Ende der nächsten Registeradresse müssen mindestens 100µS vergehen. Der Sensor benötigt diese Zeit um den jeweiligen Befehl zu bearbeiten.

```
1  void mouse_init() {
2      // Set ports
3      DDRB |= (1<<PB7);
4      PORTB &= ~(1<<PB7);
5      uint8_t i=0;
6
7      // Wait for VCC
8       delay ms(100);
9
10     mouse_sendcmd(0x00|0x80); // Config. register
11     mouse_sendcmd(0x80);      // Reset command
12     _delay_us(47);
13     mouse_sendcmd(0x00|0x80); // Config. register
14     mouse_sendcmd(0x01);      // Normal operation
15     _delay_us(47);
16 }
```

Tabelle 15: Maussensor initialisieren

Da die serielle Übertragung bei jedem Datentransfer nahezu identisch ist, bietet es sich an, dieses Code-Segment in eine eigenen Funktion auszulagern. Somit wird wertvoller Speicherplatz gespart.
Die Datenleitung des Maussensor ist bidirektional. Dies bedeutet, dass der entsprechende Pin am Mikrocontroller als Ausgang definiert werden muss, bevor Befehle an den Sensor gesendet werden können. Die anschließend am Datenpin des Sensors anliegenden Signale werden durch eine steigende Flanke auf der Taktleitung übernommen. Wie bei den meisten seriellen Protokollen, wird auch hier die Übertragung mit dem MSB begonnen.
Nachdem die Registeradresse übertragen wurde, kann entweder ein weiteres Byte gesendet werden, welches die Daten repräsentiert, die in das Register geschrieben werden. Alternativ kann der Datenpin als Eingang geschaltet und durch Takten der Taktleitung das Register ausgelesen werden. Hierbei ist jedoch zu beachten, dass im Gegensatz zu dem Übertragen vom Mikrocontroller zum Sensor, die Daten bei einer fallenden Flanke auf der Datenleitung ausgegeben werden.

```
1  uint8_t getMouseReg(uint8_t reg) {
2      uint8_t i=0;
3      int8_t data=0;
4      syslock();            // Disable taskswitch
5      mouse_sendcmd(reg);   // Register to read from
6      delay(1);             // weit 100µs
7      DDRB &= ~(1<<SDIO);   // Data input
8      PORTB &= ~(1<<SDIO); // No pullup
9      for(i=0;i<8;i++) {
10         PORTB &= ~(1<<PB7); // Clk low for min. 250ns
11         asm volatile("nop");
12         asm volatile("nop");
13         asm volatile("nop");
14         data = data<<1;
15         PORTB |= (1<<PB7); // Clk high
16         data |= ((PINB>>SDIO)&0x01); // Save bit
17     }
18     sysunlock(); // Enable task switch
19     return data;
20 }
```

Tabelle 16: Maussensor auslesen

Auf diese Weise können alle 11 Register des Sensors ausgelesen werden. Für den normalen Gebrauch werden allerdings häufig nur die Register 2 und 3 benötigt, welche die Abweichung in X- und Y-Richtung enthalten. Aus diesem Grund wurde für die Abfrage dieser beiden Register zusätzlich jeweils eine eigene Funktion implementiert.

5.4.4 Infrarotempfänger

Der Infrarotempfänger hängt, wie die Radencoder, an einem Pin des Mikrocontrollers, für den kein Interrupt definiert werden kann. Somit muss der entsprechende Eingang manuell abgefragt werden. Hinzu kommt, dass durch das RC5 Protokoll sich die Signalpegel an dem Eingang innerhalb von ~889µs ändern können. Um den Eingang in diesen relativ kurzen Abständen abfragen zu können, ist es notwendig, während des Empfangs alle Interrupts abzuschalten und in einer Schleife auf den nächsten Pegelwechsel zu warten, oder wie bereits bei den Radencoder, dies in der ISR eines Timers zu tun. Hier bietet sich wieder die ISR des Systemtimers an.

Folgendes Codesegment wird mit in die Funktion integriert und implementiert das in Abb. 34 dargestellte Aktivitätsdiagramm zum Dekodieren des RC5-Signals:

```
1  if(++g_rc5timer > RC5_PULSE_MAX) {
2      if(g_rc5tmp & 0x2000) {
3          g_rc5code = g_rc5tmp;
4          g_rc5toggle = g_rc5tmp>>11&0x01;
5      }
6      g_rc5tmp = 0;
7  }
8  if( (g_rc5lastlevel ^ PINB) & (1<<PB1) ) {
9      g_rc5lastlevel = ~g_rc5lastlevel;
10     if(g_rc5timer < RC5_PULSE_MIN) {
11         g_rc5tmp = 0;
12     }
13     if(!g_rc5tmp || g_rc5timer > RC5_PULSE_1_2) {
14         g_rc5tmp = g_rc5tmp<<1;
15         if( !(g_rc5lastlevel&(1<<PB1)) )
16             g_rc5tmp |= 1;
17         g_rc5timer = 0;
18     }
19 }
```

Tabelle 17: RC5-Dekodierung

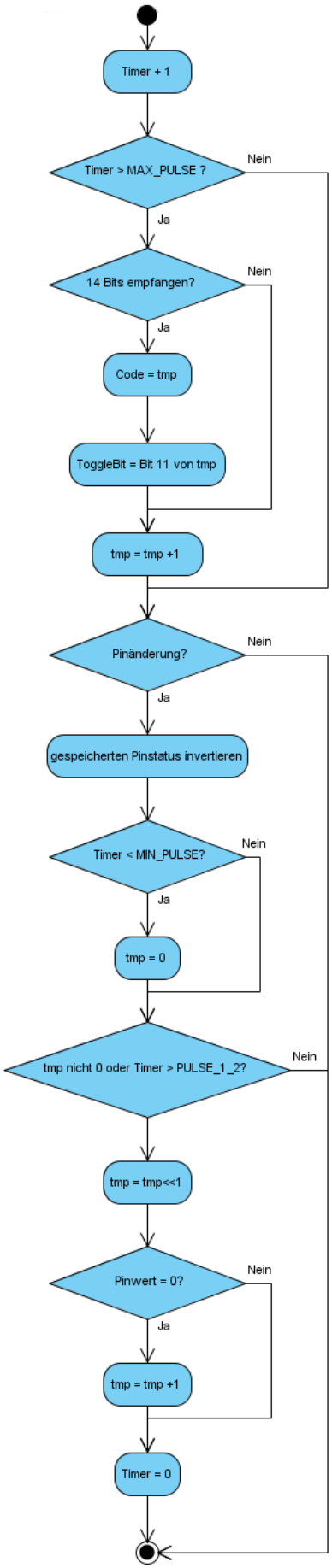


Abbildung 34: RC5-Dekodierung

5.5 Serielle Schnittstelle

Die serielle Schnittstelle des c't-Bots kann unter anderem zur Kommunikation mit vorhanden Erweiterungsboards verwendet werden. Um einen möglichst weiten Einsatzbereich zu ermöglichen und die weitere Verarbeitung zu vereinfachen, sollten alle empfangenen Daten in einem Puffer zwischengespeichert werden. Hierfür bietet sich als Zwischenspeicher ein Ringpuffer an. Das besondere an einem Ringpuffer ist, dass er eine feste Größe hat, jedoch nicht überlaufen kann. Sollte der Puffer voll sein, wird der älteste Eintrag überschrieben.
Im konkreten Fall des c't-Bots hat dies zur Folge, das nur die neuen Daten gespeichert werden, alte hingegen werden verworfen.
Für das Versenden sollte eine Funktion verwendet werden, welche die Daten sofort verschickt und nicht zwischenspeichert. Dies wird im allgemeinen auch von Benutzern erwartet, wenn sie diese entsprechende Funktion aufrufen.
Zum Einschalten des UARTs müssen lediglich der Sender und Empfänger aktiviert werden. Die meisten übertragungsrelevanten Einstellungen sind bereits standardmäßig richtig gesetzt. So ist bereits festgelegt, dass jedes Datenpaket 1 Startbit, 8 Datenbits und 1 Stoppbit enthält. Zusätzlich muss jedoch noch das RXCIE-Bit gesetzt werden. Dieses ermöglicht, dass bei jedem empfangenen Byte ein Interrupt ausgelöst wird. Hierdurch wird das kontinuierliche Abfragen des UART-Statusregister vermieden. Damit möglichst wenig Einstellungen vom Benutzer selber vorgenommen werden müssen, wird die Baudrate des UARTs bei der Initialisierung auf 19200 Bits/s eingestellt. Bei dieser Geschwindigkeit ist eine Kommunikation häufig auch dann noch möglich, wenn der Kommunikationspartner mit einem anderen Systemtakt arbeitet. Anschließend müssen noch die beiden Zeiger des Ringpuffers auf den Anfang des Arrays gesetzt werden. Der Schreibzeiger sollte mit einem Wert initialisiert werden, welche außerhalb der Größe des Speicherarrays liegt, damit später erkannt werden kann, ob bereits Daten im Ringpuffer liegen. Würden beide Zeiger mit 0 initialisiert werden, treten im späteren Programmverlauf Probleme auf, sobald mehr Daten im Array gespeichert als gelesen werden.

```
1  void uart_init(void) {
2      // 8N1, ReceiveInterrupt enable, RX enable, TX enable
3      UCSRB |= (1<<RXCIE) | (1<<RXEN) | (1<<TXEN);
4      uartSetBaud(19200);
5      g_uartRecBufferWritePos = -1;
6      g_uartRecBufferReadPos=0;
7  }
```

Tabelle 18: UART initialisieren

Die Funktion `uartSetBaud` dient zum Umrechnen und Einstellen der Baudrate. Bei den AVR-Mikrocontrollern wird die Baudrate über einen programmierbaren Prescaler festgelegt. Hierbei handelt es sich um einen abwärts zählenden Timer. Dieser wird über ein 16-Bit Register geladen und dekrementiert den eingestellten Wert bei jedem Systemtakt um Eins. Sobald 0 erreicht wird erzeugt der Timer einen Takt auf Seiten des UARTs und beginnt erneut bei dem im Register gespeicherten Wert.
Dies bedeutet, dass für das Einstellen der Baudrate, diese in einen 16-Bit Wert umgerechnet werden muss.

Aus dem Datenblatt des ATmega32 kann man folgende Formel entnehmen:

```
1  #define BAUD(x) ( (F CPU/(16L*(x)))-1 )
```

Hieran lässt sich sehr gut veranschaulichen, dass einige Baudraten nicht geeignet sind, um mit einem Takt von 16Mhz genutzt zu werden. Setzt man z.b 115200bit/s in die Formel ein, so erhält man 7,680556 als möglichen Timer Wert. Da sich jedoch nur ganze Zahlen in das Register schreiben lassen, muss der Timer entweder mit 7 oder mit 8 geladen werden, was einer Baudrate von 125kbit/s bzw. 111,11111kbit/s entspricht. Solange die Kommunikation mit einem Gerät stattfindet, das mit der selben Geschwindigkeit und Baudrate läuft, wird es keine Übertragungsfehler geben. Da jedoch bei einigen Systemen und besonders bei Softwareprogrammen die Geschwindigkeit häufig fest eingestellt ist oder nur eine vordefinierte Auswahl zulässt, kann es zu erheblichen Problemen in der Kommunikation führen.
Aus diesem Grund hat die Firma Atmel in dem Datenblatt des ATmega32 für eine an Großzahl an Standardfrequenzen die möglichen Baudraten, deren zugehöriger Registerwert sowie die mögliche Abweichung aufgelistet (Abb. 35) [22].

	f_{osc} = 16.0000 MHz			
	U2X = 0		U2X = 1	
Baud Rate (bps)	UBRR	Error	UBRR	Error
2400	416	-0.1%	832	0.0%
4800	207	0.2%	416	-0.1%
9600	103	0.2%	207	0.2%
14.4k	68	0.6%	138	-0.1%
19.2k	51	0.2%	103	0.2%
28.8k	34	-0.8%	68	0.6%
38.4k	25	0.2%	51	0.2%
57.6k	16	2.1%	34	-0.8%
76.8k	12	0.2%	25	0.2%
115.2k	8	-3.5%	16	2.1%
230.4k	3	8.5%	8	-3.5%
250k	3	0.0%	7	0.0%
0.5M	1	0.0%	3	0.0%
1M	0	0.0%	1	0.0%
Max (1)	1 Mbps		2 Mbps	

1. UBRR = 0, Error = 0.0%

Abbildung 35: Baudrate-Tabelle

Sobald nun ein Byte komplett empfangen wurde, wird von dem UART ein Interrupt ausgelöst und die entsprechende ISR aufgerufen. Innerhalb dieser muss zuallererst geprüft werden, ob der Lese-Zeiger auf die Stelle im Array zeigt, an die die neuen Daten geschrieben werden sollen. Ist dies der Fall, wird der Lese-Zeiger um eine Stelle erhöht. Zusätzlich muss noch überprüft werden, ob die neue Arrayposition gültig ist, damit der Zeiger nicht versehentlich den Wertebereich des Arrays verlässt. Anschließend wird sichergestellt, dass der Schreib-Zeiger ebenfalls auf ein gültiges Element im Array zeigt, da dies bei einem leeren Array noch nicht der Fall ist. Erst wenn beide Bedingungen erfüllt sind, können die neuen Daten gespeichert und der Schreib-Zeiger eine Stelle weiter gesetzt werden.

```
1  ISR(SIG_UART_RECV) {
2    uint8_t c = UDR; // read Data
3    /* Set readpointer to next element when equal to writepointer,
4     * data at readpointer-position will be overwritten.
5     */
6    if(g uartRecBufferWritePos == g uartRecBufferReadPos)
7      g uartRecBufferReadPos=(g uartRecBufferReadPos+1)%UARTRECBUFFERSIZE;
8
9    // Buffer is empty, so writepointer is <0
10   if(g_uartRecBufferWritePos < 0)
11     g_uartRecBufferWritePos++;
12
13   // Write data and set writepointer to next element
14   g_uartrecbuffer[g_uartRecBufferWritePos++] = UDR;
15
16   // Overflow?
17   if(g uartRecBufferWritePos > UARTRECBUFFERSIZE-1)
18     g uartRecBufferWritePos=0;
19 }
```

Tabelle 19: UART-ISR

Um ein Element aus den Ringpuffer auszulesen, müssen lediglich die Daten an der entsprechenden Position aus dem Array `g_uartrecbuffer` ausgelesen werden. Vor dem Auslesen empfiehlt es sich allerdings zu prüfen, ob überhaupt Daten gespeichert sind.
Nachdem die Daten aus dem Array zwischengespeichert wurden, kann der Lese-Zeiger um ein Element weiter geschoben werden.

```
1  char uartgetc() {
2    // Buffer empty?
3    if(g_uartRecBufferReadPos == g_uartRecBufferWritePos ||
4       g_uartRecBufferWritePos < 0)
5      return 0;
6
7    // Get data
8    char c = g_uartrecbuffer[g_uartRecBufferReadPos];
9
10   // Set new pointer position
11   g_uartRecBufferReadPos=(g_uartRecBufferReadPos+1)%UARTRECBUFFERSIZE;
12   return c;
13 }
```

Tabelle 20: Ein Zeichen aus Ringpuffer lesen

Gleichermaßen ist die Funktion `uartreadc` aufgebaut. Der einzige Unterschied besteht darin, dass am Ende der Funktion der Lese-Zeiger nicht versetzt wird. Somit ist ein mehrmaliges Auslesen des selben Zeichens möglich.
Damit zusätzlich zu einzelnen Zeichen ganze Zeichenketten ausgelesen werden können, werden die Funktionen `uartgets` und `uartreads` implementiert. Beiden Funktionen wird als Parameter der Zeiger auf einen bereits reservierten Speicherbereich, sowie die Größe dieses Bereichs übergeben. In einer Schleife wird `uartgetc` bzw. `uartreadc` so oft aufgerufen, bis der Speicherbereich voll ist oder sich keine weiteren Zeichen mehr im Ringpuffer befinden. Abschließend wird die neue Zeichenkette mit `\0` terminiert und die Anzahl der ausgelesenen Zeichen zurückgegeben.
Zum Versenden eines Zeichens muss sichergestellt sein, dass das Datenregister des UARTs leer ist. Dies wird vom UART durch setzen des UDRE-Bits im Register UCSRA signalisiert. Da die Funktion zum Senden von Daten so lange blockieren soll, bis die Daten verschickt sind, ist es möglich, das Register in einer Schleife so lange auszulesen, bis das genannte Bit gesetzt wurde. Anschließend muss nur noch das zu versendende Byte in das Datenregister des UARTs geschrieben werden. Der restliche Ablauf der Übertragung wird von der Hardware übernommen.
Um komplette Zeichenketten zu verschicken, ist es möglich, diese Zeichenweise zu übertragen bis ein Terminierungszeichen oder eine festgelegte Länge erreicht wird.
Dies kann jedoch zu Problemen führen, wenn der Benutzer Zeichenketten übertragen möchte, die gleichermaßen Text wie auch Zahlen enthalten, oder wenn zum Aufspüren von Fehler Debug-Nachrichten in für Menschen lesbarer Form an einen PC übertragen werden sollen. In beiden Fällen müsste der Benutzer die Zahlen umständlich umwandeln und zu einer Zeichenkette zusammensetzen, bevor er die Funktion zum Senden aufruft. Hier kann die Funktion `vsnprintf` Abhilfe schaffen. Diese Funktion benötigt zwar viel Platz im Programmspeicher, vereinfacht jedoch das Formatieren von Zeichenketten ungemein. Mit ihrer Hilfe ist es möglich, Variablen in unterschiedlichen Formatierungen in eine Zeichenkette einzufügen.

Der Funktion wird hierzu eine Zeichenkette übergeben, welche neben dem eigentlichen Text Platzhalter enthält, die Aufschluss über die Position und die Formatierung der einzusetzenden Daten geben. Des weiteren wird eine Länge angegeben, die die resultierende Zeichenkette nicht überschreiten darf. Als letztes wird noch eine Liste übergeben, in der die Daten enthalten sind, welche eingesetzt werden sollen.
Da allerdings nicht bekannt ist, wie viele verschiedene Variablen der Benutzer in seine Zeichenkette einfügen möchte empfiehlt es sich eine Funktion zu erstellen, welche eine „variable Parameterliste“ akzeptiert.

```
1  void uartprintf(const char *s, ...) {
2      char buffer[40]; // max 40 characters
3      uint8_t i=0;
4      va_list args;
5
6      // Create string
7      va_start(args,s);
8      vsnprintf(buffer,40,s,args);
9      va_end(args);
10
11     // Send string
12     while(buffer[i] != '\0') {
13         uartputc(buffer[i++]);
14      }
15 }
```

Tabelle 21: String per UART senden

5.6 Erweiterungsboard-Kommunikation

Für die Kommunikation mit dem Erweiterungsboard der HAW sind einige Änderungen an den im vorherigen Kapitel beschriebenen Funktionen erforderlich. Das Erweiterungsboard erwartet Daten und Befehle in einem fest vorgegebenem Format (Abb. 36). Bei dem ersten übertragenen Byte handelt es sich um einen Befehl. Von diesem ist abhängig, welche Aktion das Erweiterungsboard ausführt. Anschließend wird die Anzahl der nachfolgenden Datenbytes übertragen. Dies ermöglicht dem Erweiterungsboard, genügend Platz im Arbeitsspeicher zu reservieren. Als letztes wird die im vorherigen Byte angegebene Anzahl an Datenbytes übertragen.

Abbildung 36: Erweiterungsboard Datenpaket

Nachdem ein komplettes Datenpaket übertragen wurde, wird dieses vom Erweiterungsboard verarbeitet und ggf. mit einem Datenpaket im gleichen Format beantwortet.
Zur Zeit können die in der folgenden Tabelle aufgelisteten Befehle verarbeitet werden:

Befehl	Beschreibung	Liefert Antwort
01h	Get Sensor	Ja
02h	Send Zigbee Message	Nein
03h	Set Zigbee Identity	Nein
04h	Get Position	Ja
05h	Get Zigbee Message	Ja
06h	Set Message Mode	Nein
07h	Set Baud Rate	Nein

Tabelle 22: Erweiterungsboard Befehle

Je nach Befehl ist eine unterschiedlich große Anzahl an Datenbytes erforderlich. So benötigt z.B. der Befehl 01h keine Datenbytes, Befehl 06h 1 Datenbyte und der Befehl 03h ganze 6 Datenbytes.

Befehl	Size	Datenbytes					
		DB0	DB1	DB2	DB3	DB4	DB5
01h	0	-	-	-	-	-	-
02h	min. 2	Dest. high	Dest. low	Data 1	Data 2	Data 3	Data 4
03h	6	Addr. high	Addr. low	Color 1	Color 2	Color 3	Color 4
04h	0	-	-	-	-	-	-
05h	0	-	-	-	-	-	-
06h	1	0 or 1	-	-	-	-	-
07h	2	Baud high	Baud low	-	-	-	-

Tabelle 23: Erweiterungsboard Datenbytes

Für die Implementierung bedeutet dies nun, dass die entsprechenden Funktionen für die serielle Schnittstelle soweit angepasst und erweitert werden müssen, dass die Kommunikation über die beschriebenen Datenpakete läuft. Für den Fall, dass ein anderes Gerät an die serielle Schnittstelle des Mikrocontrollers angeschlossen werden soll, müssen die ursprünglichen Funktionen natürlich enthalten bleiben. Damit diese jedoch nicht unnötigen Speicher belegen oder mit den neuen Funktionen in Konflikt geraten, ist es sinnvoll, diese bereits beim Kompilieren durch den Präprozessor per `#ifdef ... #endif` entfernen zu lassen. Somit können sie jederzeit bei Bedarf wieder eingefügt werden.
Das Senden eines Datenpaketes erfolgt, wie bereits beschrieben, durch das aufeinander folgende Übertragen des Befehls, der Anzahl der Datenbytes und der eigentlichen Daten. Nichts anderes als diesen Ablauf stellt die Funktion `uartSendCmd` zur Verfügung. Die Funktion setzt zudem lediglich die Empfangsfunktion zurück und blockiert so lange, bis die Übertragung vollständig abgeschlossen ist.

```
1  void uartSendCmd(uint8_t cmd, uint8_t *data, uint8_t size) {
2      uint8_t i = 0;
3      uartputc(cmd);              // Send command
4      uartputc(size);             // Send size
5      while(i < size)
6          uartputc(data[i++]);  // Send data
7      uartDataValid = false;          // Mark data as invalid
8      g_uartState = UARTSTATE_CMD;    // Reset ISR
9      while( (UCSRA&(1<<TXC)) == 0); // Wait until transmit completed
10 }
```

Tabelle 24: uartSendCmd Funktion

Da das Empfangen der Datenpakete immer nach dem selben, fest vorgegebenen Muster abläuft, lässt sich dieses sehr gut über einen nichtdeterministischen Automaten darstellen (Abb. 37) und dementsprechend implementieren. Durch die Implementierung in der ISR des UARTs kann ein Zustandswechsel jedoch nur stattfinden, wenn ein neues Byte empfangen wurde. Dies hätte zur Folge, dass durch den Automatenstatus *FINISH* immer ein Byte mehr empfangen werden müsste. Die Implementierung erlaubt es jedoch, dass die Aktionen des *FINISH*-Status sowohl in *UARTSTATE_SIZE* als auch in *UARTSTATE_DATA* verschoben werden können und *FINISH* somit wegfallen kann.

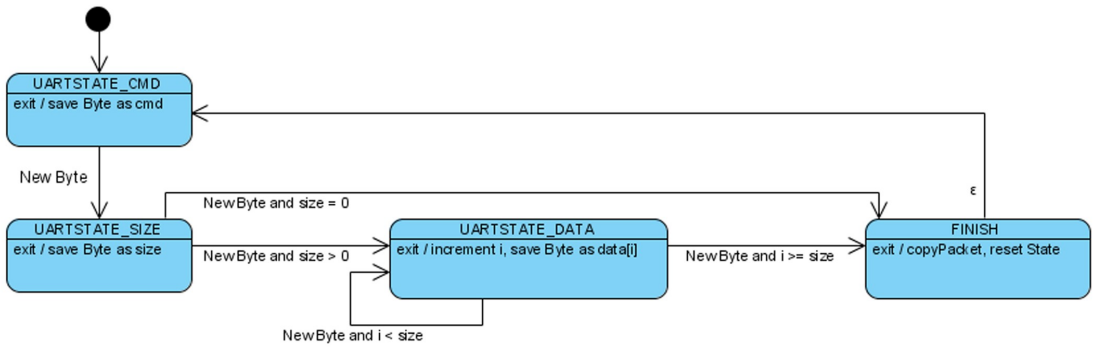


Abbildung 37: Erweiterungsboard Empfangsautomat

Die übersichtlichste Art, einen Automaten in eine Programmiersprache zu übersetzen, ist mit Hilfe einer switch-case-Anweisung gegeben. Hierbei ist es zudem sehr einfach, bei ausbleibenden oder fehlerhaften Daten in den Ursprungszustand zurückzukehren.

```
1  uint8_t c = UDR; // save byte
2      switch(g_uartState)
3          case UARTSTATE_CMD:
4          tmp.cmd = c;                    // save byte as cmd
5          g_uartState = UARTSTATE_SIZE; // next state
6          break;
7      case UARTSTATE_SIZE:
8          tmp.size = c;                   // save byte as size
9          if(tmp.size > 0) {
10             tmp.data = malloc(tmp.size); // allocate memory
11             g uartState = UARTSTATE DATA; // next state
12             i = 0; // reset byte counter
13         }
14         else {
15             tmp.data = 0;  // no data
16             g_uartState = UARTSTATE_CMD; // reset state
17             copyPacket();
18         }
19         break;
20     case UARTSTATE_DATA:
21         tmp.data[i++] = c; // save byte as data
22         if(i>= tmp.size) {
23             g_uartState = UARTSTATE_CMD; // reset state
24             copyPacket();
25         }
26         break;
27     default:
28         g_uartState = UARTSTATE_CMD;
29 }
```

Tabelle 25: Switch-Case Aufbau der UART ISR

Befehl, Anzahl an Bytes und ein Zeiger auf den reservierten Speicherbereich werden in der globalen Variablen `tmp` zwischengespeichert. Bei dieser handelt es sich um eine aus zwei 8-Bit Variablen und einem Zeiger zusammengesetzten Struktur. Nachdem ein Datenpaket vollständig empfangen wurde, wird es in der Funktion `copyPacket` in eine weitere Struktur kopiert und dem Programm über eine Variable signalisiert, das gültige Daten vorliegen. Dies ermöglicht den Empfang neuer Daten, während die alten noch vom Benutzer verarbeitet werden.

Da es für den Benutzer jedoch sehr umständlich und aufwändig sein kann, sich die Datenpakete vor dem Versenden selber zusammensetzen oder die Nutzdaten aus empfangenen Paketen zu extrahieren zu müssen, sollte dies soweit wie möglich hinter einer weiteren Anwendungsschicht verborgen werden. Hierfür werden die Funktionen `sendMessage`, `getMessage`, `setIdentity`, `getPosition` sowie `setMsgMode` implementiert. Diese dienen dazu die Kommunikation für die Befehle *Send Zigbee Message* (02h), *Get Zigbee Message* (05h), *Set Identity* (03h), *Get Position* (04h) und *Set Message Mode* (06h) zu abstrahieren. Da der Befehl Get Sensor (01h) die Werte der analogen Sensoren des Boards liefert, wird dieser mit in die `getAnalog` Funktion der Bibliothek integriert. Hierzu muss in das enum SENSOR_ANALOG noch die Werte Voltage und Current eingefügt werden. Anschließend kann in der Funktion `getAnalog` geprüft werden, ob einer der beiden Werte abgefragt werden soll. Ist dies der Fall, wird der Sensor des Erweiterungsboards abgefragt und der entsprechende Wert zurückgegeben.

```
1  if(id > 7) {
2      uartSendCmd(CMD_GETSENSOR,0,0);     // send command
3      while(newDataAvailable() != true); // wait for answer
4      if(getSize() == 4) {  // right datasize received
5          uint8_t bf[4];
6          getData(bf);
7          if(id == VOLTAGE)
8              ret = bf[0]<<8 | bf[1];
9          else
10             ret = bf[2]<<8 | bf[3];
11     }
12     else
13         ret = 0;
14 }
15 else { // do normal data conversion
```

Tabelle 26: Anpassung für analogen Sensor

Nach dem selben Prinzip wie in Tabelle 26 dargestellt, sind die fünf oben erwähnten Funktionen aufgebaut. Nachdem der Befehl an das Erweiterungsboard übertragen wurde, wird gewartet, bis eine Antwort eintrifft. Sofern die empfangene Nutzdatengröße mit der erwarteten übereinstimmt, können die Daten verarbeitet werden. Hierfür werden die Nutzdaten aus dem Paket extrahiert indem sie mit Hilfe der Funktion `getData` in einen ausreichend großen Speicherblock kopiert werden. Nach dem Aufruf dieser Funktion wird das Datenpaket zudem als bearbeitet und somit ungültig markiert, was zur Folge hat, das die Daten nicht erneut ausgelesen werden können.

```
1  void getData(uint8_t *buffer) {
2      if(uartDataValid == true)
3          memcpy(buffer,uartReceived.data,uartReceived.size); // copy data
4      uartDataValid = false; // mark packet as invalid
5  }
```

Tabelle 27: Nutzdaten eines Paketes kopieren

Eine Ausnahme bilden die Funktionen `sendMessage` und `setIdentity`. Bei diesen beiden Funktionen müssen die Nutzdaten vor dem Versenden des Paketes aus mehreren Variablen zusammengesetzt werden. Aus Gründen der Benutzerfreundlichkeit werden diese den Funktionen über mehreren Parameter mitgeteilt. Bei der Funktion `sendMessage` handelt es sich hierbei um Zieladresse der Zigbee-Nachricht, den eigentlichen Daten und deren Größe in Bytes. Die Funktion `setIdentity` hingegen erwartet eine 16-Bit große ID sowie einen Zeiger auf ein mindestens 4 Byte großes Array, welches den zur ID gehörenden Farbcode enthält. Außer der Funktion `setMsgMode` sind dies die beiden einzigen Funktionen, welche keine Antwort auf ihren Befehl erwarten.

```
1  void sendMessage(uint16_t dest, uint8_t *data, uint8_t length) {
2      uint8_t *bf = malloc(length+2); // allocate memory
3      memcpy(&bf[2],data,length);     // copy data
4      bf[0] = dest>>8;                // insert destination
5      bf[1] = dest&0x00FF;
6      uartSendCmd(CMD_SENDMSG,bf,length+2); // send command
7      free(bf);
8  }
```

Tabelle 28: Zigbee-Nachricht senden

```
1  void setIdentity(uint16_t addr, char *colour) {
2      uint8_t data[6];
3      data[0] = addr>>8;     // copy ID
4      data[1] = addr&0x00FF;
5      data[2] = colour[0];   // copy colour
6      data[3] = colour[1];
7      data[4] = colour[2];
8      data[5] = colour[3];
9      uartSendCmd(CMD_SETIDENTITY,data,6); // send command
10 }
```

Tabelle 29: Bot Identität festlegen

5.7 Motoren

Die Motoren werden, wie in Kapitel 3.4 beschrieben, über ein pulsweiten-moduliertes Signal gesteuert. Die beiden hierfür zuständigen Anschlüsse sind mit den Ausgängen einer der drei Timer im Mikrocontroller verbunden. Es bietet sich somit an, die Generierung des PWM-Signals von dem entsprechenden Timer übernehmen zu lassen.

```
1  void motor_init() {
2      DDRC |= (1<<PC6) | (1<<PC7);
3      DDRD |= (1<<PD4) | (1<<PD5);
4
5      // Clear OC1A/OC1B on compare, set at BOTTOM
6      TCCR1A |= (1<<COM1A1) | (1<<COM1B1);
7      TCCR1A |= (1<<WGM12) | (1<<WGM10); // 8-Bit Fast PWM
8
9      // Set compare-register
10     OCR1AL = 20;
11     OCR1BL = 20;
12
13     g_motorMaxPower = 255;
14     g_motorMinPower = 0;
15     motor_enable();
16 }
```

Tabelle 30: motor_init-Funktion

Im Gegensatz zu den beiden anderen Timern verfügt der verwendete über einen erweiterten Funktionsumfang. So kann u.a. zwischen einem 8, 9 und 10-Bit Zähler mit „Phase Correct PWM“, „Fast PWM“ oder „Phase and Frequency Correct PWM“ gewählt werden. Zusätzlich ist es möglich einen Frequenzteiler einzusetzen, um den Timer zu verlangsamen. Zur Ansteuerung der Motoren fiel die Wahl auf den 8-Bit „Fast PWM“ Modus. Fast-PWM bedeutet, dass der Timer von 0 bis zum max. möglichen Wert zählt, in diesem Fall 255, und anschließend wieder bei 0 beginnt. Ein Frequenzteiler (Prescaler) ist nicht erforderlich, da die Motoren durch die 8 Bit lediglich mit 16Mhz/256 = 62,5 kHz angesteuert werden, was akzeptabel ist.
Der Timer wird so konfiguriert, dass er bei jedem Null-Durchgang beide Ausgänge auf „High“ setzt. Jedes mal, wenn der Zähler sich erhöht, wird der aktuelle Wert mit denen der beiden Vergleichsregister OCR1AL und OCR1BL verglichen. Sollte eines der beiden mit dem Zählerregister übereinstimmen, wird der entsprechende Ausgang auf „Low“ geschaltet.

Neben der Geschwindigkeit soll auch die Laufrichtung der Motoren eingestellt werden können. Dies geschieht über zwei weitere Ausgänge des Mikrocontrollers.
Hierbei muss jedoch beachtet werden, dass bei gleicher Ansteuerung einer der beiden Motoren in die entgegengesetzte Richtung läuft. Damit beide in die selbe Richtung drehen, muss einer der Ausgänge also immer den invertierten Wert des anderen enthalten.

Um eine einfache Steuerung zu gewährleisten, wird der Zugriff auf die Motoren über mehrere Funktionen möglich sein. Zum einen erlaubt die Funktion `setMotorPwm` einen direkten Zugriff auf die beiden Vergleichsregister des Timers, zum anderen steht mit der Methode `setMotorPower` die Möglichkeit zur Verfügung, die Geschwindigkeit beider Motoren in einem Bereich von -100% bis +100% festzulegen. Hierbei wird durch die Prozentzahl gleichzeitig die Geschwindigkeit und die Richtung angegeben. Negative Prozentzahlen veranlassen den Motor rückwärts zu drehen, positive hingegen vorwärts. Zusätzlich kann der einstellbare Geschwindigkeitsbereich der Motoren über die Funktionen `setMotorMinPower` und `setMotorMaxPower` begrenzt werden.

```
1  void setMotorPower(uint8_t motor, int8_t power) {
2      // Drive backward?
3      if(power < 0) {
4          setMotorRichtung(motor,MOTOR_BACKWARD);
5          power *= -1; // Invert power for better calculating
6      }
7      else
8          setMotorRichtung(motor,MOTOR FORWARD);
9
10     // calculate new pwm value
11     if(motor == MOTOR_LEFT) {
12         OCR1AL = g_motorMinPower+
13                  ((uint16_t)(g_motorMaxPower-g_motorMinPower)*power/100);
14     }
15     else if(motor == MOTOR_RIGHT) {
16         OCR1BL = g_motorMinPower+
17                  ((uint16_t)(g_motorMaxPower-g_motorMinPower)*power/100);
18     }
19     if(power > 0)
20         motor_enable();
21 }
```

Tabelle 31: setMotorPower Funktion

5.8 LCD

Das LCD des c't-Bot ist aufgrund seiner Ansteuerung eines der kritischeren Elemente. Es ist, wie bereits beschrieben, über einen Seriell-Parallel Wandler mit dem Mikrocontroller verbunden. Die Daten des Displays müssen somit alle seriell ausgegeben werden, was sich erheblich in der Ablaufgeschwindigkeit der Software bemerkbar macht. Zudem muss beachtet werden, dass beim Schreiben in das Shiftregister die Daten mit dem MSB beginnend übertragen werden müssen.

```
1  for(i=0;i<8;i++) {
2      if(((cmd>>7)&0x01)==1)
3          PORTC |= (1<<PC0);
4      else
5          PORTC &= ~(1<<PC0);
6      PORTC |= (1<<PC1);
7      cmd = cmd<<1;
8      PORTC &= ~(1<<PC1);
9  }
```

Tabelle 32: LCD Shiftregister Quellcode

Das Display unterscheidet zwischen zwei Arten von Daten: Befehle und normale Daten. Mit Hilfe von Befehlen lassen sich Einstellungen in dem Display vornehmen, normale Daten hingegen werden im DDRAM (Display Data Ram) gespeichert und dienen zur Anzeige. Die Position, an der die Daten im DDRAM gespeichert werden, legt gleichzeitig ihre Ausgabeposition auf dem Display fest (Abb. 38).

DISPLAY POSITION

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
FIRST LINE	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13
SECOND LINE	40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F	50	51	52	53
THIRD LINE	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F	20	21	22	23	24	25	26	27
FOURTH LINE	54	55	56	57	58	59	5A	5B	5C	5D	5E	5F	60	61	62	63	64	65	66	67

DDRAM ADDRESS

Abbildung 38: DDRAM-Übersicht

Nach dem Einschalten der Stromversorgung benötigt das Display bis zu 40ms zum Initialisieren. Bevor Daten angezeigt werden können, müssen einige Einstellungen im Display vorgenommen werden. Als erstes muss die Anzahl der Zeilen, sowie die Zeichenhöhe festgelegt werden. Anschließend, kann der Befehl zum Einschalten des Displays und des Cursors gesendet werden. Zum Abschluss des Startvorgangs, ist der Inhalt des DDRAMs zu löschen und die Bewegungsrichtung des Cursors festzulegen. Abb. 39 verdeutlicht diesen Ablauf.

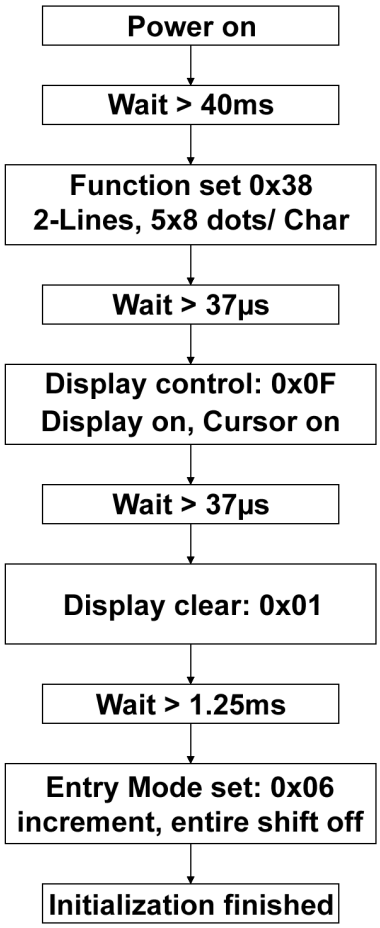


Abbildung 39: LCD-Initialisierungsablauf

Zur Ausgabe auf dem Display wird eine Funktion implementiert, die die selben Parameter akzeptiert wie das aus dem PC-Bereich bekannte *printf*. Das bedeutet die Funktion akzeptiert eine Zeichenkette, welche einen beliebigen Text und Platzhalter für die formatierte Ausgabe von Variablen enthält. Als zweiter Parameter dient eine sogenannte Parameterliste, die es ermöglicht, beliebig viele Variablen zu übergeben. Mit Hilfe dieser Funktion wird das Anzeigen von Variablen wesentlich vereinfacht.

5.9 LEDs

Die 8 LEDs sind ebenfalls über eines der Shiftregister mit dem Mikrocontroller verbunden. Daraus folgt, dass selbst für das An- oder Abschalten von nur einer LED immer ein ganzes Byte in das Shiftregister geladen werden muss. Für die softwaretechnische Umsetzung bedeutet dies, dass der aktuelle Status aller 8 LEDs entweder in einer globalen oder statischen Variablen zwischengespeichert werden muss. Andernfalls würde nur die jeweilige LED angeschaltet sein.
Da für jede LED nur der Zustand *on* oder *off* gespeichert werden muss, bietet es sich an dies in einer einfachen 8-Bit Variable zu tun. Hierbei repräsentiert jedes Bit den Zustand einer LED.
Für die Initialisierung müssen lediglich die drei Ausgänge, welche zur Steuerung des Shiftregisters nötig sind, festgelegt werden. Zusätzlich sollten noch alle LEDs abgeschaltet werden, da es passieren kann, dass einige noch von einer älteren Software angeschaltet waren, oder sich während der Programmierung aktiviert haben.

```
1  void initLeds() {
2      // Set outputs for data, storage clock, serial shift clock
3      DDRC |=  (1<<PC0) | (1<<PC1) | (1<<PC4);
4
5      g_leds = 0x00;     // All LEDs off
6
7      ledshift(g leds); // shift out
8  }
```

Tabelle 33: LEDs initialisieren

Um eine oder mehrere LEDs an- oder abzuschalten, ist es nun nur noch nötig, das entsprechende Bitmuster mit der globalen Variable `g_leds` zu verbinden und diese erneut in das Shiftregister zu übertragen. Damit es während der Übertragung nicht zu Problemen bei einem Prozesswechsel kommt, sollte dieser vor Beginn des Transfers unterdrückt werden.

```
1  void setLed(uint8_t nr) {
2      syslock();         // disable taskswitch
3      g leds |= nr;      // mark led
4      ledshift(g leds); // refresh leds
5      sysunlock();       // enable taskswitch
6  }
7
8  void clearLed(uint8_t nr) {
9      Syslock();         // disable taskswitch
10     g_leds &= ~nr;     // mark led
11     ledshift(g_leds); // refresh leds
12     Sysunlock();       // enable taskswitch
13 }
```

Tabelle 34: LEDs an-/abschalten

Die serielle Übertragung zwischen Mikrocontroller und Shiftregister läuft ähnlich ab wie bei dem Aktivieren und Deaktivieren der Sensoren. Die einzelnen Bits der 8-Bit Variablen werden mit dem MSB beginnend an den Dateneingang des Shiftregisters (PC0) angelegt und durch eine steigende Flanke am „*Shift-Register-Clock*“ (PC4) in das Shiftregister übertragen. Nachdem alle 8 Bit übertragen wurden, wird der „*Storage-Register-Clock*“ Pin (PC1) für kurze Zeit auf „*High*“ geschaltet und dem IC somit signalisiert, dass es die gespeicherten Daten ausgeben kann.

```
1  void ledshift(uint8 t value) {
2      uint8_t i;
3      for(i=0;i<8;i++) {
4          // Set data
5          if( ((value>>7)&0x01)==1 )
6              PORTC |= (1<<PC0);
7          else
8              PORTC &= ~(1<<PC0);
9
10         // serial-shift clock
11         PORTC  |= (1<<PC4);
12         value = value<<1;
13         PORTC  &= ~(1<<PC4);
14     }
15     // storage clock
16     PORTC |= (1<<PC1);
17     asm volatile ("nop");
18     PORTC &= ~(1<<PC1);
19 }
```

Tabelle 35: Shift-Methode für LEDs

5.10 Servos

Der c't-Bot ist für die Steuerung von maximal 2 analogen Servos ausgelegt. Die Position die ein Servo anfahren soll, wird diesem mit Hilfe eines PWM-Signals mitgeteilt. Die Frequenz des PWM-Signals beträgt typischerweise 50Hz, was eine Periodendauer von 20ms entspricht. Modernere Analog-Servos sowie Digital-Servos sind jedoch in der Lage sich mit höheren Frequenzen ansteuern zu lassen, wodurch der Servo schneller reagieren kann. Der Stellwinkel des Servos wird über den „High“-Teil des Signals angegeben. Der minimale Servoausschlag wird erreicht, wenn der „High“-Teil ca. 1ms andauert. Der maximale Servoausschlag hingegen liegt bei ca. 2ms (Abb. 40). Je nach Bauart und Stellbereich können diese Werte etwas variieren.

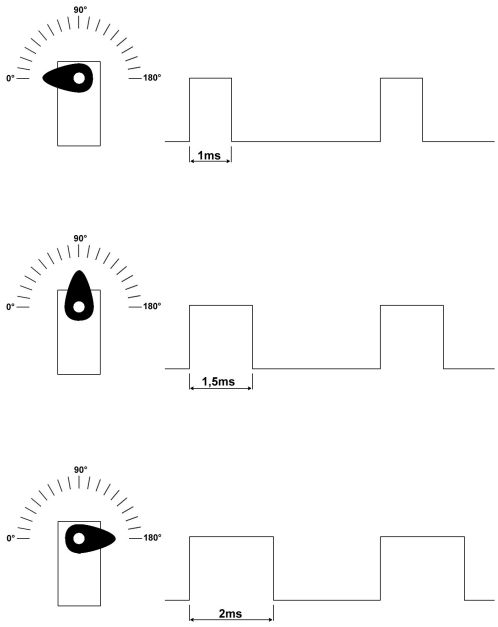


Abbildung 40: Servo-Timing

In der offiziellen Software ist es vorgesehen, dass für jeden Servo ein eigener Timer im Mikrocontroller zur Verfügung steht, welcher das Timing des PWM-Signals übernimmt. Da jedoch bereits ein Timer für die Steuerung der Motoren verwendet wird und ein weiterer mit dem Systemtimer belegt ist, müssen sich beide Servos den verbliebenen 8-Bit Timer teilen.
Eine Möglichkeit ist, den Timer in einem festen Intervall einen Interrupt generieren zu lassen. In dessen ISR könnten die Ausgangsspins des Mikrocontrollers je nach Bedarf gesetzt oder gelöscht werden. Um den vollen Stellbereich des Servos abzudecken, muss der Timer in der Lage sein, einen Bereich zwischen 1ms und 2ms möglichst genau abzudecken. Das heißt, um den Stellbereich in 256 Schritten zu erfassen, darf zwischen zwei ISR Aufrufen maximal 2ms/256Schritte = 7,81µs liegen, was einer Frequenz von ca. 128kHz entspricht. Eine niedrigere Frequenz würde zu einer längeren Schrittdauer führen und somit die Auflösung verringern, mit welcher der Drehwinkel des Servos eingestellt wird. Durch den kurzen Abstand, in den die ISR aufgerufen werden müsste, schließt sich diese Art der Implementation schon jetzt aus. Bei dieser kurzen Zeitspanne bliebe kaum Rechenzeit für das eigentliche Programm übrig, zudem würde es durch die anderen Interruptquellen immer wieder zu Überschneidungen kommen.

Eine weitaus performantere Lösung ist es, den Timer nur dann einen Interrupt generieren zu lassen, wenn ein Wechsel der Ausgangspins am Mikrocontroller nötig ist. Hierbei bietet sich der „Clear Timer on Compare Match (CTC) Mode“ des Timers an. Bei diesem wird das Zählerregister des Timers bei jedem Takt mit einem Vergleichsregister verglichen. Stimmen die Werte beider überein, wird ein Interrupt ausgelöst und der Timer beginnt wieder bei 0 mit zählen.
Bedingt durch den Prescaler des Timers bietet sich als optimale Frequenz ein Takt von 125kHz an. Bei diesem würde der Timer eine maximale Dauer von 2,048ms zwischen zwei Interrupts erreichen. Dies entspräche bei 2ms einen Zählerwert von 250.

So lange der Steuerimpuls der Servos jeweils alle 20ms erneuert wird, ist es möglich erst den einen Servo anzusteuern und anschließend den zweiten. Im Anschluss muss nur noch gewartet werden, bis die 20ms um sind und der erste Servo einen neuen Impuls benötigt.
Die Interruptroutine des Timers könnte demnach den Ausgangspin des ersten Servos auf „High“ setzen und den Timer so einstellen, dass der nächste Interrupt erst auftritt, wenn die Zeit, die der Impuls dauern soll, abgelaufen ist. Sobald nun die Interruptroutine erneut aufgerufen wird, wird der Ausgang wieder auf „Low“ gesetzt und der Timer dahingehend geändert, dass diesmal bis zum nächsten Interrupt eine Zeit von 2ms minus der „High“-Zeit des Impulses vergeht. Dasselbe geschieht bei dem zweiten Servo. Um die 20ms Dauer einzuhalten muss der Timer anschließend noch weitere 8 mal jeweils alle 2ms einen Interrupt auslösen. Somit erreicht man eine optimale Genauigkeit mit möglichst wenig Verbrauch an Speicher und Rechenzeit. Das Aktivitätsdiagramm 41 auf der folgenden Seite veranschaulicht diesen Ablauf dieses Programmteils.

```
1  ISR(SIG_OUTPUT_COMPARE2) {
2      switch(g_servoindex) {
3          case 0: PORTB |= (1<<PB3); break;  // Set Servo 1 High
4          case 1: PORTB &= ~(1<<PB3); break;// Set Servo 1 Low
5          case 2: PORTD |= (1<<PD7); break;  // Set Servo 2 High
6          case 3: PORTD &= ~(1<<PD7); break;// Set Servo 1 Low
7      }
8      // Set Compare Register
9      if(g_servoindex < 4)
10         OCR2 = g_servoPulse[g_servoindex];
11     else
12         OCR2 = SERVO_MAXTIME; // 2 ms
13     g_servoindex = (g_servoindex+1)%12;
14 }
```

Tabelle 36: ISR des Servo-Timers

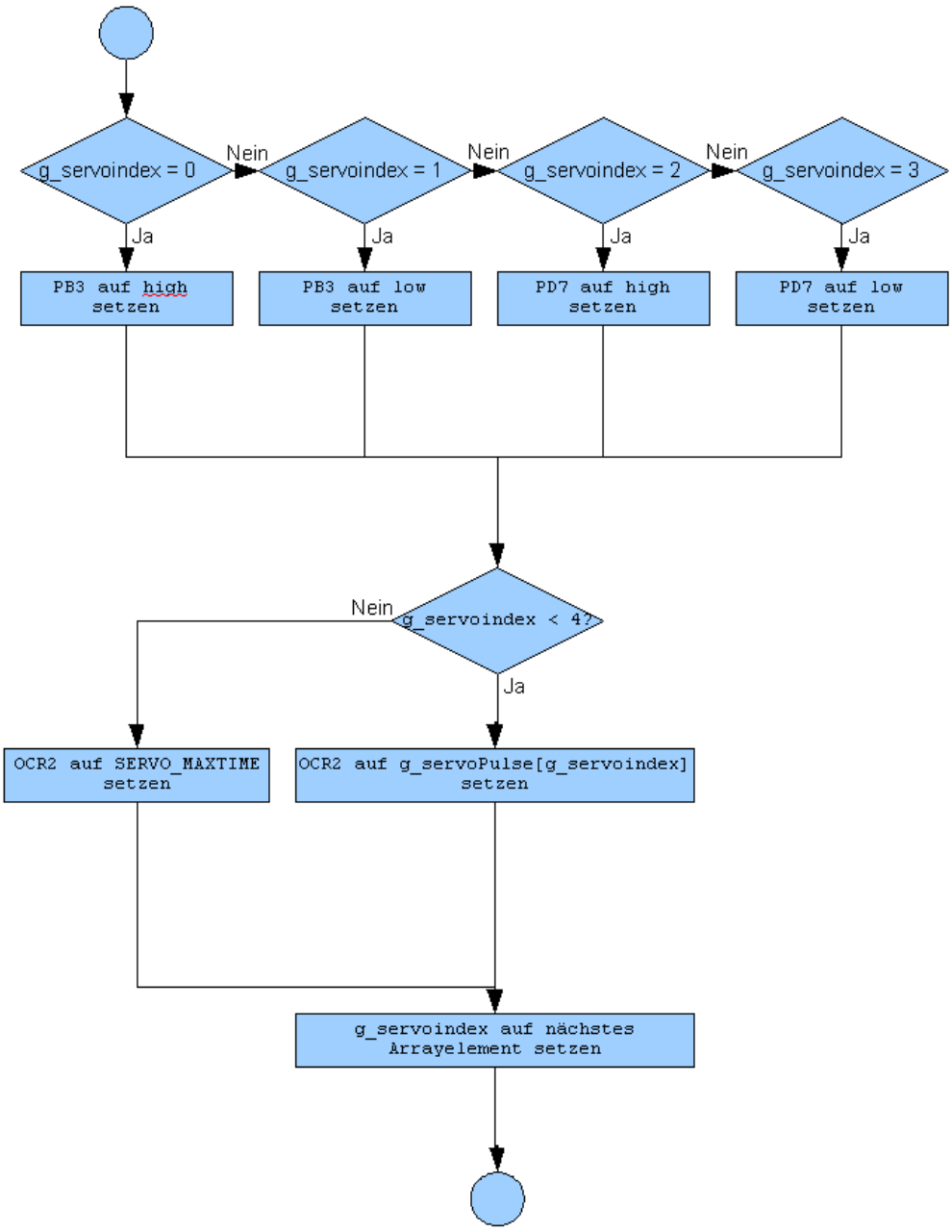


Abbildung 41: Sequenzdiagram der Servo-Interruptroutine

6. Tests

6.1 Modultests

Zur Auswertung aller Modultests ist ein optische, akustische oder Computer gebundene Ausgabe erforderlich. Da der c't-Bot mit dem Display, den LEDs und der seriellen Schnittstelle bereits drei mögliche Arten integriert hat, bietet es sich an, deren Funktionalität zu Beginn der Tests sicherzustellen und zur Ausgabe zu verwenden. Die vollständige Funktionalität der LEDs lässt sich durch einfaches An-/Abschalten aller LEDs validieren.

```
1  void ledTest(void) {
2      uint8_t i;
3      for(i=0;i<8;i++) {
4          setLed((1<<i));
5          delay(5000);
6          clearLed((1<<i));
7          delay(5000);
8      }
9  }
```

Tabelle 37: LED-Funktionalität validieren

Ähnlich unkompliziert ist auch Überprüfung des Displays. Bei diesem wird ein Text jede Sekunde um eine Zeile und ein Zeichen weiter geschoben. Hierdurch kann sehr leicht erkannt werden, ob eine der Funktionen nicht wie erwartet arbeiten sollte.

```
1  void lcdTest(void) {
2      uint8_t i;
3      for(i=0;i<4;i++) {
4          lcdclear();
5          lcdsetcursor(i,i);
6          lcdprintf("Test %d, 0x%x, %.2f",i,i,(float)i);
7          delay(10000);
8      }
9  }
```

Tabelle 38: LCD-Funktionalität validieren

Die Prüfung der seriellen Schnittstelle erfordert einen etwas komplexeren Aufbau der Testumgebung. Dies liegt einerseits an dem erweiterten Funktionsumfang, der neben den allgemeinen Funktionen auch die speziell auf das Erweiterungsboard angepassten umfasst. Auf der anderen Seite muss für die Überprüfung der allgemeinen Funktionen der c't-Bot mit einem PC verbunden werden. Dies geschieht entweder direkt über die serielle Schnittstelle oder, da viele moderne PC keine serielle Schnittstelle mehr besitzen, mit Hilfe eines Seriell-USB Wandlers über die USB Schnittstelle. Durch die standardmäßig eingestellt Übertragungsgeschwindigkeit von 19200 Baud kann auf Seiten des PCs jede Art von Terminal Programm zur Kommunikation verwendet werden.

Bei diesen Test wurde das Programm HTerm 0.8.1beta gewählt, welches sowohl für Windows als auch für Linux verfügbar ist [37].
Die Tests der allgemeinen Funktionen starten mit dem Versuch, einige Daten an den PC zu senden. Sollte dies erfolgreich sein, werden der Reihenfolge nach die vier Empfangsfunktionen `uartgetc`, `uartreadc`, `uartreads`, `uartgets` überprüft. Hierfür wird eine bestimmte Anzahl Zeichen an den Mikrocontroller geschickt, aus dem Zwischenspeicher ausgelesen und wieder an den PC übertragen. Auf diese Weise lässt sich sicherstellen, dass alle Funktionen korrekt auf den Zwischenspeicher zugreifen und die Daten fehlerlos verarbeiten.
Die Überprüfung der Funktionen für die Kommunikation mit dem Erweiterungsboard ist aufgrund des Zusammenhangs zwischen den Funktionen komplexer.
Durch Übertragen eines Datenpakets an den PC wird sichergestellt, dass die Übertragung vom Mikrocontroller zum Erweiterungsboard fehlerfrei möglich ist,
So liefert der Aufruf der Funktion `uartSendCmd(0xAACC, "Test", 4)` bei fehlerfreier Implementierung auf Seiten des PCs den im oberen Teil von Abb. 42 dargestellten Datenstring.

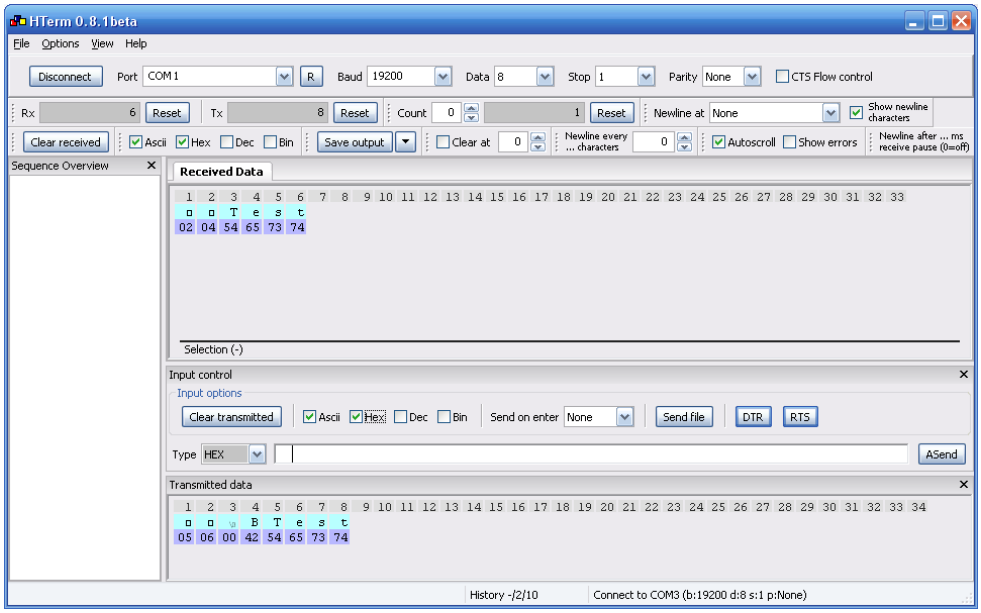


Abbildung 42: UART-Test HTerm Ausgabe

Der Empfang von Daten des Zigbee-Boards wird ebenfalls durch den PC simuliert. Hierfür wird ein Datenpaket wie im unteren Teil von Abb. 42 zu sehen ist, an den Mikrocontroller übertragen. Dieser wertet dieses aus und gibt alle extrahierten Daten auf dem LCD an. Somit erscheint bei fehlerfreier Implementierung die folgende Ausgabe auf dem Display:

CMD: 5

SIZE: 6

DATA: BTest

Nachdem nun sichergestellt ist, dass zur Ausgabe der weiteren Testergebnisse drei unterschiedliche Darstellungsarten zur Verfügung stehen, können die analogen und digitalen Sensoren sowie die Motoren und Servos überprüft werden.

Das Abfragen der analogen Sensoren lässt sich mit relativ wenig Aufwand realisieren. Da diese an dem Analog-Digital-Wandler angeschlossen sind, brauchen lediglich alle 8 Anschlüsse einmal ausgewertet werden.

```
1  for(i=0;i<8;i++) {
2     lcdclear();
3     lcdprintf("Sensor %d: %d",i,getAnalog(i));
4     delay(10000);
5  }
```

Tabelle 39: Analoge Sensoren auswerten

Die Prüfung der digitalen Sensoren hingegen gestaltet sich aufgrund der unterschiedlichen Anschlüsse komplexer. Hier muss jeder Sensor separat angegeben werden. Zu den digitalen Sensoren zählen neben jenen, die über die Funktion `getDigital` abgefragt werden können, der optische Maussensor. Da dieser vorzeichenbehaftete Daten liefert, besitzt er eigene Funktionen zum Auslesen. Damit das Verhalten der Sensoren bei Änderungen der Umwelt überprüft werden kann, laufen die Tests über einen Zeitraum von mehreren Sekunden und aktualisieren die jeweiligen Sensorwerte mehrmals in der Sekunde.

Damit die korrekte Funktionsweise der Motoren sichergestellt werden kann, wird eine Abfolge unterschiedlicher Fahrmanöver mit dem Roboter durchgeführt und mit den zu erwartenden Ergebnissen verglichen. Dies umfasst das Geradeausfahren gefolgt von einer Links- und Rechtsdrehung und dem abschließenden Vorwärtsfahren mit halber Geschwindigkeit, bei dem die Funktionen `setMotorMinPower` sowie `setMotorPwm` verwendet werden.
Um die sichere Ansteuerung der Servos zu gewährleisten, muss der Timer in der Lage sein in Zeitabständen von 1mS bis 2mS Interrupts zu generieren. Dies wird überprüft, indem beide Servos zweimal über ihren gesamten Wertebereich gedreht werden. Beim ersten Mal im vollen Bereich zwischen 0 und 255, beim zweiten Mal werden sie jedoch auf einen Bereich von +/- 10 Schritten um die Mittelstellung begrenzt. Somit ist gleichzeitig sichergestellt, dass die minimale und maximale Servoposition korrekt berechnet wird.

```
1  k=0;
2  do {
3      for(i=0;i<255;i++) {
4          setServoPos(SERVO1,i);
5          setServoPos(SERVO2,i);
6          lcdprintf("Servopos. %d",i);
7          delay(500);
8      }
9      setMinServoPos(SERVO1,117);
10     setMinServoPos(SERVO2,117);
11     setMaxServoPos(SERVO1,137);
12     k++;
13 }while(k<2);
```

Tabelle 40: Servo-Test

6.2 Integrationstests

Die Integrationstests beschäftigen sich mit dem Zusammenspiel zwischen den einzelnen Komponenten. Im speziellen dienen diese Tests dazu, das Multitasking-System sowie die zeitabhängigen Komponenten, wie den RC5-Decoder oder die Radencoder zu überprüfen. Zusätzlich wird die Kommunikation mit dem Erweiterungsboard auf einer höheren Abstraktionsebene geprüft.

Bevor das Multitasking-System getestet werden kann, muss sichergestellt sein, dass der entsprechende Timer korrekt läuft und einen Interrupt auslöst. Dies lässt sich durch einfaches Anzeigen der Systemticks überprüfen. Sobald diese sich wie erwartet erhöhen, kann mit dem eigentlichen Überprüfen des Systems begonnen werden. Hierfür wird in dem Hauptprozess `ctMain` ein weiterer Prozess erzeugt, welcher eine der LEDs aktiviert. Diese wird anschließend in dem Hauptprozess wieder deaktiviert. Hierdurch erhält man eine LED, welche je nach Ausführungsdauer unterschiedlich hell leuchtet oder blinkt.

```
1  void led(void) {
2      while(1)
3          setLed(LED1);
4  }
5  void multitaskTest(void) {
6      createTask(led,100);
7      while(1)
8          clearLed(LED1);
9  }
```

Tabelle 41: Multitasking Test

Durch den Timer wird neben dem Multitasking-System auch das Dekodieren von RC5-Signalen gesteuert, somit sollte dies nun ebenfalls ohne Probleme funktionieren. Dazu wird mit einer RC5 tauglichen Fernbedienung Signale an den c't-Bot gesendet und die empfangenen Befehle auf dem Display angezeigt. Dadurch kann leicht verglichen werden, ob die empfangenen Daten mit den erwarteten übereinstimmen.

```
1  while(1) {
2      lcdclear();
3      lcdprintf("CMD: %d",getRC5Cmd());
4      lcdsetcursor(1,0);
5      lcdprintf("ADDR: %d",getRC5Addr());
6      lcdsetcursor(2,0);
7      lcdprintf("TOGGLE: %d",getRC5Toggle());
8      delay(5000);
9  }
```

Tabelle 42: RC5 Funktionsprüfung

Ähnlich wird bei der Kontrolle der Radencoder verfahren. Diese ändern ihren Wert bei jeder Bewegung des Roboters, daher reicht als Funktionskontrolle, den c't-Bot in Bewegung zu versetzen und die aktuellen Encoderwerte anzeigen zu lassen.

Die Kontrolle der Kommunikationsfunktionen für die Verbindung zum Erweiterungsboard gestaltet sich aufwändiger als die vorherigen Tests. Bei diesem Test wird der Mikrocontroller jedoch mit dem Erweiterungsboard statt mit einem PC verbunden. Dies ist erforderlich um kontrollieren zu können, ob Nachrichten korrekt von dem Erweiterungsboard verarbeitet werden. Für den Empfang von Zigbee-Nachrichten wird ein funktionstüchtiger Sender benötigt. Bei diesem Test soll ein zweiter c't-Bot als Sender dienen und kontinuierlich Nachrichten verschicken. Damit dies möglich ist, muss zu Beginn sichergestellt werden, dass die Funktion `sendMessage` ohne Fehler abläuft. Bei dem Test wird eine Zigbee-Nachricht an die Adresse 0xAACC geschickt. Die gesendeten Nachrichten lassen sich von einem PC mit angeschlossenem Zigbee-Sniffer aufzeichnen und grafisch analysieren (Abb. 43). Auf diese Weise ist es ein Leichtes, die Funktionalität der Funktion sicherzustellen. Anschließend kann ein zweiter Roboter mit einem angepassten Programm versehen werden, so dass dieser in kontinuierlichen Zeitabständen Nachrichten verschickt.
Mit diesen ist es jetzt möglich, die Funktionen `getMessage` und `setIdentity` zu testen. Zum Abschluss der Integrationstests wird der Zugriff auf die im Erweiterungsboard gespeicherten Koordinaten des Roboters sowie die beiden, ebenfalls dort befindlichen, Sensoren überprüft.

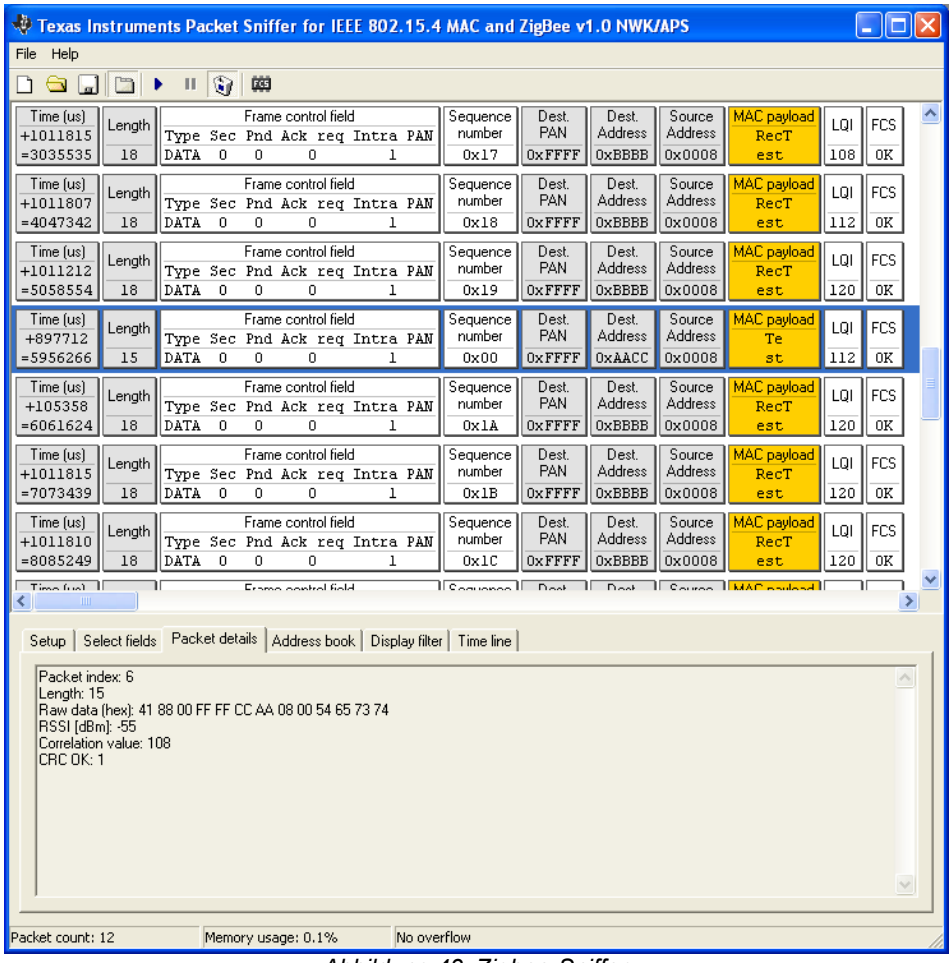


Abbildung 43: Zigbee-Sniffer

6.3 Performanztests

Für die einwandfreie Funktionalität des Systems ist es unabdingbar, dass wichtige Komponenten strenge Zeitvorgaben einhalten. Zu diesen zählen neben dem RC5-Dekoder und dem Scheduler des Multitasking-Systems unter anderem auch die Ansteuerung der Servos und die Kommunikation über die Shiftregister. Da jedoch keine Möglichkeit besteht, Zeitmessungen, die sich in einem Bereich unter 100µS bewegen, direkt auf einem der Roboter auszuführen, muss die Mehrheit der Tests rechnerisch erfolgen. Als Hilfe dient hierbei der integrierte Simulator des AVR-Studios. Mit diesem lässt sich die Anzahl an Taktzyklen zwischen zwei Punkten im Programmcode messen. Multipliziert man diese Zahl mit der Zeit, die der Mikrocontroller für einen Takt benötigt, ergibt dies die Ausführungszeit eines bestimmten Programmabschnitts.
So benötigt z.B. die in Abb. 44 gemessene Bearbeitung des RC5-Signals in der Systemtimer-ISR 25 Takte. Multipliziert mit 62,5nS, der Zeit für einen Takt, ergibt sich für diesen Abschnitt eine Ausführungszeit von nur 1,56µS. Der Scheduler benötigt mit 2 Prozessen hingegen je nach aktivem Prozess bereits zwischen 288 und 291 Takte, was einer Dauer von 18,00µS – 18,19µS entspricht. Für die Abarbeitung der kompletten ISR des Systemtimers zusammen mit dem RC5-Dekoder, der Abfrage beider Radencoder und einem Prozesswechsel sind sogar 26,43µS notwendig. Sollte kein Prozesswechsel stattfinden, benötigt derselbe Code mit knapp 13µS lediglich die Hälfte der Zeit. Bedingt durch die Konfiguration des Timers gehört diese Zeit bereits mit zur der, die dem Prozess zur Ausführung zustehen würde. Somit bleiben einem Prozess zwischen zwei Interrupts je nach Situation 73,57µS bis 87µS für die Bearbeitung seiner Aufgaben. Von dieser Zeit wird noch zusätzlich in unregelmäßigen Abständen zwischen 19,3µS und 19,7µS für die Bearbeitung der Servos benötigt. Daher empfiehlt es sich diese zu deaktivieren sofern sie nicht benötigt werden.
Als letzter kritischer Abschnitt wird die serielle Übertragung zu den Shiftregistern getestet. Stellvertretend für alle 3 Register wurde die Kommunikation mit dem Maussensor ausgewählt. Um 1 Byte an den Sensor zu übertragen werden 8µS benötigt. Das Auslesen eines Registers des Sensors wiederum zieht sich über 109,25µS hin. Dies liegt allerdings zum Hauptteil an der Zeitspanne, die der Sensor zum Bearbeiten und Bereitstellen der Daten benötigt.

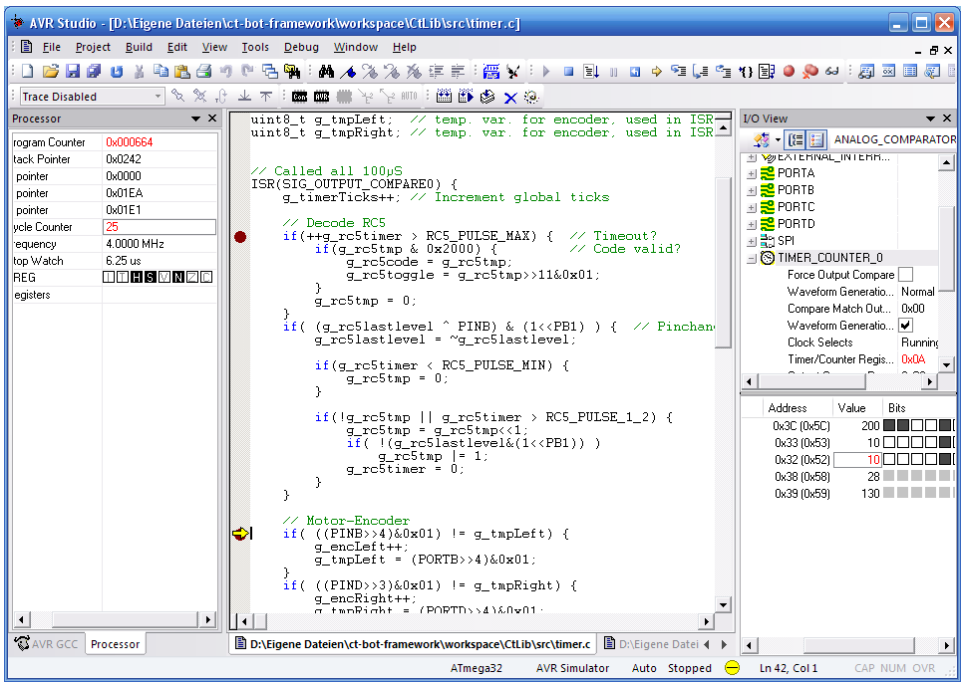


Abbildung 44: Performanztest RC5-Dekoder

7. Fazit

7.1 Zusammenfassung

Im Rahmen dieser Arbeit wurde eine Softwarebibliothek entwickelt, welche ein einfacheres Arbeiten mit dem c't-Bot ermöglicht.
Damit die neue Software bestmöglich auf den Roboter abgestimmt werden konnte, wurde zunächst der Aufbau und die einzelnen Hardwarekomponenten genau analysiert. Anschließend konnten die Rahmenbedingen und der Funktionsumfang bestimmt, sowie bereits vorhandene Systeme untersucht werden. Eine der Anforderungen der Bibliothek bestand darin, mehrere Prozesse parallel abarbeiten zu können. Hierfür wurden mehrere Multitasking-Kernel auf ihre Verwendbarkeit hin genauer untersucht. Da jedoch keiner den strengen Anforderungen genügte, war die Entwicklung eines eigenen, maßgeschneiderten Systems notwendig.
Nachdem dieses erfolgreich entwickelt und getestet worden war, konnte mit der Ansteuerung des Displays und den LEDs begonnen werden, da diese weitere Tests erleichtern. Nachdem zusätzlich sowohl alle Sensoren wie auch die Motoren angesteuert werden konnten, konnte die Kommunikation mit Hilfe der Zigbee-Funkmodule in Angriff genommen werden. Hierfür waren sowohl Änderungen in der Software des Erweiterungsboards wie auch in dem UART-Modul der Softwarebibliothek notwendig.
Das Ergebnis ist eine Software, welche weniger als 30% des vorhandenen Programmspeichers und unter 10% des Datenspeichers benötigt.
Mit Hilfe der Bibliothek ist es möglich, sowohl einfache Programme, wie auch komplexe Systeme mit mehreren Prozessen entwickeln zu können und über das vorhanden Zigbee-Modul Daten mit anderen Geräten auszutauschen. Somit kann diese Software das Standard-Framework des c't-Bots an der HAW als Entwicklungsgrundlage ablösen.

7.2 Kritik

Durch die übermäßige Beanspruchung der letzten Semester waren viele Roboter nicht mehr in bester Verfassung, was sich durch Kabelbrüche oder defekte Sensoren bemerkbar machte. Dies führte während der Implementation und den Tests mehrfach zu Problemen, insbesondere bei der seriellen Kommunikation mit dem Maussensor. Hier konnte erst durch eine mehrfache und aufwendige Simulation der Software der Sensor als Fehlerquelle ausgemacht werden. Durch Verwendung mehrerer Roboter konnten dieser und weitere Hardwarefehler jedoch relativ elegant umgangen werden.
Während der abschließenden Tests zeigte sich jedoch, dass die Software noch einige Tücken aufweist.
So wäre es wünschenswert beim Erstellen eines Prozesses die Größe seiner Prozessor-Zeit angeben zu können. Ferner ließe sich die Kommunikation mit dem Maussensor und dem Erweiterungsboard noch optimieren. Ein endgültiger Test der Bibliothek unter Projektbedingungen wäre zwar wünschenswert gewesen, ließ sich jedoch nicht realisieren.

7.3 Ausblick

Die entwickelte Bibliothek ist in ihrem Umfang voll funktionsfähig und ermöglicht durch ihren weitgehendst modularen Aufbau zudem eine leichte Erweiterung mit neuen Funktionen.
So ließe sich eine weitere Abstraktionsebene implementieren, welche eine vereinfachte Kommunikation mit anderen Robotern ermöglicht oder ein eigenes Protokoll zur Verfügung stellt.
Ferner wäre es denkbar, das Multitasking-System zu erweitern, damit Prozesse mit unterschiedlichen Prioritäten versehen werden können.
Ein weiteres interessantes Ziel wäre es, eine Gerüst zu schaffen, das es erleichtert ein wie in Kapitel 1.5 beschriebenes Subsumption-System zu implementieren.

8. Anhang A

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[1] Aude Billard, Gillian Hayes, *DRAMA, a connectionist architecture for control and learning in autonomous robots*, 1999

[2] H. Hexmoor, D. Kortenkamp: *Issues on building software for hardware agents*, In Knowledge Engineering Review,1995

[3] Hans W. Ingensiep , Anne Eusterschulte: *Philosophie der natürlichen Mitwelt*, 2002

[4] J.S. Albus, H.G. McCain, R. Lumia: *NASREM: The NASA/NBS standard reference model for telerobot control system architecture*, In NBS Technical Note 1235,1987

[5] Joseph L. Jones, Anita M. Flynn: *Mobile Robots*, 1993

[6] Keith E. Curtis: *Embedded Multitasking*, 2006

[7] Long-Ji Lin, Reid Simmons, Christopher Fedor: *Experience with a Task Control Architecture for Mobile Robots*, In CMU-RI-TR 89-29,1989

[8] Rodney A. Brooks: *A Robot that Walks; Emergent Behavior from a Carefully Evolved Network*, In Neural Computation,1989

[9] Rodney A. Brooks: *A robust layered control system for a mobile robot*, In IEEE Journal of Robotics and Automation,1986

[10] Rodney A. Brooks: *Intelligence without reason*, In IJCAI-91,1991

[11] Rodney A. Brooks: *Intelligence without representation*, In Artificial Intelligence Journal,1991

[12] Ronald C. Arkin: *Behavior-Based Robotics*, 1998

[13] Ronald C. Arkin: *Integrating behavioral, perceptual, and world knowledge in reactive navigation*, In Robotics and Autonomous Systems,1990

[14] Ronald C. Arkin: *Path Planning for a Vision-Based Autonomous Robot*, In Proceedings of the SPIE Conference on Mobile Robots,1986

[15] Terrence W. Fong: *A Computational Architecture for Semi-autonomous Robotic Vehicles*, In AIAA Computing in Aerospace conference,1993

[16] Thomas Bräunl: *Embedded Robotics: Mobile Robot Design and Applications with Embedded Systems*, 2006

[17] 6.270 - MIT's Autonomous Robot Design Competition
http://web.mit.edu/6.270/www/

[18] 74HC595 Datenblatt
http://www.nxp.com/acrobat_download/datasheets/74HC_HCT595_4.pdf

[19] ADNS-2610 Datenblatt
http://www.avagotech.com/docs/AV02-1184EN

[20] AKSEN - Controller für reaktive Roboter
http://ots.fh-brandenburg.de/?module=pagemaster&PAGE_user_op=view_page&PAGE_id=20

[21] Asuro
http://www.rn-wissen.de/index.php/Asuro

[22] Atmega32A Datasheet
http://www.atmel.com/dyn/resources/prod_documents/doc8155.pdf

[23] ATmega644 Datenblatt
http://www.atmel.com/dyn/resources/prod_documents/doc2593.pdf

[24] C-Control
http://de.wikipedia.org/wiki/C-Control

[25] c't-Bot Hauptplatine
http://www.heise.de/ct/projekte/ct-bot/pdf/schaltplan-final.pdf

[26] c't-Bot und c't-Sim
http://www.heise.de/ct/projekte/ct-bot/ct-bot.shtml

[27] c't-Bot Wiki
http://wiki.ctbot.de

[28] CNY70 Datenblatt
http://www.vishay.com/doc?83751

[29] Der RC5-Code
http://www.sprut.de/electronic/ir/rc5.htm

[30] Die Geschichte der Roboter
http://www.chemie.de/articles/d/45201/

[31] Die geschichtliche Entwicklung von Industrierobotern
http://smerobot-tools.prospektiv.de/robotic/deu/historie.html

[32] femto OS
http://www.femtoos.org

[33] freeRTOS
http://www.freertos.org

[34] Global Hawk
http://de.wikipedia.org/wiki/Global_Hawk

[35] GP12D2 Datenblatt
http://sharp-world.com/products/device/lineup/data/pdf/datasheet/gp2d12_e.pdf

[36] HD44780U Datenblatt
http://web.media.mit.edu/~ayah/documents/hd44780u.pdf

[37] HTerm Website
http://www.der-hammer.info/terminal/

[38] Infrarot Entfernungsmesser
http://www.kreatives-chaos.com/artikel/infrarot-entfernungsmesser

[39] IR-Fernbedienungsprotokolle
http://www.mikrocontroller.com/de/IR-Protokolle.php

[40] L293D Datenblatt
http://focus.ti.com/lit/ds/symlink/l293d.pdf

[41] LCD-Modul Datenblatt
http://www.display-elektronik.de/DEM20485SBH-PW-N.PDF

[42] LEGO Mindstorms
http://mindstorms.lego.com/Overview/default.aspx

[43] Merlin Robotics
http://www.merlinrobotics.co.uk

[44] Offizielle C-Control Website
http://www.c-control.de

[45] pico] OS
http://www.picoos.sourceforge.net

[46] RoboCup
http://robocup.org/02.html

[47] Robots and their Arms
http://infolab.stanford.edu/pub/voy/museum/pictures/display/1-Robot.htm

[48] Rowley Associates
http://www.rowley.co.uk/

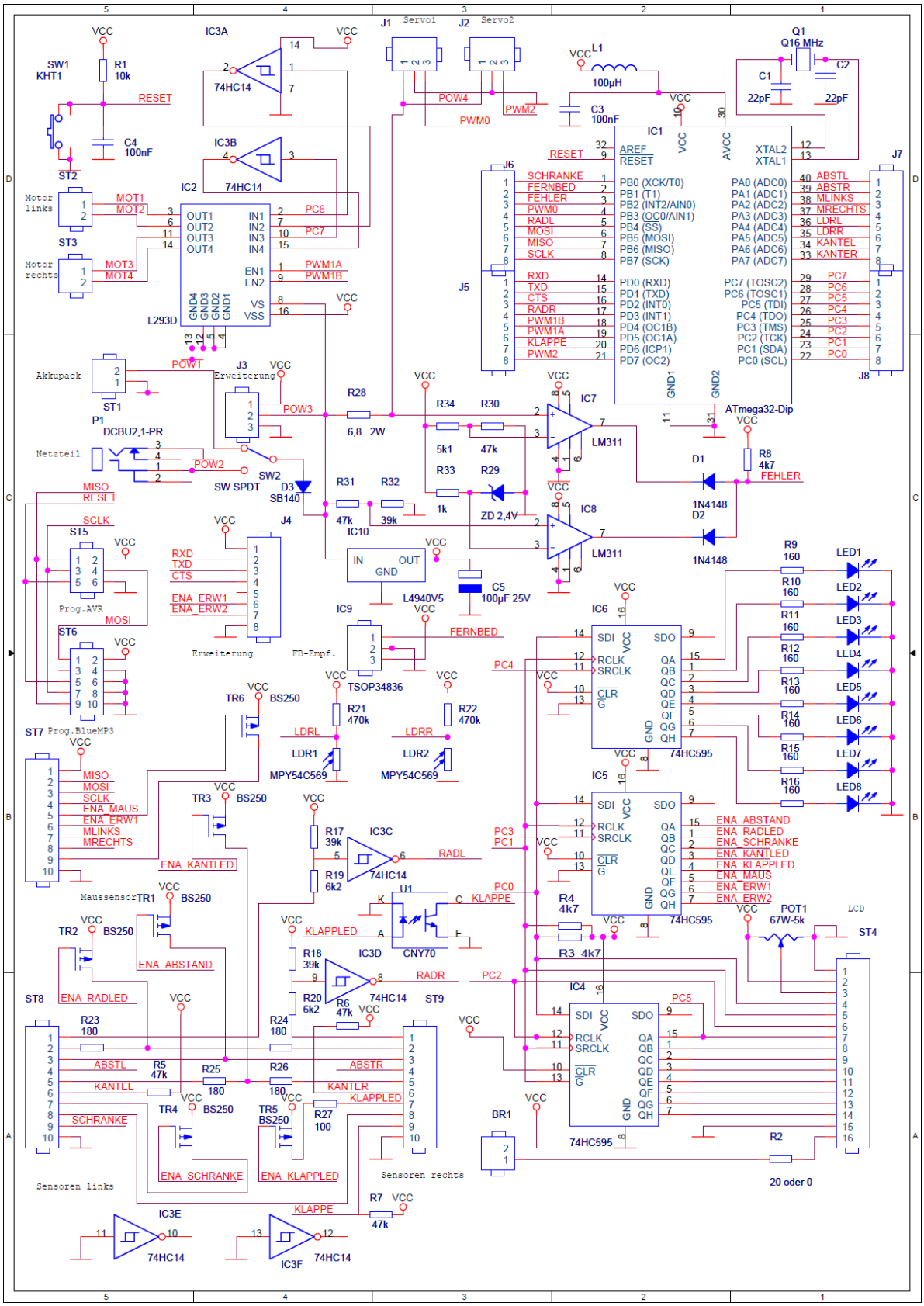
[49] RTOS for AVR
http://www.avrfreaks.net/index.php?name=PNphpBB2&file=viewtopic&t=66371

[50] The Handy Board
http://www.handyboard.com/index.html

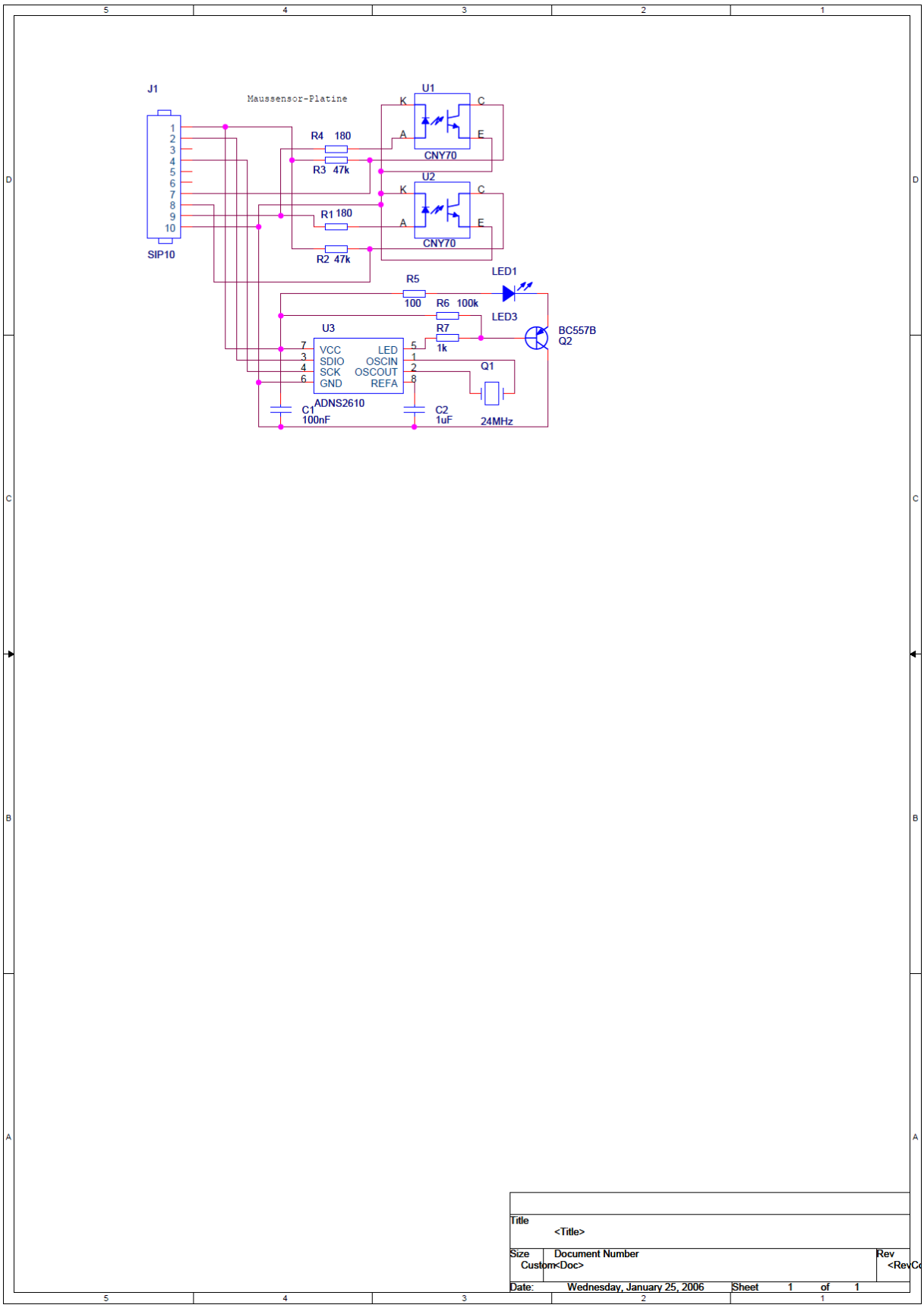
[51] TSOP 34836 Datenblatt
www.vishay.com/docs/81732/tsop348.pdf

Die Funktionstüchtigkeit aller aufgeführten Internetseiten wurde am 2. Juni 2009 überprüft.

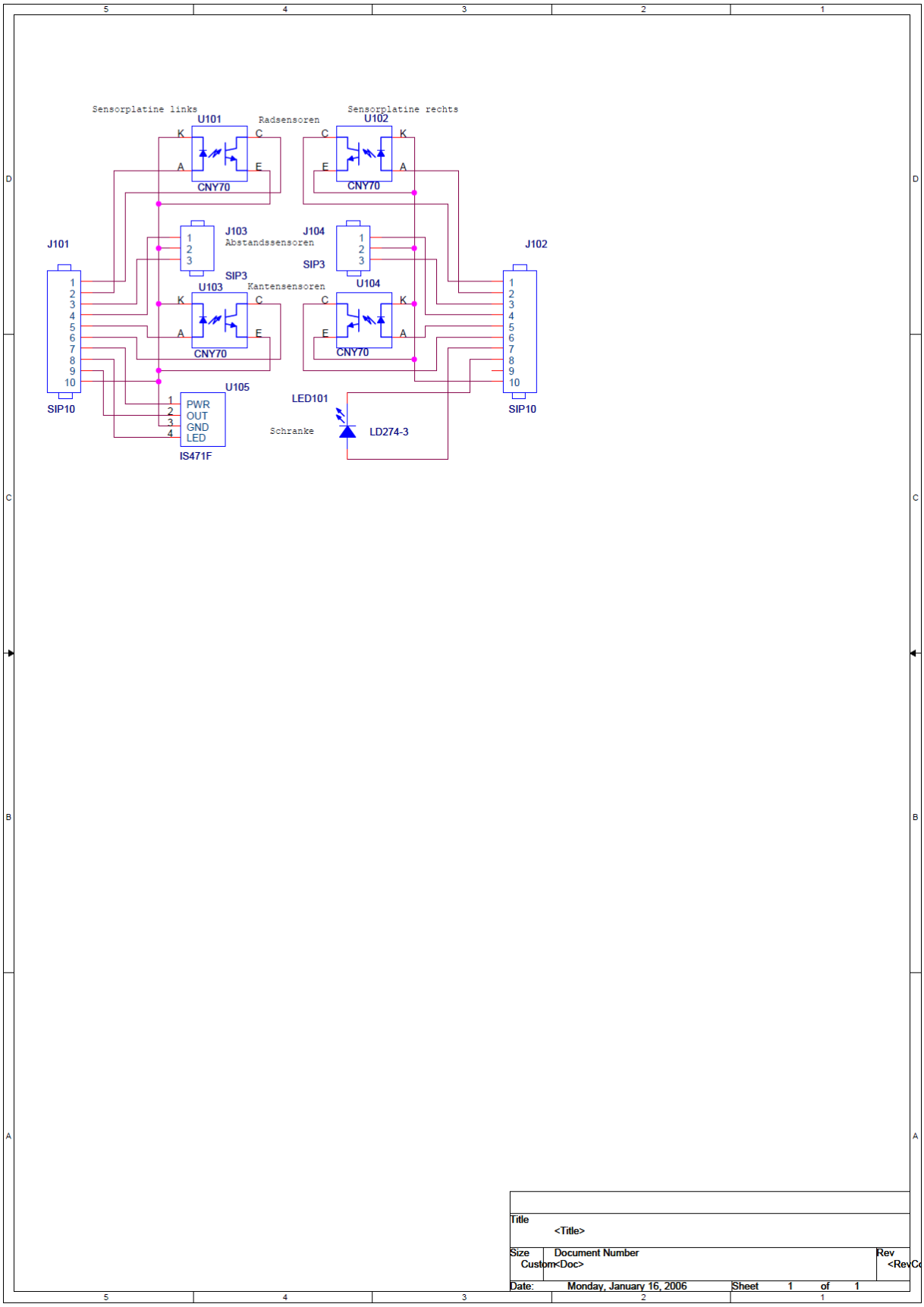
Schaltpläne



Zeichnung I: Gesamtplan



Zeichnung II: Maussensor



Zeichnung III: Sensoren

9. Anhang B

Funktionsübersicht

setLed

```
void setLed(uint8_t nr)
```

Schaltet eine LED an
Schaltet die mit Argument *nr* übergebene LED an. Bereits angeschaltete LEDs bleiben an.

Parameter:

nr
Nummer der LED, die angeschaltet werden soll. Möglich sind Werte von 1 bis 255, bzw. *LED1...LED8*

Rückgabewert:

Keiner

clearLed

```
void clearLed(uint8_t nr)
```

Schaltet eine LED ab
Schaltet die mit Argument *nr* übergebene LED ab. Andere angeschaltete LEDs bleiben an.

Parameter:

nr
Nummer der LED, die abgeschaltet werden soll. Möglich sind Werte von 1 bis 255 , bzw. *LED1...LED8*

Rückgabewert:

Keiner

lcdSetCursor

```
void lcdsetcursor(uint8_t row, uint8_t pos)
```

Setz den Cursor des Displays neu
Setzt den Cursor entsprechend den Argumenten `row` und *`pos`* neu.

Parameter:

row
Zeile in die der Cursor gesetzt werden soll. Möglich sind Werte von 0 bis 3

pos
Position innerhalb einer Zeile auf die der Cursor gesetzt werden soll. Möglich sind Werte von 0 bis 19

Rückgabewert:

Keiner

lcdClear

```
void lcdclear()
```

Löscht den Displayinhalt
Löscht den Inhalt des Displays und setzt den Cursor auf das erste Zeichen der ersten Zeile.

Parameter:

Keine

Rückgabewert:

Keiner

lcdPrintf

```
void lcdprintf(const char *s, ...)
```

Gibt einen formatierten Daten aus
Gibt eine Sequenz von Zeichen und formatierten Daten auf dem Display aus.

Parameter:

s
String, der den auszugebenen Text enthält. Er kann Formatierungzeichen enthalten, welche durch die weiteren Argumente ersetzt werden.

%[BREITE].[PREZISION]Bezeichner

Bezeichner	Typ	Ausgabebeispiel
c	Einzelnes Zeichen	a
d	Integer mit Vorzeichen	-15
e	Gebrochen Rationale Zahl in Exponentialdarstellung	3.985e+6
E	Gebrochen Rationale Zahl in Exponentialdarstellung	3.985E+6
f	Gebrochen Rationale Zahl in Kommadarstellung	398.5
g	Gebrochen Rationale Zahl in Komma- oder Exponentialdarstellung. Je nach dem, was kürzer ist	398.5
G	Gebrochen Rationale Zahl in Komma- oder Exponentialdarstellung, Je nach dem, was kürzer ist	398.5
o	Oktalzahl ohne Vorzeichen	230
s	Zeichenkette	Hallo!
u	Integer ohne Vorzeichen	34566
x	Hexadezimalzahl ohne Vorzeichen	8ea
X	Hexadezimalzahl ohne Vorzeichen	8EA
p	Adresse eines Zeigers	C600:2000
%	Ausgabe von %	%

Weitere Argumente
Abhängig von dem Argument `s` können beliebig viele weitere angegeben werden. Jedes Argument muss einen Wert enthalten der anstelle des entsprechenden %-Tags eingesetzt werden kann

Rückgabewert:

Keiner

enable

```
void enable(SENSORTYPE id)
```

Schaltet einen Sensor an
Schaltet die Stromversorgung für den in Argument *id* angegebenen Sensor an.

Parameter:

id
ID des Sensors

ID	Sensor
ABSTAND	Abstandssensoren
RAD	Radsensoren
SCHRANKE_EN	Sensor der Schranke
KANTEN	Kantensensoren vorne unter dem Bot
KLAPPE_EN	Sensor der Transportklappe
MAUS	Liniensensoren unter dem Bot
ERW1	Erweiterungsboard 1
ERW2	Erweiterungsboard 2

Rückgabewert:

Keiner

disable

```
void disable(SENSORTYPE id)
```

Schaltet einen Sensor ab
Schaltet die Stromversorgung für den in Argument *id* angegebenen Sensor ab.

Parameter:

id
ID des Sensors. Siehe Funktion *enable*.

Rückgabewert:

Keiner

getAnalog

```
uint16_t getAnalog(SENSOR_ANALOG id)
```

Liefert den Wert eines Sensors
Liefert den Wert des in Argument *id* übergebenen Analogsensors.

Parameter:

id
ID des Sensors

ID	Sensor
ABSTL	Linker Abstandssensor
ABSTR	Rechter Abstandssensor
LINEL	Linker Liniensensor
LINER	Rechter Linien
LDRL	Linker LDR
LDRR	Rechter LDR
KANTEL	Linker Kantensensor
KANTER	Rechter Kantensensor

Rückgabewert:

Ergebnis der Analog-Digital-Wandlung als 16-Bit unsigned Integer.
Der maximal mögliche Wert beträgt jedoch nur 1023.

getDigital

```
uint8_t getDigital(SENSOR_DIGITAL id)
```

Liefert den Wert eines Sensors
Liefert den Wert des in Argument *id* übergebenen Digitalsensors.

Parameter:

id
ID des Sensors

ID	Sensor
SCHRANKE	Sensor der Schranke
RADL	Linker Radsensor
KLAPPE	Sensor der Transportklappe
RADR	Rechter Radsensor

Rückgabewert:

Der aktuell am entsprechenden Eingang des Mikrocontrollers anliegende Wert als 8-Bit unsigned Integer.

getEncLeft

```
getEncLeft()
```

Liefert die Anzahl der Motorencoder-Schritte
Diese Funktion existiert nur als Makro. Sie liefert die Anzahl der Schritte des linken Motorencoders seit Programmstart als 32-Bit Wert.

Parameter:

Keine

Rückgabewert:

Den Wert des linken Radencoders als der 32-Bit unsigned Integer.

GetEncRight

```
getEncRight()
```

Liefert die Anzahl der Motorencoder-Schritte
Diese Funktion existiert nur als Makro. Sie liefert die Anzahl der Schritte des rechten Motorencoders seit Programmstart als 32-Bit Wert.

Parameter:

Keine

Rückgabewert:

Den Wert des rechten Radencoders als der 32-Bit unsigned Integer.

getRC5Cmd

```
getRC5Cmd()
```

Liefert den Kommando-Teil eines RC5-Befehls
Diese Funktion existiert nur als Makro. Sie extrahiert aus dem zuletzt empfangenen RC5-Befehl den Kommando-Teil.

Parameter:

Keine

Rückgabewert:

Kommando-Teil des RC5-Codes als 8-Bit unsigned Integer

getRC5Toggle

```
getRC5Toggle()
```

Liefert das Toggle-Bit eines RC5-Befehls
Diese Funktion existiert nur als Makro. Sie liefert das Toggle des letzten empfangenen RC5-Befehls

Parameter:

id
Keine

Rückgabewert:

Toggle-Bit des RC5-Codes als 8-Bit unsigned Integer

clearRC5Toggle

```
clearRC5Toggle()
```

Löscht das Toggle-Bit eines RC5-Befehls
Diese Funktion existiert nur als Macro. Sie setzt das Toggle des letzten empfangenen RC5-Befehls auf -1.

Parameter:

Keine

Rückgabewert:

Keiner

clearRC5Code

```
clearRC5Code()
```

Löscht einen kompletten RC5-Befehls
Diese Funktion existiert nur als Macro. Sie setzt den kompletten RC5-Befehl auf 0.

Parameter:

Keine

Rückgabewert:

Keiner

setMotorRichtung

```
void setMotorRichtung(uint8_t motor, uint8_t direction)
```

Stellt die Drehrichtung des Motors ein
Mit Hilfe dieser Funktion wird die Drehrichtung des in Argument *motor* angegebenen Motors festgelegt.

Parameter:

motor
ID des Motors. Möglich sind die Werte *MOTOR_LEFT* und *MOTOR_RIGHT*, bzw. *0* und *1*

direction
Richtung in die sich der Motor drehen soll. Möglich sind die Werte *MOTOR_FORWARD* und *MOTOR_BACKWARD*, bzw. *0* und *1*.

Rückgabewert:

Keiner

setMotorPower

```
void setMotorPower(uint8_t motor, int8_t power)
```

Legt die Geschwindigkeit und Richtung des Motors fest
Mit Hilfe dieser Funktion kann die Geschwindigkeit und Richtung des in Argument *motor* übergebenen Motors im Bereich von -100% bis +100% in 1%-Schritten festgelegt werden.

Parameter:

motor
ID des Motors. Möglich sind die Werte *MOTOR_LEFT* und *MOTOR_RIGHT*, bzw. *0* und *1*

power
Die Geschwindigkeit mit der der Motor laufen soll. Möglich sind Werte zwischen -100 bis +100. Negative Werte lassen den Motor rückwärts drehen, positive vorwärts.

Rückgabewert:

Keiner

setMotorMaxPower

```
void setMotorMaxPower(uint8_t maxPower)
```

Legt die maximale Geschwindigkeit der Motoren fest
Mit Hilfe dieser Funktion kann die maximal einstellbare Geschwindigkeit, mit der die Motoren laufen, festgelegt werden. Diese Funktion aktualisiert nicht die momentane Geschwindigkeit.

Parameter:

maxPower
Die maximal einstellbare Geschwindigkeit. Möglich sind Werte zwischen 1 und 255.

Rückgabewert:

Keiner

setMotorPwm

```
void setMotorPwm(uint8_t motor, uint8_t power)
```

Erlaubt direkten Zugriff auf das PWM-Signal
Dieser Funktion erlaubt direkten Manipulationen des für das PWM-Signal zuständigen Registers des in Argument *motor* angegebenen Motors.

Parameter:

motor
ID des Motors. Möglich sind die Werte *MOTOR_LEFT* und *MOTOR_RIGHT*, bzw. *0* und *1*

power
Der Wert, der in das PWM-Signal Register geschrieben werden soll. Möglich sind Werte zwischen 0 und 255

Rückgabewert:

Keiner

uartreadc

```
char uartreadc()
```

Liefert ein empfangenes Zeichen
Liefert, sofern vorhanden, das erste Zeichen aus dem Empfangsbuffer, ohne es zu löschen.

Parameter:

Keine

Rückgabewert:

Das Zeichen, was zu erst in den Empfangsbuffer geschrieben wurde. Sollte der Empfangsbuffer leer sein, ist der Rückgabewert 0

uartgetc

```
char uartgetc()
```

Liefert ein empfangenes Zeichen
Liefert, sofern vorhanden, das erste Zeichen aus dem Empfangsbuffer und löscht es anschließend daraus.

Parameter:

Keine

Rückgabewert:

Das Zeichen, was zu erst in den Empfangsbuffer geschrieben wurde. Sollte der Empfangsbuffer leer sein, ist der Rückgabewert 0

uartreads

```
uint8_t uartreads(char *buffer, uint8_t length)
```

Liefert eine empfangene Zeichenkette
Speichert, sofern vorhanden, die in Argument *length* angegebene Anzahl an zuerst empfangenen Zeichen aus dem Empfangsbuffer an die in Argument *buffer* angegebene Adresse.

Parameter:

buffer
Zeiger auf den Speicherplatz an denen die Zeichen gespeichert werden sollen. Der Speicherbereich muss vorher allokiert werden.

length
Anzahl der Zeichen inkl. terminierender 0, die maximal ausgelesen werden sollen.

Rückgabewert:

Die Anzahl der Zeichen, die aus dem Empfangsbuffer ausgelesen werden konnten.

uartSetBaud

```
void uartSetBaud(uint32_t baud)
```

Ändert die Baudrate des UART
Diese Funktion dient zum ändern der Baudrate des UART.

Parameter:

baud
Übertragungsrate die eingestellt werden soll. Mögliche fehlerfreie Werte sind 300, 1200, 2400, 4800, 9600, 14400, 19200, 38400, 76800, 250000, 500000, 1000000

Rückgabewert:

Keiner

uartgets

```
uint8_t uartgets(char *buffer, uint8_t length)
```

Liefert eine empfangene Zeichenkette
Wie *uartreads*, mit der Erweiterung, das ausgelesene Zeichen aus dem Empfangsbuffer gelöscht werden

Parameter:

siehe *uartreads*

Rückgabewert:

siehe *uartreads*

uartputs

```
void uartputc(char c)
```

Sendet ein Zeichen
Diese Funktion schreibt das in Argument *c* übergebene Zeichen in das entsprechende Register des UARTs.

Parameter:

c
Das Zeichen, welches gesendet werden soll.

Rückgabewert:

Keiner

uartprintf

```
void uartprintf(const char *s, ...)
```

Sendet eine formatierte Zeichenkette
Versendet eine Sequenz von Zeichen und formatierten Daten mit Hilfe des UARTs.

Parameter:
s
String, der den auszugebenen Text enthält. Er kann Formatierungzeichen enthalten, welche durch die weiteren Argumente ersetzt werden.

%[BREITE].[PREZISION]Bezeichner

Bezeichner	Typ	Ausgabebeispiel
c	Einzelnes Zeichen	a
d	Integer mit Vorzeichen	-15
e	Gebrochen Rationale Zahl in Exponentialdarstellung	3.985e+6
E	Gebrochen Rationale Zahl in Exponentialdarstellung	3.985E+6
f	Gebrochen Rationale Zahl in Kommadarstellung	398.5
g	Gebrochen Rationale Zahl in Komma- oder Exponentialdarstellung. Je nach dem, was kürzer ist	398.5
G	Gebrochen Rationale Zahl in Komma- oder Exponentialdarstellung, Je nach dem, was kürzer ist	398.5
o	Oktalzahl ohne Vorzeichen	230
s	Zeichenkette	Hallo!
u	Integer ohne Vorzeichen	34566
x	Hexadezimalzahl ohne Vorzeichen	8ea
X	Hexadezimalzahl ohne Vorzeichen	8EA
p	Adresse eines Zeigers	C600:2000
%	Ausgabe von %	%

Weitere Argumente
Abhängig von dem Argument `s` können beliebig viele weitere angegeben werden. Jedes Argument muss einen Wert enthalten der anstelle des entsprechenden %-Tags eingesetzt werden kann

Rückgabewert:

Keiner

setServoPos

```
void setServoPos(uint8_t servo, uint8_t pos)
```

Stellt den Servo ein
Mit Hilfe dieser Funktion lässt sich die Position des in Argument *servo* übergebenen Servos einstellen.

Parameter:

servo
Die ID des Servos, der geändert werden soll. Möglich sind die Werte *SERVO1* und *SERVO2*, bzw. *0* und *1*.

pos
Die Position des Servos. Möglich sind Werte von 0 bis 255.

Rückgabewert:

Keiner

setMinServoPos

```
void setMinServoPos(uint8_t servo, uint8_t pos)
```

Legt die Ausschlagsgrenze des Servos fest
Mit Hilfe dieser Funktion lässt sich der maximale Ausschlag des Servos zu einer Seite festlegen. Sie hat jedoch keine Auswirkungen auf die aktuelle Position.

Parameter:

servo
Die ID des Servos, der geändert werden soll. Möglich sind die Werte *SERVO1* und *SERVO2*, bzw. *0* und *1*.

pos
Die minimale Position die der Servo anfahren kann. Je höher dieser Wert, desto kleiner ist der Aktionsradius des Servos. Möglich sind Werte zwischen 1 und 255

Rückgabewert:

Keiner

setMaxServoPos

```
void setMaxServoPos(uint8_t servo, uint8_t pos)
```

Legt die Ausschlagsgrenze des Servos fest
Mit Hilfe dieser Funktion lässt sich der maximale Ausschlag des Servos zu einer Seite festlegen. Sie hat jedoch keine Auswirkungen auf die aktuelle Position.

Parameter:

servo

Die ID des Servos, der geändert werden soll. Möglich sind die Werte *SERVO1* und *SERVO2*, bzw. *0* und *1*.

pos

Die maximale Position die der Servo anfahren kann. Je niedriger dieser Wert, desto kleiner ist der Aktionsradius des Servos. Möglich sind Werte zwischen 1 und 255

Rückgabewert:

Keiner

readDY

```
int8_t readDY(void)
```

Liefert die Bewegung der Maus in Y-Richtung
Diese Funktion liefert die relative Pixelbewegung der Y-Achse der Maus seit dem letzten Funktionsaufruf.

Parameter:

Keine

Rückgabewert:

Die Anzahl der Pixel um die sich die Y-Achse der Maus bewegt hat im Bereich von -128 bis +127.

readDX

```
int8_t readDX(void)
```

Liefert die Bewegung der Maus in X-Richtung
Diese Funktion liefert die relative Pixelbewegung der X-Achse der Maus seit dem letzten Funktionsaufruf.

Parameter:

Keine

Rückgabewert:

Die Anzahl der Pixel um die sich die X-Achse der Maus bewegt hat im Bereich von -128 bis +127.

createTask

```
uint8_t createTask(void (* func), uint16_t stacksize)
```

Erstellt einen neuen Prozess
Mit Hilfe dieser Funktion lässt sich die als Argument *func* übergebene Funktion als eigener Prozess nutzen.

Parameter:

func
Zeiger auf die Funktion, die als Prozess dienen soll.

stacksize
Die Anzahl der Bytes, die der Stack des Prozesses belegen darf.

Rückgabewert:

Die ID des Prozesses, sofern dieser erfolgreich angelegt wurde. Ansonsten -1.

getThisId

```
getThisId()
```

Liefert die ID des aktuellen Prozesses
Diese Funktion existiert nur als Macro. Sie liefert die ID des momentan aktiven Prozesses und ist äquivalent zu der Variablen *this*.

Parameter:

Keine

Rückgabewert:

Die ID des aktiven Prozesses als 8-Bit unsigned Integer.

sleep

```
void sleep(uint16_t ticks)
```

Unterbricht den aktuellen Prozess
Mit Hilfe dieser Funktion lässt sich der momentan laufende Prozess unterbrechen und wird erst nach der in Argument *ticks* angegebenen Anzahl von Systemticks (100µs) fortgesetzt.

Parameter:

ticks
Die Anzahl an Systemticks für die der Prozess unterbrochen werden soll.

Rückgabewert:

Keiner

suspendTask

```
void suspendTask(uint8_t id)
```

Unterbricht einen aktuellen Prozess
Mit Hilfe dieser Funktion lässt ein beliebiger Prozess für ca. 6,5 Sekunden unterbrechen.

Parameter:

id
Die ID des Prozesses, der unterbrochen werden soll.

Rückgabewert:

Keiner

resumeTask

```
void suspendTask(uint8_t id)
```

Setzt einen unterbrochenen Prozess fort
Mit Hilfe dieser Funktion lässt ein beliebiger unterbrochener Prozess vorzeitig wieder aktivieren.

Parameter:

id
Die ID des Prozesses, der wieder aktiviert werden soll.

Rückgabewert:

Keiner

lock

```
lock()
```

Schützt einen Codeabschnitt
Diese Funktion existiert nur als Macro. Mit ihrer Hilfe lässt sich ein Codeabschnitt schützen, indem verhindern wird, dass zu einem anderen Prozess gewechselt werden kann.

Parameter:

Keine

Rückgabewert:

Keiner

unlock

```
unlock()
```

Beendet einen geschützten Codeabschnitt
Diese Funktion existiert nur als Macro. Sie beendet einen geschützten Codeabschnitt

Parameter:

Keine

Rückgabewert:

Keiner

getSystemTicks

```
getSystemTicks()
```

Liefert die Anzahl der Systemticks
Diese Funktion existiert nur als Macro. Sie liefert die Anzahl der Systemticks seit Programmstart.

Parameter:

Keine

Rückgabewert:

Keiner

sendMessage

```
void sendMessage(uint16_t dest, uint8_t *data, uint8_t length)
```

Überträgt eine Nachricht über das Zigbee-Modul
Diese Funktion dient zur Übertragung von Daten mit Hilfe des Zigbee-Moduls

Parameter:

dest
Zieladresse der Nachricht

data
Zeiger auf den Speicherbereich in dem die Daten liegen

length
Anzahl an Bytes, die übertragen werden sollen

Rückgabewert:

Keiner

getMessage

```
uint8_t getMessage(uint16_t *src, uint8_t *data, uint8_t length)
```

Ruft eine Nachricht ab
Diese Funktion ruft eine Empfangene Nachricht ab

Parameter:

src
Zeiger auf eine Variable, in der nach dem Aufruf der Absender der Nachricht steht

data
Zeiger auf den Speicherbereich in dem die Daten gespeichert werden sollen

length
Anzahl an Bytes, die maximal gespeichert werden können

Rückgabewert:

Anzahl der tatsächlich gespeicherten Bytes

setIdentity

```
void setIdentity(uint16_t addr, char *colour)
```

Legt die Identität des Roboters fest
Legt die Identität des Roboters im Erweiterungsboard fest.

Parameter:

addr
Empfangsadresse für Zigbee-Nachrichten

colour
Zeiger auf ein 4 Byte großes Array, welches den Farbcode für die Kamera enthält

Rückgabewert:

Keiner

getPosition

```
void getPosition(uint8_t *x, uint8_t *y)
```

Liest die gespeicherten Koordinaten aus
Liest die im Erweiterungsboard gespeicherten Koordinaten aus

Parameter:

x
Zeiger auf eine Variable, in der nach dem Aufruf die X-Koordinate steht

y
Zeiger auf eine Variable, in der nach dem Aufruf die Y-Koordinate steht

Rückgabewert:
Keiner

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, 07. Juli 2009
Ort, Datum

Unterschrift

CD Inhalt

- Diese Arbeit im PDF-Format
- Quellcode der entwickelten Bibliothek