UNIVERSITÄT KOBLENZ · LANDAU
Fachbereich 4: Informatik

AGAS
Arbeitsgruppe Aktives Sehen

Bau eines Roboters für den RoboCupSoccer Wettbewerb

Studienarbeit
im Studiengang Computervisualistik

vorgelegt von

Andreas Klöber und Michael Strack

Betreuer:	Prof. Dr.-Ing. Dietrich Paulus, Institut für Computervisualistik, Fachbereich Informatik
Erstgutachter:	Prof. Dr.-Ing. Dietrich Paulus, Institut für Computervisualistik, Fachbereich Informatik
Zweitgutachter:	Dipl.-Inf. Johannes Pellenz, Institut für Computervisualistik, Fachbereich Informatik

Koblenz, im Juli 2006

Erklärung

Wir versichern, dass wir die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt haben und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Die Richtlinien der Arbeitsgruppe für Studien- und Diplomarbeiten haben wir gelesen und anerkannt, insbesondere die Regelung des Nutzungsrechts.

Mit der Einstellung dieser Arbeit in die Bibliothek sind wir einverstanden.	ja ☐	nein ☐
Der Veröffentlichung dieser Arbeit im Internet stimmen wir zu.	ja ☐	nein ☐

Koblenz, den Andreas Klöber Michael Strack

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die Zeitschrift *c't* stellte in der Ausgabe 02/2006 einen Bausatz für einen kleinen mobilen Roboter vor, den *c't-Bot*, der diese Studienarbeit inspirierte. Dieser Bausatz sollte die Basis eines Roboters darstellen, der durch eine Kamera erweitert und mit Hilfe von Bildverarbeitung in der Lage sein sollte am *RoboCupSoccer-Wettbewerb* teilzunehmen.

Während der Planungsphase veränderten sich die Ziele: Statt einem Fußballroboter, sollte nun ein Roboter für die neu geschaffene *RoboCup-Rescue-League* entwickelt werden. In diesem Wettbewerb sollen Roboter in einer für sie unbekannten Umgebung selbstständig Wege erkunden, bzw. Personen in dieser Umgebung finden.

Durch diese neue Aufgabenstellung war sofort klar, dass der c't-Bot nicht ausreichte und es musste ein neuer Roboter entwickelt werden, der mittels Sensoren die Umgebung wahrnehmen, durch eine Kamera Objekte erkennen und mit Hilfe eines integrierten Computers diese Bilder verarbeiten sollte. Die Entstehung dieses Roboters ist das Thema dieser Studienarbeit.

Kapitel 2
Zeigt die Entwicklung, angefangen beim c't-Bot, über die Planung bis hin zum Aufbau des eigenen Roboters.

Kapitel 3
Zu Beginn werden die verwendeten Platinen genauer beschrieben, danach wird der Mikrocontroller vorgestellt.

Kapitel 4
Beschreibt die Programmierung des Mikrocontrollers zur Ansteuerung von Sensoren, Motoren und sonstiger Hardware des Roboters.

Kapitel 5
Die Ansteuerung des Roboters erfolgt über eine spezielle Steuersoftware. Dieses Kapitel beschreibt die zugehörigen Klassen und, basierend darauf, die Entwicklung der eigentlichen Steuerung in Form diverser Anwendungen.

Kapitel 6
Fasst die Eigenschaften des Roboters zusammen und gibt einen Ausblick auf mögliche Erweiterungen.

Kapitel 2

Planung und Aufbau

Diese Studienarbeit wurde von einem Artikel in der Zeitschrift *c't* inspiriert. Dort wurde ein kleiner Roboter, der *c't-Bot*, vorgestellt, den die Redakteure der Zeitschrift entwickelt haben. Ziel dieses Projektes war es, einen günstigen und einfachen Roboter zu entwickeln, der sich mit Hilfe von diversen Sensoren autonom in einem Raum bewegen und ein Objekt suchen kann, um dieses gegebenenfalls an einen anderen Ort zu bringen. Im ersten Unterkapitel werde ich diesen c't-Bot genauer vorstellen. Im zweiten werde ich die Planungen zu unserem eigenen Roboter erläutern und im dritten Unterkapitel werde ich den Aufbau dieses Roboters beschreiben.

2.1 Grundidee: Eine günstige, kleine mobile Plattform

Der c't-Bot ist selbst als Bausatz erhältlich, welcher alle benötigten Bauteile enthält. Er hat eine runde Grundfläche, deren Durchmesser 12cm beträgt und besitzt im Grundzustand zwei Ebenen, auf denen die gesamte Elektronik untergebracht ist. Durch die runde Form ist ein anecken nicht möglich und größtmögliche Wendigkeit wird garantiert. Angetrieben wird der c't-Bot durch zwei auf der Mittelachse der Grundplatte befestigte Motoren mit integriertem 33:1 Untersetzungsgetriebe. Damit er nicht umkippt, besitzt er noch einen Stützpin am hinteren Ende der Grundplatte.

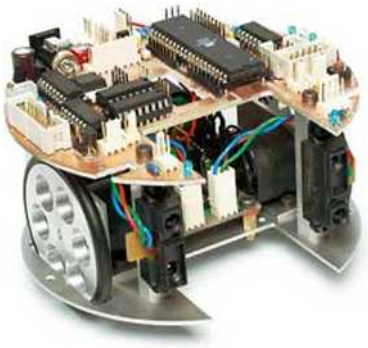


Bild 2.1: Der c’t-Bot (Quelle: c’t Magazin)

Zur Umgebungserkennung kommen diverse Sensoren zum Einsatz. So hat er vorne, zur Entfernungsmessung, zwei Infrarotsensoren des Typs *GP2D12* von *Sharp*, deren Messbereich zwischen 10 und 80cm liegt. Des Weiteren besitzt er noch vier nach unten gerichtete Optokoppler des Typs *TCST1103*. Zwei davon sind im vorderen äußeren Bereich der Grundplatte befestigt und sorgen dafür, dass Abgründe rechtzeitig erkannt werden. Die beiden anderen liegen mittig im hinteren Bereich der Grundplatte und dienen dazu, eine schwarze Linie auf dem Boden zu erkennen und zu verfolgen. Durch zwei an der Vorderseite befestigte Photosensoren, ist der c’t-Bot auch in der Lage einer Lichtquelle zu folgen.

Die Odometriebestimmung übernehmen ebenfalls zwei Optokoppler des Typs *TCST1103*. Diese tasten die auf den beiden Rädern geklebten Kodierscheiben ab und liefern so die zurückgelegte Strecke der einzelnen Räder. Dieses Verfahren ist relativ ungenau, da durchdrehende Räder die gelieferten Werte verfälschen, so dass diese nicht mehr der tatsächlich zurückgelegten Strecke entsprechen. Aus diesem Grund besitzt der c’t-Bot noch einen nach unten gerichteten Maussensor, der die Bewegung des Roboters registriert. Hier wird aus der Positionsveränderung des Roboters die zurückgelegte Strecke genau beschrieben.

Die Steuerung übernimmt ein *Atmel ATmega32* Mikrocontroller. Er verfügt über 32 KByte Flash-Speicher, um die nötigen Programme zu speichern. Durch seine RISC-Architektur und seinen 16 MHz Systemtakt, hat er auch genug Leistung, um die Programme schnell genug abzuarbeiten. Im Kapitel 3.2 wird dieser Mikrocontroller noch genauer betrachtet.

Die Spannungsversorgung übernehmen 4 NiMh-Akkus, die zusammen 6 V und 2,3 Ah liefern, was für eine Betriebsdauer von ca. 9 Stunden reichen sollte. Durch Anschließen eines Netzteils, kann die Betriebsdauer noch erhöht werden. Ein interner Lademechanismus ist nicht vorgesehen.

2.2 Planung des Roboters

Da wir auf unserem Roboter Bilder verarbeiten wollten, war sofort klar, dass die c't-Bot-Plattform zu klein ist, da dazu noch ein geeigneter Prozessor und mehr Speicher vonnöten sind, die auf diesem Roboter keinen Platz finden würden. Deshalb entschieden wir uns für eine Eigenentwicklung, die folgende Eigenschaften erfüllen sollte:

- Der Roboter sollte nicht zu groß sein, d.h. Länge, Breite und Höhe des Roboters sollten so klein wie möglich gehalten werden.
- Er sollte eine gute Manövrierfähigkeit haben, d.h. er sollte sich auf der Stelle drehen können, er sollte leichte Steigungen meistern können, er sollte einen guten Geradeauslauf haben und er sollte eine angemessene Fahrgeschwindigkeit haben.
- Die Möglichkeit der Bildverarbeitung sollte gegeben sein, d.h. auf dem Roboter sollte sich ein Computer mit genügend Rechenleistung und Speicher befinden, die eine Bildverarbeitung ermöglichen. Des Weiteren sollte sich auch eine Kamera auf dem Roboter befinden, die die zu verarbeitenden Bilder liefert.
- Als Betriebssystem sollte Linux zu Einsatz kommen.
- Der Roboter sollte über genügend Sensoren verfügen, damit er die Umgebung auswerten kann und nirgendwo aneckt bzw. runterfällt.
- Der Roboter sollte modular aufgebaut sein, d.h. man sollte gegebenenfalls Teile einfach austauschen können bzw. den Roboter ohne größeren Aufwand erweitern können.
- Die Möglichkeit der Odometriebestimmung sollte ebenfalls gegeben sein.

- Die Laufzeit des Roboters sollte mindestens 45 Minuten betragen, wenn alle Verbraucher genutzt werden.
- Die Kosten des Roboters dürfen 750 EUR nicht übersteigen.

Um die benötigte Größe des Roboters herauszufinden, mussten wir zuerst einmal die zu verbauenden Teile zusammenstellen. Wir begannen mit der Suche nach einem geeigneten Mainboard, welches nicht nur stromsparend, sondern auch über genügend Rechenleistung verfügen sollte, bei geringstmöglichem Platzbedarf.

Zuerst kam ein *PC104-Board* in die nähere Auswahl, da diese Art der Mainboards sehr geringe Ausmaße besitzt (96mm * 147mm) und auch über genügend Rechenleistung verfügt (es gibt sie inzwischen mit bis zu 1.8 GHz Pentium M Prozessor). Leider sind diese sehr teuer, da sie in relativ kleiner Stückzahl produziert werden, und eine Erweiterung ist meist nur durch Spezialhardware möglich.

Aus diesem Grund entschieden wir uns für ein *Mini-ITX-Board*. Diese Art der Boards sind bei vergleichbarer Leistung wesentlich günstiger und, da es sich um eine normales Mainboard handelt, können Standard-PC-Bauteile eingesetzt werden, die die Kosten im Rahmen halten. Das von uns verwendete *VIA Epia M-10000* Mainboard besitzt neben einem 1 GHz *Via C3* Prozessor, auch noch eine integrierte Grafikkarte und benötigt nur ca. 45 Watt unter Volllast. Mit einer Größe von 170mm * 170mm ist es auch relativ klein, so dass sich die Breite des Roboters im Rahmen hält.

Der nächste Punkt, um die Größe des Roboters zu ermitteln war die Art der Spannungsversorgung. Wir benötigten genügend Leistung, um den Roboter für ca. 45 Minuten mit Strom zu versorgen. Dass wir eine 12 V Spannung einsetzen wollten war von vornherein klar, da dies schon das Mainboard verlangt. Es kam also nur noch auf die Stärke des Akkus an. Die benötigte Leistung des Roboters konnten wir zu diesem Zeitpunkt nur abschätzen, die einzigen Anhaltspunkte waren die Stromaufnahme des Mainboards, ca. 4 A, und des c't-Bots, ca. 250mA. Mit diesen beiden Werten kamen wir auf einen Akku, der mindestens 4,5 A liefern sollte und entschieden uns für einen 7,2 Ah Blei-Gel-Akku, der über genügend Reserven verfügt, um den Roboter für die gewünschte Zeit mit Strom zu versorgen. Ein weiterer Vorteil der Wahl dieses Akkus war, dass diese auch in den anderen Robotern der Arbeitsgruppe verwendet werden und so von der Anschaffung eines neuen Ladegeräts

abgesehen werden konnte. Da der Akku innerhalb des Roboters Platz finden sollte, musste das gesuchte Fahrgestell eine innere Mindestbreite von 10cm haben.

Nach diesen Voraussetzungen begannen wir mit der Suche nach einem geeigneten Fahrgestell und entschieden uns für einen *RC-Panzer* der Firma *Hen Long* im Maßstab 1:16. Dieses Modell hat neben den geeigneten Maßen auch noch zwei getrennte Motoren, ein integriertes Untersetzungsgetriebe, einzeln gefederte Kettenräder und ein weiterer Vorteil ist, dass die Ketten aus Einzelgliedern bestehen.

Während wir auf den Panzer warteten, kümmerten wir uns um die restlichen Dinge, die der Roboter erhalten sollte. Die Motorsteuerung und die Sensordatenauswertung sollte in unserem Roboter ebenfalls der *ATmega32* von *Atmel* übernehmen. Da die Eigenentwicklung eines Controllerboards sehr aufwändig gewesen wäre, entschieden wir uns für ein preisgünstiges Board der Firma *Embedit Mikrocontrollertechnik*, das *Robo32-Board*. Dieses Board hat neben dem Mikrocontroller auch noch gleich einen Motortreiber für zwei 2A-Motoren und eine integriert I^2C-Schnittstelle. Eine genaue Beschreibung des Boards finden sie in Kapitel 3.1.3. Als Sensoren planten wir zwei Infrarotsensoren des Typs *GP2D120* von *Sharp*, diese haben einen Erfassungsbereich von ca. 4cm bis 30cm und dienen dazu Abgründe bzw. Treppen zu erkennen. Zusätzlich haben wir noch einen Ultraschallsensor des Typs *SRF08* der Firma *Devantech* mit I^2C-Schnittstelle zur Entfernungsmessung im vorderen Bereich eingeplant.

Um den Ladezustand des verwendeten Akkus immer im Blick zu haben, entschlossen wir uns noch eine Spannungsanzeige zu integrieren, welche wir auf Basis einer im Internet[1] veröffentlichten Spannungsüberwachung entwickelt haben. Des Weiteren sollte auch ein Adapter zum Programmieren des ATmega32 eingebaut werden. Beides wurde auf einer eigenen Platine realisiert, die die gleichen Ausmaße wie das Robo32-Board hat, damit beide übereinander montiert werden konnten.

Mit der Lieferung des Panzermodells begann die Planung für unsere Karosserie. Dafür musste das Modell komplett zerlegt werden, um die benötigte Bodengruppe des Modells zu erhalten. Aus der Wanne entfernten wir noch das Batteriefach und weitere Halterungen. Das einzige, was wir behielten, waren die Getriebe und zunächst noch die Motoren.

[1] *http://www.team-iwan.de*

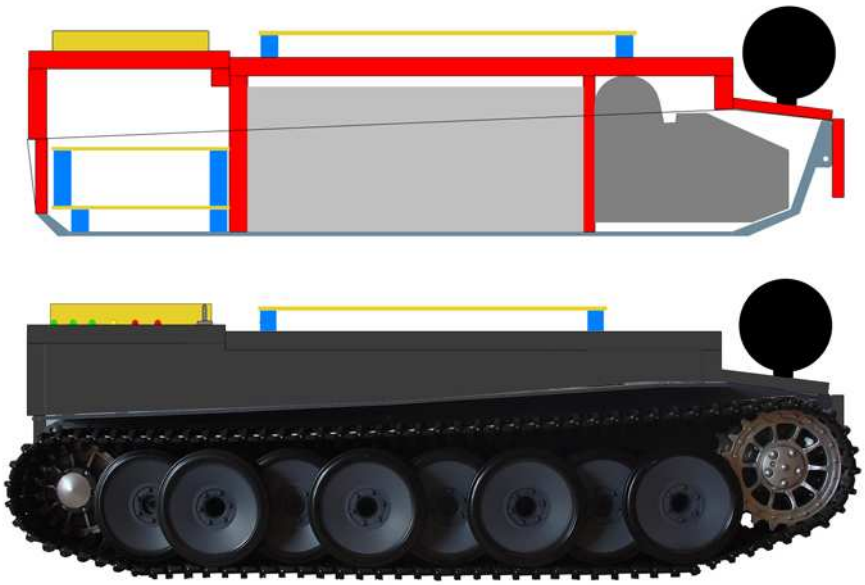


Bild 2.2: Der erste Entwurf auf Basis des Panzerfahrgestells

Nachdem alles entfernt wurde, begannen wir das Modell zu vermessen, um die benötigten Kunststoffteile für das Chassis zu entwerfen. Auf Bild 2.2 sehen sie den geplanten Roboter auf Basis dieses Fahrgestells.

Da alle Komponenten des Roboters, von uns *MiniBot* getauft, außer dem Mainboard und der Webcam unter einer Kunststoffkarosserie verschwinden sollten, musste diese Karosserie auch dementsprechend groß sein. Des Weiteren sollte der Akku in einem separaten Fach untergebracht sein. Somit ergab sich eine Dreiteilung des Bodenwanne. Im vorderen Bereich befinden sich Getriebe, Motoren und die Taktgeberscheiben in Verbindung mit den Gabellichtschranken zur Odometriebestimmung. Im mittleren Bereich ist wegen der Gewichtsverteilung der Akku untergebracht und im hinteren Bereich befinden sich die beiden Platinen, das Robo32-Board und eine eigens entworfene Platine, auf der sich die Spannungsüberwachung, der Programmieradapter für das Robo32-Board und der Schalteranschluss befinden.

Als Anhaltspunkt für die Höhe der Karosserie dienten uns die Motor- bzw. Getriebehalterungen, die noch etwas höher als der Akku sind. Mit der Breite und der Länge der Karos-

serie richteten wir uns nach der Bodenwanne des RC-Modells.

Um eine hohe Stabilität des Roboters zu erhalten, verwendeten wir 8mm starken Kunststoff namens *Polyamid 6E* (bis auf die inneren Trennwände, die Grundplatte des Ultraschallsensors und die untere hintere Heckplatte, die jeweils 5mm stark sind). Wir entschieden uns für dieses Material, da es trotz der hohen Härte, doch weich genug ist, um es mit einfachsten Mitteln zu bearbeiten und man einfach Gewinde hinein drehen kann, ohne dass das Material gleich reißt.

Genaue Zeichnungen der einzelnen Kunststoffteile und weitere Entwurfszeichnungen finden Sie im Anhang A. Die Karosserieteile wurden von der *Comco Plastik GmbH* aus Fachbach nach diesen Entwürfen gefertigt.

2.3 Aufbau des MiniBots

Der Aufbau des MiniBots begann mit dem Einbau der inneren Trennwände. Dies sind die einzigen Teile, die fest mit der Bodenwanne des Panzermodells verklebt sind. Sie trennen das Modell in die gewünschten drei Teile.

Die Trägerplatte des Ultraschallsensors wurde im nächsten Schritt vorne an zwei vorhandenen Ösen angeschraubt. Die untere Heckplatte, in der sich die Ladebuchse, die Netzteilbuchse und die RS232-Schnittstelle des Programmieradapters befinden, wurde nach geringen Anpassungen in die vorhandenen Nuten gesetzt.

Die Grundplatine mit der Spannungsanzeige, dem Programmieradapter und dem Schalteranschluss wurden nun mittels Abstandhalter mit der Wanne verschraubt. Die drei Buchsen der Heckplatte konnten danach angeschlossen werden. Damit man später die Platine ausbauen kann, wurden alle Anschlüsse durch Pfostenstecker realisiert.

Als nächstes wurden die Seitenteile mit den inneren Trennwänden verschraubt und die Taktgeberscheiben an der Achse des zweiten Getrieberitzels befestigt. Danach konnte die Signalplatine eingesetzt werden. Nachdem der Ultraschallsensor und der vordere Infrarotsensor festgeschraubt waren, konnten auch die ersten Kabel eingezogen werden. Im mittleren Akkufach wurden noch kleine Abstandhalter zur Fixierung des Akkus montiert.

Nachdem der Akku eingesetzt und an die Grundplatine angeschlossen wurde, konnten wir mit der restlichen Verkabelung dieser Platine beginnen. Wir begannen mit dem Schalter und den LEDs der Spannungsanzeige, die danach in den hinteren Deckel eingebaut wurden. Nun wurde der Programmieradapteranschluss und die Spannungsversorgung für das Robo32-Board angeschlossen, die auch gleich an diesem befestigt wurden. Schließlich folgte die Spannungsversorgung für das Mainboard, dessen Kabel zunächst nur bis in das mittlere Fach des Roboters gezogen wurde, da es erst zum Schluss benötigt wurde. Da die Grundplatine nun fertig angeschlossen war, konnten wir jetzt das Robo32-Board auf dessen Abstandhalter schrauben.

Die Kabel der vorderen Sensoren und der Signalplatine konnten nun mit dem Robo32-Board verbunden werden. Des Weiteren wurden die Motoren an dieses Board angeschlossen. Nun fehlte nur noch der hintere Infrarotsensor, der an die obere Heckabdeckung angeschraubt wurde. Das auf diesem Board festgelötete serielle Anschlusskabel konnte danach in den mittleren Bereich des Roboters gezogen werden. Somit war auch diese Platine fertig angeschlossen.

Um das Heck zu komplettieren, wurden die Kontroll-LEDs, der Ein- und der Reset-Schalter des Mainboards, sowie die 2,5“-Festplatte auf dessen Deckel befestigt und deren Anschlusskabel ebenfalls in den mittleren Bereich gezogen. Da nun alle Kabel in der Mitte lagen, konnten wir diese durch die Öffnung des mittleren Deckels ziehen und diesen ebenfalls mit dem Roboter verschrauben. Um die Karosserie zu schließen, verschraubten wir noch das Vorderteil und den vorderen Deckel, auf dem wir den Halter der Webcam befestigten.

Die erste Testfahrt brachte zwei Dinge zu Tage: Erstens, waren die Motoren zu schwach und zweitens, wurde der Motortreiber des Robo32-Boards zu heiß und benötigte deshalb eine bessere Kühlung. Das erste Problem lösten wir durch zwei neue *Graupner*[2]-Motoren mit integriertem 3:1-Getriebe. Der Einbau der neuen Motoren setzte zwar geringe Änderungen der ersten Zwischenwand voraus, sie wurden dann aber so eingebaut, dass ein späterer Austausch kein Problem darstellt. Das zweite Problem wurde durch den Einbau eines 40mm-Lüfters in die obere Rückwand und Lüftungslöchern in der rechten Seitenwand gelöst. Um gegen Kurzschlüsse gewappnet zu sein, wurden in der rechten Seitenwand zwei

[2] *http://www.graupner.de/*

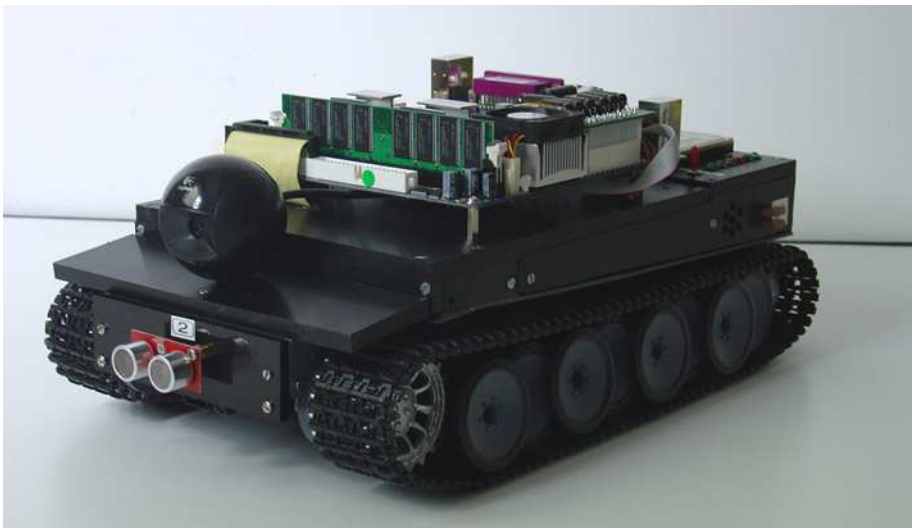


Bild 2.3: Fertiggestellter MiniBot

Sicherungen eingebaut, die einmal das Mainboard und einmal das Robo32-Board absichern. Schließlich wurde das Mainboard auf dem mittleren Deckel befestigt und angeschlossen. Mit der Installation des Betriebssystems (siehe Anhang G) und dem Aufspielen der Software (siehe Kapitel 5) wurde der MiniBot fertiggestellt.

Einige ausgewählte Bilder zum Aufbau befinden sich im Anhang B. Alle Bilder sind auf der beiliegenden CD im Verzeichnis `images/` zu finden.

2.4 Blockschaltbild

Das folgende Blockschaltbild zeigt die Zusammenhänge der einzelnen Komponenten des MiniBots, sowie die Kommunikationsart der einzelnen Bauteile untereinander.

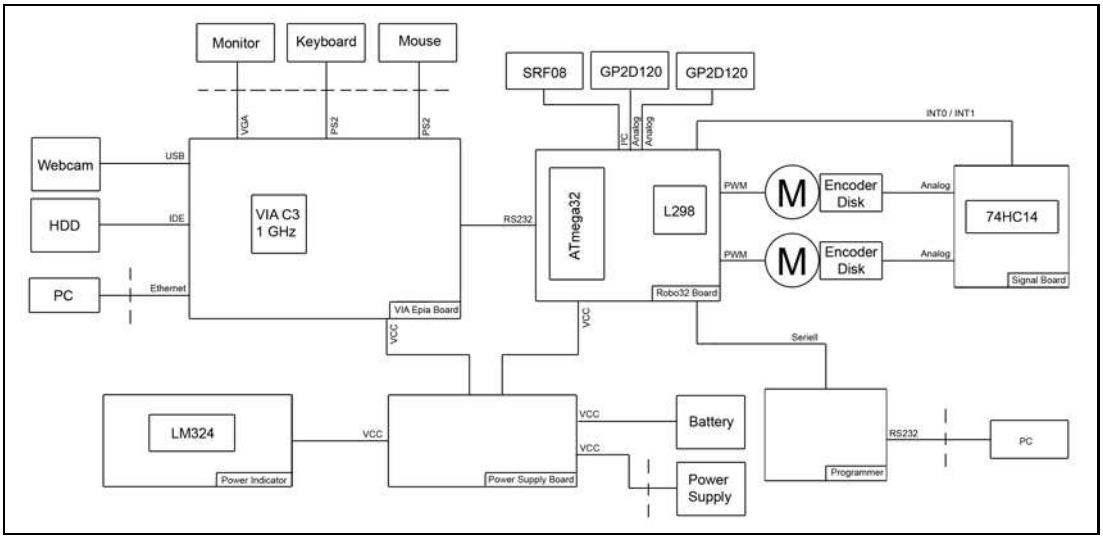


Bild 2.4: Blockschaltbild des MiniBot

Kapitel 3

Hardware

In diesem Kapitel wird die verwendete Hardware des Roboters beschrieben. Im ersten Unterkapitel werden die drei eingebauten Platinen erläutert. Im zweiten Unterkapitel wird der verwendete Mikrocontroller und dessen Einstellungen beschrieben.

3.1 Platinen

Für den Roboter war es notwendig, zu der gekauften *Robo32* Platine noch zwei Platinen zu entwickeln. Auf der ersten, der *Grundplatine* befinden sich die Spannungsanzeige auf Basis eines *LM324 Chips*, der Programmieradapter, um den ATmega32 des Robo32-Boards zu programmieren, und die Schalteranschlüsse. Die zweite, die *Signalplatine*, dient dazu, die unsauberen Sinussignale der Gabellichtschranken in saubere Rechtecksignale umzuwandeln. In den folgenden Unterkapiteln werde ich die beiden selbst gebauten Platinen und das Robo32-Board genauer erklären.

3.1.1 Spannungsanzeige und Programmieradapter

Da wir bei unserem Roboter mit sehr beengten Platzverhältnissen zurecht kommen müssen, haben wir die Grundplatine so entworfen, dass sie die gleiche Größe hat wie das

Robo32-Board, um sie übereinander befestigen zu können. Diese Platine ist in drei Sektionen eingeteilt (siehe Abb. 3.1), auf der linken Seite ist die *Spannungsanzeige*, in der Mitte befindet sich der *Programmieradapter* und auf der rechten Seite sind die Anschlüsse für den Schalter, mit dem man die Art der Spannungsversorgung auswählen kann.

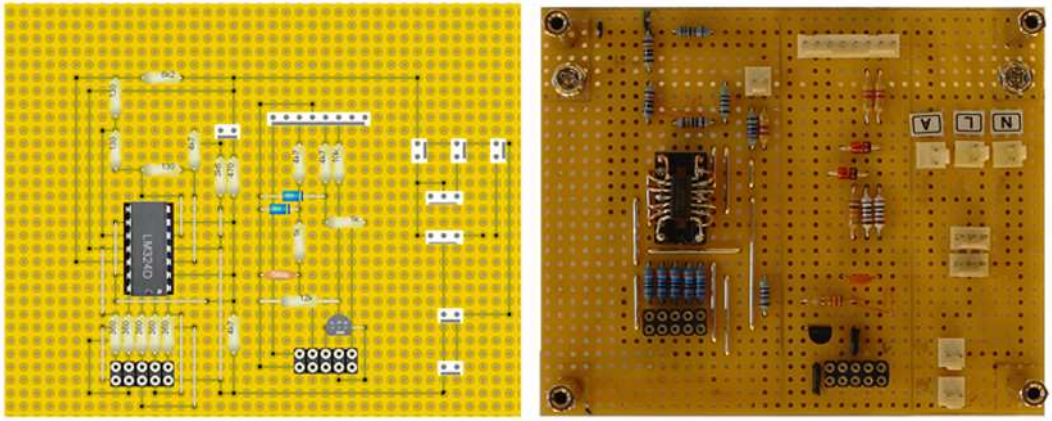


Bild 3.1: Geplante Platine (links) und fertiggestellte Platine (rechts)

Spannungsanzeige

Nach einer im Internet veröffentlichten Spannungsüberwachung[1] und mit Hilfe des Programms *MultiSIM 9*[2] von *Electronics Workbench*, haben wir eine Spannungsanzeige erstellt, die den Status des internen Akkus über 5 LEDs anzeigt[Fed04].

Basis dieser Spannungsüberwachung ist ein 4facher Operationsverstärker, der *LM324*[3] von *STMicroelectronics*. Jeder einzelne Operationsverstärker besitzt einen invertierten und einen nicht-invertierten Eingang liegt an dem nicht-invertierten Eingang eine höhere Spannung an, als an dem invertierten Eingang, so schaltet der Operationsverstärker die anliegende Versorgungsspannung auf den Ausgang, ansonsten liegt am Ausgang Masse an.

Die Referenzspannung, die an allen vier invertierten Eingängen anliegt, wird durch den Spannungsteiler mit den Widerständen $R2$ und $R3$ erzeugt. Die Spannungen, die an den

[1] *http://www.team-iwan.de*
[2] *http://www.ewb.de*, als 45-Tage Testversion erhältlich
[3] siehe *datasheets/LM324.pdf*

nicht-invertierten Eingängen anliegen, werden durch eine Widerstandsreihe erzeugt, d. h. es sind in diesem Fall fünf in Reihe geschaltete Widerstände ($R4$ bis $R9$), bei denen man zwischen jedem Widerstand eine andere Spannung abgreifen kann.

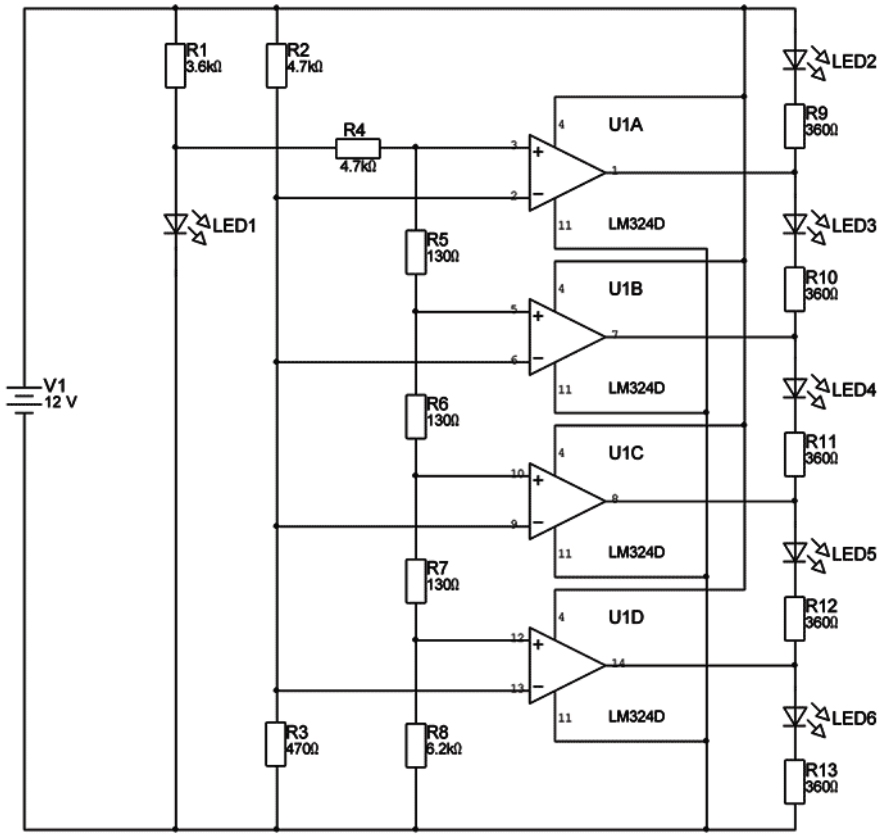


Bild 3.2: Schaltplan der Spannungsüberwachung

Bei einem Spannungsteiler [Bau04] werden im Normalfall zwei Widerstände in Reihe geschaltet (siehe Abbildung 3.3). Die Eingangsspannung U_e liegt an der Gesamtschaltung an. Die Ausgangsspannung U_a wird zwischen den Widerständen R_a und R_b abgegriffen und kann mit der folgenden Formel berechnet werden:

$$U_a = U_e \cdot \frac{R_b}{R_a + R_b}$$

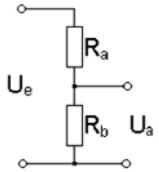


Bild 3.3: Spannungsteiler

Die Eingangsspannung unserer Widerstandsreihe wird durch den Widerstand $R1$ und die LED1 auf 2V herabgesetzt. Es ergibt sich für den ersten Operationsverstärker ein $R_a = R4 = 4.7k\Omega$ und ein $R_b = R5 + R6 + R7 + R8 = 6.59k\Omega$ am nicht-invertierten Eingang. Beim zweiten Operationsverstärker ist $R_a = R4 + R5 = 4.83k\Omega$ und $R_b = R6 + R7 + R8 = 6.46k\Omega$, beim dritten ist $R_a = R4 + R5 + R6 = 4.96k\Omega$ und $R_b = R7 + R8 = 6.33k\Omega$ und beim vierten Operationsverstärker ist $R_a = R4 + R5 + R6 + R7 = 5.09k\Omega$ und $R_b = R8 = 6.2k\Omega$.

Liegt nun die volle Akkuspannung an, so sind alle Spannungen an den nicht-invertierten Eingängen niedriger als die Referenzspannung an den invertierten Eingängen und keiner der Operationsverstärker schaltet. Dies bedeutet, dass am Ausgang des ersten Operationsverstärkers Masse anliegt und somit die LED2 zu leuchten beginnt, da deren Anode direkt mit Plus verbunden ist und deren Kathode über den Widerstand $R9$ auf Masse liegt.

Fällt nun die Spannung auf ca. 13V, so ist die Spannung, die zwischen $R4$ und $R5$ abgegriffen wird und am nicht-invertierten Eingang des ersten Operationsverstärkers anliegt, größer als die Referenzspannung und der Ausgang dieses Operationsverstärkers, Pin 1 des LM324, schaltet die Akkuspannung durch, so dass sowohl an der Anode der LED2 als auch an deren Kathode eine positive Spannung anliegt und diese dadurch erlischt. Die LED3 beginnt nun zu leuchten, da aufgrund der positiven Spannung am Ausgang des ersten Operationsverstärkers und der Masse, die sie über den Ausgang des zweiten Operationsverstärkers, Pin 7, und den Widerstand $R10$ erhält, eine Spannung anliegt.

Die weiteren Abstufungen der Spannungsanzeige sind:

- LED4 leuchtet ab ca. 12,7V
- LED5 leuchtet ab ca. 12,3V
- LED6 leuchtet ab ca. 12V

Eine Anmerkung noch zum Aussehen des LM324 auf der fertigen Platine. Da der benötigte IC nur in SMD-Bauweise erhältlich war, musste er mittels Drahtbrücken auf einen IC-Sockel gelötet werden, damit man ihn, falls er ausfällt, trotzdem noch ohne weitere Lötarbeiten auf der Platine austauschen kann.

Programmieradapter

Da das Robo32-Board nur über eine *SPI-Schnittstelle*[4] verfügt, über die man Software auf den ATmega32 übertragen kann, wird ein Adapter benötigt, mit dessen Hilfe man einen PC an diese Schnittstelle anschließen und diesen unter Verwendung von PonyProg 2000[5] beschreiben kann. Wir haben uns für einen selbstgebauten seriellen Programmieradapter entschieden, dessen Bauanleitung im Internet[6] zu finden ist. Ziel eines solchen Adapters ist es, die Signale der SPI-Schnittstelle in ein Signal umzuwandeln, das über die serielle Schnittstelle übertragen werden kann.

Die Kommunikation zwischen PC und Roboboard geschieht über vier Signalleitungen:

- **MISO:**
  Master-in-Slave-out, dies ist die Kommunikationsleitung vom Slave, also dem Roboboard, zum Master, also dem programmierenden PC.
- **MOSI:**
  Master-out-Slave-in, dies ist die Kommunikationsleitung vom Master zum Slave.
- **SCK:**
  System-Clock, auf dieser Leitung wird der Synchronisationstakt übertragen, der vom PC aus generiert wird.
- **Reset:**
  Auf dieser Leitung wird ein Low-Signal während des Schreibens übertragen, damit der ATmega32 in den beschreibbaren Modus versetzt wird.

[4] *Serial Peripheral Interface*, u. a. Programmierschnittstelle des ATmega32
[5] siehe Anhang G
[6] *http://s-huehn.de/elektronik/*

Eine Erläuterung der Basisprogrammierung erfolgt im Kapitel 4.2. Die Verwendung der Software zur Programmierung folgt in Anhang G. Den genauen Ablauf einer Programmiersequenz können Sie im Datenblatt des ATmega32[7] nachlesen.

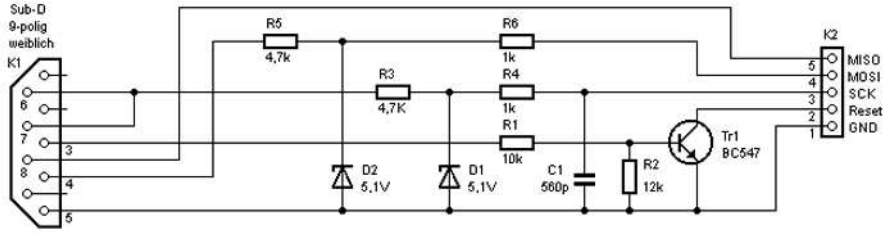


Bild 3.4: Schaltplan des Programmieradapters (Quelle: *http://s-huehn.de*)

Die Spannungsversorgung des Adapters wird von der seriellen Schnittstelle übernommen. Damit der ATmega32 keinen Schaden nimmt, werden die beiden Eingangssignale `MOSI` und `SCK` durch die Z-Dioden auf ca. 5V begrenzt. Das Reset-Signal wird durch den Transistor erzeugt, der bei anliegender Spannung die Masse durchschaltet. Der Kondensator `C1` soll laut Angaben von Scott-Frank Hühn, der diese Schaltung entworfen hat, Kommunikationsprobleme beseitigen, er ist aber nicht zwingend notwendig.

Dieser Programmieradapter ist über einen Pfostenstecker mit der SPI-Schnittstelle des Robo32-Boards verbunden. Da der ATmega32 über das sogenannte *ISP* (*In-System-Programming*) verfügt, kann er im eingebauten Zustand ohne Probleme programmiert werden. Die serielle Schnittstelle ist nach außen geführt, damit der Adapter genutzt werden kann, ohne dass der Roboter zerlegt werden muss.

Schalteranschluss

An den Schalteranschluss wird ein 2-Poliger Ein-Aus-Ein-Schalter angeschlossen. Durch diesen Schalter kann die Art der Spannungsversorgung (Akku oder Netzteil) gewählt werden. Wird Akku als Spannungsversorgung gewählt, so wird die komplette Versorgungsspannung des Roboters vom Akku geliefert. Wählt man das Netzteil als Spannungsver-

[7] siehe *datasheets/atmega32.pdf*, ab Seite 268

sorgung, so wird einerseits die Versorgungsspannung über das Netzteil, welches an der Netzteilbuchse angeschlossen sein sollte, geliefert und zum Anderen kann man bei dieser Schalterstellung den Akku über die Ladebuchse aufladen, ohne diesen ausbauen zu müssen.

3.1.2 Odometriebestimmung

Die Signalplatine dient dazu saubere High- und Low-Signale der Gabellichtschranken an die Interrupt-Eingänge des Robo32-Board zu übertragen. Ein sauberes High-Signal entspricht ca. 5V und eine sauberes Low-Signal entspricht ca. 0V.

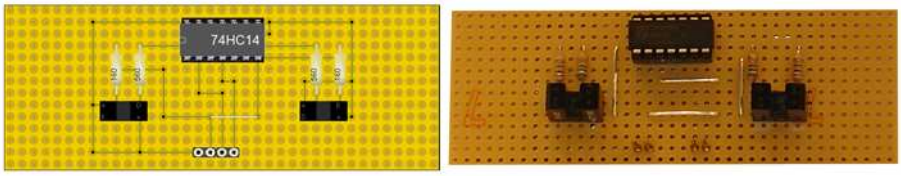


Bild 3.5: Geplante Platine (links) und fertiggestellte Platine (rechts)

Für die Odometriebestimmung wurde an eine Getriebewelle eine Taktgeberscheibe mit 120 Impulsen befestigt, die mittels einer Gabellichtschranke an das Robo32-Board übertragen werden. Da wir aber relativ günstige Gabellichtschranken verwendeten, konnten diese keinen der Impulsbalken erkennen. Abhilfe schaffte eine neue Einteilung der Taktgeberscheiben. Wir entschieden uns für nur noch 12 Impulsbalken, die wir durch eine einfache bedruckte Klebefolie auf die original Taktgeberscheiben aufklebten (siehe Bild 3.6).

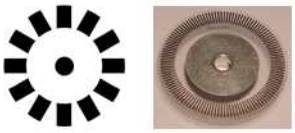


Bild 3.6: Neue Einteilung (links) und original Taktgeberscheibe (rechts

Die Gabellichtschranken erkannten zwar jetzt die Takte, lieferten aber keine sauberen High- und Low-Signale. Das High-Signal entsprach ca. 3,4V und das Low-Signal entsprach ca. 2,1V. Dieser geringe Unterschied konnte das Robo32-Board nicht auseinanderhalten, da der Spannungsunterschied zu gering war. Deshalb haben wir uns entschlossen einen Schmitt-Trigger IC zu verwenden, der diese Werte unterscheiden kann und saubere Signale ausgibt.

Wir verwenden einen Schmitt-Trigger IC des Typs *M74HC14*[8] der Firma *STMicroelectronics* . Dieser schaltet ab ca. 2,8V die Versorgungsspannung, also 5V durch, und liefert so ein sauberes High-Signal. Bis zu einer Spannung von ca. 2,3V liefert er die Masse, also 0V. Da der verwendete IC auch schnell genug schaltet, kann das Robo32-Board die Signale verarbeiten. Eine ausführlichere Erklärung der Funktionsweise der Gabellichtschranken können Sie in Kapitel 4.5 nachlesen.

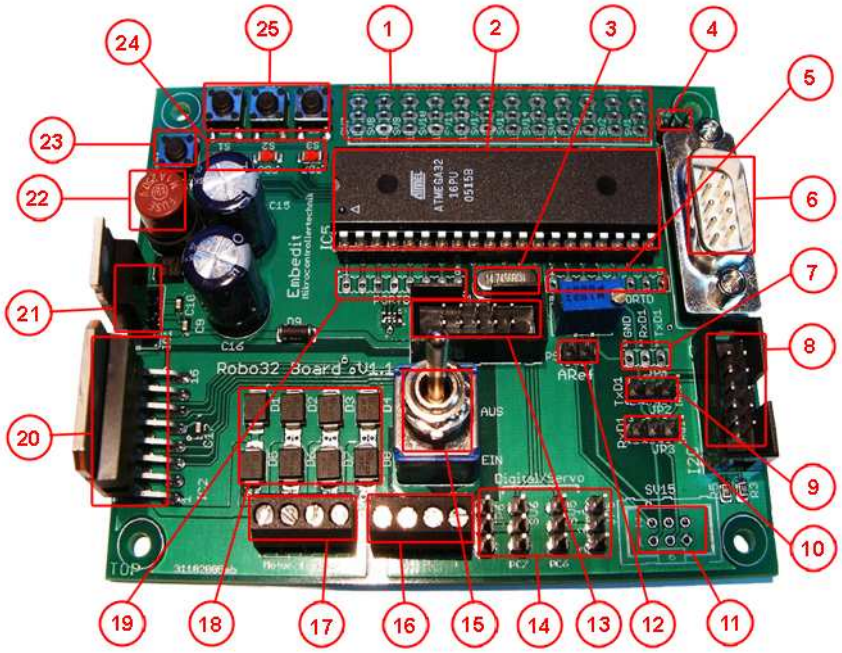
3.1.3 Steuerplatine: Robo32-Board

Da die Eigenentwicklung einer Mikrocontrollerplatine mit integriertem Motortreiber zu aufwändig wäre, entschieden wir uns für den Kauf einer fertigen Platine, die all unsere Bedürfnisse erfüllt. Nach diversen Recherchen im Internet fanden wir eine Platine auf Basis eines Atmel ATmega32 Mikrocontrollers, das *Robo32-Board* von *Embedit Mikrocontrollertechnik*[9], welches alle unsere Ansprüche erfüllte.

In diesem Unterkapitel stelle ich dieses Board genauer vor, dem Mikrocontroller selbst ist ein eigenes Kapitel gewidmet.

[8] siehe *datasheets/74HC14.pdf*
[9] *http://www.embedit.de*

Der Aufbau des Boards



[10]

(1) **8 Digital/Analog- und 4 reine Digital-I/O-Ports**
Jeder dieser Ports besteht aus einer 3-poligen Lochleiste. An Lochreihe 1 befinden sich die Signalein-, bzw. Ausgänge, die direkt an den Anschlüssen `PA0 - PA7`, für die Digital-/Analog-, und `PC2 - PC5`, für die reinen Digital-I/O-Ports, an dem ATmega32 angeschlossen sind. Die Lochleiste 2 ist mit Masse belegt und an der Lochleiste 3 liegt eine Spannung von 5V an. Wir haben diese Anschlüsse unter anderem dazu genutzt, um eine saubere 5V Spannung abzugreifen, z.B. für den Hecklüfter und die Signalplatine. Des Weiteren sollte man bei der Programmierung des Mikrocontrollers darauf achten, dass diese Ports entweder als Eingang oder als Ausgang definiert werden, da sonst Probleme auftreten können, wie z.B. falsche Signale, die schwer zu finden sind.

[10] Quelle: Embedit Microelectronics, überarbeitet

(2) **Der ATmega32**
Dies ist der Mikrocontroller, für den das Robo32-Board entwickelt wurde. Er wird im folgenden Unterkapitel genauer beschrieben.

(3) **14,7456 MHz Quarz**
Dieser Quarz wird anstelle des internen 1MHz Quarzes des Mikrocontrollers genutzt, um eine höhere Rechenleistung des Mikrocontrollers zu erreichen. Dieser Quarz ist über die Anschlüsse `XTAL1` und `XTAL2` mit dem ATmega32 verbunden und muss bei der Programmierung des Controllers aktiviert werden. Eine genaue Beschreibung der Einstellungen finden Sie im Anhang G „Programmierumgebung für den Mikrocontroller“.

(4) **Jumper `Enable RS232`**
Dieser Jumper dient dazu die serielle Schnittstelle (siehe Punkt 6) zu aktivieren. Ist er gesteckt, so kann die vorhandene Schnittstelle verwendet werden.

(5) **Port D**
Diese 8-polige Lochreihe beinhaltet auf dem Board verwendeten Ports, wie z.B. die Pulsweitenmodulation der Motoren[11], die natürlich auch direkt mit dem Mikrocontroller verbunden sind. Da wir die zweite serielle Schnittstelle des Boards nicht verwenden, können wir die beiden Interrupts an Port `PD3` und Port `PD4`, für unsere Zwecke nutzen. Über diese beiden Ports werden die Signale der Gabellichtschranken eingelesen, die dann im Mikrocontroller weiterverarbeitet werden können. Diese beiden Eingänge müssen aus den gleichen Gründen, wie unter Punkt 1 beschrieben, als Eingänge definiert werden.

(6) **Die serielle Schnittstelle (RS232-0)**
Über diese Schnittstelle wird die Kommunikation mit dem Barebone-Motherboard betrieben. Aus Platzproblemen haben wir die vorhandene Buchse aus- und an dieser Stelle ein 9-poliges Kabel eingelötet. Das Kabel wird mittels eines 10-poligen Pfostensteckers direkt auf dem Mainboard aufgesteckt. Zu beachten ist, dass das Kabel eine Nullmodemkabel-Belegung haben muss, damit beide miteinander kommunizieren können. Um nicht immer das interne Mainboard des Roboters benutzen

[11] siehe *datasheets/Robo32.pdf*, Seite 13

zu müssen, haben wir einen einfachen Adapter gebaut, mit dessen Hilfe man das Robo32-Board an einen externen PC anschließen kann, um z.B. die erstellten Programme mit diesem zu testen. Der Adapter besteht aus der ausgelöteten seriellen Buchse, an die einfach ein Stück Kabel mit einer Stiftleiste angelötet wurde, die genau in die Pfostenleiste passt. Eine Erläuterung der Programmierung der RS232-Schnittstelle finden Sie im Kapitel 4.3.

(7) **Jumper 4**
Dieser Anschluss liefert die seriellen Signale einer zweiten RS232-Schnittstelle.

(8) **I²C-Schnittstelle**
An dieser Pfostensteckerleiste können bis zu 112 I²C-Module angeschlossen werden, in unserem Fall nur der Ultraschallsensor. Da der I²C-Bus nur zwei Datenleitungen, `SDA` (Datenleitung) und `SCL` (Taktleitung), benötigt, wird er auch *Two-Wire-Interface* genannt. Die Funktionsweise des Ports und des Ultraschallsensors werden in Kapitel 4.6 genauer erklärt. Die Belegung des Ports ist in der Robo32-Board-Dokumentation[12] nachzulesen.

(9) **Jumper 2**
An diesem Jumper wird die Verwendung des Interrupts `INT0` eingestellt, wird er auf der Position 1-2 gebrückt, so steht `INT0` dem I²C-Port zur Verfügung. Will man die zweite RS232-Schnittstelle benutzen, so muss man die Pins 2-3 brücken. Benötigt man weder die zweite serielle Schnittstelle noch den `INT0` am I²C-Bus, wie in unserem Fall, so darf man keinen Jumper setzen. Dann kann man `INT0` an dem Erweiterungsport `SV15`, bzw. falls dieser auch nicht genutzt wird, an Port `PD3` nutzen.

(10) **Jumper 3**
Dieser Jumper regelt die Einstellungen für den Interrupt `INT1`. Es gelten die gleichen Richtlinien wie bei Jumper 2.

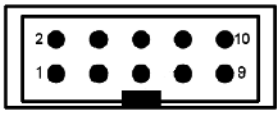
(11) **Erweiterungsport `SV15`**
Dieser Port kann genutzt werden, falls keine serielle Schnittstelle und keine der

[12] siehe *datasheets/Robo32.pdf*, Seite 11

beiden Interrupts `INT0` und `INT1` benötigt werden. Eine genaue Belegung des Ports ist in der Dokumentation[13] nachzulesen.

(12) **Referenzspannungsanschluss `ARef`**
An dieser Stiftleiste kann man eine externe Referenzspannung anlegen, die dann im Mikrocontroller verwendet wird. Sie ist direkt mit dessen Ports `AREF` und `GND` verbunden. Brückt man diesen Anschluss, so wird der Mikrocontroller zerstört, da keine Absicherung des Ports vorliegt.

(13) **SPI**
Das *Serial Peripheral Interface* ist eine allgemeine Kommunikationsschnittstelle des ATmega32. In unserem Fall wird sie aber nur zur Programmierung des Mikrocontrollers genutzt. Die Belegung des Ports sehen Sie in Bild 3.7. An Pin 2 des Pfosten-



Pin 1: MOSI	Pin 2: VCC
Pin 3: Not Connected	Pin 4: GND
Pin 5: RESET	Pin 6: GND
Pin 7: SCK	Pin 8: GND
Pin 9: MISO	Pin 10: GND

Bild 3.7: Die Belegung des SPI (Quelle: *datasheets/Robo32.pdf*)

steckers liegt eine Spannung von 5V und an den Stiften 4, 6, 8 und 10 liegt Masse an. Deshalb sollte man darauf achten, dass die Stifte nicht gebrückt werden, da sonst ein Kurzschluss entstehen könnte, der zur Zerstörung des Mikrocontrollers führen würde. Die LEDs (siehe Punkt 24) nutzen die gleichen Ports (Port `PB5` bis Port `PB7`) des Mikrocontrollers, weshalb diese während des Programmierens flackern. Die Beschreibung einer Programmiersequenz mit Hilfe des Programms *PonyProg 2000* wird in Unterkapitel G beschrieben.

(14) **Digital-Servo-Anschluss**
An den Anschlüssen `SV5` und `SV6` können digitale Servos von *Conrad*, *Futaba*

[13] siehe *datasheets/Robo32.pdf*, Seite 16

oder *Robbe* angeschlossen werden. Sie nutzen eine andere Pin-Belegung, hier ist die positive Spannung in der Mitte und die Masse außen. Diese Anschlüsse sind direkt mit dem Mikrocontroller über die Ports `PC6` und `PC7` verbunden. Die Jumper `JP5` und `JP6` stellen die Art der Versorgungsspannung ein. Sind die Pins 1 und 2 gebrückt, so werden die Servos über die 5V-Spannung des Boards versorgt. Sind aber die Pins 2 und 3 gebrückt, so wird eine externe Spannung, über den Anschluss `X3-S` (siehe Punkt 16), verwendet.

(15) **Schalter**
Mit diesem Schalter kann man das Robo32-Board ein- bzw. ausschalten.

(16) **Spannungsversorgung**
Hier wird die Spannungsversorgung angeschlossen. An „`GND`“ wird die Masse angeschlossen, an „+“ die positive Versorgungsspannung, die nicht größer als 30 V sein sollte, da das sonst zur Zerstörung des Robo32-Boards führen würde. Da diese Spannung intern nicht abgesichert ist, sollte man sie vorher absichern. Deshalb haben wir eine der beiden Sicherungen in die Seitenwand des Roboters eingesetzt. Am Anschluss „`5V`“ kann man 5V für externe Geräte abgreifen. Dieser Anschluss ist über die interne 1A-Sicherung (Punkt 22) abgesichert. An `S` wird gegebenenfalls eine externe Spannungsquelle für die Sensoren SV5 und SV6 (siehe Punkt 14) angeschlossen.

(17) **Motoranschluss**
Hier werden die beiden Gleichstrommotoren angeschlossen. Alternativ ist auch die Verwendung eines Schrittmotors möglich. Die Motoren dürfen jeweils einen Strom von 2A nicht übersteigen, da dies auch zur Zerstörung des Boards führen würde. Eine Erklärung der Motorsteuerung lesen Sie unter Punkt 20 und eine Erklärung der Pulsweitenmodulation folgt in Kapitel 4.4.1.

(18) **Freilaufdioden**
Es stehen pro Motor 4 Schutzdioden zur Verfügung, die die Motoren entstören und somit das Board vor den möglichen Spannungsspitzen schützen.

(19) **Port B**

Diese 8-polige Lochreihe beinhaltet ebenfalls die auf dem Board verwendeten Ports, in diesem Fall die Richtungseinstellung des zweiten Motors, die drei Taster (siehe Punkt 25) und die drei LEDs (siehe Punkt 24)[14]. Alle Ports sind natürlich auch direkt mit dem Mikrocontroller verbunden.

(20) **Motortreiber**

Als Motortreiber wird der *L298*-Baustein[15] eingesetzt. Dieser IC liefert den benötigten Strom für die beiden Motoren, der max. 2A pro Motor beträgt. Pro Motor hat der IC zwei Eingänge, die mit den Ports `PB1` und `PB2` (Richtung Motor 2), für den einen, und den Ports `PD7` und `PD8` (Richtung Motor 1), für den anderen Motor, des ATmega32 verbunden sind. Über diese Ports wird die Drehrichtung der Motoren bestimmt. Allein eine Spannung auf diesen Ports genügt noch nicht, um die Motoren zu einer Bewegung zu treiben. Man muss auch noch ein Signal auf die Enable-Ports des IC legen. Diese sind mit den Ports `PD5` (PWM2) und `PD6` (PWM1) des Mikrocontrollers verbunden. Über die Pulsweitenmodulation (siehe Kapitel 4.4.1) wird dann die Geschwindigkeit der Motoren eingestellt. Da in unserem Fall die Motoren in entgegengesetzter Richtung laufen müssen, damit der Roboter geradeaus fährt, müssen auch die Input-Signale, die der L298 erhält, entgegengesetzte Signale sein.

(21) **Spannungswandler**

Auf dem Robo32-Board ist ein Spannungswandler-Baustein des Typs *7805*[16]. Dieser IC regelt die anliegende Versorgungsspannung auf 5V herunter, und versorgt mit dieser Spannung das komplette Board. Der abgegriffene Strom darf 1A nicht übersteigen, sonst wird der Baustein zerstört. Diese 1A beziehen sich auf das komplette Board, incl. aller an den analogen bzw. digitalen Schnittstellen angeschlossenen Sensoren und/oder Servos.

(22) **Sicherung**

Diese 1A-Sicherung sorgt dafür, dass die max. 1A des Spannungswandlers nicht überschritten wird.

[14] siehe *datasheets/Robo32.pdf*, Seite 13
[15] siehe *datasheets/L298.pdf*
[16] siehe *datasheets/7805.pdf*

(23) **Reset-Taster**
Mit diesem Taster kann man den Mikrocontroller neu starten, anstatt das komplette Board aus- und wieder einzuschalten.

(24) **LEDs**
Diese drei LEDs sind direkt an die Ports `PB5`, `PB6` und `PB7` des Mikrocontrollers angeschlossen. Man kann diese direkt über den ATmega32 ansprechen. Sie haben keinen weiteren Zweck, sind aber recht sinnvoll, da man diese als einfache Kontrollwerkzeuge bei der Programmierung einsetzen kann.

(25) **Taster**
Die drei Taster sind ebenfalls direkt mit dem ATmega32 verbunden und zwar über die Ports `PB2`, `PB3` und `PB4`. Mit den Tastern kann man ebenfalls die Programmierung überprüfen, z.B. wenn das Programm auf ein Eingangssignal wartet. Sie haben sonst aber ebenfalls keinen weiteren Zweck.

Aktuelle Konfiguration des Robo32-Boards

Zur Zeit sind an dem Robo32-Board drei Sensoren angeschlossen. Die beiden Infrarotsensoren des Typs `GP2D120` von Sharp sind an den Digital/Analogen Eingängen des Boards angeschlossen. Der Ultraschallsensor wird über den I^2C-Bus angesteuert. Des Weiteren sind die Signalleitungen der Signalplatine an den Interrupt-Eingängen `PD3` und `PD4` angeschlossen. Der Lüfter, sowie die Signalplatine erhalten ihre Spannung über die Pins 2 und 3 zweier unbenutzter Digital/Analog-Eingänge.

3.2 Mikrocontroller: ATmega32

In diesem Unterkapitel wird der *ATmega32* von *Atmel* genauer beschrieben. Im ersten Teil werden die Eigenschaften des Mikrocontrollers beschrieben. Im zweiten Teil werden die auf dem Robo32-Board verwendeten Ports und Einstellungen beschrieben.

3.2.1 Eigenschaften im Überblick

Der Mikrocontroller ATmega32 nutzt eine erweiterte *RISC*-Architektur[17], d.h. er besitzt u.a. 131 auf dem Chip gespeicherte Befehle, die meist in einem Taktzyklus ausgeführt werden können. Das bedeutet, dass er bis zu 16 Millionen Befehle pro Sekunde verarbeiten kann (bei einen Systemtakt von 16MHz). Intern arbeitet der Mikrocontroller mit einem Takt von 1 MHz, der aber durch die Verwendung eines externen Quarzes auf bis zu 16 MHz erhöht werden kann.

Es stehen drei Arten von internem Speicher zur Verfügung: Ein 32 KByte großer Flash-Speicher, der die Programme aufnehmen kann, ein 2 KByte großer SRAM-Speicher, der als Datenspeicher fungiert und ein 1 KByte großer EEPROM-Speicher, der ebenfalls als Datenspeicher benutzt wird. In einem 32 Byte großen Register, dem *General Purpose Working* Register, werden die Werte abgelegt, die die ALU dann mit den integrierten Befehlen verarbeitet. Dieses Register teilt sich in zwei Teile `R0` bis `R15` und `R16` bis `R31`, wobei die Register `R26` bis `R31` auch als Pointer-Register verwendet werden können.

Des Weiteren besitzt dieser Mikrocontroller eine JTAG-, USART-, TWI- und eine SPI-Schnittstelle. Die *JTAG*-Schnittstelle[18] dient dem Debuggen von Programmcode auf dem Controller, d.h. man kann über diese Schnittstelle erstellten Programmcode überprüfen. Über die *USART*-Schnittstelle[19] kommuniziert der ATmega32 mit dem PC, es ist eine serielle Kommunikationsschnittstelle, die nur zwei Leitungen benötigt, erstens `TxD` (Transceive Data) und zweitens `RxD` (Receive Data). Die *TWI*-Schnittstelle[20] ist eine einfache aber leistungsfähige und flexible Kommunikationsschnittstelle. Sie wird in Kapitel 4.6 genauer beschrieben. Die *SPI*-Schnittstelle[21] ist eine universelle Hochgeschwindigkeits-Schnittstelle, die synchrone Datenübertragung zwischen mehreren Mikrocontrollern oder einem Mikrocontroller und einem PC mittels Adapter erlaubt. Über diese Schnittstelle wird auch unser Programmieradapter angeschlossen, der, da der ATmega32 das *In-System-Programming* unterstützt, eine Programmierung in eingebautem Zustand erlaubt.

[17] *RISC = Reduced Instruction Set Computing*
[18] *JTAG = Joint Test Action Group = IEEE-Standard 1149.1*
[19] *USART = Universal Synchronous and Asynchronous Receiver and Transmitter*
[20] *TWI = Two Wire Interface = I^2C = Inter-Integrated Circuit*
[21] *SPI = Serial Peripheral Interface*

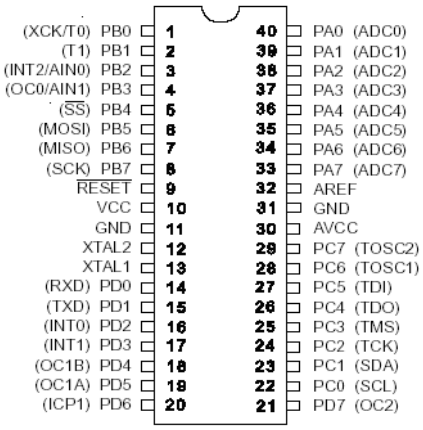


Bild 3.8: Pinbelegung des ATmega32 (Quelle: *datasheets/atmega32.pdf*)

Neben einem 8-Kanal, 10-Bit A/D-Wandler besitzt der ATmega32 noch folgende Eigenschaften:

- vier Pulsweitenmodulations-Kanäle
- zwei 8-Bit Timer/Counter
- ein 16-Bit Timer/Counter
- 32 frei programmierbare Ein- bzw. Ausgänge
- ein Analog-Komparator

3.2.2 Verwendete Ports und Einstellungen

Die Pinbelegung des ATmega32 und deren Verwendung

In der anschließenden Tabelle werden die Funktion der einzelnen Pins des ATmega32 beschrieben.

PIN	FUNKTION
PD0	Entweder ein Timer-Eingang oder ein externer Takteingang für den USART
PD1	Ein weiterer Timer-Eingang
PD2	Entweder der externer Interrupt 2 oder der eine Eingang des Analog-Komparators
PD3	Entweder der Output-Compare-Ausgang des Timer 0 oder der zweite Eingang des Analog-Komparators
PD4	Ein Pin der SPI-Schnittstelle, der benötigt wird um den richtigen Slave am Bus zu wählen
PD5	Datenausgang der SPI-Schnittstelle
PD6	Dateneingang der SPI-Schnittstelle
PD7	Taktleitung der SPI-Schnittstelle
RESET	Wie der Name schon sagt, um den Controller zurückzusetzen
VCC	Versorgungsspannung
GND	Masse
XTAL2	Eingang eines externen Taktgebers (Quarz)
XTAL1	Der zweite Eingang eines externen Taktgebers (Quarz)
PD0	Eingang der USART-Schnittstelle
PD1	Ausgang der USART-Schnittstelle
PD2	Externer Interrupt 0
PD3	Externer Interrupt 1
PD4	1. PWM-Ausgang des Timers 1
PD5	2. PWM-Ausgang des Timers 1
PD6	Eingang der Capture-Funktion des integrierten Zählerbausteins
PD7	PWM-Ausgang des Timers 2
PC0	Clockleitung der I^2C-Schnittstelle
PC1	Datenleitung der I^2C-Schnittstelle
PC2	Test Clock der JTAG-Schnittstelle
PC3	Test Mode Select Input der JTAG-Schnittstelle
PC4	Test Data Output der JTAG-Schnittstelle
PC5	Test Data Input der JTAG-Schnittstelle
PC6	Anschluss 1 eines externen Uhrenquarzes

PC7	Anschluss 2 eines externen Uhrenquarzes
AVCC	Spannungsanschluss für den AD-Wandler
GND	Masse
AREF	Referenzspannung für den AD-Wandler
PA7	8. Eingang des AD-Wandlers
PA6	7. Eingang des AD-Wandlers
PA5	6. Eingang des AD-Wandlers
PA4	5. Eingang des AD-Wandlers
PA3	4. Eingang des AD-Wandlers
PA2	3. Eingang des AD-Wandlers
PA1	2. Eingang des AD-Wandlers
PA0	1. Eingang des AD-Wandlers

In der folgenden Tabelle sind die benutzten Pins des Mikrocontrollers und deren Verwendung aufgeführt. Alle anderen Pins des Controllers sind zwar auf dem Robo32-Board ausgeführt, spielen aber bei unserem Roboter zur Zeit noch keine Rolle.

VERWENDUNG	PINS
Drehrichtung rechter Motor	PD7, PD8
Drehrichtung linker Motor	PB1, PB2
Geschwindigkeit rechter Motor	PD6
Geschwindigkeit linker Motor	PD5
Infrarotsensor vorne	PA1
Infrarotsensor hinten	PA3
Ultraschallsensor	PC0, PC1
Programmieradapter	PB4, PB5, PB6, PB7
Gabellichtschranken	PD3, PD4
Spannungsversorgung	VCC, GND
Serielle Schnittstelle	PB0, PD0, PD1

Die Fuse-Bits

Mit diesen Bits werden Einstellungen des ATmega32 konfiguriert, es wird z.B. dem Mikrocontroller mitgeteilt, dass ein externer Quarz verwendet werden soll. Eine Beschreibung

der einzelnen Fuse-Bits sehen Sie in der folgenden Tabelle. Des Weiteren ist in dieser Tabelle aufgeführt, ob die Bits gesetzt sind oder nicht.

FUSE-BIT	BEDEUTUNG	GESETZT
`CKSEL0`, `CKSEL1`, `CKSEL2`, `CKSEL3`	Mit diesen vier Bits wird die Quelle des Systemtaktes eingestellt, da bei unserem System ein externer Quarz verwendet wird, sind alle vier Bits gesetzt[22]	ja
`SUT0`, `SUT1`	Hier wird die Startzeit eingestellt, die abhängig vom verwendeten Taktgeber gewählt werden sollte[23]	ja
`BODEN`	Über dieses Bit wird eine Versorgungsspannungsüberwachung (Brown-out Detector) aktiviert, die gegebenfalls einen RESET des Mikrocontrollers durchführt[24]	ja
`BODLEVEL`	Durch dieses Bit wird die Spannung eingestellt, ab der ein RESET durchgeführt wir, in unserem Fall 4 V	ja
`BOOTRST`	Mit diesem Bit wird die Bootquelle nach einem RESET eingestellt, entweder Adresse 0, oder wie in unserem Fall der Bootloader-Bereich	ja
`BOOTSZ0`, `BOOTSZ1`	Durch diese beiden Bits wird der Bereich und die Größe des Bootloaderbereichs eingestellt[25]	nein
`EESAVE`	Mit Setzten dieses Bits wird der EEPROM-Speicher bei jedem Flashen gelöscht. Wird dieses Bit nicht gesetzt, so ist der Speicher geschürzt	ja
`CKOPT`	Durch dieses Bit kann ein Verstärkung des Taktsignals eingestellt werden[26]	ja
`SPIEN`	Wird dieses Bit gesetzt, kann man den Mikrocontroller nicht mehr über das SPI programmieren, also **niemals** dieses Bit setzen	nein
`JTAGEN`	Dieses Bit aktiviert die JTAG-Schnittstelle des Mikrocontrollers	ja
`OCDEN`	Durch dieses Bit wird der OnChip-Debugger aktiviert	ja

Eine ausführlichere Beschreibung des Mikrocontrollers finden Sie im Datenblatt[27].

[27] siehe *datasheets/atmega32.pdf*

Kapitel 4

Programmierung des ATmega32-Mikrocontrollers

Die hardwarenahe Steuerung des MiniBots erfolgt über ein *Robo32-Board*[1], das mit einem *ATmega32*[2]-Mikrocontroller[Gad00] von *Atmel*[3] bestückt ist. Tabelle 4.1 zeigt die Hauptmerkmale des Boards. Für weitere Details zum verwendeten Robo32-Board, siehe Kapitel 3.1.3, bzw. Kapitel 3.2 für weitere Informationen zum ATmega32-Mikrocontroller.

Die Programmierung des Mikrocontrollers erfolgt in *AVR-Assembler*[Mor02, RHB06], einer von Atmel speziell für deren Mikrocontroller entwickelten Assemblersprache. Unter *http://www.mikrocontroller.net/tutorial/* steht ein gutes Tutorial zum Einstieg in die Programmierung mit AVR-Assembler zur Verfügung. Eine ausführliche Befehlsreferenz befindet sich auf der beiliegenden CD unter `specifications/avr.pdf`.

In Anhang G werden die zur Programmierung verwendeten Tools und deren Einrichtung erläutert.

[1] siehe *datasheets/robo32.pdf*
[2] siehe *datasheets/atmega32.pdf*
[3] *http://www.atmel.com/*

Tabelle 4.1: Merkmale des Robo32-Boards mit ATmega32

ATmega32 mit 14,7456 MHz
RS232-Schnittstelle
L298-Motortreiber für zwei 2A-Motoren
5V Spannungsregler für variable Eingangsspannung
I^2C-Port
STK200 kompatibler ISP-Programmieranschluss
8 AD-Wandler Eingänge
6 digitale I/O-Ports
3 Taster und 3 LEDs

4.1 Client-Server-Kommunikation

Die Kommunikation zwischen Mikrocontroller und Barebone-Motherboard findet über eine *serielle RS232-Schnittstelle*[Kai89, Sey88] nach dem *Client-Server*-Prinzip statt. Das Barebone-Motherboard fungiert als Client und sendet Bytes bzw. Befehle an den Mikrocontroller, der diese als Server verarbeitet und eine Antwort an den Client zurücksendet. So kann der Client den MiniBot durch Senden bestimmter Befehle steuern oder Statusinformationen abfragen. Die Kommunikation zwischen Barebone-Motherboard und Mikrocontroller bzw. Client und Server verläuft nach einem festgelegten Protokoll, das in Tabelle 4.2 dargestellt ist. Die Protokollcodes sind in `Protocol.asm`[4] als Aliase hinterlegt. Aliase dienen bei AVR-Assembler zur Definition von benannten Konstanten, die durch den Präprozessor des AVR-Compilers ersetzt werden (vergleichbar mit der `#define`-Direktive des `C++`-Präprozessors).

Mit `REQUEST_TEST` kann die serielle Verbindung, sowie die korrekte Funktionsweise des Mikrocontrollers getestet werden. Das als Parameter gesendete Byte wird inkrementiert und an den Client zurückgesendet.

Durch Senden von `REQUEST_ADC` wird an den Pins, an denen die Infrarot-Sensoren angeschlossen sind, eine Analog-Digital-Wandlung ausgelöst. Anschließend wird dann

[4] siehe *prog/MiniBot/Protocol.asm*

Tabelle 4.2: Protokoll der Client-Server-Kommunikation

Request	Request-Parameter	Response	Response-Parameter
REQUEST_TEST	X	inc(X)	-
REQUEST_ADC	-	RESPONSE_REQUEST_SUCCESSFUL	FH FL RH RL
REQUEST_PWM_SET_VELOCITY	LV RV	RESPONSE_REQUEST_SUCCESSFUL	-
REQUEST_PWM_START_ACTION	AC	RESPONSE_REQUEST_SUCCESSFUL RESPONSE_ACTION_UNKNOWN	- -
REQUEST_PWM_START_ACTION_INTERVAL	AC IH IL	RESPONSE_REQUEST_SUCCESSFUL RESPONSE_ACTION_UNKNOWN	- -
REQUEST_PWM_STOP_ACTION	-	RESPONSE_REQUEST_SUCCESSFUL	-
REQUEST_I2C_START_MEASUREMENT	-	RESPONSE_REQUEST_SUCCESSFUL RESPONSE_I2C_ERROR	- EC
REQUEST_I2C_GET_DISTANCE	-	RESPONSE_REQUEST_SUCCESSFUL RESPONSE_I2C_ERROR	DH DL EC
REQUEST_GET_STATE	-	RESPONSE_REQUEST_SUCCESSFUL	AS
sonst	-	RESPONSE_REQUEST_UNKNOWN	-

Abkürzungen:

FH	=	Front-Sensor High-Byte	IH	=	Intervall High-Byte
FL	=	Front-Sensor Low-Byte	IL	=	Intervall Low-Byte
RH	=	Rear-Sensor High-Byte	EC	=	Error-Code
RL	=	Rear-Sensor Low-Byte	AC	=	Action-Code
LV	=	Linke Geschwindigkeit	AS	=	Action-State
RV	=	Rechte Geschwindigkeit	X	=	beliebiges Byte

RESPONSE_REQUEST_SUCCESSFUL gefolgt von jeweils Low- und High-Byte der ermittelten Werte an den Client gesendet (siehe Kapitel 4.7).

Die Geschwindigkeit der beiden Motoren kann mit REQUEST_PWM_SET_VELOCITY gefolgt von den zwei Bytes für die linke und rechte Geschwindigkeit getrennt eingestellt werden (siehe Kapitel 4.4). Mit RESPONSE_REQUEST_SUCCESSFUL wird dies vom Server bestätigt. Um eine Bewegung des Roboters mit der gewählten Geschwindigkeit zu starten, kann der Client REQUEST_PWM_START_ACTION mit anschließendem *Action-Code*[5], der die Art der gewünschten Bewegung codiert, senden (siehe Kapitel 4.4). Der Server bestätigt dies wieder mit RESPONSE_REQUEST_SUCCESSFUL bzw. mit RESPONSE_ACTION_UNKNOWN, wenn eine unbekannte Bewegung gefordert wurde. Mit REQUEST_PWM_START_ACTION_INTERVAL kann eine Bewegung eines bestimmten Intervalls ausgelöst werden. Hierbei wird nach dem *Action-Code* zusätzlich High- und Low-Byte der gewünschten Anzahl von Taktimpulsen der Taktgeberscheiben gesendet. Näheres hierzu in Kapitel 4.5. Die Bewegung kann jederzeit wieder mit REQUEST_PWM_STOP_ACTION gestoppt werden, was der Server wiederum durch Senden von RESPONSE_REQUEST_SUCCESSFUL bestätigt (siehe Kapitel 4.4).

Mit REQUEST_I2C_START_MEASUREMENT kann der Client eine Entfernungsmessung des Ultraschall-Sensors auslösen. Mit RESPONSE_REQUEST_SUCCESSFUL wird dies bei Erfolg vom Server bestätigt, andernfalls sendet er RESPONSE_I2C_ERROR gefolgt von dem I^2C-Fehlercode, wenn ein Fehler bei der I^2C-Kommunikation aufgetreten ist. Nach 65ms kann der Client mit REQUEST_I2C_GET_DISTANCE den gemessenen Wert in cm abrufen. Der Client sendet dann RESPONSE_REQUEST_SUCCESSFUL und anschließend High- und Low-Byte der gemessenen Entfernung bzw. einen Fehlermeldung analog zu REQUEST_I2C_START_MEASUREMENT. Eine genaue Beschreibung folgt in Kapitel 4.6.

Den Bewegungszustand des MiniBots kann der Client mit REQUEST_GET_STATE erfragen. Der Server antwortet mit RESPONSE_REQUEST_SUCCESSFUL gefolgt von dem derzeitigen Zustand (siehe Kapitel 4.4). Hiermit kann beispielsweise festgestellt werden, ob eine intervallbasierte Bewegung abgeschlossen wurde.

[5] siehe *prog/MiniBot/Actions.asm*

Sendet der Client einen undefinierten Protokollcode, so antwortet der Server, um dies zu signalisieren, mit `RESPONSE_REQUEST_UNKNOWN`.

4.2 Basisprogrammierung des Mikrocontrollers

Die Programmierung des Mikrocontrollers ist in Module gegliedert, die im Basismodul `MiniBot.asm`[6] über `.include`-Anweisungen importiert werden. Dort wird am Anfang `m32def.inc` inkludiert. Hier befinden sich spezielle Aliase für den ATmega32. Damit kann beispielsweise auf die I/O-Register mit einem Namen anstatt ihrer Speicheradresse zugegriffen werden. Danach wird allerdings der CPU-Takt `CLOCK` auf `14,7456 MHz` gesetzt, da der in `m32def.inc` definierte Takt `4 MHz` beträgt, was später zu falschen Berechnungen führen würde. Anschließend werden `Protocol.asm` und `Actions.asm` inkludiert, die die bereits erwähnten Aliase für das Protokoll und die Action-Codes enthalten. Die inkludierte Datei `Registers.asm` enthält eindeutige Aliase für die global verwendeten Standard-Register, um eine doppelte Verwendung auszuschließen.

Anschließend folgt der Interruptvektor (siehe Listing 4.1), der die Einsprungpunkte für die Interruptroutinen enthält. Da nur drei Interrupts benutzt werden, bleiben die anderen Stellen des Interruptvektors leer. An Adresse `0x000` beginnt die Abarbeitung der Befehle nach jedem Reset. Dort wird die Routine `RESET` aufgerufen. An der Adresse `INT0addr`, dem Einsprungpunkt für den ersten externen Interrupt (weiteres zu den externen Interrupts siehe Kapitel 4.5), wird ein Sprung zu Routine `INT_0` eingetragen. Analoges gilt für Adresse `INT1addr`, der Einsprungadresse des zweiten externen Interrupts. Hier wird als Interruptroutine `INT_1` eingetragen. An Adresse `URXCaddr` (*USART Read Complete*) (siehe Kapitel 4.3) wird als Interruptroutine `INT_RS232` eingetragen. Dieser Interrupt wird ausgelöst, wenn ein Byte bzw. ein Befehl des Clients über die serielle RS232-Schnittstelle empfangen wurde. Hierdurch wird die weitere Verarbeitung dieses Befehls angestoßen.

Die `RESET`-Routine (siehe Listing 4.2) wird bei jedem Neustart des Mikrocontrollers aufgerufen. Nachdem der Stackpointer initialisiert wurde, wird zunächst `SLEEP_INIT` auf-

[6] siehe *prog/MiniBot/MiniBot.asm*

Listing 4.1: MiniBot.asm

```
13 ;interrupt-vector
14 .org 0x000                          ;start of execution after reset
15   rjmp RESET
16 .org INT0addr                       ;interrupt-address for external interrupt 0
17   rjmp INT_0
18 .org INT1addr                       ;interrupt-address for external interrupt 1
19   rjmp INT_1
20 .org URXCaddr                       ;interrupt-address for URXC interrupt
21   rjmp INT_RS232
```

gerufen. Dort wird der Sleep-Modus aktiviert und auf *idle* gesetzt. Im *idle*-Modus wird zwar der CPU-Takt angehalten, alle anderen Module, wie etwa USART, I^2C, Pulsweitenmodulation, etc. bleiben aber aktiv. Andernfalls könnte der Mikrocontroller beispielweise nicht durch den Interrupt eines ankommenden Bytes bzw. Befehls vom Client aufgeweckt werden. Außerdem würde die Pulsweitenmodulation und somit die Motoren beim Übergang in den Sleep-Modus gestoppt. Daraufhin wird `LED_INIT` aufgerufen, wodurch lediglich die LED-Pins `B5`, `B6` und `B7` als Ausgangsports definiert werden. Anschließend werden diverse Initialisierungsroutinen für RS232-Interface, A/D-Wandler, Pulsweitenmodulation, I^2C-Interface und die externen Interrupts aufgerufen. Ihre Funktionsweise wird in den dazugehörigen Kapiteln erklärt. Nachdem der aktuelle Bewegungszustand auf `ACTION_NONE` gesetzt wurde (siehe Kapitel 4.4), werden die Interrupts global aktiviert und der Mikrocontroller in den Sleep-Modus versetzt. Dies befindet sich in einer Schleife, sodass der Mikrocontroller, nach Abarbeitung einer durch einen Interrupt ausgelösten Interruptroutine, sofort wieder in den Sleep-Modus versetzt wird.

Die Routine `INT_RS232` (siehe Anhang D.5) übernimmt die Abarbeitung der Client-Befehle. Im Normalzustand befindet sich der Mikroprozessor im Sleep-Modus. Sobald der Client ein Byte bzw. einen Befehl (siehe Tabelle 4.2 auf Seite 43) sendet, wird der Interrupt `URXC` (*USART Read Complete*) ausgelöst und die Interruptroutine `INT_RS232` durch den Sprungbefehl an Adresse `URXCaddr` des Interruptvektors aufgerufen. Nachdem die temporären Register und das Statusregister auf dem Stack gesichert wurden, wird der Interrupt für die RS232 Schnittstelle deaktiviert und anschließend die globale Interruptbehandlung wieder aktiviert. Die generelle Interruptbehandlung muss sofort wieder aktiviert werden,

Listing 4.2: MiniBot.asm

```
25 ;The reset routine is called on every reset of the microcontroller.
26 RESET:
27   ldi TEMPL, LOW(RAMEND)                ;initialize stackpointer
28   ldi TEMPH, HIGH(RAMEND)
29   out SPL, TEMPL
30   out SPH, TEMPH
31
32   rcall SLEEP_INIT                      ;initialize sleep-mode
33   rcall LED_INIT                        ;initialize LEDs
34   rcall RS232_INIT                      ;initialize RS232-interface
35   rcall ADC_INIT                        ;initialize ADC-unit
36   rcall PWM_INIT                        ;initialize PWM-unit
37   rcall I2C_INIT                        ;initialize I2C-interface
38   rcall INT_INIT                        ;initialize external interrupts
39
40   ldi TEMP1, ACTION_NONE                ;set initial state
41   mov ACTION_STATE, TEMP1
42   sei                                   ;enable interrupts
43   SLEEP_LOOP:
44     sleep                               ;put mc in idle-mode
45     rjmp SLEEP_LOOP
```

da bei einer Reaktivierung am Ende der Interruptroutine, eventuell dazwischen aufgetretene externe Interrupts durch Impulse der Gabellichtschranken (siehe Kapitel 4.5) nicht registriert würden, weil die Abarbeitung von `INT_RS232` unter Umständen sehr viel Zeit in Anspruch nimmt. Wäre der RS232-Interrupt jedoch vorher nicht explizit deaktiviert worden, so würden bei der nun folgenden Client-Server-Kommunikation weitere RS232-Interrupts ausgelöst und jedesmal die Routine `INT_RS232` erneut aufgerufen, was keinesfalls passieren darf.

Nun wird der empfangene Befehlscode mit den im Protokoll definierten Befehlscodes verglichen und die zugehörige Bearbeitungsroutine `FUNCTION_*` aufgerufen. Erwähnt seien an dieser Stelle lediglich `FUNCTION_TEST`, die ein weiteres Byte empfängt, inkrementiert und zurücksendet, und `FUNCTION_UNKNOWN`, die bei einem unbekannten Befehlscode aufgerufen wird und `RESPONSE_REQUEST_UNKNOWN` zurücksendet. Die übrigen Bearbeitungsroutinen werden in den folgenden Kapiteln erklärt.

Sobald die entsprechende Routine beendet wurde, wird die globale Interruptbehandlung

deaktiviert und der RS232-Interrupt wieder aktiviert. Nachdem die temporären Register und das Statusregister wieder vom Stack geladen wurden, wird mit dem Verlassen der Routine die generelle Interruptbehandlung wieder aktiviert. Der Mikrocontroller geht zurück in den Sleep-Modus und ist bereit, neue Befehle des Clients entgegenzunehmen, die wiederum über den RS232-Interrupt `URXC` die `INT_RS232`-Routine aufrufen.

4.3 Programmierung der RS232-Schnittstelle

Das Barebone-Motherboard schickt die Befehle (siehe Protokoll in Tabelle 4.2 auf Seite 43) über eine *serielle RS232-Schnittstelle*[Kai89, Sey88] an den Mikrocontroller. Dieser besitzt ein integriertes *USART-Modul* (= *Universal Synchronous and Asynchronous Receiver and Transmitter*) zur seriellen Kommunikation. Die zugehörigen *RXD*- und *TXD*-Leitungen zum Senden und Empfangen sind an den Pins `D0` bzw `D1` des Mikrocontrollers ausgeführt. Sie führen über einen *MAX3222*-Baustein[7] zur Anpassung des Pegels zu dem integrierten seriellen Port des Mikrocontroller-Boards.

Die Routinen zur Kommunikation über die serielle RS232-Schnittstelle befinden sich in `RS232.asm`[8]. Zum Initialisieren des USART-Moduls dient die Routine `RS232_INIT` (siehe Listing 4.3). Zunächst wird die Baudrate, indem der Wert `RS232_UBRRVAL` in `UBRRH:UBRRL` gespeichert wird, eingestellt. Dieser Wert berechnet sich nach folgender Formel[9]:

$$\texttt{UBRRH:UBRRL} = \frac{\texttt{CPU-Frequenz}}{\texttt{Baudrate} * 16 - 1}$$

Hier wird die Baudrate `RS232_BAUD` von `9600` eingestellt. Daraufhin wird das Frame-Format auf 8 Bit eingestellt und der *Receive-Complete-Interrupt* (`URXC`) aktiviert, der ausgelöst wird, wenn ein Byte erfolgreich empfangen wurde, was bedeutet, dass der Client einen Befehl gesendet hat. Zum Schluss werden Empfangs- und Sendeeinheit des USART-Moduls aktiviert.

[7] siehe *datasheets/MAX3222.pdf*
[8] siehe *prog/MiniBot/RS232.asm*
[9] siehe *datasheets/atmega32.pdf*

Listing 4.3: RS232.asm

```
11 ;This routine initializes the RS232-interface.
12 RS232_INIT:
13   ldi TEMP1, LOW(RS232_UBRRVAL)        ;set RS232-baudrate
14   out UBRRL, TEMP1
15   ldi TEMP1, HIGH(RS232_UBRRVAL)
16   out UBRRH, TEMP1
17   ldi TEMP1, (1<<URSEL)|(3<<UCSZ0)     ;set frame-format 8-bit
18   out UCSRC, TEMP1
19   sbi UCSRB, RXCIE                     ;enable RS232-Interrupt
20   sbi UCSRB, RXEN                      ;enable RX
21   sbi UCSRB, TXEN                      ;enable TX
22 ret
```

Listing 4.4: RS232.asm

```
24 ;This routine receives a byte on the RS232-interface and stores
25 ;it in RS232_VALUE.
26 RS232_RECEIVE_BYTE:
27   sbis UCSRA, RXC                      ;wait for byte
28   rjmp RS232_RECEIVE_BYTE
29   in RS232_VALUE, UDR                  ;store byte in RS232_VALUE
30 ret
```

Das Senden und Empfangen läuft über das Register RS232_VALUE[10]. Dort wird das empfangene Byte bzw. das zu sendende Byte gespeichert.

Das Empfangen übernimmt die Routine RS232_RECEIVE_BYTE (siehe Listing 4.4). Dort wird gewartet bis das RXC-Bit (*Receive-Complete*) gesetzt ist, was einen fertigen Empfangsvorgang signalisiert. Dann wird das empfangene Byte aus dem UDR-Register in RS232_VALUE gespeichert.

Die Routine RS232_SEND_BYTE (siehe Listing 4.5 sendet ein Byte und funktioniert ähnlich. Es wird gewartet bis das Modul bereit ist. Dann wird das Byte in RS232_VALUE durch Schreiben in das UDR-Register gesendet.

[10] siehe *Registers.asm*

Listing 4.5: RS232.asm

```
32 ;This routine sends the byte in RS232_VALUE on the RS232-interface.
33 RS232_SEND_BYTE:
34   sbis UCSRA, UDRE                ;wait until interface is ready
35   rjmp RS232_SEND_BYTE
36   out UDR, RS232_VALUE            ;send byte
37 ret
```

4.4 Ansteuerung der Motoren

Zur Ansteuerung der Motoren verfügt das Mikrocontroller-Board über einen *L298* 2-Kanal-Motortreiber[11] mit dem 2 Motoren bei einer Stromstärke von je maximal 2 Ampere betrieben werden können. Abbildung 4.1 zeigt den Beschaltungsplan des Motortreibers.

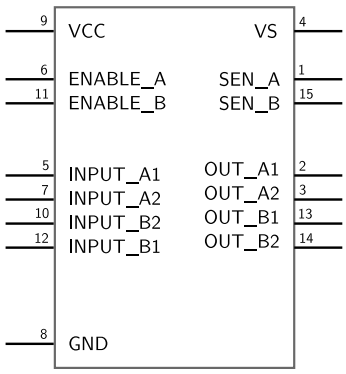


Bild 4.1: Aufbau des L298-Motortreibers

Die Logik des Bausteins wird über VCC versorgt. Die Anschlüsse SEN_A und SEN_B dienen zum Deaktivieren des Bausteins und sind dauerhaft mit Masse verbunden, die ebenfalls an GND angeschlossen ist. Die Motoren werden mit der an VS angelegten Spannung versorgt. Die Laufrichtung der Motoren A und B wird über die Steuereingänge (INPUT_*) geregelt. Tabelle 4.3 zeigt die möglichen Beschaltungen der Steuereingänge und ihren Einfluss auf die Motoren. Je nach Laufrichtung wird die volle Versorgungsspannung VS an die entsprechenden Ausgänge (OUT_*) des Motortreibers durchgeschaltet.

[11] siehe *datasheets/L298.pdf*

Tabelle 4.3: Beschaltung der Steuereingänge

INPUT_N1	INPUT_N2	Auswirkung auf Motor N
low	low	Stop
low	high	Rückwärtslauf
high	low	Vorwärtslauf
high	high	Stop

Das Durchschalten der Spannung wird über die ENABLE_*-Eingänge aktiviert. Durch das Anlegen einer *Pulsweitenmodulation*[Cat05] mit hinreichend großer Frequenz an die ENABLE_*-Eingänge des Bausteins, kann somit die an den Motoren anfallende Spannung gesteuert werden, was wiederum Auswirkungen auf die Umdrehungsgeschwindigkeit hat. In Kapitel 4.4.1 wird die Funktionsweise der Pulsweitenmodulation erklärt, woraufhin in Kapitel 4.4.2 deren Programmierung erläutert wird.

4.4.1 Funktionsweise Pulsweitenmodulation

Bei der Pulsweitenmodulation wird die Spannung bei einer konstanten Grundfrequenz zwischen high und low gewechselt, wobei das Verhältnis beider Schaltzeiten zueinander (auch als *Tastverhältnis* bezeichnet) während einer Periode variiert werden kann. Wählt man die Grundfrequenz ausreichend hoch, so fällt durch die Tiefpasswirkung der Motoren ein dem Tastverhältnis entsprechender Anteil der Versorgungsspannung an den Motoren ab. Durch Variation des Tastverhältnisses kann somit also die Geschwindigkeit der Motoren geregelt werden. Abbildung 4.2 zeigt Pulsweitenmodulation mit Tastverhältnissen von 20%, 50% und 90% mit der zugehörigen Betriebsspannung U_{eff}, die an den Motoren abfällt.

4.4.2 Programmierung der Motoransteuerung

Der ATmega32 verfügt über drei getrennte Timer-Einheiten, die für die Ausgabe von Pulsweitenmodulationen verwendet werden können. Die hier genutzte Timer-Einheit 1 verfügt

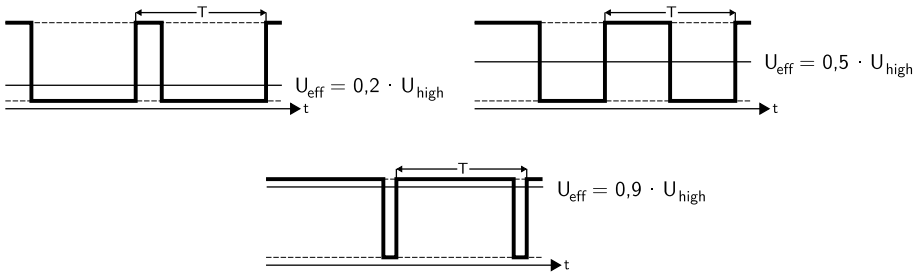


Bild 4.2: Beispiele Pulsweitenmodulation

über zwei getrennte Ausgabekanäle, die je für die Ansteuerung eines Motors genutzt werden. Hierzu sind die Pins `D4` und `D5` des Mikrocontrollers, die die Timer-Einheit 1 verwendet, direkt mit den `ENABLE_*`-Eingängen des Motortreibers verbunden.

Als Ansteuerungsmodus kommt *Fast-PWM* zur Anwendung. Hierbei wird der Ausgang zunächst auf `high` gesetzt. Dann zählt der Timer bis zu dem in `OCR1NH:OCR1NL` gespeicherten Wert hoch und setzt dann den Ausgang auf `low`. Anschließend zählt er bis zu der in `ICR1H:ICR1L` gespeicherten oberen Schranke, setzt den Ausgang wieder auf `high` und startet erneut bei Null. Abbildung 4.3 veranschaulicht diesen Ablauf am Beispiel des ersten Motors.

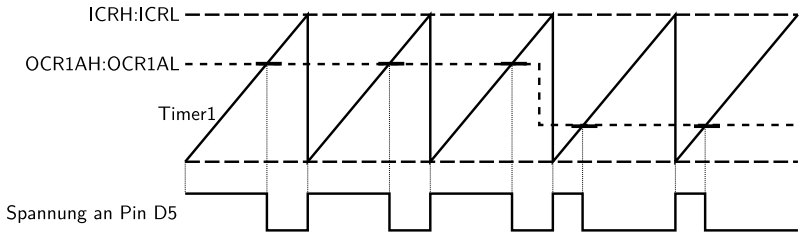


Bild 4.3: Ablauf der Pulsweitenmodulation

In `PWM.asm`[12] befinden sich die Routinen zur Low-Level-Steuerung der Motoren. Zu ihnen gehört die Routine `PWM_INIT` (siehe Listing 4.6), die die Initialisierung übernimmt. Zuerst wird der Inhalt der `OCR1*`-Register gelöscht. Anschließend wird die Pulsweitenmodulation über die Register `TCCR1A` und `TCCR1B` konfiguriert. Die ersten 4 Bit in `TCCR1A` legen die oben beschriebene Funktionsweise fest. Die zwei folgenden Bit spielen

[12] siehe *prog/MiniBot/PWM.asm*

Listing 4.6: PWM.asm

```
7  ;This routine initializes Timer unit 1.
8  PWM_INIT:
9    ldi TEMP1, 0x00         ;clear output control registers
10   out OCR1AH, TEMP1
11   out OCR1AL, TEMP1
12   out OCR1BH, TEMP1
13   out OCR1BL, TEMP1
14   ldi TEMP1, 0b10100010   ;configure PWM
15   out TCCR1A, TEMP1
16   ldi TEMP1, 0b00011011
17   out TCCR1B, TEMP1
18   ldi TEMPH, 0x00         ;configure timer resolution
19   ldi TEMPL, 0xFF
20   out ICR1H, TEMPH
21   out ICR1L, TEMPL
22   sbi DDRB, PB0           ;set data direction to output
23   sbi DDRB, PB1
24   sbi DDRD, PD6
25   sbi DDRD, PD7
26 ret
```

keine Rolle und die zwei letzten Bit legen zusammen mit Bit 4 und 3 in `TCCR1B` als Modus Fast-PWM mit oberer Grenze in `ICR1H:ICR1L` fest. Die ersten 3 Bit in `TCCR1B` sind irrelevant. Die 3 letzten Bit setzen die Taktfrequenz des Timers auf $\frac{1}{64}$ der CPU-Frequenz. Daraufhin wird die obere Grenze in `ICR1H:ICR1L` auf `0xFF = 255` gesetzt. Nun kann man daraus mit folgender Gleichung die Frequenz der Pulsweitenmodulation berechnen:

$$f_{PWM} = \frac{\texttt{CPU-Frequenz}}{64 \cdot (1 + \texttt{ICR1H:ICR1L})}$$

Es ergibt sich eine Frequenz der Pulsweitenmodulation f_{PWM} von 900 Hz. Am Ende der Routine werden die Pins, die mit den Steuerleitungen des Motortreibers (`INPUT_*`) verbunden sind, als Ausgang konfiguriert.

Die Geschwindigkeit der Motoren wird mit der Routine `PWM_SET_VELOCITY` gesetzt (siehe Listing 4.7). Die gewünschten Geschwindigkeiten befinden sich in den Registern `PWM_LEFT_VELOCITY` und `PWM_RIGHT_VELOCITY`, wobei die Wertebereiche je von `0x00` bis `0xFF` gehen. Bei Aufruf dieser Routine werden diese Werte in die `Low`-Bytes der `OCR1*`-Register geladen. Die `High`-Bytes der `OCR1*`-Register bleiben konstant auf

Listing 4.7: PWM.asm

```
28 ;This routine sets left velocity to PWM_LEFT_VELOCITY and
29 ;right velocity to PWM_RIGHT_VELOCITY.
30 PWM_SET_VELOCITY:
31   out OCR1AL, PWM_LEFT_VELOCITY
32   out OCR1BL, PWM_RIGHT_VELOCITY
33 ret
```

`0x00`, da die obere Schranke in `ICR1H:ICR1L` lediglich `0x00FF` beträgt und höhere Werte keinen Sinn machen würden. Dadurch kann somit das Tastverhältnis der Pulsweitenmodulation bzw. die Geschwindigkeit beider Motoren getrennt zwischen 0% ($\equiv$ `0x00`) und 100% ($\equiv$ `0xFF`) variiert werden.

Der Client kann einer Änderung der Geschwindigkeit durch Senden des Protokollcodes `REQUEST_PWM_SET_VELOCITY` auslösen, was wiederum zum Aufruf der Routine `FUNCTION_PWM_SET_VELOCITY` führt. Dort werden zunächst die Werte für die linke und rechte Geschwindigkeit empfangen und anschließend in `PWM_LEFT_VELOCITY` bzw. `PWM_RIGHT_VELOCITY` gespeichert. Schließlich werden die Geschwindigkeiten durch Aufruf von `PWM_SET_VELOCITY` geändert und zur Bestätigung der Protokollcode `RESPONSE_REQUEST_SUCCESSFUL` zum Client zurückgesendet.

Die Drehrichtung der Motoren wird mit den Routinen `PWM_FORWARD`, `PWM_BACKWARD`, `PWM_LEFT` und `PWM_RIGHT` (siehe Anhang D.6) eingestellt. Mit dem Aufruf dieser Routinen werden die Pins, die mit den Steuerleitungen des Motortreibers (`INPUT_*`) verbunden sind, entsprechend der gewünschten Bewegungsrichtung (Vorwärts- / Rückwärtsfahrt oder Links- / Rechtsdrehung) auf `high` bzw. `low` gesetzt.

Diese Routinen werden in `FUNCTION_PWM_START_ACTION` (siehe Anhang D.1) und `FUNCTION_PWM_START_ACTION_INTERVAL` aufgerufen. Wenn der Client über das Senden des Protokollcodes `REQUEST_PWM_START_ACTION` eine permanente (im Gegensatz zu `REQUEST_PWM_START_ACTION_INTERVAL`, vgl. Kapitel 4.5) Bewegung des Roboters starten will, wird die Routine `FUNCTION_PWM_START_ACTION` aufgerufen. Dort werden die Pins, an denen die Eingänge des Motortreibers angeschlossen sind, zunächst durch Maskierung des Datenrichtungsregisters als Eingangsports konfiguriert. Dies ist nötig, damit sich nachfolgende Änderungen an den Steuerpins nicht direkt

auf die Motoren auswirken. Ansonsten würde beispielsweise beim Start einer Geradeausfahrt durch `PWM_FORWARD` der Motor an `B0/B1` minimal früher starten, als der Motor an `D6/D7`, was zu einem minimalen Drehimpuls zu Beginn der Fahrt führen würde. Anschließend wird der *Action-Code*[13] vom Client empfangen, der die gewünschte Bewegungsart übermittelt. Dann wird je nach *Action-Code* zu der entsprechenden Low-Level-Routine (`PWM_*`) verzweigt oder `RESPONSE_ACTION_UNKNOWN` (bei unbekanntem *Action-Code*) an den Client zurückgeschickt. Nach der Ausführung der Low-Level-Funktionen wird, entsprechend der ausgeführten Aktion, der in `ACTION_STATE` gespeicherte Bewegungs-Zustand auf `ACTION_MOVING` bzw. `ACTION_TURNING` gesetzt. Nachdem der Protokollcode `RESPONSE_REQUEST_SUCCESSFUL` an den Client gesendet wurde, werden die Steuerpins wieder als Ausgangsports konfiguriert, wodurch sich die gemachten Änderungen synchron auf beide Motoren auswirken.

Der Client kann den in `ACTION_STATE` gespeicherten Bewegungs-Zustand durch Senden des Protokollcodes `REQUEST_GET_STATE` auslesen. Hierdurch wird die Routine `FUNCTION_GET_STATE` (siehe Anhang D.1) aufgerufen, in der lediglich der Protokollcode `RESPONSE_REQUEST_SUCCESSFUL` gefolgt von dem aktuellen Zustand gesendet wird.

Über die Routine `REQUEST_PWM_START_ACTION_INTERVAL` kann eine Bewegung eines bestimmten Intervalls, der vom Client übermittelt wird, ausgelöst werden. Die Funktionsweise dieser Routine wird im Zusammenhang mit der Behandlung, der von den Gabellichtschranken ausgelösten externen Interrupts, in Kapitel 4.5 erklärt.

Durch Senden des Protokollcodes `REQUEST_PWM_STOP_ACTION`, kann der Client die Motoren stoppen. Hierdurch wird `FUNCTION_PWM_STOP_ACTION` aufgerufen (siehe Anhang D.1). Hier werden die Motoren durch Konfiguration der Steuerpins als Eingangsports (wie oben beschrieben) gestoppt. Dann wird zur Bestätigung der Protokollcode `RESPONSE_REQUEST_SUCCESSFUL` zurück an den Client gesendet und der in `ACTION_STATE` gespeicherte Bewegungs-Zustand auf `ACTION_NONE` zurückgesetzt. Zum Schluss werden die eventuell aktivierten externen Interrupts deaktiviert (siehe Kapitel 4.5).

[13] siehe *prog/MiniBot/Actions.asm*

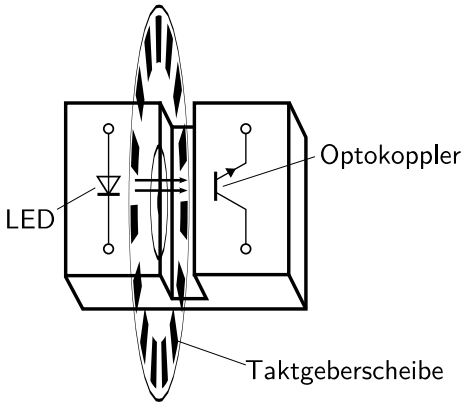


Bild 4.4: Aufbau Gabellichtschranke

4.5 Gabellichtschranken und externe Interrupts

Der MiniBot verfügt an jedem der beiden Getriebe über eine *Taktgeberscheibe*, auf deren transparenter Oberfläche lichtundurchlässige Markierungen angebracht sind. Die Taktgeberscheiben sind von *Gabellichtschranken* vom Typ *TCST 1103*[14] umschlossen. Die Gabellichtschranken bestehen aus einer LED, die Licht der Wellenlänge 950nm aussendet. Gegenüber ist ein *Optokoppler* angebracht, der durchschaltet, wenn sich keine Markierung der Taktgeberscheibe zwischen ihm und der LED befindet und das Licht ungehindert ankommt (siehe Skizze 4.4).

Sobald sich Motor und Getriebe drehen, rotiert auch die entsprechende Taktgeberscheibe, wodurch ein Taktsignal am Ausgang des Optokopplers erzeugt wird. Bei jeder Markierung der Taktgeberscheibe ist der Ausgang des Optokopplers auf `Low`-Pegel, zwischen zwei Markierungen ist er auf `High`-Pegel. Über die Anzahl der Takte kann auf die Umdrehungen der Taktgeberscheiben geschlossen werden.

Um die Taktimpulse zu registrieren, sind die Ausgänge der beiden Optokoppler über einen Schmitt-Trigger zur Signalstabilisierung an die Pins `D2` bzw. `D3` des Mikrocontrollers angeschlossen. Dort befinden sich die Eingänge für die externen Interrupts und 0 und 1. Die Konfiguration der externen Interrupts wird in der Routine `INT_INIT` (siehe Listing 4.8) vorgenommen.

[14] siehe *datasheets/TCST1103.pdf*

Listing 4.8: MiniBot.asm

```
458 ;This routine initializes the external interrupts
459 INT_INIT:
460   cbi DDRD, PD2                    ;set the pins of the external interrupts
461   cbi DDRD, PD3                    ;to input
462   in TEMP1, MCUCR
463   ori TEMP1, (3<<ISC10)|(3<<ISC00)  ;configure external interrupts for initiation
464   out MCUCR, TEMP1                 ;on every rising edge
465 ret
```

Nachdem die Pins, an denen die Optokoppler angeschlossen sind, als Eingangsports konfiguriert wurden, werden die beiden externen Interrupts durch Schreiben von `3` an die entsprechenden Stellen im `MCUCR`-Register konfiguriert. Diese Konfiguration bewirkt, dass bei jeder steigenden Taktflanke an einem der beiden Pins, der zugehörige externe Interrupt ausgelöst wird, was wiederum zum Aufruf der jeweiligen Interruptroutine führt. Die Skizze in Abbildung 4.5 zeigt den Zusammenhang zwischen Stellung der Taktgeberscheibe, Ausgangspegel des Optokopplers, Auslösen des externen Interrupts und Aufruf der entsprechenden Routine am Beispiel von Pin `D2` und dem zugehörigen externen Interrupt 0.

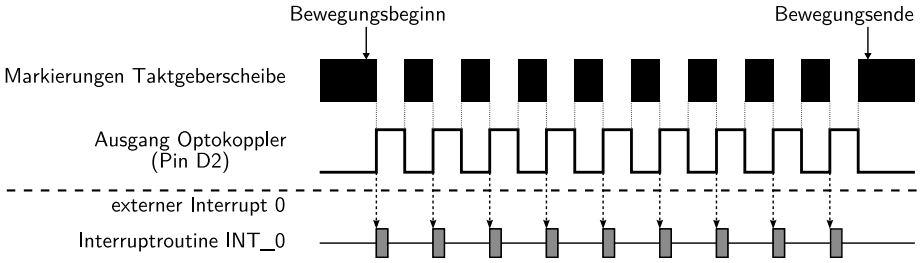


Bild 4.5: Auslösen des externen Interrupts 0

Bei jeder steigenden Taktflanke an Pin `D2` wird der externe Interrupt 0 ausgelöst. Dies führt zum Sprung zu Adresse `INT0addr` des Interruptvektors (siehe Listing 4.1) und somit zum Aufruf der Interruptroutine `INT_0` (siehe Listing 4.9). Analoges gilt für Pin `D3`. Hier wird der externe Interrupt 1 ausgelöst und die Interruptroutine `INT_1` aufgerufen.

Benötigt werden die externen Interrupts, wenn der Client durch Senden des Protokollcodes

REQUEST_PWM_START_ACTION_INTERVAL eine Bewegung eines bestimmten Intervalls auslösen will (im Gegensatz zu REQUEST_PWM_START_ACTION, siehe Kapitel 4.4). Dadurch wird die Routine FUNCTION_PWM_START_ACTION_INTERVAL aufgerufen, die ähnlich aufgebaut ist, wie FUNCTION_PWM_START_ACTION, allerdings sendet der Client nach dem *Action-Code*[15] zusätzlich die gewünschte Anzahl der Takte. Diese wird im Registerpaar INTERVAL_COUNTER_H:INTERVAL_COUNTER_L gespeichert. Entsprechend der ausgeführten Aktion, wird der in ACTION_STATE gespeicherte Bewegungs-Zustand auf den zugehörigen Wert ACTION_MOVING_INTERVAL bzw. ACTION_TURNING_INTERVAL gesetzt. Am Ende der Routine werden die externen Interrupts dann durch Setzen der entsprechenden Bits im GICR-Register aktiviert.

Bei jedem Taktimpuls, ausgelöst durch die Taktgeberscheiben an den Getrieben, wird nun der entsprechende externe Interrupt ausgelöst und die zugehörige Interruptroutine aufgerufen. Der Aufbau der Interruptroutinen INT_0 bzw. INT_1 ist nahezu identisch.

In der Routine INT_0 (siehe Listing 4.9) wird, nachdem die temporären Register und das Status-Register auf dem Stack gesichert wurden, der Leuchtzustand der LED an Pin B7 gewechselt. Dies erleichtert die Kontrolle der ordnungsgemäßen Funktion der Gabellichtschranken, durch die auf dem Robo32-Board angebrachten LEDs. Anschließend wird der Intervall-Wert in INTERVAL_COUNTER_H:INTERVAL_COUNTER_L dekrementiert. Ist dieser Null, so wurde die Bewegung für die gewünschte Anzahl von Takten durchgeführt. In diesem Fall werden die Motoren angehalten, die externen Interrupts wieder deaktiviert und der in ACTION_STATE gespeicherte Bewegungszustand zurück auf ACTION_NONE gesetzt. Dadurch wird es möglich, durch Abfragen des Zustandes mittels REQUEST_GET_STATE, festzustellen, ob die Bewegung abgeschlossen wurde. Am Ende der Interruptroutine wird der ursprüngliche Inhalt der Register wieder vom Stack geladen.

Auf gleiche Weise funktioniert die Routine INT_1 (siehe Anhang D.1), mit dem Unterschied, dass hier die LED an Pin B5 zur Signalisierung der Takte benutzt wird.

Der Client hat somit die Möglichkeit für jede Bewegung eine gewünschte Taktzahl zu übermitteln, um somit die Wegstrecke, die zurückgelegt werden soll, bzw. der Drehwinkel, der erreicht werden soll, zu beeinflussen. Die Bewegung kann natürlich auch durch Senden

[15] siehe *prog/MiniBot/Actions.asm*

Listing 4.9: MiniBot.asm

```
133 ;This interrupt routine is executed on every rising edge on pin D2
134 ;when external interrupt 0 is triggered.
135 INT_0:
136   push TEMP1                                    ;save temporary and status registers
137   in TEMP1, SREG
138   push TEMP1
139
140   sbis PORTB, PB7                               ;toggle state of LED on pin B7
141   rjmp INT_0_ON
142   cbi PORTB, PB7
143   rjmp INT_0_NEXT
144   INT_0_ON:
145   sbi PORTB, PB7
146   INT_0_NEXT:
147
148   sbiw INTERVAL_COUNTER_H:INTERVAL_COUNTER_L, 1 ;decrease interval
149   brne INT_0_END                                ;check if interval is zero
150   in TEMP1, DDRD                                ;stop engines
151   andi TEMP1, 0xCF
152   out DDRD, TEMP1
153   in TEMP1, GICR                                ;disable external interrupts
154   andi TEMP1, ~((1<<INT0)|(1<<INT1))
155   out GICR, TEMP1
156   ldi TEMP1, ACTION_NONE                        ;set action state
157  mov ACTION_STATE, TEMP1
158
159   INT_0_END:
160     pop TEMP1                                   ;restore temporary and status registers
161     out SREG, TEMP1
162     pop TEMP1
163 reti
```

von `REQUEST_PWM_STOP_ACTION` jederzeit gestoppt werden, wodurch die externen Interrupts deaktiviert und der Bewegungszustand auf `ACTION_NONE` gesetzt wird.

Die Umrechnung der Längeneinheiten und Gradzahlen in die korrespondierende Anzahl von Takten wird in der Steuersoftware des Clients vorgenommen, da hierfür Fließkomma-Arithmetik erforderlich ist, deren Implementierung auf dem Mikrocontroller zu aufwendig wäre.

4.6 Ansteuerung des SRF08-Ultraschallmoduls

Zur Erkennung von Hindernissen, die sich unmittelbar vor dem Roboter befinden, wurde an dessen Front ein Ultraschall-Entfernungsmesser vom Typ *SRF08*[16] montiert. Die Datenübertragung zwischen Mikrocontroller und Ultraschallmodul erfolgt über den *I²C-Bus*, dessen grundlegende Funktionsweise in Kapitel 4.6.1 erklärt wird. Anschließend wird in Kapitel 4.6.2 die Funktionsweise des SRF08-Ultraschallmoduls erläutert, woraufhin dessen Ansteuerung mittels Assemblerprogrammierung in Kapitel 4.6.3 behandelt wird.

4.6.1 Funktionsweise des I²C-Busses

Der I²C-Bus[17][DP97] wurde in den 80er Jahren von Philips Semiconductors entwickelt, um Geräte mit niedriger Datenübertragungsrate an Mikrocontroller oder Hauptplatinen anschließen zu können. Der Adressraum von 7 Bit ermöglicht maximal 112 Knoten (16 Adressen sind für Sonderzwecke reserviert).

Der I²C-Bus benötigt nur 2 Leitungen: *serial data* (*SDA*) und *serial clock* (*SCL*). Von Atmel wird er daher als *Two-Wire-Interface* (*TWI*) bezeichnet. Jedes der angeschlossenen Geräte kann als Sender oder Empfänger fungieren. Im Ausgangszustand liegen beide Leitungen durch einen Pull-Up-Widerstand auf `High`-Pegel. Der Datenaustausch funktioniert nach dem Master-Slave-Prinzip: Der Master initiiert den Datentransfer und erzeugt ein Taktsignal auf der SCL-Leitung. Alle anderen Geräte werden damit automatisch zum Slave.

Die Kommunikation beginnt mit dem Senden einer *start-condition* durch den Master, indem dieser einen Übergang von `high` nach `low` auf der SDA-Leitung bewirkt, während der Takt auf `High`-Pegel liegt. Abgeschlossen wird die Übertragung mit dem Senden einer *stop-condition* durch den Master, indem ein Übergang von `low` nach `high` auf der SDA-Leitung vollzogen wird, während der Takt auf `High`-Pegel liegt. Das erneute Senden einer start-condition, bevor eine stop-condition gesendet wurde, wird als *repeated-start* bezeichnet. Die eigentlichen Daten werden byteweise übertragen, wobei die höherwertigen Bits

[16] siehe *datasheets/srf08.pdf*
[17] siehe *specifications/i2c.pdf*

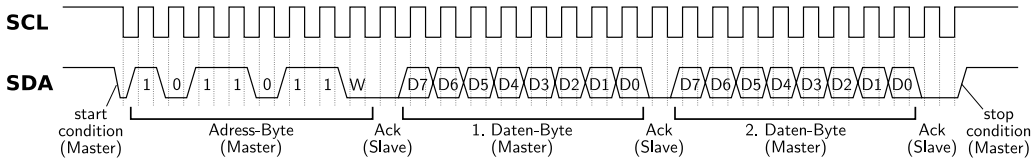


Bild 4.6: Senden von 2 Bytes auf dem I^{2}C-Bus

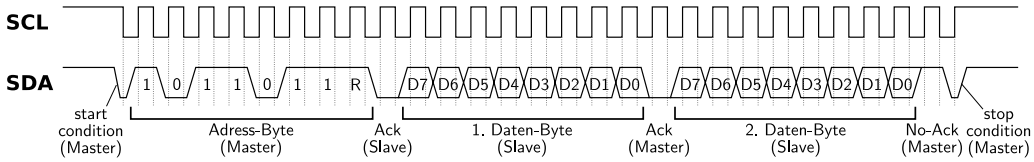


Bild 4.7: Lesen von 2 Bytes auf dem I^{2}C-Bus

zuerst gesendet werden. Nach jedem übertragenen Byte folgt ein *acknowledge*, indem der Empfänger den Pegel auf der SDA-Leitung während des 9. Taktimpulses auf `low` zieht, um ein weiteres Byte anzufordern.

Um das Gerät zu identifizieren, das als Slave mit dem Master kommuniziert, wird die Adresse des Slaves in dem ersten Datenbyte gesendet. Die 7 Bit langen Adressen sind jedem Gerät eindeutig zugeordnet und können bei Bedarf geändert werden. Nach der 7 Bit langen Adresse wird an 8. Position ein *data direction bit* gesendet. Eine Null bedeutet, dass Daten an das Gerät gesendet werden sollen; eine Eins bedeutet, dass Daten von dem Gerät gelesen werden sollen. Meistens wird das data direction bit zur Adresse gezählt, so auf jedes Gerät zwei Adressen entfallen - eine Lese- und eine Schreibadresse.

Das Schema in Abbildung 4.6 zeigt, wie der Master 2 Bytes an Adresse `0x5B` sendet. Nach der start-condition überträgt er die 7 Bit der Adresse, gefolgt von einer Null, um zu signalisieren, dass er auf das Gerät schreiben will. Der Slave bestätigt den Empfang mit einem acknowledge. Nun sendet der Master die beiden Daten-Bytes, die vom Slave je mit einem acknowledge bestätigt werden. Zum Schluss beendet der Master die Datenübertragung mit einer stop-condition.

Das Schema in Abbildung 4.7 zeigt, wie der Master 2 Bytes von Adresse `0x5B` liest. Der Kommunikationsbeginn unterscheidet sich in dem 8. Bit des Adress-Bytes. Dort wird nun eine eins gesendet, um dem Slave zu signalisieren, dass der Master lesen will. Nachdem

der Slave dies mit einem acknowledge bestätigt hat, sendet er die Daten-Bytes. Das erste Byte bestätigt der Master mit einem acknowledge, um zu signalisieren, dass er ein weiteres Byte empfangen will. Nach dem zweiten Byte sendet er kein weiteres acknowledge, worauf die Kommunikation schließlich durch das Senden einer stop-condition vom Master beendet wird.

Möchte der Master nach der Datenübertragung (Lesen oder Schreiben) mit einem weiteren Gerät kommunizieren oder zwischen Schreib- und Lesemodus wechseln, so kann er statt einer stop-condition und erneuter start-condition, den bereits erwähnten repeated-start durchführen. Anschließend sendet er die neue Adresse und fährt mit der Kommunikation wie oben beschrieben fort.

4.6.2 Funktionsweise des SRF08-Ultraschallmoduls

Das SRF08-Ultraschallmodul erkennt Hindernisse in einer Entfernung von 3cm bis 6m bei einer Auflösung von 1cm. Hierzu werden Schallwellen im Ultraschallspektrum mit einer Frequenz von 40kHz ausgesandt. Durch die Zeitdifferenz bis zum Eintreffen eines Echos kann die Entfernung zu einem Hindernis errechnet werden.

Tabelle 4.4: Register des SRF08-Ultraschallmoduls

Register	Lesen	Schreiben
0x00	Software Version	Befehlsregister
0x01	Licht Sensor	Verstärkungsregister
0x02	1. Echo `High`-Byte	Reichweitenregister
0x03	1. Echo `Low`-Byte	-
...	...	...
0x22	17. Echo `High`-Byte	-
0x23	17. Echo `Low`-Byte	-

Die Kommunikation läuft über den I^2C-Bus. Das Modul hat im Auslieferungszustand die Adresse `0xE0` zum Schreiben und `0xE1` zum Lesen (Das 8. Byte wurde hier mit zur Adresse gezählt). Der Datenaustausch mit dem Modul geschieht über 36 Register, die im

Schreib- und im Lesemodus allerdings unterschiedliche Bedeutungen haben. Tabelle 4.4 zeigt alle Register mit den zugehörigen Bedeutungen.

Aus Register `0x00` kann die Version der Software ausgelesen werden. Register `0x01` liefert die Helligkeit des integrierten Lichtsensors. Die Register `0x02` bis `0x23` enthalten jeweils abwechselnd `High`- und `Low`-Byte der bis zu 17 empfangenen Echo-Signale.

Die Tabelle 4.5 zeigt die zulässigen Befehlscodes, die in das Befehlsregister mit Adresse `0x00` geschrieben werden können, mit zugehöriger Bedeutung.

Tabelle 4.5: Befehlscodes und ihre Bedeutungen

Befehlscode	Aktion
0x50	Messung auslösen - Ergebnis in Zoll
0x51	Messung auslösen - Ergebnis in Zentimeter
0x52	Messung auslösen - Ergebnis in Mikrosekunden
0x53	ANN-Mode - Ergebnis in Zoll
0x54	ANN-Mode - Ergebnis in Zentimeter
0x55	ANN-Mode - Ergebnis in Mikrosekunden
0xA0	Erstes Byte in Sequenz zur Änderung der Moduladresse
0xA1	Zweites Byte in Sequenz zur Änderung der Moduladresse
0xA2	Drittes Byte in Sequenz zur Änderung der Moduladresse

Die Befehlscodes `0x50` bis `0x52` dienen zum Auslösen einer Messung in der Einheit Zoll, Zentimeter, bzw. Mikrosekunden. Im *ANN-Mode* werden die Multi-Echo-Daten zur Weiterverarbeitung in einem künstlichen neuronalen Netz vorbereitet (siehe Datenblatt). Mit den Befehlscodes `0xA0` bis `0xA2` kann die Adresse des Moduls geändert werden.

Über das Verstärkungsregister kann die maximale analoge Verstärkung eingestellt werden. Die normale Zeitdauer eines Messvorgangs dauert ca. 65ms. Sie kann jedoch durch Herabsetzen des Wertes im Reichweitenregister verringert werden. Dadurch wird jedoch die Reichweite zur Erkennung eines Hindernisses ebenfalls begrenzt und eine Anpassung des Verstärkungsregisters wird nötig. Der nicht lineare Zusammenhang zwischen

Reichweiten- und Verstärkungsregister ist im Datenblatt des SRF08[18] beschrieben.

Bei der Kommunikation mit dem Ultraschallmodul über den I²C-Bus sind zwei Fälle zu unterscheiden:

1. **Schreiben eines Wertes in ein Register**:

   Hier wird im ersten Daten-Byte das Register adressiert, in das geschrieben werden soll. Das zweite Datenbyte enthält den zu schreibenden Wert. Abbildung 4.8 zeigt das Schreiben des Befehlscodes zum Start einer Messung in Zentimetern (`0x51`) in das Befehlsregister (`0x00`) des Ultraschallmoduls mit der Standardadresse `0xE0`.

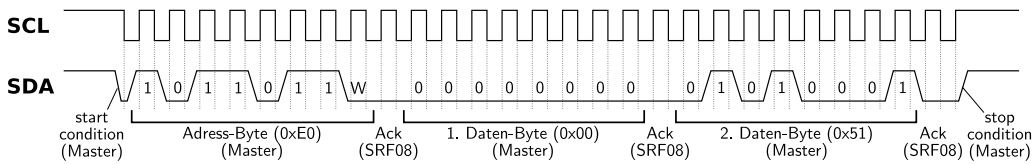


Bild 4.8: Schreiben in ein Register des Ultraschall-Moduls

2. **Lesen eines Wertes aus einem Register**:

   Zum Lesen eines Wertes wird zunächst analog zu Fall 1 das Register adressiert. Dann wird in den Lesemodus gewechselt werden, indem ein repeated-start (siehe Kapitel 4.6.1) gefolgt von der Leseadresse durchgeführt wird. Nun überträgt das Modul die Daten. Abbildung 4.9 zeigt das Auslesen des ersten Echo-Signals (`0x02`) des Moduls an der Standardadresse `0xE0`.

4.6.3 Programmierung der SRF08-Ansteuerung

Der Atmel ATmega32 verfügt über ein eigenes Modul, dass die Kommunikation mit dem *Two-Wire-Interface* (*TWI*), wie der I²C-Bus bei Atmel genannt wird, steuert. Dadurch erspart man sich zwar eine Software-Implementierung des I²C-Protokolls, doch die Programmierung geschieht immer noch auf sehr niedriger Ebene, wodurch man sich nach

[18] siehe *datasheets/srf08.pdf*

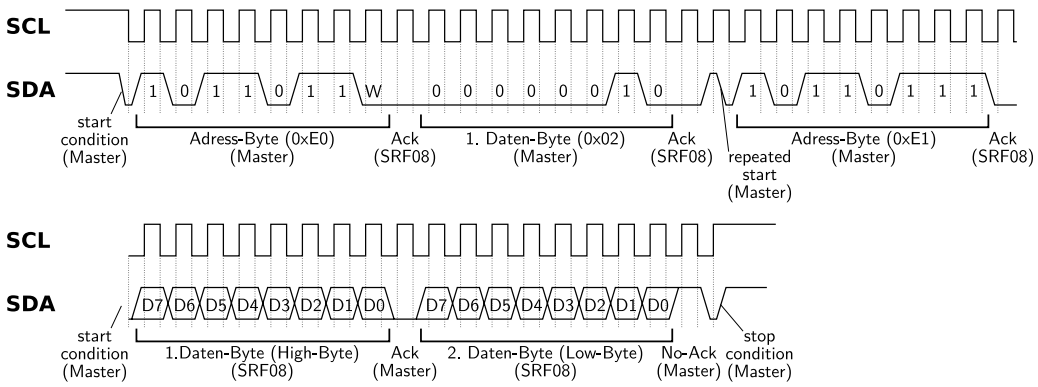


Bild 4.9: Lesen aus einem Register des Ultraschall-Moduls

wie vor um die genauen Abfolgen beim Senden der start-/stop-conditions und Acknowledges kümmern muss. Die SCL- und SDA-Ausgänge des TWI-Moduls befinden sich an den Pins `C0` und `C1` des Mikrocontrollers und sind auf dem Robo32-Board an der I²C-Pfostenbüchse ausgeführt[19]. Dort kann das SRF08-Ultraschallmodul direkt angeschlossen werden.

Die grundlegenden Routinen zur Abwicklung der I²C-Kommunikation befinden sich in `I2C.asm`[20]. Dort sind auch die Codes definiert, die den Ablauf der Übertragung protokollieren (siehe Listing 4.10). Näheres dazu allerdings später.

Listing 4.10: I2C.asm

```
11 ;I2C-Codes:
12 .equ I2C_START_SUCCESS = 0x08
13 .equ I2C_REPEATED_START_SUCCESS = 0x10
14 .equ I2C_WRITE_SLAW_SUCCESS = 0x18
15 .equ I2C_WRITE_SLAR_SUCCESS = 0x40
16 .equ I2C_WRITE_SUCCESS = 0x28
17 .equ I2C_READ_SUCCESS = 0x50
18 .equ I2C_READLAST_SUCCESS = 0x58
19 .equ I2C_NO_ERROR = 0x00
```

Zur Initialisierung dient die Routine `I2C_INIT` (siehe Listing 4.11). Sie setzt die Bitrate

[19] siehe *datasheets/robo32.pdf*
[20] siehe *prog/MiniBot/I2C.asm*

Listing 4.11: I2C.asm

```
21 ;This routine initializes the I2C-unit.
22 I2C_INIT:
23   ldi TEMP1, I2C_BITRATE          ;set bitrate
24   out TWBR, TEMP1
25   ldi TEMP1, I2C_PRESCALER        ;set prescaler
26   out TWSR, TEMP1
27 ret
```

im TWBR-Register auf I2C_BITRATE (= 17) und die Prescaler-Bits im TWSR-Register auf I2C_PRESCALER (= 1). Mit der folgenden Formel aus dem Datenblatt des ATmega32[21] führt dies zu einer SCL-Frequenz von 97,01 kHz[22], was unterhalb der Maximal-Frequenz von 100 kHz liegt, die der Standard vorschreibt[23]:

$$\texttt{SCL-Frequenz} = \frac{\texttt{CPU-Frequenz}}{16 + 2 \cdot \texttt{TWBR} \cdot 4^{\texttt{TWPS}}}$$

Die eigentlichen Routinen lassen sich in zwei Gruppen unterteilen. Die erste Gruppe von Routinen bietet eine komfortable Möglichkeit, Daten von einem I²C-Gerät zu lesen oder zu schreiben. Zu Ihnen zählen die Routinen I2C_STORE_BYTE, I2C_LOAD_BYTE und I2C_LOAD_WORD. Diese rufen ihrerseits die Low-Level-Routinen auf, die die einzelnen Schritte in der Sequenz einer I²C-Kommunikation ausführen.

Die Routine I2C_STORE_BYTE (Listing 4.12), speichert das Byte in I2C_VALUE in das Register I2C_REGISTER des Slaves mit der Adresse I2C_ADDRESS. Dazu wird zuerst die Routine I2C_START zum Senden einer start-condition aufgerufen. Anschließend wird durch Aufruf von I2C_WRITE_SLAW das Adressbyte für den Schreibmodus gesendet. Dann wird die Adresse des Registers in das TWDR-Register geladen und durch I2C_WRITE gesendet. Schließlich wird das eigentliche zu speichernde Byte gleichermaßen in das TWDR-Register geladen und ebenfalls gesendet. Beendet wird die Kommunikation durch das Senden der stop-condition mittels I2C_STOP. Danach wird der Code I2C_NO_ERROR in das Register I2C_ERRORCODE geladen, um zu signalisieren, dass die Übertragung erfolgreich war. Durch diese Aufrufsequenz der Low-Level-Funktionen,

[21] siehe *datasheets/atmega32.pdf*

[22] bei einer CPU-Frequenz von 14,7456 MHz

[23] siehe *specifications/i2c.pdf*

Listing 4.12: I2C.asm

```
29 ;This routine stores the value of I2C_VALUE in register
30 ;I2C_REGISTER of device at address I2C_ADDRESS.
31 I2C_STORE_BYTE:
32    rcall I2C_START             ;send start condition
33    rcall I2C_WRITE_SLAW        ;configure device for write access
34    out TWDR, I2C_REGISTER      ;load register-address
35    rcall I2C_WRITE             ;send register-address
36    out TWDR, I2C_VALUE         ;load value
37    rcall I2C_WRITE             ;send value
38    rcall I2C_STOP              ;send stop condition
39    ldi TEMP1, I2C_NO_ERROR     ;load I2C_NO_ERROR in I2C_ERRORCODE
40    mov I2C_ERRORCODE, TEMP1    ;to indicate that no error occured
41 ret
```

wird der in Abbildung 4.8 skizzierte Ablauf exakt nachvollzogen. Analog dazu, liest die Routine `I2C_LOAD_BYTE` (Listing 4.13), das Byte aus Register `I2C_REGISTER` des Slaves mit der Adresse `I2C_ADDRESS` und speichert es in `I2C_VALUE`. Nach dem Senden der Registeradresse wird jedoch durch Aufruf der Routine `I2C_REPEATED_START` ein repeated-start durchgeführt. Anschließend wird mit `I2C_WRITE_SLAR` das Adress-Byte für den Lesemodus gesendet. Dann wird das eigentliche Byte mit `I2C_READLAST` vom Slave empfangen, in `I2C_VALUE` gespeichert und die Kommunikation durch Senden einer stop-condition via `I2C_STOP` beendet. Um den Funktionsumfang zu komplettieren, bietet die Routine `I2C_LOAD_WORD` (siehe Anhang D.7) die Möglichkeit gleich zwei Bytes zu lesen. Hierbei wird nach dem Empfang des ersten Bytes ein weiteres Byte über ein Acknowledge des Masters mittels `I2C_READ` angefordert, was bei `I2C_READLAST` nicht der Fall ist. Die empfangenen Bytes werden in `I2C_VALUE_H:I2C_VALUE_L` gespeichert. Diese Aufrufsequenz entspricht dem in Abbildung 4.9 skizzierten Ablauf. Die Low-Level-Routinen, die von diesen drei Routinen aufgerufen werden, sind prinzipiell alle nach dem gleichen Schema aufgebaut:

1. Ausführung der gewünschten Aktion
2. Überprüfung auf Erfolg, durch Inspektion des `TWSR`-Registers
3. Im Fehlerfall: Laden des Fehlercodes in `I2C_ERRORCODE` und Abbruch der übergeordneten High-Level-Routine

Listing 4.13: I2C.asm

```
43 ;This routine loads the byte from register I2C_REGISTER
44 ;of the device at address I2C_ADDRESS into I2C_VALUE.
45 I2C_LOAD_BYTE:
46    rcall I2C_START             ;send start condition
47    rcall I2C_WRITE_SLAW        ;configure device for write access
48    out TWDR, I2C_REGISTER      ;load register-address
49    rcall I2C_WRITE             ;send register-address
50    rcall I2C_REPEATED_START    ;send repeated start condition
51    rcall I2C_WRITE_SLAR        ;configure device for read access
52    rcall I2C_READLAST          ;load value without acknowledge
53    in I2C_VALUE, TWDR          ;save value in I2C_VALUE
54    rcall I2C_STOP              ;send stop condition
55    ldi TEMP1, I2C_NO_ERROR     ;load I2C_NO_ERROR in I2C_ERRORCODE
56    mov I2C_ERRORCODE, TEMP1    ;to indicate that no error occured
57 ret
```

Dies lässt sich gut an `I2C_START` zum Senden einer start-condition demonstrieren (Listing 4.14). Zuerst werden die erforderlichen Bits im `TWCR`-Register gesetzt. Das Schreiben einer logischen Eins in das `TWINT`-Bit löscht dieses, wodurch die I^2C-Operation gestartet wird. Gleichzeitig muss das `TWSTA`-Bit zum Senden einer start-condition und das `TWEN`-Bit zur Aktivierung des I^2C-Moduls gesetzt werden. Anschließend wird das `TWCR`-Register in einer Schleife solange ausgelesen, bis das `TWINT`-Bit wieder auf Eins gesetzt ist, wodurch das Ende der Operation angezeigt wird. Daraufhin wird das `TWSR`-Register ausgelesen. Nachdem die Prescaler-Bits maskiert wurden, wird der verbleibende Wert des Status-Registers mit dem, für ein erfolgreiches Senden erforderlichen Wert (`I2C_START_SUCCESS`), verglichen. Sind beide Werte identisch, so wird die Routine verlassen.

Andernfalls wird zu `I2C_ERROR` (Listing 4.15) gesprungen. Dort wird der auf den Fehler hindeutende Wert des Statusregisters in `I2C_ERRORCODE` geladen und eine stop-condition gesendet, um den Bus wieder freizugeben. Beim Verlassen der Routine, würde jetzt die nächste Low-Level-Routine (beispielsweise `I2C_WRITE_SLAW`), trotz des Fehlers beim Senden der start-condition, aufgerufen. Dies ist natürlich unerwünscht, denn die darüberliegende Routine soll ebenfalls verlassen werden, was durch Veränderung des Stack-Pointers realisiert wird, da beide Return-Adressen auf dem Stack liegen. Der Stack-Pointer wird also um `2` erhöht und zeigt jetzt auf den Return-Wert der aufrufenden Rou-

Listing 4.14: I2C.asm

```
78 ;This routine sends a start condition on I2C.
79 I2C_START:
80   ldi TEMP1, (1<<TWINT)|(1<<TWSTA)|(1<<TWEN) ;set bits in control register
81   out TWCR, TEMP1
82   WAIT_I2C_START_SENT:                       ;wait for completion
83     in TEMP1, TWCR
84     sbrs TEMP1, TWINT                        ;interrupt-flag indicates completion
85     rjmp WAIT_I2C_START_SENT                 ;not yet completed
86   in TEMP1, TWSR                             ;load status register and clear the
87   andi TEMP1, 0xF8                           ;prescaler-configuration-bits
88   ldi TEMP2, I2C_START_SUCCESS               ;compare value of status register
89   cpse TEMP1, TEMP2                          ;with estimated value I2C_START_SUCCESS
90   rjmp I2C_ERROR                             ;an error occured
91 ret
```

Listing 4.15: I2C.asm

```
198 ;This branch is executed if an error has occured, the errorcode is saved
199 ;and the I2C-communication is stopped.
200 I2C_ERROR:
201   mov I2C_ERRORCODE, TEMP1          ;save errorcode
202   rcall I2C_STOP                    ;send stopcondition
203   in zh, sph                        ;increase the stack-pointer by 2 - now, the
204   in zl, spl                        ;address on the stack is the return address
205   adiw zh:zl, 2                     ;of the calling routine
206   out sph, zh                       ;so on return the calling routine is aborted too
207   out spl, zl
208 ret
```

tine. Schließlich wird die Routine verlassen und die Verarbeitung fährt nach dem Aufruf der übergeordneten Routine fort, womit die verbleibenden Low-Level-Routinen, aufgrund des aufgetretenen Fehlers, übersprungen wurden. Nach diesem Schema arbeiten alle Low-Level-Routinen, was dazu führt, dass aufgetretene Fehler durch Auswertung des Wertes in `I2C_ERRORCODE` identifiziert werden können.

Die Routinen `I2C_REPEATED_START` und `I2C_STOP` (siehe Anhang D.7) dienen zum Senden eines repeated-start bzw. einer stop-condition. `I2C_REPEATED_START` unterscheidet sich von `I2C_START` nur durch einen anderen Wert, mit denen das `TWSR`-Register nach der Operation verglichen wird (`I2C_REPEATED_START_SUCCESS`). Bei

I2C_STOP reicht es, das TWSTO-Bit anstatt des TWSTA-Bits zu setzen. Ein Warten und anschließende Überprüfung des TWSR-Registers ist hier nicht nötig, da beim Senden einer stop-condition kein Fehler auftreten kann.

Die Routinen I2C_WRITE_SLAW und I2C_WRITE_SLAR (siehe Anhang D.7) senden das Adress-Byte für die in I2C_ADDRESS gespeicherte Adresse im Schreib- bzw. Lesemodus. Bei I2C_WRITE_SLAW wird die Adresse aus I2C_ADDRESS in das TWDR-Register geladen. Daraufhin wird das Senden durch Setzen der TWINT- und TWEN-Bits im TWCR-Register gestartet. Nach anschließendem Warten auf Beendigung der Operation, erfolgt der Test auf Fehler analog zu I2C_START. Bei I2C_WRITE_SLAR wird die Adresse aus I2C_ADDRESS vor dem Senden inkrementiert, um das R/W-Bit für den Lesemodus zu setzen.

Tabelle 4.6: Routinen zur I²C-Kommunikation

Routine	Zweck
I2C_STORE_BYTE	schreibt Inhalt von I2C_VALUE in Register I2C_REGISTER an Adresse I2C_ADDRESS
I2C_LOAD_BYTE	liest Byte aus Register I2C_REGISTER an Adresse I2C_ADDRESS in I2C_VALUE
I2C_LOAD_WORD	liest Word ab Register I2C_REGISTER an Adresse I2C_ADDRESS in I2C_VALUE_H:I2C_VALUE_L
I2C_START	sendet start-condition
I2C_REPEATED_START	sendet repeated-start
I2C_STOP	sendet stop-condition
I2C_WRITE_SLAW	sendet Adress-Byte für Adresse in I2C_ADDRESS im Schreibmodus
I2C_WRITE_SLAR	sendet Adress-Byte für Adresse in I2C_ADDRESS im Lesemodus
I2C_WRITE	sendet Byte im TWDR-Register
I2C_READ	empfängt Byte in TWDR-Register mit anschließendem Acknowledge
I2C_READLAST	empfängt Byte in TWDR-Register ohne anschließendes Acknowledge

Zum Senden und Empfangen der Daten sind die Routinen I2C_WRITE, I2C_READ und I2C_READLAST (siehe Anhang D.7) verantwortlich. I2C_WRITE funktioniert ähnlich wie I2C_WRITE_SLAW, lediglich das Laden der Adresse in das TWDR-Register entfällt, da dessen Inhalt direkt gesendet wird. I2C_READ und I2C_READLAST sind ebenfalls analog aufgebaut, mit dem Unterschied, dass bei I2C_READ zusätzlich das TWEA-Bit

gesetzt wird, um durch ein Acknowledge ein weiteres Daten-Byte anzufordern. Ob empfangen oder gesendet werden soll, weiß das I²C-Modul anhand des im Adress-Byte gesendeten R/W-Bits.Tabelle 4.6 zeigt alle Routinen zur I²C-Kommunikation und deren Zweck im Überblick.

Die eigentliche Ansteuerung des SRF08-Ultraschallmoduls bei entsprechenden Requests des Clients geschieht in den folgenden zwei Routinen:

Die Routine `FUNCTION_I2C_START_MEASUREMENT` (siehe Anhang D.1) wird aufgerufen, wenn der Client durch Senden von `REQUEST_I2C_START_MEASUREMENT` eine Messung auslöst. In der Routine wird zuerst die Standardadresse `0xE0` des Moduls in `I2C_ADDRESS` geladen, in `I2C_REGISTER` wird die Adresse `0x00` des Kommandoregisters geladen und schließlich wird der Befehlscode `0x52` zum Auslösen einer Messung in cm in `I2C_VALUE` gespeichert. Dann werden die Daten mit `I2C_STORE_BYTE` übertragen. Anschließend wird überprüft, ob der Inhalt von `I2C_ERRORCODE` dem Wert von `I2C_NO_ERROR` entspricht und das Auslösen der Messung erfolgreich war. Ist dies der Fall, so wird der Wert von `RESPONSE_REQUEST_SUCCESSFUL` an den Client zurückgeschickt, andernfalls der in `I2C_ERRORCODE` befindliche Fehlercode.

Sendet der Client `REQUEST_I2C_GET_DISTANCE`, so wird der vom Sensor gemessene Wert mit `FUNCTION_I2C_GET_DISTANCE` (siehe Anhang D.1) ausgelesen und an den Client gesendet. Dazu wird in `I2C_ADDRESS` wieder die Adresse `0xE0` geladen und die Adresse `0x02` des ersten Echoregisters wird in `I2C_REGISTER` gespeichert. Die gemessene Entfernung wird mit `I2C_LOAD_WORD` ausgelesen. Zeigt `I2C_ERRORCODE` eine erfolgreiche I²C-Kommunikation an, so wird `RESPONSE_REQUEST_SUCCESSFUL` zur Bestätigung, gefolgt von `High`- und `Low`-Byte der gemessenen Entfernung, an den Client geschickt. Tritt ein Fehler bei der Kommunikation auf, so wird genauso verfahren, wie bei `FUNCTION_I2C_GET_DISTANCE`.

4.7 Ansteuerung der Infrarot-Sensoren

Zur Erkennung von Abgründen und Hindernissen wurde sowohl vorne, als auch hinten je ein *SHARP GP2D120*[24] Infrarot-Sensor montiert. Die Sensoren sind schräg nach unten gerichtet und ermöglichen die Messung von Entfernungen zwischen 4cm und 30cm. Die Ausgabe der ermittelten Entfernung erfolgt über einen analogen Pegel zwischen 0,3V und 3V. Der Zusammenhang zwischen Entfernung des Objekts, an dem der Infrarotstrahl reflektiert wurde, und resultierender Ausgangsspannung, lässt sich dem Diagramm 4 des Datenblatts entnehmen. Die Ausgänge der Sensoren sind mit den Pins `A1` und `A3` des Mikrocontrollers verbunden, an denen eine *Analog-Digital-Wandlung*[Cat05] vorgenommen wird, um von dem analogen Signal zu einem digitalen 10-Bit-Wert zu gelangen.

4.7.1 Funktionsweise der A/D-Wandlung

Die Umwandlung der analogen Mess-Spannung V_{in} in einen digitalen 10-Bit-Wert erfolgt durch *sukzessive Approximation*. Hierbei wird ein digitaler 10-Bit-Wert iterativ inkrementiert und mit einem D/A-Wandler, der von einer Referenz-Spannung V_{ref} versorgt wird, in eine analoge Spannung umgewandelt. Diese wird dann mit dann mit der Mess-Spannung V_{in} über einen Komparator verglichen. Dies wird solange wiederholt bis der erzeugte Pegel des D/A-Wandlers höher als die Mess-Spannung V_{in} oder der 10-Bit-Wert gleich dem Maximalwert (=`0x3FF`) ist. Der 10-Bit-Wert wird dann zurückgegeben. Die folgende Gleichung beschreibt den Zusammenhang zwischen dem ermittelten Digitalwert ADC, der Referenz-Spannung V_{ref} und der Mess-Spannung V_{in}:

$$ADC = \frac{V_{in} \cdot 1023}{V_{ref}}$$

4.7.2 Programmierung der Infrarot-Sensor-Ansteuerung

Die beiden Routinen für die A/D-Wandlung befinden sich `ADC.asm`[25]. Zur Initialisierung des A/D-Moduls dient die Routine `ADC_INIT` (siehe Listing 4.16). Als Referenz-

[24] siehe *datasheets/gp2d120.pdf*
[25] siehe *prog/MiniBot/ADC.asm*

Listing 4.16: ADC.asm

```
7  ;This routine initializes the A/D-unit.
8  ADC_INIT:
9    ldi TEMP1, 3<<REFS0                    ;select internal reference-voltage
10   out ADMUX, TEMP1
11   cbi DDRA, 1                            ;configure ports as input
12   cbi DDRA, 3
13 ret
```

Spannung für den D/A-Wandler wird hier die interne 2,56V Referenz-Spannung gewählt. Eine Mess-Spannung größer als 2,56V führt also zu dem Maximalwert `1023=0x3FF`. Anschließend werden die Pins, an denen die Infrarot-Sensoren angeschlossen sind, als Eingangsports konfiguriert.

Über die Routine `ADC_START` (siehe Listing 4.17) wird der eigentlichen Messung durchgeführt. Vor der Messung muss in Register `ADC_SOURCE` die Quelle bzw. der Pin für die A/D-Wandlung gespeichert werden. Vor Beginn der Messung werden die Bits des `ADMUX`-Registers, die die Quelle für die A/D-Wandlung bestimmen, gelöscht. Dann wird die neue Quelle aus `ADC_SOURCE` im `ADMUX`-Register gespeichert. Vor der Messung wird das Interrupt-Flag-Bit `ADIF` (*AD interrupt flag*), das die Beendigung einer A/D-Wandlung signalisiert, gelöscht. Nun wird die A/D-Wandlung durch Setzen der entsprechenden Bits im `ADCSRA`-Register gestartet. Durch Setzen von `ADEN` (*AD enable*) wird das A/D-Modul aktiviert. Das Setzen des Prescalers `ADPS0` (*AD prescaler*) auf `7`, bewirkt eine Frequenz des A/D-Moduls von $\frac{1}{128}$ der CPU-Frequenz (=115,2kHz). Der eigentliche Start der Messung wird durch das `ADSC`-Bit (*AD start conversion*) ausgelöst. Anschließend wird in einer Schleife gewartet bis das Interrupt-Flag-Bit `ADIF` die Beendigung der A/D-Wandlung anzeigt. Der ermittelte 10-Bit-Wert in `ADCH:ADCL` wird schließlich in dem Registerpaar `ADC_VALUE_H:ADC_VALUE_L` gespeichert.

Benutzt wird diese Routine in `FUNCTION_ADC` (siehe Anhang D.1), die aufgerufen wird, wenn der Client, durch Senden des Protokollcodes `REQUEST_ADC`, die Messung der Entfernungen beider Infrarotsensoren auslösen will. Nach dem Senden des Protokollcode `RESPONSE_REQUEST_SUCCESSFUL` zurück an den Client, wird zunächst `0x01` als Quelle für den Pin `A1` in `ADC_SOURCE` geladen. Nach Ausführung der A/D-Wandlung durch Aufruf von `ADC_START`, wird das in `ADC_VALUE_H:ADC_VALUE_L` befindli-

Listing 4.17: ADC.asm

```
15 ;This routine starts an A/D-conversion.
16 ADC_START:
17    in TEMP1, ADMUX                                  ;erase previous ADC-source
18    andi TEMP1, 0xE0
19    or TEMP1, ADC_SOURCE                             ;select new ADC-source
20    out ADMUX, TEMP1
21    sbi ADCSR, ADIF                                  ;flag löschen
22    ldi TEMP1, (1<<ADEN)|(7<<ADPS0)|(1<<ADSC)        ;start conversion
23    out ADCSRA, TEMP1
24    WAIT_ADC:                                        ;wait for conversion
25       sbis ADCSRA, ADIF
26       rjmp WAIT_ADC
27    in ADC_VALUE_L, ADCL                             ;store result in memory
28    in ADC_VALUE_H, ADCH
29 ret
```

che Ergebnis an den Client gesendet. Auf gleiche Weise wird mit der Quelle `0x03` für den Pin `A3` verfahren.

Die Umrechnung der 10-Bit-Digitalwerte in die zugehörige Entfernung erfolgt in der Steuersoftware des Clients, da hierfür Fließkommaoperationen notwendig sind.

4.8 Zustandsübergangsdiagramm

Nachdem nun alle benutzten Komponenten und deren Funktionsweise erklärt wurden, soll das Zustandsübergangsdiagramm in Abbildung 4.10 einen Gesamtüberblick über die Funktionsweise des Mikrocontrollers im Zusammenhang mit dem bereits erklärten Protokoll in Tabelle 4.2 auf Seite 43 geben. Es ist konform zu den in *UML 2.0*[26][Amb05] definierten Zustandsdiagrammen.

Die obere Hälfte des Diagramms zeigt den aktuellen Verarbeitungszustand des Mikrocontrollers. Nach der Initialisierung befindet er sich im Zustand `sleep`. Durch einen ankommenden Request des Clients und den dadurch ausgelösten `URXC`-Interrupt, geht er in den Zustand `evaluate request` über, in dem er den gerade empfangenen Befehl

[26] *http://www.omg.org/technology/documents/modeling_spec_catalog.htm#UML*

auswertet. Je nach Art des Befehls geht er in den zugehörigen Folgezustand. Nach der Abarbeitung des Befehls sendet er die entsprechende Antwort an den Client zurück und geht wieder in Zustand `sleep`, um neue Befehle des Clients entgegenzunehmen.

In der unteren Hälfte befindet sich der derzeit aktuelle *action state*. Dort kann man Erkennen, welche Ereignisse eine Änderung des action states bewirken. Dies kann entweder ein gezieltes Kommando des Clients sein, das direkten Einfluss auf den Bewegungszustand hat. Oder aber ein externer interrupt, der den Intervall dekrementiert und, sobald der Intervall Null erreicht, ebenfalls eine Zustandsänderung auslöst.

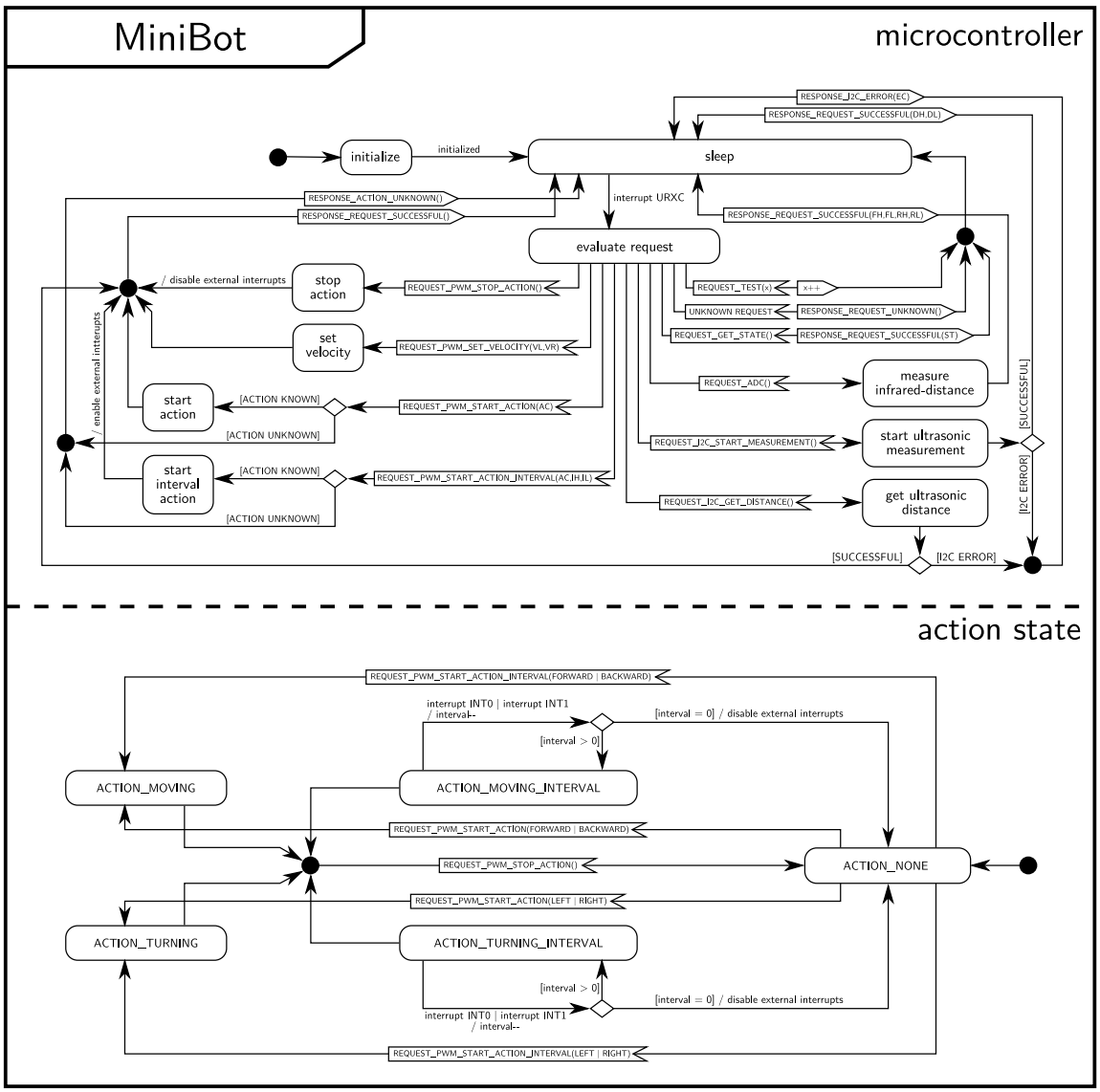


Bild 4.10: Zustandsübergangsdiagramm des Mikrocontrollers

Kapitel 5

Steuersoftware des MiniBots auf dem Barebone-PC

Die Steuersoftware des MiniBots befindet sich in `prog/` auf der beiliegenden CD bzw. in `/usr/src/MiniBot/` auf den MiniBots. Sie besteht aus drei Teilen:

RoboControl, *USBGrabber* und *MiniBotGUI*

Diese befinden sich in den gleichnamigen Unterverzeichnissen. Eine Inhaltsangabe der Verzeichnisse, sowie eine Erklärung der Funktionsweise folgt in den Kapiteln 5.1, 5.2 und 5.3. Anhang G erläutert den Aufbau der Softwareentwicklungsumgebung und gibt eine Übersicht über die verwendeten Tools und Bibliotheken.

5.1 RoboControl

Das Verzeichnis `RoboControl` enthält die `RoboControl`-Klasse, die ein einfach zu bedienendes Interface zur Steuerung des MiniBots zur Verfügung stellt (siehe Kapitel 5.1.1). Außerdem befindet sich dort das Programm `RoboControl` (siehe Kapitel 5.1.2). Es bietet eine einfache Konsolenoberfläche, über die Kommandos an den MiniBot gesendet oder Statusinformationen abgefragt werden können, wobei die bereits oben erwähnte `RoboControl`-Klasse verwendet wird.

Tabelle 5.1: Dateien in `RoboControl`

Datei	Inhalt
RoboControl.h	Headerdatei der RoboControl-Klasse
RoboControl.cpp	Implementierung der RoboControl-Klasse
RoboProtocol.h	Konstantendefinitionen für die Protokollcodes
RoboSetup.h	Konstantendefinitionen für die Konfiguration des MiniBots
Mutex.h	Headerdatei der Mutex-Klasse
Mutex.cpp	Implementierung der Mutex-Klasse
main.cpp	Implementierung des RoboControl-Programms
Makefile	Makefile zur Übersetzung

Tabelle 5.1 listet alle Dateien in diesem Verzeichnis mit ihrem Inhalt auf.

5.1.1 Die `RoboControl`-Klasse

Die Steuerung des MiniBots wird über die Klasse `RoboControl` abgewickelt. Hierzu zählen sowohl das Senden von Fahrbefehlen, als auch die Abfrage von Statusinformationen, wie etwa Sensordaten oder aktueller Bewegungszustand. Die Kommunikation mit dem MiniBot erfolgt nach dem bereits erklärten Protokoll in Tabelle 4.2 auf Seite 43.

Da die Kontrolle des MiniBots in der Regel auf mehrere Threads verteilt ist, wurde die Klasse *threadsafe* und nach dem *Singleton*-Pattern entwickelt, damit alle Threads den MiniBot über dasselbe Objekt ansteuern[CH97]. Wenn ein Thread Zugriff auf den MiniBot möchte, so kann er sich mittels `RoboControl* getInstance()` einen Handle auf die einzige Instanz holen. Das eigenständige Erzeugen eines Objektes vom Typ `RoboControl` ist nicht möglich, da der Konstruktor als `private` deklariert wurde. Bei Aufruf von `getInstance` wird anhand des internen Zählers `m_instanceCount`, der die Anzahl der Handles auf das Objekt in der statischen Variable `m_instance` speichert, überprüft, ob bereits ein Objekt vom Typ `RoboControl` erzeugt wurde. Ist dies nicht der Fall, so wird ein neues Objekt erzeugt. Daraufhin wird der Zähler `m_instanceCount` inkrementiert und ein Zeiger auf dieses Objekt zurückgegeben.

Wenn ein Thread den Handle auf das Objekt nicht mehr benötigt, so kann er ihn mittels `void freeInstance()` wieder freigeben. Die Funktionsweise ist symmetrisch zum Aufruf von `getInstance`. Zunächst wird `m_instanceCount`, der Zähler für die Handles, wieder dekrementiert. Dann wird überprüft, ob noch weitere Handles existieren. Ist dies nicht der Fall, so wird das in `m_instance` gespeicherte Objekt gelöscht.

Um die Funktionen von `RoboControl` darüberhinaus *threadsafe* zu machen wurde eine `Mutex`-Klasse[CH97] implementiert. Sie stellt eine `lock()`- und eine `unlock()`-Funktion zur Verfügung. Intern arbeitet sie mit `pthread_mutex_t` aus der `pthread`-Bibliothek[BN96]. Die `RoboControl`-Klasse verfügt über solch ein in `m_mutex` gespeichertes statisches `Mutex`-Objekt. Am Anfang jeder Funktion wird dessen `lock`-Funktion aufgerufen. Beim Verlassen wird dessen `unlock`-Funktion aufgerufen. Wenn ein Thread nun eine Funktion aufruft und ein `lock` durchführt, so werden weitere Threads beim Versuch ein `lock` durchzuführen solange blockiert, bis der Thread die Funktion verlässt und das `Mutex` mit `unlock` wieder freigibt. Jetzt kann ein weiterer Thread ein `lock` erhalten und die gewünschte Funktion betreten.

In `int init()` (siehe Listing 5.1) wird die Initialisierung der Verbindung zwischen Steuersoftware und MiniBot durchgeführt. Dort wird zuerst der serielle Port durch Aufruf der Funktion `int openSerialPort()`geöffnet. Der Pfad zum seriellen Port ist im Namensraum `RoboSetup` in der Setup-Datei `RoboSetup.h` als Konstante gespeichert. Bei erfolgreichem Öffnen des Ports gibt `openSerialPort` den Filedescriptor des Ports zurück, andernfalls `-1`. Durch Aufruf von `void configureSerialPort()` wird dieser nach erfolgreichem Öffnen konfiguriert[Kai89]. Hier werden die gleichen Einstellungen für Baudrate, Stoppbits, Flusskontrolle, etc. vorgenommen, wie sie auch auf dem Mikrocontroller gemacht wurden. Schließlich wird die Verbindung und die korrekte Funktionsweise des Mikrocontrollers durch Aufruf von `int testConnection()` getestet. Dort wird der im Protokoll beschriebene Test durch Senden von `REQUEST_TEST` gefolgt von einem Byte mit dem Wert `97` durchgeführt. Anschließend wird überprüft, ob der Mikrocontroller das gesendete Byte inkrementiert (Wert `98`) zurückgeschickt hat. Ist dies der Fall, so funktionieren Verbindung und Mikrocontroller einwandfrei und die Funktion gibt `0` zurück, andernfalls wird `-1` zurückgegeben. Wenn alle Funktionsaufrufe von `init` erfolgreich ausgeführt wurden, gibt auch `init` `0` zurück, andernfalls `-1`.

Listing 5.1: RoboControl.cpp

```
61 int RoboControl::init(){
62   m_mutex.lock();
63   int retval = 0;
64   if(!m_initialized){
65     //connection not already initialized
66     m_fileDescriptor = openSerialPort();  //openserial port
67     if(m_fileDescriptor != -1){
68       configureSerialPort();              //configure serial port
69       if(testConnection() != -1){         //test connection and functionality
70         //everything allright
71         m_initialized = true;
72         retval = 0;
73       }else{
74         retval = -1;
75       }
76     }else{
77       retval = -1;
78     }
79   }
80   m_mutex.unlock();
81   return retval;
82 }
```

Die Protokollcodes zur Kommunikation sind in der Datei `RoboProtocol.h` im Namensraum `RoboProtocol` als Konstanten hinterlegt.

Die Methoden zur Kommunikation mit dem MiniBot verlaufen alle nach dem selben Schema. Dieses soll am Beispiel von `int getState(roboState& state)` (siehe Listing 5.2) zur Abfrage des Bewegungszustandes erläutert werden.

Nachdem das Mutex gelockt und der Return-Wert initialisiert wurde, wird ein Array für den Datenaustausch angelegt. Der Protokollcode `REQUEST_GET_STATE` wird an erster Stelle des Arrays eingetragen. Anschließend wird dieses Byte gesendet. Daraufhin wird die Antwort empfangen und ebenfalls in diesem Array gespeichert. Sobald beim Senden oder Empfangen ein Fehler auftritt, wird der Return-Wert auf `-1` gesetzt. Andernfalls wird überprüft, ob das erste Byte den Wert `RESPONSE_REQUEST_SUCCESSFUL` hat. Ist dies nicht der Fall, so ist ein Fehler aufgetreten und der Return-Wert wird auf den Wert dieses ersten Bytes gesetzt. Ansonsten war die Abfrage erfolgreich und der im zweiten Byte übertragene Zustand wird in der per *call-by-reference* übergebenen Variable `state`

Listing 5.2: RoboControl.cpp

```
364 int RoboControl::getState(roboState& state){
365   m_mutex.lock();
366   int retval = 0;
367   unsigned char buffer[2];
368   buffer[0] = RoboProtocol::REQUEST_GET_STATE;
369   if(write(m_fileDescriptor, buffer, 1) != 1){
370     retval = -1;
371   }else if(read(m_fileDescriptor, buffer, 2) != 2){
372     retval = -1;
373   }else if(buffer[0] != RoboProtocol::RESPONSE_REQUEST_SUCCESSFUL){
374     retval = buffer[0];
375   }else{
376     state = static_cast<roboState>(buffer[1]);
377   }
378   m_mutex.unlock();
379   return retval;
380 }
```

gespeichert. Vor Beendigung der Funktion wird auf dem Mutex wieder ein Unlock durchgeführt. Dann wird der Return-Wert `0` zurückgegeben.

Nach diesem Schema funktionieren fast alle Funktionen. Lediglich die Methoden zur Steuerung des Ultraschall-Sensors beinhalten eine weitere Abfrage, da hier zusätzlich die Antwort `RESPONSE_I2C_ERROR` gefolgt von dem Fehlercode möglich ist. Ansonsten unterscheidet sich die Abfolge jedoch nicht.

Da beide Motoren unterschiedliche Laufeigenschaften aufweisen, kommt es bei gleicher Ansteuerung zu unangenehmen Nebeneffekten. So fährt der Roboter nach der Übertragung identischer Werte für die Pulsweitenmodulation des rechten und linken Motors mittels `REQUEST_PWM_SET_VELOCITY` nicht genau geradeaus. Außerdem unterscheiden sich die Werte, die für eine normalschnelle Geradeausfahrt nötig sind, von denen für eine normalschnelle Drehung. Daher wurde die einstellbare Geschwindigkeit auf drei Level begrenzt, die in der Enumeration `roboSpeed` definiert sind. Dabei steht `min` für die minimale, `med` für die mittlere und `max` für die maximale Geschwindigkeit. Die erforderlichen Werte für die Pulsweitenmodulationen beider Motoren wurden durch Messungen festgestellt und sind in den Arrays `RoboSetup::leftSpeedTable[][]` und `RoboSetup::rightSpeedTable[][]` gespeichert. Für jede der drei möglichen Ge-

schwindigkeiten und für jede mögliche Bewegung wurden hier die an den Roboter zu übertragenden Werte für den linken bzw. rechten Motor eingetragen. Diese werden von der Funktion `postVelocity`, die aufgerufen wird, wenn Geschwindigkeit oder Art der Bewegung geändert werden, an den MiniBot übertragen. Diese führt dazu, dass der Roboter bei jeder der Geschwindigkeiten geradeaus fährt und eine Geschwindigkeitsanpassung, beispielsweise bei dem Übergang von normalschneller Vorwärtsfahrt zu normalschneller Drehung, entfällt.

Leider unterscheiden sich diese Werte bei beiden MiniBots. Daher muss vor dem Kompilieren die Umgebungsvariable `MINIBOT` auf `MINIBOT_1` bzw. `MINIBOT_2` gesetzt werden, je nach dem für welchen MiniBot die Steuersoftware kompiliert werden soll. Dadurch werden über Präprozessordirektiven in `RoboSetup.h` die zu diesem Roboter gehörigen Werte benutzt. Ist diese Umgebungsvariable nicht gesetzt, so wird beim Kompilieren eine Fehlermeldung angezeigt.

Bei intervall-basierten Bewegungen wird dem Mikrocontroller nur die entsprechende Anzahl der Taktimpulse von den Taktgeberscheiben übergeben (siehe Kapitel 4.5). Die Parameter der zugehörigen Steuer-Funktionen haben jedoch als Einheit Zentimeter für Translationen bzw. Grad für Rotationen, so dass eine Umrechnung in die zugehörige Anzahl von Taktimpulsen erforderlich ist. Dies wird über die beiden Konstanten `TICKS_PER_CM` und `TICKS_PER_DEGREE` in der Setup-Datei `RoboSetup.h` geregelt. Durch Multiplikation mit diesen beiden Faktoren, werden die Einheiten der Routinen in die korrespondierende Anzahl von Taktimpulsen umgerechnet.

Da der Roboter nach dem Erreichen der berechneten Anzahl von Ticks jedoch nicht sofort anhält, müssen spezielle, für den jeweiligen Geschwindigkeits-Level angepasste, Korrekturwerte eingerechnet werden. Diese sind in den Arrays `TRANSLATION_CORRECTION` (für Translationen) und `ROTATION_CORRECTION` (für Rotationen) gespeichert. Es handelt sich um die Anzahl von Korrekturticks für die drei Geschwindigkeitslevel, die aus obigem Grund von der errechneten Anzahl subtrahiert werden müssen. Hierdurch wird, das durch den größeren Impuls bei höheren Geschwindigkeiten bedingte Überschreiten der geforderten Tickzahl, kompensiert.

Wie bereits erwähnt, liefert der Mikrocontroller für die Entfernung der Infrarot-Sensoren lediglich 10-Bit-Digitalwerte, die durch eine A/D-Wandlung des analogen Ausgangspe-

gels der Sensoren entstehen. Leider ist der Zusammenhang zwischen den Ausgangspegeln bzw. den Digitalwerten und der zugehörigen Entfernung zum reflektierenden Objekt nicht linear[1]. Die Umrechnung erfolgt durch Aufruf der Approximierungsfunktion `int digitalValueToDistance(int value)`. Um die Approximierungsfunktion zu bestimmen, wurden zunächst die gemessenen Digitalwerte bei ansteigende Entfernung in Zentimeter-Schritten in einer Wertetabelle notiert (siehe Anhang E). Die Werte wurden nur in dem Intervall zwischen `4cm` und `30cm` gemessen, da dies der Spezifikation der Sensoren entspricht.

Zur Ermittlung der Approximierungsfunktion wurde die Trial-Version des Programms *FindGraph*[2] von *UNIPHIZ Lab*[3] benutzt (siehe Anhang G).

Dies führte zu folgender Approximierungsfunktion:

$$x = -1.458224732336748 + \frac{6156.057917536882}{y} - \frac{262380.5447723502}{y^2}$$

Hierbei ist `y` der Digitalwert und `x` die zugehörige Entfernung in cm. In Abbildung 5.1 sind die gemessenen Wertepaare und die errechnete Approximierungsfunktion eingezeichnet. Die Koeffizienten der Approximierungsfunktion wurden bei der Implementierung in `digitalValueToDistance` durch Präprozessordirektiven definiert, um die Lesbarkeit zu erhöhen.

5.1.2 Das Steuerprogramm `RoboControl`

Das Programm `RoboControl` wurde zur einfachen Steuerung des MiniBots und zum Testen aller Funktionen verwendet. Es stellt eine Konsolenoberfläche zur Verfügung, über die durch spezielle Tasten (siehe Tabelle 5.4 auf Seite 88) Befehle an den MiniBot gesendet oder Statusinformationen abgefragt werden können.

Zur Ein- und Ausgabe wurde die *ncurses*-Bibliothek[4][Str86] verwendet. Sie bietet diverse Vorteile gegenüber der stream-basierten Ein- und Ausgabe. Zum Einen besteht die Mög-

[1] siehe *datasheets/gp2d120.pdf*
[2] *http://www.uniphiz.com/findgraph.htm*
[3] *http://www.uniphiz.com/*
[4] *http://www.gnu.org/software/ncurses/ncurses.html*

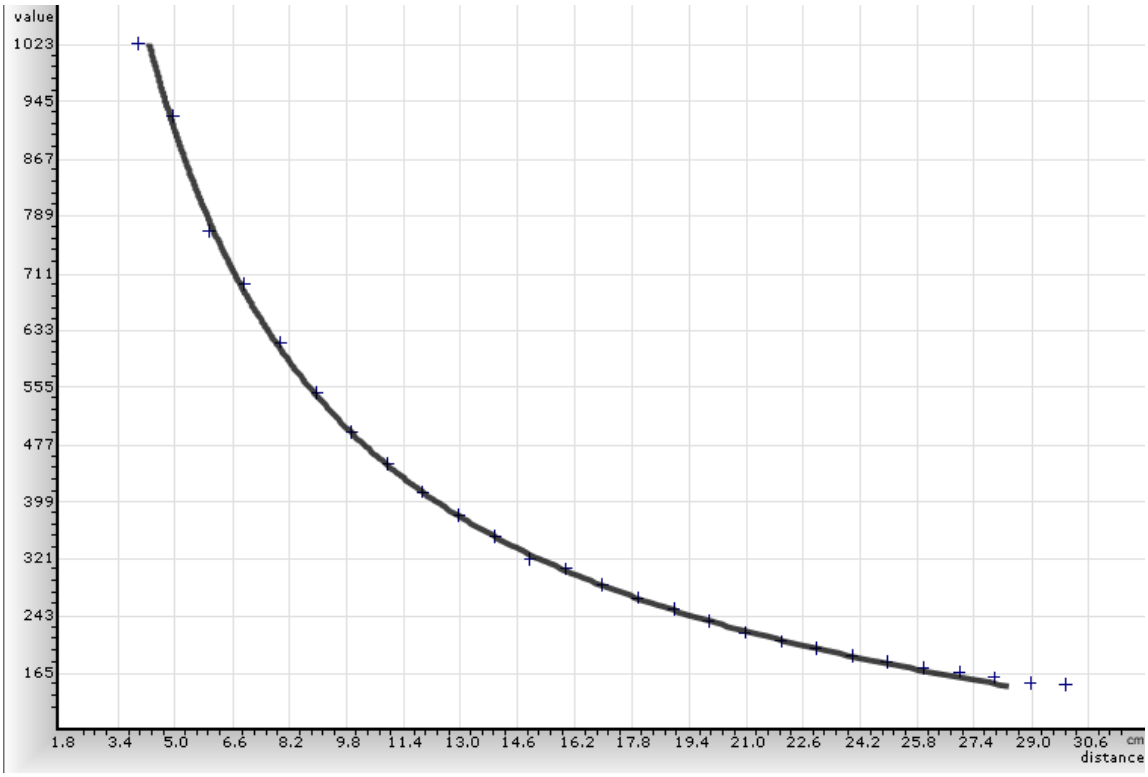


Bild 5.1: Gemessene Werte und Approximierungsfunktion

lichkeit einer ungepufferten Eingabe, wodurch die direkte Steuerung des MiniBots ohne bestätigende Return-Eingaben ermöglicht wird. Zum Anderen kann die Cursorposition bei der Ausgabe beliebig gesetzt werden, wodurch die Positionierung bestimmter Ausgaben erleichtert wird.

Alle Daten des MiniBots werden in globalen Variablen gespeichert, die mit `Robo*` beginnen. Tabelle 5.2 listet alle globale Variablen mit der Beschreibung ihres Inhalts auf. Während die ersten drei Variablen Parameter zur Bewegung des Roboters speichern, enthalten die folgenden vier Variablen Statusinformationen über den MiniBot. Ein Refresh findet nur beim Senden des entsprechenden Befehls an den MiniBot statt. Einen Auswahlcode, der beschreibt welche Werte durch Drücken von `+` oder `-` inkrementiert bzw. dekrementiert werden sollen enthält `RoboSelection`. Tabelle 5.3 zeigt die zulässigen Auswahlcodes mit zugehöriger Beschreibung. Sie sind über Präprozessordirektiven de-

Tabelle 5.2: Globale Variablen des `RoboControl`-Programms

Globale Variable	Beschreibung
roboSpeed RoboSpeed	Geschwindigkeits-Level des MiniBots
int RoboDistance	Wegstrecke in cm
int RoboAngle	Drehwinkel in °
int RoboUltraSonicRange	Gemessene Entfernung des Ultraschall-Sensors
int RoboFrontIRRange	Gemessene Entfernung des vorderen Infrarot-Sensors
int RoboRearIRRange	Gemessene Entfernung des hinteren Infrarot-Sensors
roboState RoboState	Bewegungszustand des MiniBots
int RoboSelection	Auswahl der Werte, die geändert werden sollen
char RoboMode	Bewegungs-Modus des MiniBots
RoboControl∗ Robo	Zeiger auf das Steuer-Objekt
struct timespec I2CWaitTime	Struct zum Warten auf die Ultraschall-Messung
bool Quit	Flag zum Beenden des Programms

Tabelle 5.3: Auswahlcodes mit Beschreibung

Auswahlcode	Beschreibung
ROBO_SELECTION_VELOCITY	Änderung des Geschwindigkeits-Levels
ROBO_SELECTION_DISTANCE	Änderung der Wegstrecke
ROBO_SELECTION_ANGLE	Änderung des Drehwinkels

finiert. Die Variable `RoboMode` enthält den derzeitigen Bewegungsmodus. `p` steht für permanente Bewegung, `i` für intervall-basierte Bewegung. `Robo` enthält einen Zeiger auf das `RoboControl`-Objekt, das zur Steuerung und Kommunikation mit dem MiniBot benutzt wird. `I2CWaitTime` enthält ein Struct, um mit `nanosleep` zwischen Auslösen einer Messung des Ultraschall-Sensors und Abfrage des Ergebnisses, die in `RoboSetup::I2C_WAIT_MSEC` gespeicherte Anzahl von Millisekunden zu warten. Das Flag `Quit` dient zur Beendigung des Programms.

Nach dem Start der `main()`-Methode (siehe Listing 5.3), wird zunächst ein Handle für das `RoboControl`-Objekt mit `getInstance()` besorgt und in der globalen Varia-

Listing 5.3: main.cpp

```
459 int main(int argc, char** argv){
460   cout << "Starting initialization:" << endl;
461   Robo = RoboControl::getInstance();
462   if(Robo->init()){
463     cout << "Error during initialization!" << endl;
464     RoboControl::freeInstance();
465     return -1;
466   }
467   cout << "Initilization successful!" << endl;
468   initCurses();
469   initRobotValues();
470   RoboSetVelocity();
471   printDescription();
472   printRobotValues();
473   Quit = false;
474   printw("Enter command: ");
475   while(!Quit){
476     if(handleCommand(getch()) == -1){
477       printMessage("Unknown command!");
478     }
479   }
480   endwin();
481   RoboControl::freeInstance();
482   return 0;
483 }
```

ble `Robo` gespeichert. Anschließend wird die Verbindung zum Roboter durch Aufruf von `init()` initialisiert und getestet. Schlägt dies fehl, so wird das Handle sofort wieder mit `freeInstance()` freigegeben und die `main()`-Methode mit Rückgabewert `-1` verlassen. Andernfalls wird *ncurses* durch Aufruf von `initCurses()` initialisiert und konfiguriert und die globalen Variablen werden durch `initRobotValues()` mit Startwerten gefüllt. Anschließend wird eine Beschreibung der Tastenkürzel zur Steuerung des Programms mit `printDescription()` auf der Konsole ausgegeben. Darunter wird mit `printRobotValues()` der derzeitige Inhalt der globalen Variablen angezeigt. Nachdem das Flag zum Beenden auf `false` gesetzt wurde, werden in einer Schleife Keycodes bzw. Befehle des Users durch drücken von bestimmten Tasten mittels `getch()` empfangen und durch `handleCommand()` ausgewertet bis `Quit` auf `true` gesetzt wird. Dann wird *ncurses* mit `endwin()` deinitialisiert, der Handle auf das `RoboControl`-Objekt

Listing 5.4: main.cpp

```
425 int handleCommand(int command){
426   int retval = 0;
427   switch(command){
428     case KEY_LEFT: RoboTurnLeft(); break;
429     case KEY_RIGHT: RoboTurnRight(); break;
430     case KEY_UP: RoboMoveForward(); break;
431     case KEY_DOWN: RoboMoveBackward(); break;
432     case KEY_END: RoboStop(); break;
433     case 'u': RoboGetUltraSonicRange(); break;
434     case 'i': RoboGetIRRanges(); break;
435     case 's': RoboGetState(); break;
436     case 't': toggleMode(); break;
437     case '+': adjustSelection(true); break;
438     case '-': adjustSelection(false); break;
439     case 'v': RoboSelection = ROBO_SELECTION_VELOCITY; break;
440     case 'd': RoboSelection = ROBO_SELECTION_DISTANCE; break;
441     case 'a': RoboSelection = ROBO_SELECTION_ANGLE; break;
442     case 'q': Quit = true; break;
443     default: retval = -1;
444   }
445   return retval;
446 }
```

freigegeben und die `main()`-Methode mit einem Return-Wert von `0` verlassen.

Die Methode `handleCommand()` (siehe Listing 5.4) wird aufgerufen, sobald der User eine Taste gedrückt hat. Sie wertet den empfangenen Keycode aus und ruft die zugehörige Funktion (`Robo*`) auf, die lediglich die entsprechende Funktion des `RoboControl`-Objekts aufruft.

Ausnahmen hierzu stellen nur folgende Tasten dar:

Bei Drücken von `t` wird der in `RoboMode` gespeicherte Bewegungsmodus durch Aufruf von `toggleMode()` umgeschaltet. Mit `q` wird das Flag `Quit` zum Beenden des Programms auf `true` gesetzt. Die Tasten `v`, `d` oder `a` setzen die in `RoboSelection` gespeicherte Auswahl auf die entsprechenden Werte (siehe Tabelle 5.3). Die zugehörigen Werte können dann mit `+` bzw. `-` inkrementiert bzw. dekrementiert werden, wobei die Funktion `adjustSelection(bool)` benutzt wird. Für Wegstrecke und Drehwinkel werden die in `RoboSetup.h` definierten Schrittweiten `DISTANCE_STEPPING` bzw.

`ANGLE_STEPPING` benutzt. Außerdem wird sichergestellt, dass Wegstrecke und Drehwinkel nicht `DISTANCE_MAX` bzw. `ANGLE_MAX` übersteigen.

Wenn dem empfangenen Keycode keine Funktion zugeordnet ist, gibt `handleCommand -1` zurück, andernfalls `0`.

Tabelle 5.4 zeigt einen Überblick, mit welchen Funktionen die in dem Programm benutzten Tasten belegt sind.

Tabelle 5.4: Tasten und Funktionen des `RoboControl`-Programms

Taste	Funktion
←	Linksdrehung
→	Rechtsdrehung
↑	Vorwärtsfahrt
↓	Rückwärtsfahrt
`Ende`	Bewegung anhalten
`t`	Umschaltung permanente ↔ intervall-basierte Bewegung
`+`	Werte der aktuellen Auswahl inkrementieren
`-`	Werte der aktuellen Auswahl dekrementieren
`v`	Auswahl auf Geschwindigkeits-Level setzen
`d`	Auswahl auf Wegstrecke setzen
`a`	Auswahl auf Drehwinkel setzen
`u`	Ultraschall-Messung durchführen
`i`	Infrarot-Messung durchführen
`s`	Bewegungszustand des MiniBots abfragen
`q`	Programm beenden

5.2 USBGrabber

Im Verzeichnis `USBGrabber` befindet sich die `USBGrabber`-Klasse, die Funktionen bietet, um USB-Kameras zu konfigurieren und Bilder von ihnen zu empfangen (siehe

Tabelle 5.5: Dateien in `USBGrabber`

Datei	Inhalt
USBGrabber.h	Headerdatei der USBGrabber-Klasse
USBGrabber.cpp	Implementierung der USBGrabber-Klasse
main.cpp	Implementierung des USBGrabber-Programms
Makefile	Makefile zur Übersetzung

Kapitel 5.2.1). Das Programm `USBGrabber` verwendet diese Klasse, um beliebig viele Bilder von einer angeschlossenen USB-Kamera aufzunehmen und im *ppm*-Format zu speichern (siehe Kapitel 5.2.2). Eine Auflistung aller Dateien in diesem Verzeichnis mit Beschreibung ihres Inhalts befindet sich in Tabelle 5.5.

5.2.1 Die `USBGrabber`-Klasse

Damit der MiniBot seine Umgebung wahrnehmen kann, wurde an dessen Front eine USB-Webcam von Typ `Logitech Communicate STX`[5] angebracht. Diese Kamera hat die Vorteile, dass sie kompatibel zu *USB 2.0* ist, einen Autofokus besitzt und es existiert ein funktionsfähiges Kernel-Modul zur Ansteuerung der Kamera unter Linux.

Die Ansteuerung von Webcams unter Linux erfolgt über die *v4l* (*video for linux*) API[6]. Dies ist eine *video capture* API, die das Frontend zur Kommunikation mit jeglicher Art von Geräten, die Bilder (evtl. mit Audiosignalen) liefern, darstellt. Das Backend wird von einem hardware-spezifischen Kernelmodul gebildet, das in der Lage ist, die jeweilige Kamera anzusteuern. Zur Ansteuerung der Logitech Communicate STX eignet sich das *spca5xx*-Kernelmodul[7], das jedoch zunächst kompiliert werden muss. Nach dem Kompilieren und Installieren des Kernel-Moduls, kann die korrekte Funktionsweise durch Anschließen der Kamera an einen freien USB-Port getestet werden. Dann sollten sowohl das `videodev`-Modul, als auch das `spca5xx`-Modul geladen werden. Außerdem sollte

[5] siehe *datasheets/LogitechCommunicateSTX.pdf*
[6] *http://linux.bytesex.org/v4l2/*
[7] *http://mxhaard.free.fr/spca5xx.html*

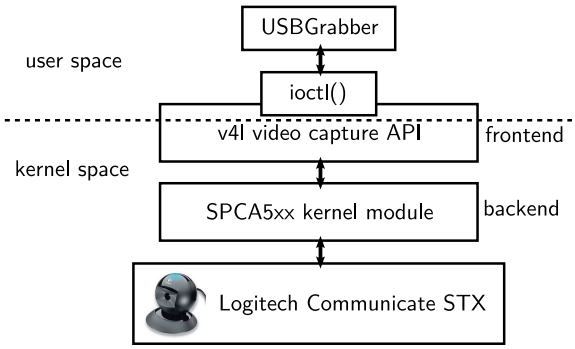


Bild 5.2: Zusammenspiel der Komponenten bei der v4l-API

`dmesg` eine Eintrag für die erkannte Kamera ausgeben. Darüberhinaus sollte ein *Device-Node* `/dev/video*` erzeugt werden, über den auf die Kamera zugegriffen werden kann. Nun kann der Zugriff auf die v4l-API durch Aufrufe der Funktion `ioctl()` erfolgen (siehe unten). Das Schema in Abbildung 5.2 verdeutlicht dieses Zusammenspiel zwischen `ioctl()`, v4l-API und `spca5xx`-Kernelmodul.

Die `USBGrabber`-Klasse stellt eine Reihe von Funktionen zur Verfügung, um die Kamera durch Zugriff auf die v4l-API zu konfigurieren und Bilder von ihr zu empfangen. Die Bilder können entweder als Rohdaten oder als *SpatialDomainImage* bereitgestellt werden. SpatialDomainImage ist die in der *PUMA*-Bibliothek[8] verwendete Bildklasse. Um diese jedoch benutzen zu können, muss PUMA sauber installiert sein und das Makro `PUMA` muss in der Toplevel-Makefile vor der Kompilierung gesetzt werden.

Nachdem ein Objekt von Typ `USBGrabber` erzeugt wurde, muss dieses durch Aufruf von `int init(const char* device)` (siehe Listing 5.5) initialisiert werden. Der Parameter gibt dabei den Pfad zum Device-Node (`/dev/video*`) des zugehörigen v4l-Geräts an. Wenn der Device-Node erfolgreich geöffnet wurde, erfolgt die weitere Kommunikation über den zurückgegebenen Filedescriptor.

Die Steuerung verläuft über Aufrufe von `ioctl(int fd, int req, ...)`. Diese Funktion dient zur Manipulation von Parametern spezieller Geräte-Dateien. Hierbei ist `fd` der Filedescriptor des Geräts und `req` ein gerätespezifischer Request-Code. Es folgt ein

[8] *http://www.uni-koblenz.de/FB4/Institutes/ICV/AGPaulus/puma*

Listing 5.5: USBGrabber.cpp

```
39 int USBGrabber::init(const char* device){
40   cout << "Initializing USB-Camera on '" << device << "':" << endl;
41   m_device = device;
42   //open filedescriptor
43   m_fd = open(m_device, O_RDWR);
44   if(m_fd == -1){
45     cout << "Error: Unable to open device '" << device << "'!" << endl;
46     return -1;
47   }
48   //try to get the parameters of video capability associated with that
49   //filedescriptor
50   if(ioctl(m_fd, VIDIOCGCAP, &m_videocap) == -1){
51     cout << "Error: '" << device << "' seems not to be a v4l-device" << endl;
52     return -1;
53   }
54   cout << "Camera found: " << m_videocap.name << endl;
55   //clear structs for camera parameters
56   memset(&m_videopict, 0, sizeof(struct video_picture));
57   memset(&m_videowin, 0, sizeof(struct video_window));
58   memset(&m_videochan, 0, sizeof(struct video_channel));
59   memset(&m_videobuf, 0, sizeof(struct video_mmap));
60   //fill structs with actual parameters of the camera
61   if(getPictureParameters() || getWindowParameters() || getChannelParameters() != 0){
62     return -1;
63   }
64   cout << "Initialization of USB-Camera completed successfully!" << endl;
65   return 0;
66 }
```

optionales Argument, das zu dem Request-Code gehört. Beispielsweise sendet der Aufruf

```
ioctl(m_fd, VIDIOCGCAP, &m_videocap)
```

in Zeile 50 der Initialisierungsroutine den Request-Code `VIDIOCGCAP` zur Abfrage von Informationen über das zum Filedescriptor `m_fd` zugehörige v4l-Gerät. Diese Informationen werden dann in dem als Argument übergebenen Struct `m_videocap` vom Typ `struct video_capability` gespeichert.

Die Request-Codes sind in `linux/videodef.h` definiert, ebenso wie die verschiedenen Structs, die zum Austausch von Informationen und Einstellungen mit dem v4l-Gerät dienen. Tabelle 5.6 zeigt die hier verwendeten Structs und ihre Bedeutung.

Tabelle 5.6: In `USBGrabber` verwendete Structs

Variable	Typ	Bedeutung
m_videocap	struct video_capability	allgemeine Einstellungen
m_videochan	struct video_channel	Kanal-Einstellungen
m_videopict	struct video_picture	Bildton-Einstellungen
m_videowin	struct video_window	Bildformat-Einstellungen
m_videobuf	struct video_mmap	Memory-Mapping-Einstellungen

In der Initialisierungsroutine werden diese Structs zunächst initialisiert und durch Aufruf der Funktionen `getPictureParameters()`, `getWindowParameters()` und `getChannelParameters()` mit den aktuellen Einstellungen des v4l-Geräts gefüllt. Dies wird wiederum durch Aufruf von `ioctl` mit den Request-Codes `VIDIOCGPICT`, `VIDIOCGWIN` bzw. `VIDIOCGCHAN`.

Die Funktion `int setSize(int width, int height)` ermöglicht eine Änderung der Auflösung. Hierbei wird die neue horizontal und vertikale Auflösung an den entsprechenden Stellen des Structs `m_videowin` eingetragen. Dann werden diese Einstellungen durch Aufruf von `setWindowParameters()` übertragen, wobei wiederum `ioctl` in Verbindung mit dem Request-Code `VIDIOCSWIN` und dem zugehörigen Struct `m_videowin` als Argument benutzt wird. Anschließend wird durch nochmaligen Aufruf von `getPictureParameters()` überprüft, ob die Einstellungen korrekt gemacht wurden.

Das Ändern der Farbpalette geschieht mit `int setPalette(int palette)` und funktioniert ähnlich. Die möglichen Farbpaletten sind in Tabelle 5.7 aufgelistet und erklärt. Ihre Definitionen befinden sich in `linux/videodef.h`. Zuerst wird die der gewählten Farbpalette entsprechende Farbtiefe gesetzt. Anschließend werden Farbpalette und Farbtiefe im Struct `m_videopict` eingetragen und durch `setPictureParameters()` übertragen. Dann wird wieder überprüft, ob alle Einstellungen korrekt gemacht wurden. Leider wird die Farbpalette `VIDEO_PALETTE_GREY` von dieser Kamera nicht unterstützt und kann somit nicht benutzt werden.

Zur eigentliche Aufnahme eines Bildes dient die private Funktion `int grab()` (sie-

Tabelle 5.7: Farbpaletten mit zugehörigem Bitformat

Farbpalette	bpp	Bitformat (R=rot,G=grün,B=blau,A=alpha,X=grau)
VIDEO_PALETTE_GREY	8	[XXXXXXXX]
VIDEO_PALETTE_RGB565	16	[RRRRRGGGGGGBBBBB]
VIDEO_PALETTE_RGB24	24	[BBBBBBBBGGGGGGGGRRRRRRRR]
VIDEO_PALETTE_RGB32	32	[AAAAAAAARRRRRRRRGGGGGGGGBBBBBBBB]

he Listing 5.6). Dort wird zunächst anhand von Auflösung und Farbpalette die benötigte Puffergröße berechnet und mit der aktuellen Puffergröße verglichen. Sind die beiden Größen ungleich, so muss die Puffergröße verändert werden. Zum Austausch der Bilddaten wird *Memory-Mapping* benutzt. Dabei wird durch die Funktion `mmap` eine bestimmte Anzahl von Bytes aus einer Datei (hier das v4l-Gerät) in den Speicher gemapped. Über den zurückgegebenen Zeiger kann dann auf die Daten so zugegriffen werden, als würden sie sich im Speicher befinden. Es wird allerdings zuvor überprüft, ob schon ein Mapping stattgefunden hat. Ist dies der Fall, so wird nur die Größe des Mappings mit `mremap` angepasst, andernfalls wird eine neues Mapping mit `mmap` angelegt. Anschließend enthält `m_buffer` einen Zeiger auf den gemappten Speicherbereich.

Die Aufnahme eines Bildes wird wieder mit einem Aufruf von `ioctl` ausgelöst. Hierbei wird als Request-Code `VIDIOCMCAPTURE` und als Argument `m_videobuf` übergeben. Das Struct `m_videobuf` enthält nötige Informationen über die Größe des Puffers, der beim Memory-Mapping benutzt wird. Die Werte in `m_videobuf` werden bei jedem Aufruf von `getPictureParameters` und `getWindowParameters` auf dem aktuellsten Stand gehalten. Anschließend muss gewartet werden bis das Bild komplett übertragen wurde, was durch den Request-Code `VIDIOCSYNC` mit Struct `m_videobuf` als Argument bewerkstelligt wird. Nach erfolgter Synchronisierung zeigt `m_buffer` auf die Bilddaten in dem gewählten Format und die Aufnahme ist beendet.

Benutzt wird `grab` von den Funktionen `unsigned char* grabRawImage()` und `SpatialDomainImage* grabImage()`. Die Funktion `grabRawImage` gibt lediglich einen Zeiger auf eine Kopie der Bilddaten im Rohformat zurück. Eine Kopie ist deshalb nötig, da der Inhalt des gemappten Speicherbereichs bei der Aufnahme des näch-

sten Bildes wieder überschrieben wird. Die Funktion `grabImage` hingegen gibt ein *SpatialDomainImage*, wie es in PUMA verwendet wird zurück. Um die `USBGrabber`-Klasse mit dieser Funktion zu kompilieren muss, wie bereits erwähnt, PUMA sauber installiert und das Makro `PUMA` im Toplevel-Makefile gesetzt sein. Das erzeugte SpatialDomainImage enthält dann je nach gewählter Farbpalette intern ein *GreyLevelImage* (bei Farbpalette `VIDEO_PALETTE_GREY`) oder ein *ColorImage* (sonst).

Listing 5.6: USBGrabber.cpp

```
265 int USBGrabber::grab(){
266   //compute needed buffersize in bytes
267   int bufferSize = m_videobuf.width * m_videobuf.height * m_videopict.depth
268                    >> 3;
269   //compare actual buffersize with needed buffersize
270   if(m_bufferSize != bufferSize){
271     // new size of framebuffer needed
272     if(m_buffer){
273       //buffer has already been mapped, so use mremap
274       m_buffer = (unsigned char*) mremap(m_buffer, m_bufferSize, bufferSize,
275                                          MREMAP_MAYMOVE);
276     }else{
277       //buffer hasn't already been mapped, so use the normal mmap
278       m_buffer = (unsigned char*) mmap(0, bufferSize, PROT_READ|PROT_WRITE,
279                                        MAP_SHARED, m_fd, 0);
280     }
281     //save new buffersize
282     m_bufferSize = bufferSize;
283   }
284   //start capture
285   if(ioctl(m_fd, VIDIOCMCAPTURE, &m_videobuf) == -1){
286     cout << "Error: Capture from camera failed" << endl;
287     return -1;
288   }
289   //wait for synchronization that indicates that the capture is completed
290   if(ioctl(m_fd, VIDIOCSYNC, &m_videobuf) == -1){
291     cout << "Error: Synchronization failed" << endl;
292     return -1;
293   }
294   return 0;
295 }
```

Tabelle 5.8: Parameter des `USBGrabber`-Programms

Parameter	Erklärung
−d device	Angabe eines beliebigen Device-Nodes (default: /dev/video0)
−r resolution	Angabe einer Auflösung (0=640x480 [default]; 1=320x240)
−c count	Anzahl der aufzunehmenden Bilder (default: 1)
−h	Anzeige der Hilfe

5.2.2 Das Programm `USBGrabber`

Zum Testen der Funktionsweise von Kamera mit zugehörigem Kernelmodul und der eigentlichen `USBGrabber`-Klasse, wurde das `USBGrabber`-Programm implementiert. Es bietet die Möglichkeit unter Benutzung der `USBGrabber`-Klasse beliebig viele Bilder von einem ausgewählten v4l-Gerät aufzunehmen und im *ppm*-Format[9] zu speichern.

Tabelle 5.8 gibt eine Übersicht über die möglichen Parameter, die dem Programm übergeben werden können. Wenn keine Parameter angegeben werden, versucht das Programm ein Bild vom v4l-Gerät, das zum Device-Node `/dev/video0` gehört, in der Auflösung `640x480` aufzunehmen und in `image1.ppm` zu speichern. Mit der Option `-d` kann ein anderer Device-Node gewählt werden. Die Auflösung kann mit dem Parameter `-r` geändert werden. Um mehrere Bilder aufzunehmen kann man mit der Option `-c` eine beliebige Anzahl angeben. Die Bilder werden dann unter einem Dateinamen mit dem Präfix `image` gefolgt von einer laufenden Nummer und der Endung `.ppm` gespeichert. Mit `-h` wird eine kurze Erklärung des Programms ausgegeben.

Die `main()`-Funktion verarbeitet zunächst alle angegebenen Parameter durch Aufruf von `void handleOptions()`. Dort werden die eingestellten Standard-Werte durch die eventuell über Parameter angegebenen Werte überschrieben. Anschließend wird ein Objekt der Klasse `USBGrabber` erstellt, initialisiert und konfiguriert. Dann werden die Einstellungen zur Kontrolle auf der Konsole ausgegeben. Nun werden in einer Schleife solange Bilder aufgenommen und im ppm-Format gespeichert bis die gewünschte Anzahl erreicht ist.

[9] siehe *specifications/ppm.pdf*

In der Funktion `void writeImage(unsigned char* image, int index)` erfolgt die Speicherung im ppm-Format. Nachdem der Dateiname erzeugt und die zugehörige Datei geöffnet wurden, wird zuerst der Header ausgegeben. Er hat folgenden Aufbau:

```
P6
Breite Höhe
Farbwert
```

Hierbei ist `P6` die Kennung für das binäre ppm-Format. `Breite` und `Höhe` geben die Auflösung des Bildes in Pixeln an. `Farbwert` steht für den maximalen Farbwert pro Kanal. Nach dem Header folgen unmittelbar die eigentlichen Farbwerte der Pixel von oben links bis unten rechts. Die Werte der einzelnen Kanäle werden in der Reihenfolge rot-grün-blau angegeben.

5.3 MiniBotGUI

Die `MiniBotGUI`-Applikation wurde entwickelt, um alle Informationen, die dem MiniBot zur Verfügung stehen in einer einzigen Anwendung zu visualisieren und zusätzlich dessen Steuerung zu ermöglichen. Zu den angezeigten Informationen zählen insbesondere die von der Kamera aufgenommenen Bilder, die von die GUI in Echtzeit angezeigt werden. Außerdem werden die Entfernungsdaten der Sensoren durch Balken dargestellt. Hierzu wird auf die bereits erklärten Klassen `RoboControl` (siehe Kapitel 5.1.1) und `USBGrabber` (siehe Kapitel 5.2.1) zurückgegriffen. Die Steuerung des MiniBots erfolgt wahlweise durch Klicken auf die dafür vorgesehenen Schaltflächen der GUI oder durch Drücken bestimmter Tasten auf der Tastatur, die als Shortcuts definiert sind. Tabelle 5.9 listet alle Dateien in diesem Verzeichnis mit ihrem Inhalt auf.

Zur Implementierung wurde die Programmierbibliothek *Qt*[10] von *Trolltech*[11] in der Version 4.1 verwendet. Sie ermöglicht die plattformunabhängige Erstellung von graphischen

[10] *http://www.trolltech.com/products/qt*
[11] *http://www.trolltech.com/*

Tabelle 5.9: Dateien in `MiniBotGUI`

Datei	Inhalt
FrameGrabber.h	Headerdatei der FrameGrabber-Klasse
FrameGrabber.cpp	Implementierung der FrameGrabber-Klasse
InfraRedDetector.h	Headerdatei der InfraRedDetector-Klasse
InfraRedDetector.cpp	Implementierung der InfraRedDetector-Klasse
UltraSonicDetector.h	Headerdatei der UltraSonicDetector-Klasse
UltraSonicDetector.cpp	Implementierung der MainWidget-Klasse
MainWidget.h	Headerdatei der MainWidget-Klasse
MainWidget.cpp	Implementierung der UltraSonicDetector-Klasse
main.cpp	Implementierung der MiniBotGUI-Applikation
Makefile	Makefile zur Übersetzung

Benutzeroberflächen und bietet darüberhinaus eine umfangreiche Klassenbibliothek mit häufig benötigten Hilfsmitteln, wie etwa Container-Klassen, Multi-Threading-Unterstützung, etc.[Bla06]

Die graphischen Elemente einer Qt-Applikation werden als *Widgets* bezeichnet. Hierzu zählen beispielsweise Fenster, Buttons, Beschriftungen, Auswahlboxen, etc. Sie sind alle (evtl. auch indirekt) von der Klasse `QWidget` abgeleitet und hierarchisch organisiert. So enthält beispielsweise ein Fenster-Widget verschiedene Sub-Widgets, die ihrerseits auch wieder Sub-Widgets enthalten können, usw.

Zum eigentlichen Start einer Qt-Applikation muss lediglich ein `QApplication`-Objekt erzeugt werden. Dann wird ein Haupt-Widget erzeugt und angezeigt. Nun kann die Kontrolle, durch Aufruf der `exec`-Methode, an das `QApplication`-Objekt übergeben werden und der Benutzer kann mit der graphischen Oberfläche interagieren.

Die Kommunikation zwischen verschiedenen Widgets erfolgt über *Signale* und *Slots*. Ein, unter Umständen auch selbst geschriebenes, Widget kann durch eine bestimmte Interaktion des Benutzers ein Signal auslösen. Wenn dieses Signal zuvor mit einem Slot eines anderen Widgets verbunden wurde, so wird dieser Slot nach dem Auslösen des Signals, ähnlich einer ganz normalen Methode, aufgerufen. Signale und Slots können auch selbst

implementiert werden. Das Auslösen eines Signals erfolgt mit `emit signal`. Das Verbinden eines Signals mit einem Slot geschieht über die Methode `connect`, die jedes Qt-Objekt besitzt. Signale können durch Parameter auch Daten übertragen und diese dem Slot zur weiteren Verarbeitung übergeben.

Da insbesondere die Aufnahme und Übertragung der Kamerabilder sehr viel Zeit in Anspruch nimmt, werden diese Aktionen in einem eigenen Thread ausgeführt, um die GUI währenddessen nicht zu blockieren[CH97]. Die Klasse `FrameGrabber`, die von der Klasse `QThread` abgeleitet ist, implementiert den zugehörigen Thread zur Aufnahme und Übertragung der Kamera-Bilder.

Zuerst ruft das `MainWidget` aus dem Haupt-Thread die Funktion `grabFrame()` der Klasse `FrameGrabber` auf, um ein Bild anzufordern. Da der `FrameGrabber`-Thread noch nicht gestartet wurde, wird dies nun getan, wodurch die `run()`-Methode (siehe Listing 5.7) des Threads aufgerufen wird. Dort erfolgt die Konfiguration der Kamera über das `USBGrabber`-Objekt. Anschließend beginnt eine Schleife, die pro angefordertem Bild einmal ausgeführt wird, bis das Flag `m_stopExecution` durch den Destruktor gesetzt wird. In dieser Schleife wird, nachdem die Rohdaten eines Bildes empfangen wurden, daraus ein QImage zur Anzeige im Haupt-Widget konstruiert. Dann wird dieses Bild durch Auslösen des Signals `newFrame` mit dem erzeugten Bild als Parameter an den Haupt-Thread gesendet. Anschließend legt sich der Thread bis zur Anforderung des nächsten Bildes schlafen. Sollte bei der Aufnahme eines Bildes ein Fehler auftreten, so wird das Signal `cameraError` ausgelöst.

Das Signal `newFrame` ist mit dem Slot `displayFrame` des `MainWidgets` verbunden, wodurch dieser Slot beim Auslösen des Signals `newFrame` ausgeführt wird. Dieser ersetzt lediglich das alte Bild im Label `m_frameLabel` durch das soeben empfangene neue Bild und fordert wiederum durch Aufruf von `grabFrame` das nächste Bild an.

Da der Thread allerdings schon gestartet wurde, wird er nun im Gegensatz zum ersten Aufruf von `grabFrame` nicht gestartet, sondern an der Stelle in der `run`-Methode, an der er sich schlafen gelegt hat, aufgeweckt. Dann vollzieht er in der `run`-Methode einen weiteren Schleifendurchlauf und sendet das nächste Bild durch Auslösen des `newFrame`-Signals, worauf er sich wieder schlafen legt und sich das ganze wiederholt. Der Ablauf

Listing 5.7: FrameGrabber.cpp

```
29 void FrameGrabber::run(){
30   //initialize and configurre camera via USBGrabber
31   if(m_usbGrabber->init("/dev/video0") == 0){
32     m_usbGrabber->setSize(320, 240);
33     m_usbGrabber->setPalette(VIDEO_PALETTE_RGB32);
34     m_usbGrabber->printPictureParameters();
35     m_usbGrabber->printWindowParameters();
36     //loop where the images are captured
37     while(!m_stopExecution){
38       //grab one single image
39       unsigned char* frameData = m_usbGrabber->grabRawImage();
40       if(frameData == 0){
41         //an error occured during capture, so emit 'cameraError'-signal
42         emit cameraError();
43       }else{
44         //create a QImage out of the captured raw-image
45         QImage frame(320, 240, QImage::Format_RGB32);
46         memcpy(frame.bits(), frameData, frame.numBytes());
47         //deallocate memory of raw-image
48         free(frameData);
49         //lock mutex to avoid deadlocks
50         m_mutex.lock();
51         //send image by emitting the 'newFrame'-signal with the image as argument
52         emit newFrame(frame);
53         //check if execution should be stopped
54         if(m_stopExecution){
55           m_mutex.unlock();
56           return;
57         }
58         //send thread to sleep and wait for the next call of grabFrame() to wake
59         //up again. While in sleep-mode the mutex is unlocked, by the wait-
60         //condition.
61         m_waitCondition.wait(&m_mutex);
62         //unlock mutex again
63         m_mutex.unlock();
64       }
65     }
66   }else{
67     //an error occured during initialization, so emit 'cameraError'-signal
68     emit cameraError();
69   }
70 }
```

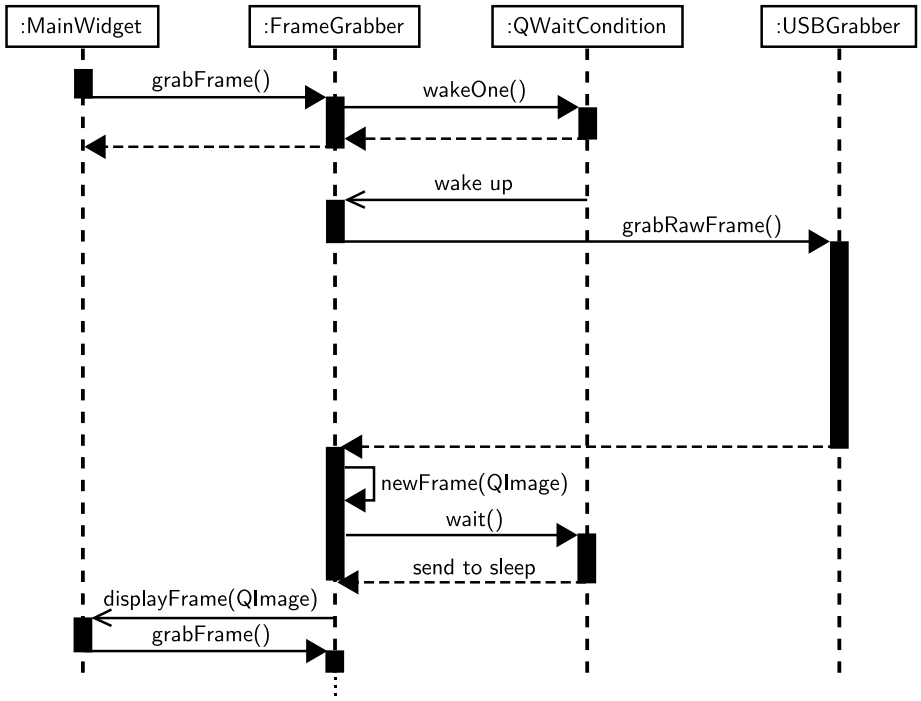


Bild 5.3: Sequenzdiagramm zum Ablauf der Bildübertragung

der sich wiederholenden Bildübertragung ist in dem zu UML 2.0[12][Amb05] konformen Sequenzdiagramm in Abbildung 5.3 dargestellt.

Die Anforderung der Sensordaten erfolgt nach dem gleichen Prinzip. Die zugehörigen Threads wurden in den Klassen `InfraRedDetector` und `UltraSonicDetector` implementiert. Der einzige Unterschied liegt in der erneuten Anforderung der Sensordaten. Während das nächste Kamerabild sofort nach dem Empfang des letzten Bildes angefordert wird, erfolgt die Anforderung der nächsten Sensordaten jeweils nach einer Pause von 200 ms. Deshalb wurden die zu `grabFrame` analogen Methoden dieser Klassen als Slots implementiert, wodurch sie mit der Methode `QTimer::singleShot` zeitversetzt ausgeführt werden können. Dadurch wird die Auslastung der Applikation deutlich reduziert.

In der Klasse `MainWidget` ist, wie bereits erwähnt, das Haupt-Widget implementiert. Im

[12] *http://www.omg.org/technology/documents/modeling_spec_catalog.htm#UML*

Tabelle 5.10: Shortcuts von `MiniBotGUI`

Taste	Funktion
↑	Vorwärtsfahrt
↓	Rückwärtsfahrt
←	Linksdrehung
→	Rechtsdrehung
`Ende`	Anhalten der Bewegung
`i`	Bewegungsmodus Umschalten
`+`	Geschwindigkeits-Level erhöhen
`-`	Geschwindigkeits-Level erniedrigen

Konstruktor wird der Inhalt dieses Haupt-Widgets durch Aufruf von `setupWidget()` erzeugt. Hier werden die Sub-Widgets angelegt und konfiguriert. Anschließend werden die Verbindungen zwischen Signalen und Slots durch die Methode `initConnections()` hergestellt. Nach erfolgreicher Initialisierung des `RoboControl`-Objekts, werden dann die ersten Sensordaten und Kamerabilder angefordert, wodurch die Informationsübermittlung zwischen den Threads in Gang gesetzt wird.

Die Applikation kann über diverse Tasten gesteuert werden. Tabelle 5.10 listet die zur Verfügung stehenden Shortcuts mit zugehöriger Funktion auf.

Kapitel 6

Zusammenfassung

Mit den beiden MiniBots stehen zwei mobile Roboter mit einer umfangreichen Ausstattung zur Verfügung. Das verwendete Barebone-Motherboard stellt ausreichend Rechenleistung bereit, um die von der angeschlossenen Kamera aufgenommenen Bilder in späteren Projekten verarbeiten zu können. Die angeschlossenen Sensoren bieten Informationen über die Beschaffenheit der direkten Umgebung des MiniBots. Weitere Sensoren können ohne großen Aufwand hinzugefügt werden, da genügend freie Ports und Schnittstellen existieren, um die Roboter zu erweitern.

Zur Ansteuerung des MiniBots wurde die Klasse `RoboControl` implementiert, die eine komfortable Steuerung des Roboters ermöglicht. Hierbei kann der MiniBot durch Aufruf von Funktionen gesteuert werden, ohne auf die technischen und hardware-spezifischen Besonderheiten achten zu müssen. Die Klasse `USBGrabber`, die ebenfalls eine leicht zu bedienende Schnittstelle bietet. übernimmt das Auslesen der Bilder. Darüberhinaus wird durch die Bereitstellung von `SpatialDomainImages` eine direkte Anbindung an PUMA zur Weiterverarbeitung möglich.

Alle verwendeten Klassen wurden in den zugehörigen Tools benutzt, wodurch, neben der Durchführung von Tests, auch die Anwendung der Klassen demonstriert wird. Die `MiniBotGUI`-Applikation stellt eine Zusammenfassung der zur Verfügung stehenden Steuersoftware dar. Sie ermöglicht die Steuerung des MiniBots und visualisiert Sensordaten und Kamerabilder in Echtzeit.

Als weitere Schritte können verschiedene Eigenschaften der Roboter noch verbessert werden: Die Implementierung eines Regelkreises auf dem Mikrocontroller, der die Geschwindigkeit beider Motoren auf dem gleichen Level hält, wäre wünschenswert. Darüberhinaus könnte die Federung der Roboter verbessert werden, da diese nur für das ursprüngliche Gewicht ausgelegt war, das im jetzigen Ausbauzustand jedoch deutlich überschritten wird. Um die mögliche Fahrzeit zu erhöhen, könnten die verwendeten Bleiakkus durch Hochleistungsakkus ersetzt werden. Außerdem wäre das Anbringen einer Abdeckung, um das noch freiliegende Barebone-Motherboard zu schützen, sinnvoll. Weitere Sensoren könnten eine umfassendere Hinderniserkennung ermöglichen und die Installation einer Funknetzwerkkarte wäre vorteilhaft, um den Roboter von einem Remote-Rechner aus steuern zu können. Hierdurch würde das lästige Anschließen von Monitor, Tastatur, etc. am Roboter entfallen.

Da wir ein sehr verbreitetes RC-Modell als Basis für unseren Roboter gewählt haben, sind auf verschiedenen Fanseiten und Foren zu diesem Modell Tipps und Tricks, z.B. zur Verbesserung des Fahrgestells oder Veränderung der Getriebeuntersetzung zu finden. Des weiteren wird in einigen Modellbaushops auch noch diverses Umbaumaterial angeboten. Die folgenden Links stellen eine kleine Auswahl dieser Seiten dar:

- *http://www.panzer-team.de/heng_long_tiger.htm*
- *http://www.razyboard.com/system/user_Heng_long_Panzer_Forum.html*
- *http://www.rc-mm.de/wbb2/hmportal.php*
- *http://www.axels-modellbau-shop.de/katalog/116-Militaer/Tiger-I-Tamiya-Heng-Long:::1_2.html*
- *http://www.tankzone.co.uk/cart/tz_henglong.htm*

Anhang A

Konstruktionszeichnungen

Alle Maße der Konstruktionszeichnungen sind in Millimeter gehalten. Die Bohrlöcher entsprechen den folgenden Größen:

(1) 2,6 mm

(2) 3,5 mm

(3) 4 mm

(4) 5 mm

(5) 6 mm

(6) 6,5 mm

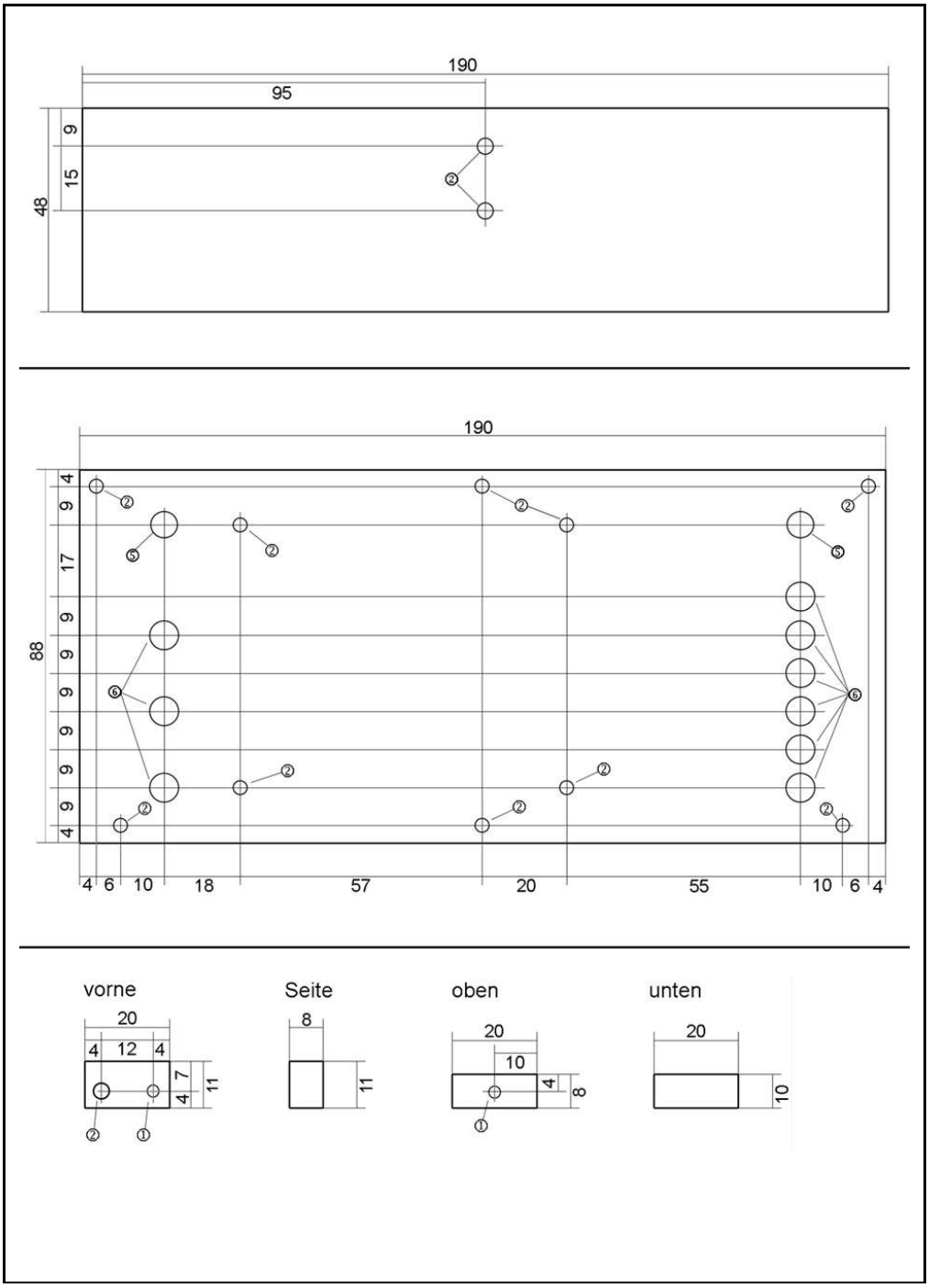
(7) 10 mm

(8) 11 mm

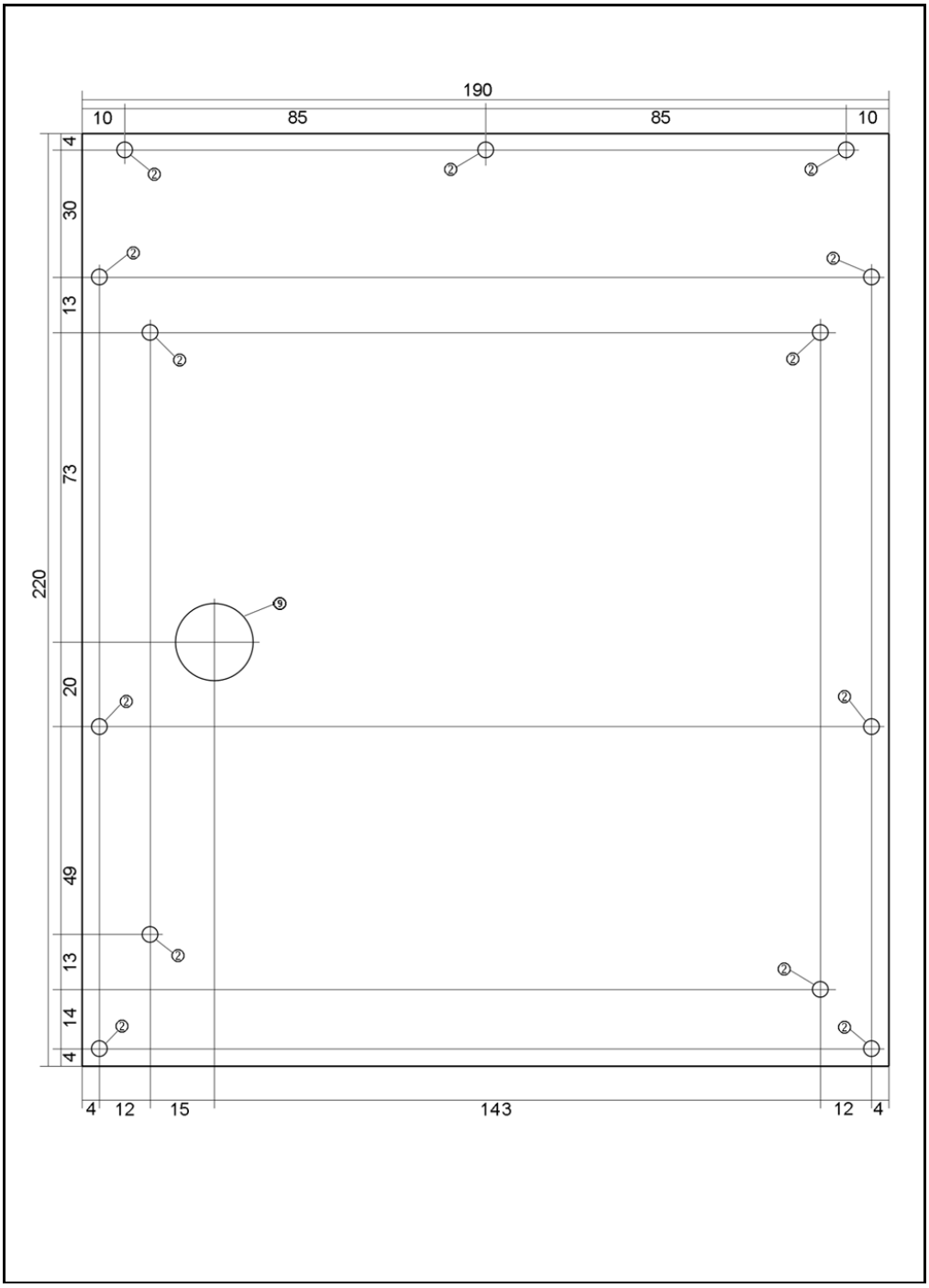
(9) 18 mm

(10) 35 mm

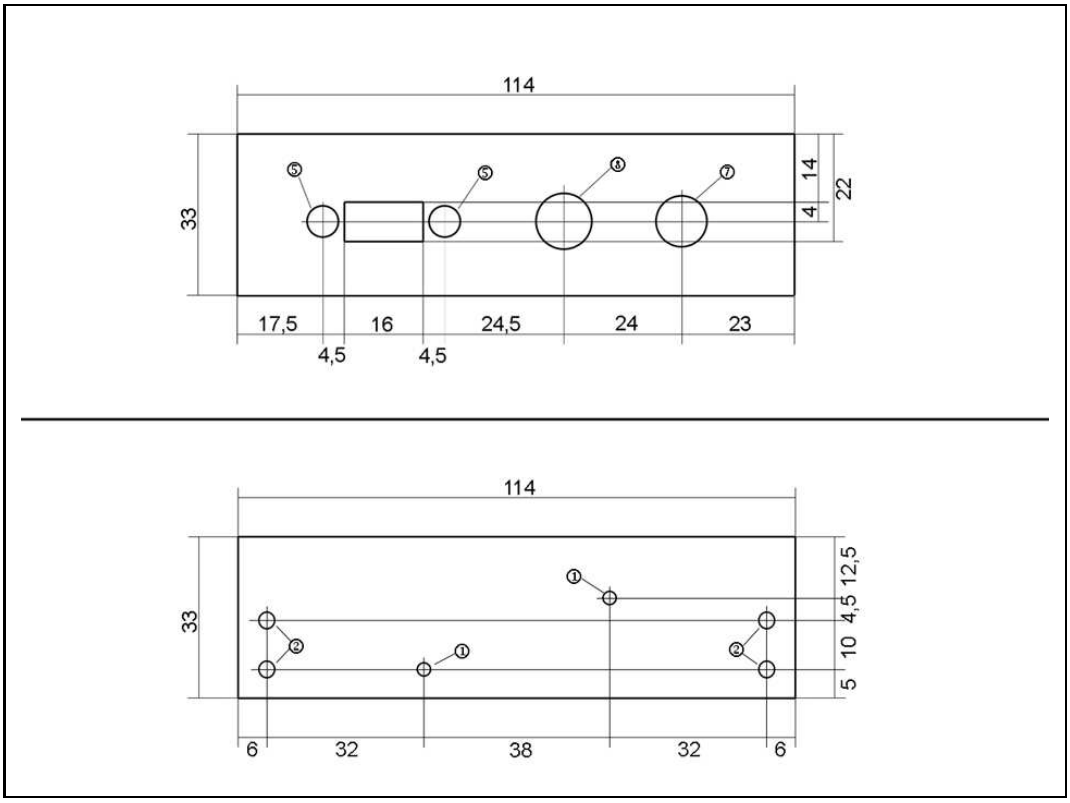
Vorderer (oben) und hinterer Deckel (mitte), Deckelhalter (unten)



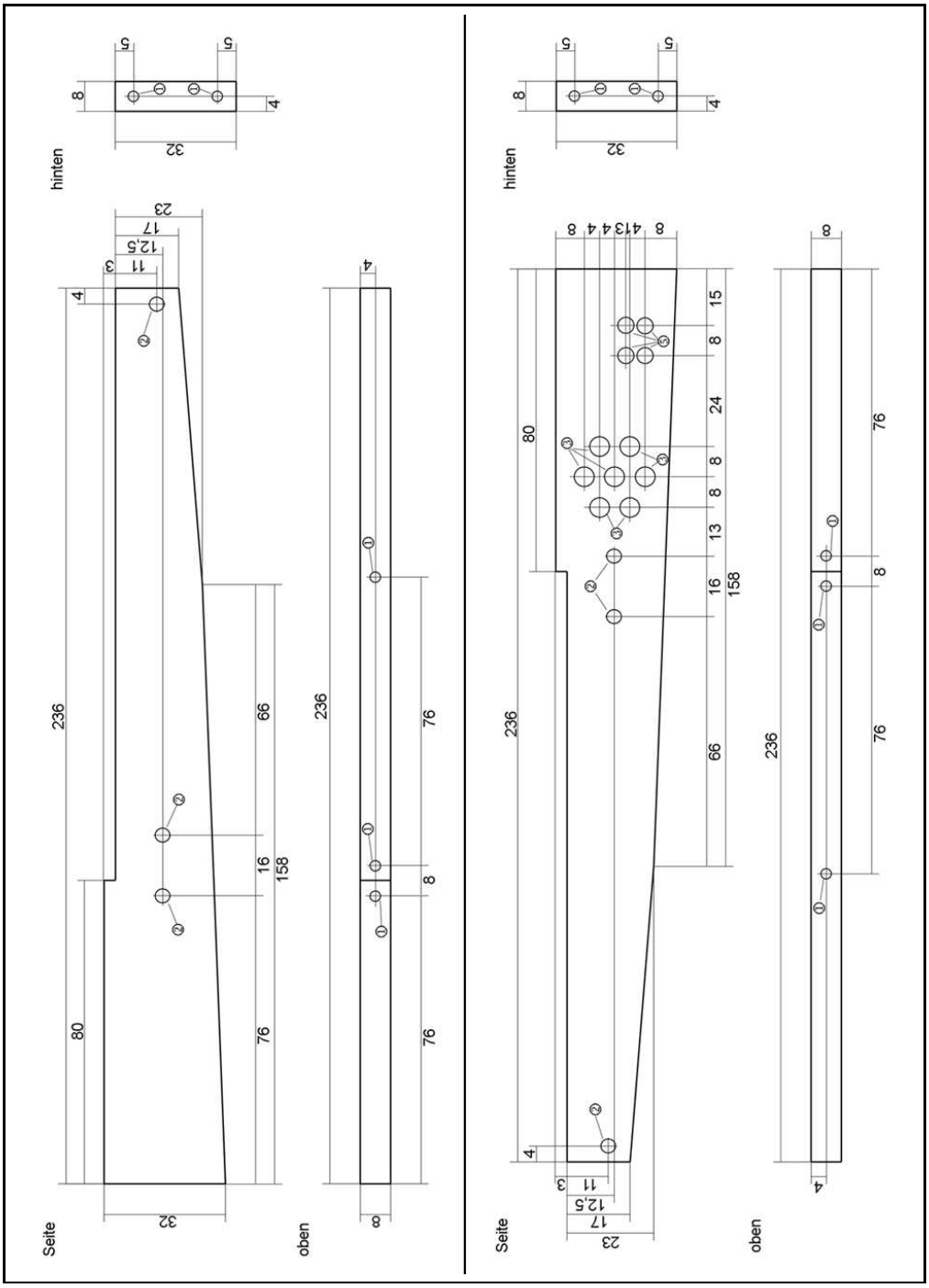
Mittlerer Deckel



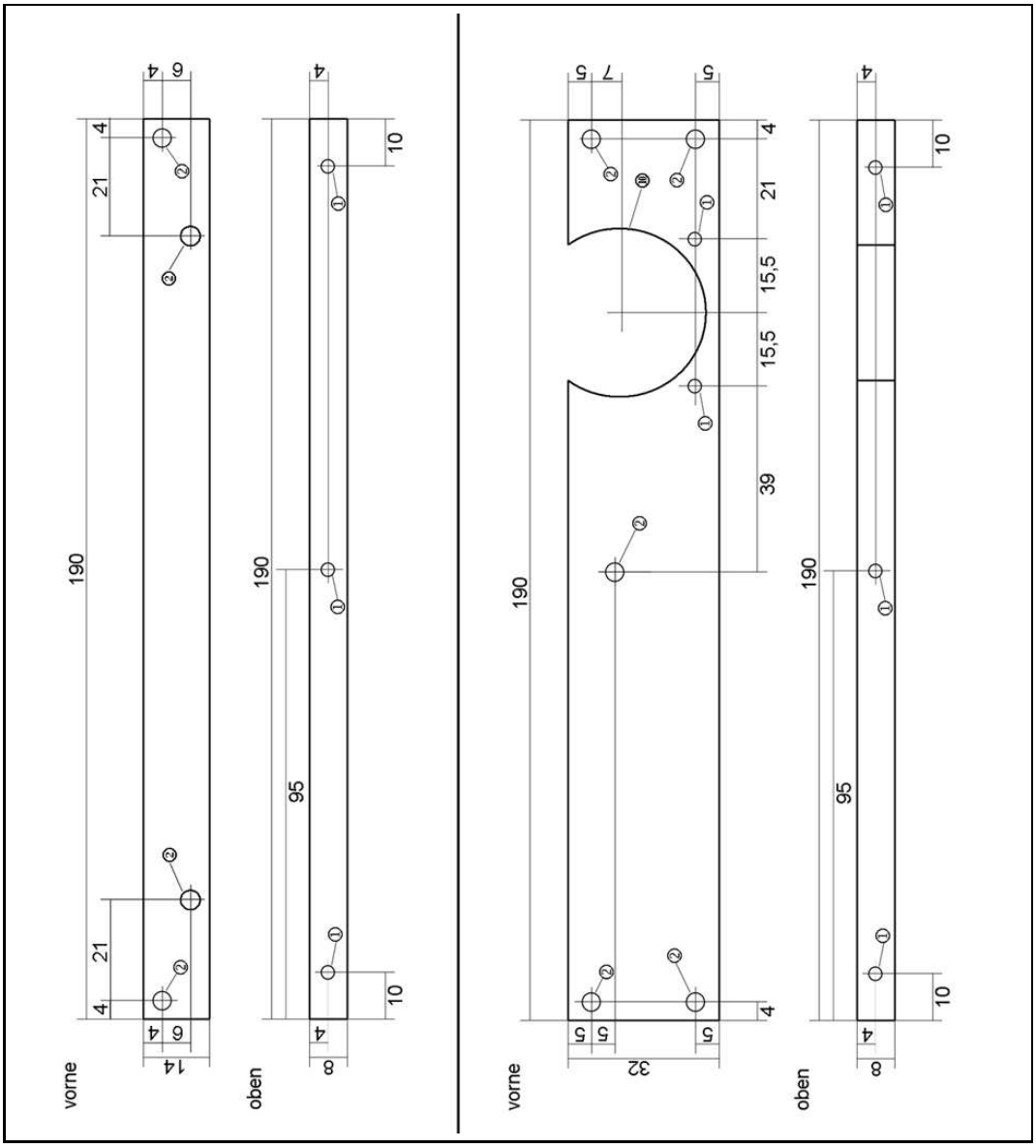
Untere Rückwand (oben), Ultraschallsensorhalter (unten)



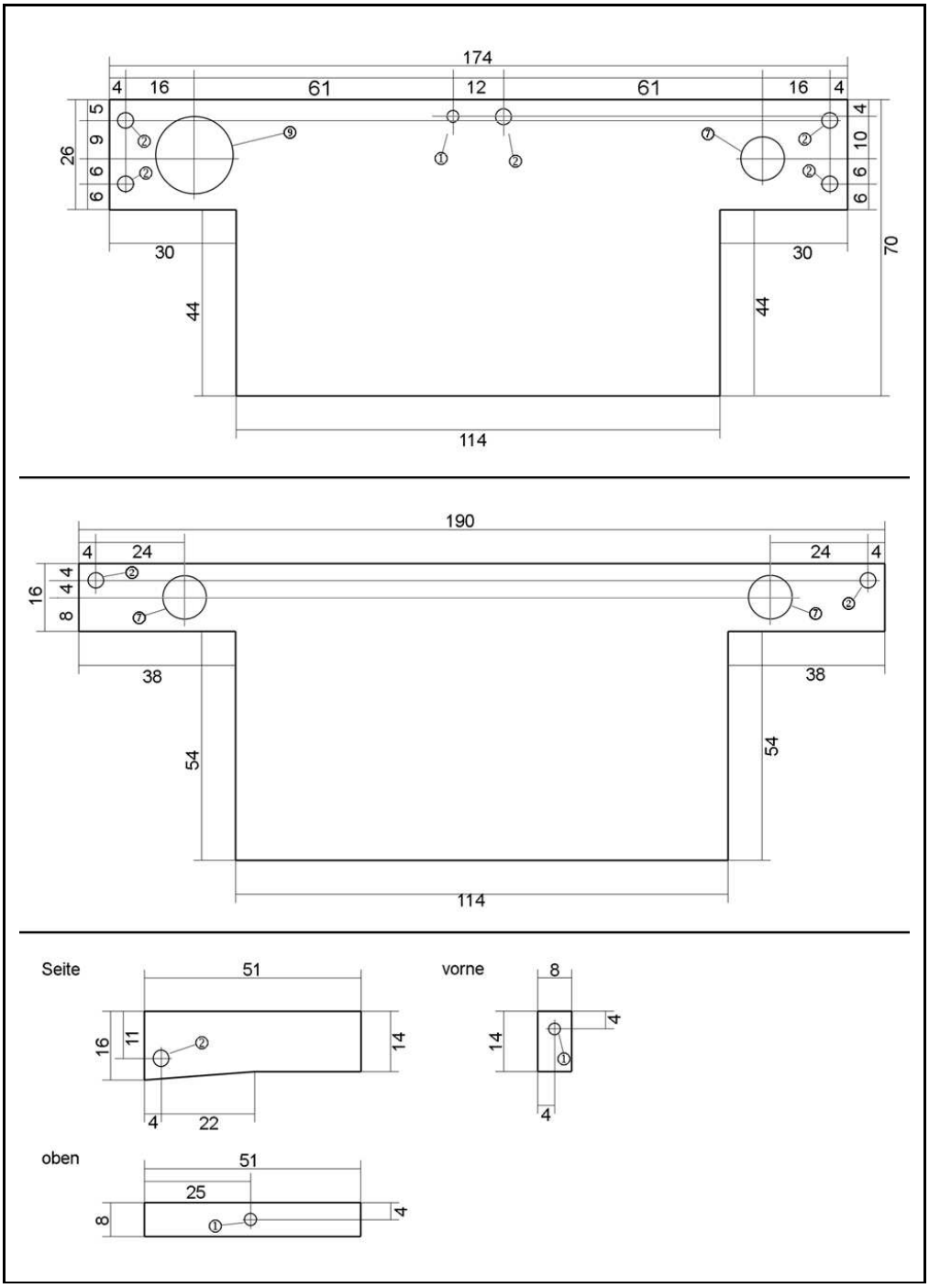
Rechtes (links) und linkes Seitenteil (rechts)



Vorderwand (links) und obere Rückwand (rechts)



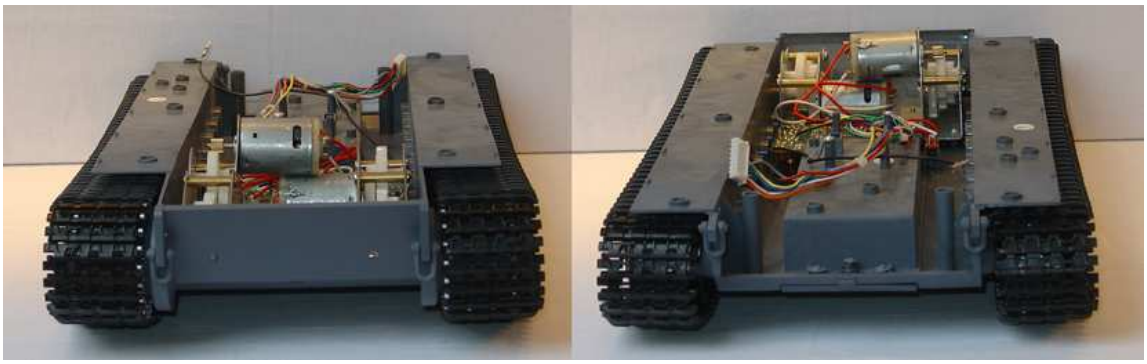
Hintere (oben) und vordere Zwischenwand (mitte), vordere Seitenwände (unten)



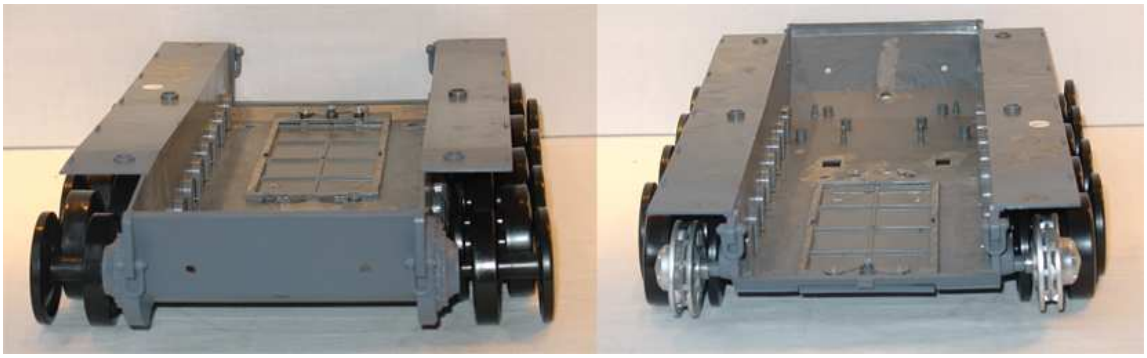
Anhang B

Ausgewählte Bilder des Aufbaus

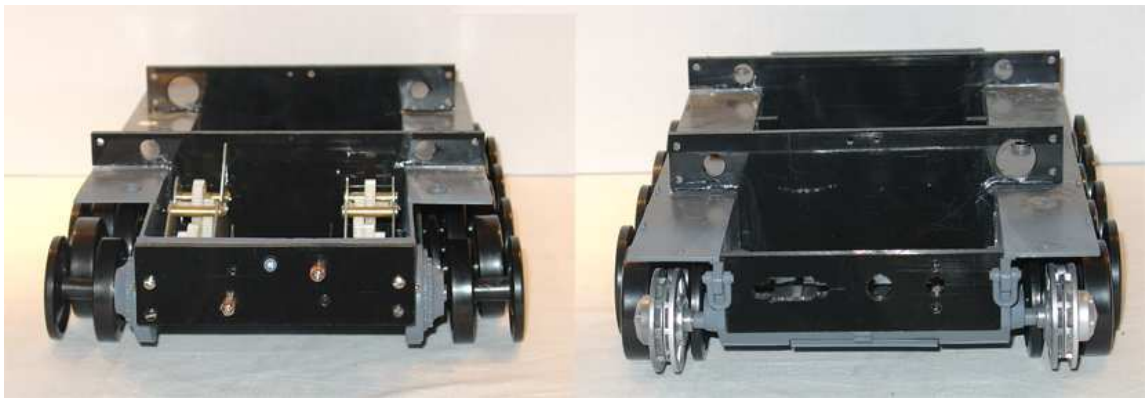
Hier sehen Sie eine kleine Bilddokumentation über den Aufbau des MiniBots:



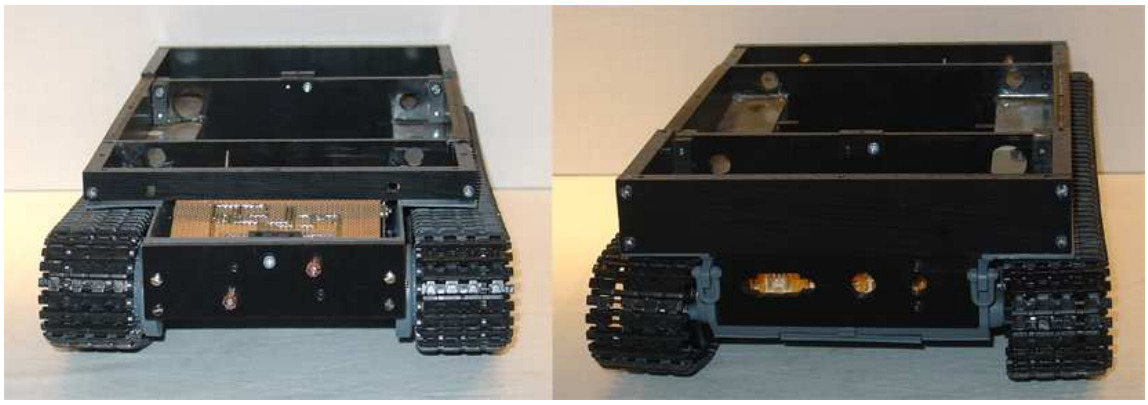
Bodenwanne des Panzermodells im Auslieferungszustand



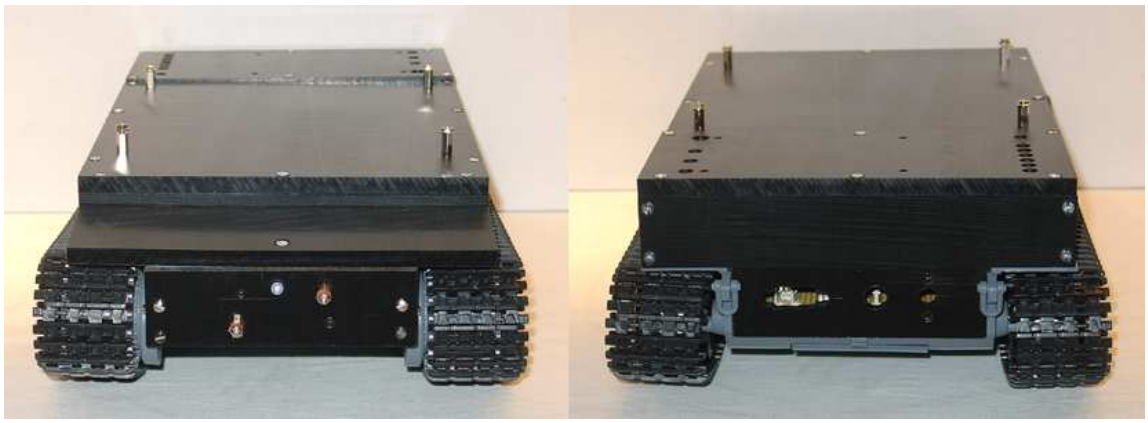
Leergeräumte Bodenwanne vor dem Neuaufbau



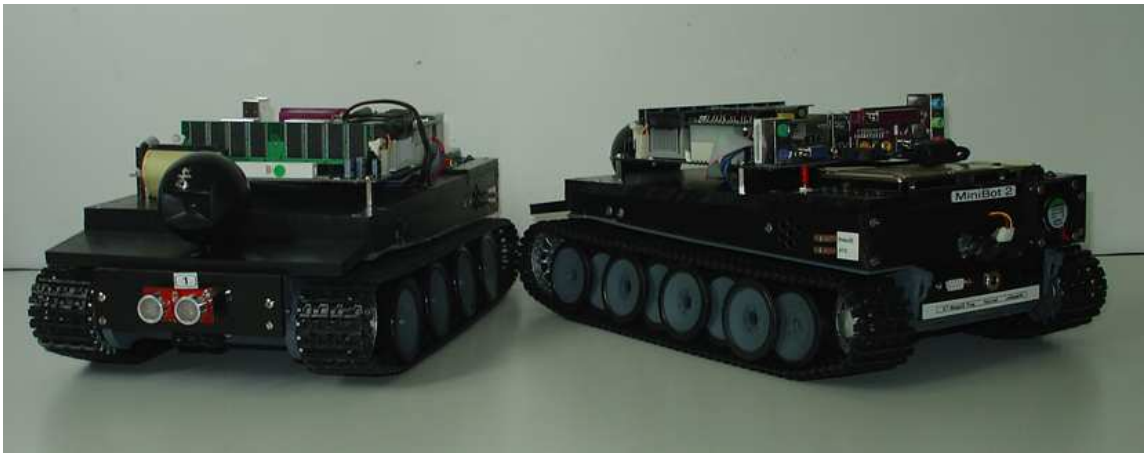
Bodenwanne nach Einbau der Zwischenwände, des Ultraschallsensorhalters und der unteren Rückwand



Hier sind bereits die Seitenwände, die Signalplatine und die Grundplatine verbaut



Dieses Bild zeigt die endgültige Karosserie des Roboters



Der fertige MiniBot

Weiter Bilder über den Aufbau sind auf der beiliegenden CD zu finden.

Anhang C

Materiallisten und Technische Daten

Materiallisten

In diesem Abschnitt werden alle Bauteile aufgelistet, die für den Bau eines Roboters benötigt werden.

EmbedIt Mikrocontrollertechnik

Bezugsadresse:

EmbedIt Mikrocontrollertechnik
Markus Burrer
Kiefernweg 7
36100 Petersberg
http://www.embedit.de

ANZAHL	BEZEICHNUNG	VERWENDUNG
1	Atmega32-16PU	Mikrocontroller
1	Robo32 Board	Mikrocontroller-Board

Reichelt

Bezugsadresse:
Reichelt Elektronik e.Kfr.
Elektronikring 1
26452 Sande
http://www.reichelt.de

Signalplatine

ANZAHL	BEZEICHNUNG	VERWENDUNG
2	1/4W 160	160 Ohm Widerstand
2	1/4W 560	560 Ohm Widerstand
2	CNY 36	Gabellichtschranken
1	74HC 14	Schmitt-Trigger
1	GS 14	14-pol. IC-Sockel

Programmieradapter

ANZAHL	BEZEICHNUNG	VERWENDUNG
2	1/4W 1,0K	1 kOhm Widerstand
2	1/4W 4,7K	4.7 kOhm Widerstand
1	1/4W 10K	10 kOhm Widerstand
1	1/4W 12K	12 kOhm Widerstand
2	ZD 5,1	5.1V Z-Diode
1	KERKO-500 560P	560pF Kondensator
1	BC547 C	Transistor
1	D-SUB BU 09	D-SUB-Buchse
1	PS 25/8G WS	8-pol. Steckverbinder
1	PRBL 10D	10-pol. Buchsenleiste
1	PFL 10	10-pol. Pfostenbuchse
1	LPV 10	10-pol. Pfostenstecker

Spannungsüberwachung

ANZAHL	BEZEICHNUNG	VERWENDUNG
3	1/4W 130	130 Ohm Widerstand
5	1/4W 360	360 Ohm Widerstand
1	1/4W 470	470 Ohm Widerstand
1	1/4W 3,6K	3.6 kOhm Widerstand
2	1/4W 4,7K	4.7 kOhm Widerstand
1	1/4W 6,2K	6.2 kOhm Widerstand
1	LM 324 D SMD	Operationsverstärker
1	GS 14	14-pol. IC-Sockel
2	LED 5MM ST RT	LED rot
1	LED 5MM ST GE	LED gelb
3	LED 5MM ST GN	LED grün
6	MONTAGERING 5MM	LED Fassung
1	PS 25/2G WS	2-pol. Steckverbinder
1	PRBL 10D	10-pol. Buchsenleiste
1	LPV 10	10-pol. Pfostenstecker

PC-Teile

ANZAHL	BEZEICHNUNG	VERWENDUNG
1	VIA EP M-10000	Mini-ITX Motherboard 1000MHz
1	DDR-PC333 512MB	Arbeitsspeicher
1	SAMSUNG MP0402H	Festplatte
1	LOGITECH QCC STX	Webcam
1	LED 5MM ST GN	LED grün
2	LED 5MM ST BL	LED blau
3	MONTAGERING 5MM	LED Fassung
1	T 113A GN	Taster grün
1	T 113A RT	Taster rot
1	ATA 44-40A	Festplattenadapter

Zusätzliches Material

ANZAHL	BEZEICHNUNG	VERWENDUNG
1	LCR-12V 7,2P-1	7,2V Blei-Gel Akku
1	HEBLM 25	Hohlstecker-Buchse
1	KARTENHALTER	Platinenhalter
1	H25PR050	Lochrasterplatine
1	H25PR100	Lochrasterplatine
1	AWG 28-10G 3M	10-pol. Flachbandkabel
5	PS 25/3G WS	3-pol. Steckverbinder
2	PS 25/2G WS	2-pol. Steckverbinder
1	MS 169	2-pol. Schalter
2	FK1 5A	Flachsicherungen
2	DA 10MM	Abstandshalter
2	DI 10MM	Abstandshalter
4	DI 25MM	Abstandshalter
4	DI 15MM	Abstandshalter
1	LÜFTER-4010 5V	Lüfter

Des weiteren wurden noch Löststifte, Steckschuhe, Silberdraht, Schrumpfschlauch und Kabel benötigt.

Micromaus-Systemtechnik

Bezugsadresse:

Micromaus-Systemtechnik
Feuerhausstraße 13
94356 Kirchroth
http://www.micromaus.de

ANZAHL	BEZEICHNUNG	VERWENDUNG
2	GP2D120	Infrarotsensor
2	Taktscheibe 120-Impulse	Radencoderscheiben

RC Model-Universe

Bezugsadresse:

RC Model-Universe
Hägewiesen 109
30657 Hannover
http://www.rc-model-universe.de

ANZAHL	BEZEICHNUNG	VERWENDUNG
1	Tiger-Panzer	Fahrgestell

Roboterteile

Bezugsadresse:

EDV-Beratung & Robotertechnik Jörg Pohl
Baluschekstr. 9
01159 Dresden
http://www.roboter-teile.de

ANZAHL	BEZEICHNUNG	VERWENDUNG
1	SRF08	Ultraschallsensor

Visual-Data

Bezugsadresse:

Visual-Data & MCOM Visual-Data GmbH
Holbeinstr.6
50733 Köln
http://www.visual-data.de

ANZAHL	BEZEICHNUNG	VERWENDUNG
1	PW120	Mainboardnetzteil

Hinweis

Alle Bestellformulare befinden sich auf der CD unter `orders/`. Die Bestellung bei Reichelt lief über das Webformular der AGAS, zu finden unter:
http://serres.uni-koblenz.de/aswiki/index.php/Bestellungen.

Technische Daten

In den folgenden Tabellen sehen Sie die technischen Daten der verwendeten Bauteile.

Sensoren

GP2D120

Bezeichnung:	GP2D120
Hersteller:	Sharp
Typ:	Infrarot-Sensor
Betriebsspannung:	5 V
Max. Stromaufnahme:	0,050 A
Abtastwinkel:	ca. 2°
Reichweite:	4cm - 30cm
Rückgabewert:	Analoges Spannungssignal
Weitere Informationen:	`datasheet/gp2d120.pdf`

SRF08

Bezeichnung:	SRF 08
Hersteller:	Devantech
Typ:	Ultraschall-Sensor
Betriebsspannung:	5 V
Max. Stromaufnahme:	0,015 A
Anschluss:	I^2C
Reichweite:	3cm - 6m
Abtastwinkel:	45°
Weitere Informationen:	`datasheet/srf08.pdf`

Motoren

Bezeichnung:	Speed 280 FG3
Hersteller:	Graupner
Typ:	Elektromotoren
Nennspannung:	6 V
Betriebsspannung::	4,5V - 7.2V
Max. Stromaufnahme:	1,58A
Laufrichtung:	R und L
Getriebe:	3:1
Weitere Informationen:	*http :// www.graupner.de*

Kamera

Bezeichnung:	QuickCam Communicate STX
Hersteller:	Logitech
Typ:	Webcam
Anschluss:	USB 2.0
Auflösung:	640 x 480
Zusatzausstattung:	Mikrofon
Weitere Informationen:	*http :// www.logitech.com*

PC

Mainboard:	VIA EPIA M10000
Prozessor:	VIA C3, 1000 MHz
Grafikkarte:	interne VIA Unichrome AGP
Festplatte:	40 GB
Arbeitsspeicher:	512 MB
externe Schnittstellen:	RJ45, 2*USB 2.0, Parallelport, RS232, RCA Port (SPDIF oder TV out), S-Video, VGA, 2* PS/2
interne Schnittstellen:	2*USB 2.0, 2*IEEE 1394, RS232, SMBUS(I^2C), FIR, CIR
Weitere Informationen:	*http :// www.via.com.tw/en/products/mainboards / mini_itx /epia_m*

Anhang D

Assemblerquellcode des Mikrocontrollers

Listing D.1: prog/MiniBot/MiniBot.asm

```
1  ;*****************************************************************************
2  ;*
3  ;*    MiniBot.asm
4  ;*
5  ;*****************************************************************************
6
7  .include "m32def.inc"
8  .equ CLOCK = 14745600
9  .include "Protocol.asm"
10 .include "Actions.asm"
11 .include "Registers.asm"
12
13 ;interrupt-vector
14 .org 0x000                          ;start of execution after reset
15   rjmp RESET
16 .org INT0addr                       ;interrupt-address for external interrupt 0
17   rjmp INT_0
18 .org INT1addr                       ;interrupt-address for external interrupt 1
19   rjmp INT_1
20 .org URXCaddr                       ;interrupt-address for URXC interrupt
21   rjmp INT_RS232
22
23 .org INT_VECTORS_SIZE
24
25 ;The reset routine is called on every reset of the microcontroller.
```

```
26 RESET:
27   ldi TEMPL, LOW(RAMEND)                    ;initialize stackpointer
28   ldi TEMPH, HIGH(RAMEND)
29   out SPL, TEMPL
30   out SPH, TEMPH
31
32   rcall SLEEP_INIT                          ;initialize sleep-mode
33   rcall LED_INIT                            ;initialize LEDs
34   rcall RS232_INIT                          ;initialize RS232-interface
35   rcall ADC_INIT                            ;initialize ADC-unit
36   rcall PWM_INIT                            ;initialize PWM-unit
37   rcall I2C_INIT                            ;initialize I2C-interface
38   rcall INT_INIT                            ;initialize external interrupts
39
40   ldi TEMP1, ACTION_NONE                    ;set initial state
41   mov ACTION_STATE, TEMP1
42   sei                                       ;enable interrupts
43   SLEEP_LOOP:
44     sleep                                   ;put mc in idle-mode
45     rjmp SLEEP_LOOP
46
47 .include "RS232.asm"
48 .include "ADC.asm"
49 .include "PWM.asm"
50 .include "I2C.asm"
51
52 ;This interrupt routine is executed when a byte arrives via RS232-interface
53 ;and interrupt URXC is triggered.
54 INT_RS232:
55   push TEMP1                                ;save temporary and status-registers
56   push TEMP2
57   push TEMP3
58   push TEMP4
59   push TEMPH
60   push TEMPL
61   in TEMP1, SREG
62   push TEMP1
63
64   cbi UCSRB, RXCIE                          ;deactivate RS232-interrupt
65   sei                                       ;reactivate other interrupts
66
67   in TEMP1, UDR                             ;read byte that arrived from RS232-interface
68   cpi TEMP1, REQUEST_TEST                   ;test what kind of request has been sent
69   breq BRANCH_TEST
70   cpi TEMP1, REQUEST_ADC
71   breq BRANCH_ADC
72   cpi TEMP1, REQUEST_PWM_SET_VELOCITY
73   breq BRANCH_PWM_SET_VELOCITY
```

```
74    cpi TEMP1, REQUEST_PWM_START_ACTION
75    breq BRANCH_PWM_START_ACTION
76    cpi TEMP1, REQUEST_PWM_START_ACTION_INTERVAL
77    breq BRANCH_PWM_START_ACTION_INTERVAL
78    cpi TEMP1, REQUEST_PWM_STOP_ACTION
79    breq BRANCH_PWM_STOP_ACTION
80    cpi TEMP1, REQUEST_I2C_START_MEASUREMENT
81    breq BRANCH_I2C_START_MEASUREMENT
82    cpi TEMP1, REQUEST_I2C_GET_DISTANCE
83    breq BRANCH_I2C_GET_DISTANCE
84    cpi TEMP1, REQUEST_GET_STATE
85    breq BRANCH_GET_STATE
86    ;...
87    rjmp BRANCH_UNKNOWN                    ;an unknown request has been sent
88    BRANCH_TEST:                           ;branches for the different requests
89       call FUNCTION_TEST
90       rjmp INT_RS232_END
91    BRANCH_ADC:
92       call FUNCTION_ADC
93       rjmp INT_RS232_END
94    BRANCH_PWM_SET_VELOCITY:
95       call FUNCTION_PWM_SET_VELOCITY
96       rjmp INT_RS232_END
97    BRANCH_PWM_START_ACTION:
98       call FUNCTION_PWM_START_ACTION
99       rjmp INT_RS232_END
100   BRANCH_PWM_START_ACTION_INTERVAL:
101      call FUNCTION_PWM_START_ACTION_INTERVAL
102      rjmp INT_RS232_END
103   BRANCH_PWM_STOP_ACTION:
104      call FUNCTION_PWM_STOP_ACTION
105      rjmp INT_RS232_END
106   BRANCH_I2C_START_MEASUREMENT:
107      call FUNCTION_I2C_START_MEASUREMENT
108      rjmp INT_RS232_END
109   BRANCH_I2C_GET_DISTANCE:
110      call FUNCTION_I2C_GET_DISTANCE
111      rjmp INT_RS232_END
112   BRANCH_GET_STATE:
113      call FUNCTION_GET_STATE
114      rjmp INT_RS232_END
115   ;...
116   BRANCH_UNKNOWN:
117      call FUNCTION_UNKNOWN
118   INT_RS232_END:
119
120   cli                                    ;disable all interrupts
121   sbi UCSRB, RXCIE                       ;reenable RS232-interrupt
```

```
122
123   pop TEMP1                                    ;restore temporary and status-registers
124   out SREG, TEMP1
125   pop TEMPL
126   pop TEMPH
127   pop TEMP4
128   pop TEMP3
129   pop TEMP2
130   pop TEMP1
131 reti
132
133 ;This interrupt routine is executed on every rising edge on pin D2
134 ;when external interrupt 0 is triggered.
135 INT_0:
136   push TEMP1                                   ;save temporary and status registers
137   in TEMP1, SREG
138   push TEMP1
139
140   sbis PORTB, PB7                              ;toggle state of LED on pin B7
141   rjmp INT_0_ON
142   cbi PORTB, PB7
143   rjmp INT_0_NEXT
144   INT_0_ON:
145   sbi PORTB, PB7
146   INT_0_NEXT:
147
148   sbiw INTERVAL_COUNTER_H:INTERVAL_COUNTER_L, 1 ;decrease interval
149   brne INT_0_END                               ;check if interval is zero
150   in TEMP1, DDRD                               ;stop engines
151   andi TEMP1, 0xCF
152   out DDRD, TEMP1
153   in TEMP1, GICR                               ;disable external interrupts
154   andi TEMP1, ~((1<<INT0)|(1<<INT1))
155   out GICR, TEMP1
156   ldi TEMP1, ACTION_NONE                       ;set action state
157   mov ACTION_STATE, TEMP1
158
159   INT_0_END:
160     pop TEMP1                                  ;restore temporary and status registers
161     out SREG, TEMP1
162     pop TEMP1
163 reti
164
165 ;This interrupt routine is executed on every rising edge on pin D3
166 ;when external interrupt 1 is triggered.
167 INT_1:
168   push TEMP1                                   ;save temporary and status registers
169   in TEMP1, SREG
```

```
170    push TEMP1
171
172    sbis PORTB, PB5                          ;toggle state of LED on pin B5
173    rjmp INT_1_ON
174    cbi PORTB, PB5
175    rjmp INT_1_NEXT
176    INT_1_ON:
177    sbi PORTB, PB5
178    INT_1_NEXT:
179
180    sbiw INTERVAL_COUNTER_H:INTERVAL_COUNTER_L, 1 ;decrease interval
181    brne INT_1_END                           ;check if interval is zero
182    in TEMP1, DDRD                           ;stop engines
183    andi TEMP1, 0xCF
184    out DDRD, TEMP1
185    in TEMP1, GICR                           ;disable external interrupts
186    andi TEMP1, ~((1<<INT0)|(1<<INT1))
187    out GICR, TEMP1
188    ldi TEMP1, ACTION_NONE                   ;set action state
189    mov ACTION_STATE, TEMP1
190
191    INT_1_END:
192      pop TEMP1                              ;restore temporary and status registers
193      out SREG, TEMP1
194      pop TEMP1
195  reti
196
197  ;This routine is for testing the RS232-interface when the client sends
198  ;REQUEST_TEST.
199  ;The byte that has been sent is increased and sent back to the client.
200  FUNCTION_TEST:
201    call RS232_RECEIVE_BYTE                  ;read the byte that has arrived
202    inc RS232_VALUE                          ;increase it
203    call RS232_SEND_BYTE                     ;send it back to the client
204  ret
205
206  ;This routine makes an A/D-conversion on the pins where the infrared-sensors
207  ;are attached and sends the digital values back to the client.
208  ;It is executed when the client sends REQUEST_ADC.
209  FUNCTION_ADC:
210    ldi TEMP1, RESPONSE_REQUEST_SUCCESSFUL   ;send byte to indicate success
211    mov RS232_VALUE, TEMP1
212    call RS232_SEND_BYTE
213    ldi TEMP1, 0x01                          ;load source of pin A1
214    mov ADC_SOURCE, TEMP1
215    call ADC_START                           ;do A/D-conversion on pin A1
216    mov RS232_VALUE, ADC_VALUE_H             ;send high-byte
217    call RS232_SEND_BYTE
```

```
218   mov RS232_VALUE, ADC_VALUE_L                ;send low-byte
219   call RS232_SEND_BYTE
220   ldi TEMP1, 0x03                             ;load source of pin A2
221   mov ADC_SOURCE, TEMP1
222   call ADC_START                              ;do A/D-conversion
223   mov RS232_VALUE, ADC_VALUE_H                ;send high-byte
224   call RS232_SEND_BYTE
225   mov RS232_VALUE, ADC_VALUE_L                ;send low-byte
226   call RS232_SEND_BYTE
227 ret
228
229 ;This routine is exexuted when the velocities of the engines should be
230 ;changed via REQUEST_PWM_SET_VELOCITY.
231 FUNCTION_PWM_SET_VELOCITY:
232   call RS232_RECEIVE_BYTE                     ;receive left velocity
233   mov PWM_LEFT_VELOCITY, RS232_VALUE          ;save left velocity
234   call RS232_RECEIVE_BYTE                     ;receive right velocity
235   mov PWM_RIGHT_VELOCITY, RS232_VALUE         ;save right velocity
236   rcall PWM_SET_VELOCITY                      ;set velocities
237   ldi TEMP1, RESPONSE_REQUEST_SUCCESSFUL      ;send byte to indicate success
238   mov RS232_VALUE, TEMP1
239   call RS232_SEND_BYTE
240 ret
241
242 ;This routine is executed when a permanent action should be started
243 ;via REQUEST_START_ACTION.
244 FUNCTION_PWM_START_ACTION:
245   in TEMP1, DDRD                              ;deactivate engines
246   andi TEMP1, 0xCF
247   out DDRD, TEMP1
248   call RS232_RECEIVE_BYTE                     ;receive action-code
249   mov TEMP1, RS232_VALUE
250   cpi TEMP1, FORWARD
251   breq PWM_START_ACTION_FORWARD               ;forward movement requested
252   cpi TEMP1, BACKWARD
253   breq PWM_START_ACTION_BACKWARD              ;backward movement requested
254   cpi TEMP1, LEFT
255   breq PWM_START_ACTION_LEFT                  ;left turn requested
256   cpi TEMP1, RIGHT
257   breq PWM_START_ACTION_RIGHT                 ;right turn requested
258   ldi TEMP1, RESPONSE_ACTION_UNKNOWN          ;unknown action received
259   mov RS232_VALUE, TEMP1                      ;send RESPONSE_ACTION_UNKNOWN to client
260   call RS232_SEND_BYTE
261   rjmp PWM_START_ACTION_END
262   PWM_START_ACTION_FORWARD:
263     call PWM_FORWARD                          ;do forward movement
264     rjmp PWM_START_ACTION_MOVING
265   PWM_START_ACTION_BACKWARD:
```

```
266     call PWM_BACKWARD                       ;do backward movement
267     rjmp PWM_START_ACTION_MOVING
268   PWM_START_ACTION_LEFT:
269     call PWM_LEFT                           ;do left turn
270     rjmp PWM_START_ACTION_TURNING
271   PWM_START_ACTION_RIGHT:
272     call PWM_RIGHT                          ;do right turn
273     rjmp PWM_START_ACTION_TURNING
274
275   PWM_START_ACTION_MOVING:                  ;set state to ACTION_MOVING
276     ldi TEMP1, ACTION_MOVING
277     mov ACTION_STATE, TEMP1
278     rjmp PWM_START_ACTION_SUCCESSFUL
279   PWM_START_ACTION_TURNING:                 ;set state to ACTION_TURNING
280     ldi TEMP1, ACTION_TURNING
281     mov ACTION_STATE, TEMP1
282
283   PWM_START_ACTION_SUCCESSFUL:
284   ldi TEMP1, RESPONSE_REQUEST_SUCCESSFUL    ;send RESPONSE_REQUEST_SUCCESSFUL
285   mov RS232_VALUE, TEMP1                    ;back to client
286   call RS232_SEND_BYTE
287   PWM_START_ACTION_END:
288   in TEMP1, DDRD                            ;reactivate engines
289   ori TEMP1, 0x30
290   out DDRD, TEMP1
291 ret
292
293 ;This routine is executed when an interval action should be started
294 ;via REQUEST_START_ACTION_INTERVAL.
295 FUNCTION_PWM_START_ACTION_INTERVAL:
296   in TEMP1, DDRD                                ;deactivate engines
297   andi TEMP1, 0xCF
298   out DDRD, TEMP1
299   call RS232_RECEIVE_BYTE                       ;receive action-code
300   mov TEMP1, RS232_VALUE
301   call RS232_RECEIVE_BYTE                       ;receive high-byte of interval
302   mov INTERVAL_COUNTER_H, RS232_VALUE           ;save high-byte of interval
303   call RS232_RECEIVE_BYTE                       ;receive low-byte of interval
304   mov INTERVAL_COUNTER_L, RS232_VALUE           ;save low-byte of interval
305   cpi TEMP1, FORWARD
306   breq PWM_START_ACTION_FORWARD_INTERVAL        ;forward movement requested
307   cpi TEMP1, BACKWARD
308   breq PWM_START_ACTION_BACKWARD_INTERVAL       ;backward movement requested
309   cpi TEMP1, LEFT
310   breq PWM_START_ACTION_LEFT_INTERVAL           ;left turn requested
311   cpi TEMP1, RIGHT
312   breq PWM_START_ACTION_RIGHT_INTERVAL          ;right turn requested
313   ldi TEMP1, RESPONSE_ACTION_UNKNOWN            ;unknown action received
```

```
314    mov RS232_VALUE, TEMP1                    ;send RESPONSE_ACTION_UNKNOWN to client
315    call RS232_SEND_BYTE
316    rjmp PWM_START_ACTION_INTERVAL_END
317    PWM_START_ACTION_FORWARD_INTERVAL:
318       call PWM_FORWARD                       ;do forward movement
319       rjmp PWM_START_ACTION_INTERVAL_MOVING
320    PWM_START_ACTION_BACKWARD_INTERVAL:
321       call PWM_BACKWARD                      ;do backward movement
322       rjmp PWM_START_ACTION_INTERVAL_MOVING
323    PWM_START_ACTION_LEFT_INTERVAL:
324       call PWM_LEFT                          ;do left turn
325       rjmp PWM_START_ACTION_INTERVAL_TURNING
326    PWM_START_ACTION_RIGHT_INTERVAL:
327       call PWM_RIGHT                         ;do right turn
328       rjmp PWM_START_ACTION_INTERVAL_TURNING
329
330    PWM_START_ACTION_INTERVAL_MOVING:         ;set state to ACTION_MOVING_INTERVAL
331       ldi TEMP1, ACTION_MOVING_INTERVAL
332       mov ACTION_STATE, TEMP1
333       rjmp PWM_START_ACTION_INTERVAL_SUCCESSFUL
334    PWM_START_ACTION_INTERVAL_TURNING:        ;set state to ACTION_TURNING_INTERVAL
335       ldi TEMP1, ACTION_TURNING_INTERVAL
336       mov ACTION_STATE, TEMP1
337
338    PWM_START_ACTION_INTERVAL_SUCCESSFUL:     ;send RESPONSE_REQUEST_SUCCESSFUL
339    ldi TEMP1, RESPONSE_REQUEST_SUCCESSFUL    ;back to client
340    mov RS232_VALUE, TEMP1
341    call RS232_SEND_BYTE
342    PWM_START_ACTION_INTERVAL_END:
343    in TEMP1, DDRD                            ;reactivate engines
344    ori TEMP1, 0x30
345    out DDRD, TEMP1
346    in TEMP1, GICR                            ;enable external interrupts
347    ori TEMP1, (1<<INT0)|(1<<INT1)
348    out GICR, TEMP1
349 ret
350
351 ;This routine is executed when the robot should be stopped
352 ;via REQUEST_PWM_STOP_ACTION.
353 FUNCTION_PWM_STOP_ACTION:
354    in TEMP1, DDRD                            ;set data direction to input
355    andi TEMP1, 0xCF                          ;to stop the robot
356    out DDRD, TEMP1
357    ldi TEMP1, RESPONSE_REQUEST_SUCCESSFUL    ;send RESPONSE_REQUEST_SUCCESSFUL
358    mov RS232_VALUE, TEMP1                    ;back to client
359    call RS232_SEND_BYTE
360    PWM_STOP_ACTION_END:
361    ldi TEMP1, ACTION_NONE
```

```
362    mov ACTION_STATE, TEMP1
363    in TEMP1, GICR                                  ;disable external interrupts
364    andi TEMP1, ~((1<<INT0)|(1<<INT1))
365    out GICR, TEMP1
366  ret
367
368  ;This routine is executed when the client requests for the robots
369  ;action state via REQUEST_GET_STATE.
370  FUNCTION_GET_STATE:
371    ldi TEMP1, RESPONSE_REQUEST_SUCCESSFUL        ;send RESPONSE_REQUEST_SUCCESSFUL
372    mov RS232_VALUE, TEMP1                        ;back to client
373    call RS232_SEND_BYTE
374    mov RS232_VALUE, ACTION_STATE                 ;send action state to client
375    call RS232_SEND_BYTE
376  ret
377
378  ;This routine is executed when the client wants to initiate a distance-
379  ;measurement of the ultrasonic-sensor via REQUEST_I2C_START_MEASUREMENT.
380  FUNCTION_I2C_START_MEASUREMENT:
381    ldi TEMP1, 0xE0                               ;address of device
382    ldi TEMP2, 0x00                               ;address of command-register
383    ldi TEMP3, 0x51                               ;command-code for new measurement
384    mov I2C_ADDRESS, TEMP1                        ;load address of device
385    mov I2C_REGISTER, TEMP2                       ;load address of register
386    mov I2C_VALUE, TEMP3                          ;load command-code
387    rcall I2C_STORE_BYTE                          ;initiate new measurement
388    mov TEMP1, I2C_ERRORCODE
389    cpi TEMP1, I2C_NO_ERROR                       ;check if an error has occured
390    brne FUNCTION_I2C_START_MEASUREMENT_ERROR
391    ldi TEMP1, RESPONSE_REQUEST_SUCCESSFUL        ;no error - send RESPONSE_REQUEST_SUCCESSFUL
392    mov RS232_VALUE, TEMP1                        ;back to client
393    rcall RS232_SEND_BYTE
394    rjmp FUNCTION_I2C_START_MEASUREMENT_END
395
396    FUNCTION_I2C_START_MEASUREMENT_ERROR:         ;an error has occured
397      ldi TEMP1, RESPONSE_I2C_ERROR               ;send RESPONSE_I2C_ERROR to client
398      mov RS232_VALUE, TEMP1
399      rcall RS232_SEND_BYTE
400      mov RS232_VALUE, I2C_ERRORCODE              ;send errorcode to client
401      rcall RS232_SEND_BYTE
402
403    FUNCTION_I2C_START_MEASUREMENT_END:
404  ret
405
406  ;This routine is executed when the client wants to get the distance that has
407  ;been measured by the ultrasonic-sensor via REQUEST_I2C_GET_DISTANCE.
408  FUNCTION_I2C_GET_DISTANCE:
409    ldi TEMP1, 0xE0                             ;address of device
```

```
410   ldi TEMP2, 0x02                          ;address of first echo-register-pair
411   mov I2C_ADDRESS, TEMP1                   ;load address of device
412   mov I2C_REGISTER, TEMP2                  ;load address of register
413   rcall I2C_LOAD_WORD                      ;read word out of first echo-register-pair
414   mov TEMP1, I2C_ERRORCODE
415   cpi TEMP1, I2C_NO_ERROR                  ;check if an error has occured
416   brne FUNCTION_I2C_GET_DISTANCE_ERROR
417   ldi TEMP1, RESPONSE_REQUEST_SUCCESSFUL   ;no error - send RESPONSE_REQUEST_SUCCESSFUL
418   mov RS232_VALUE, TEMP1                   ;back to client
419   rcall RS232_SEND_BYTE                    ;send high-byte of measured distance
420   mov RS232_VALUE, I2C_VALUE_H
421   rcall RS232_SEND_BYTE                    ;send low-byte of measured distance
422   mov RS232_VALUE, I2C_VALUE_L
423   rcall RS232_SEND_BYTE
424   rjmp FUNCTION_I2C_GET_DISTANCE_END
425
426   FUNCTION_I2C_GET_DISTANCE_ERROR:         ;an error has occured
427     ldi TEMP1, RESPONSE_I2C_ERROR          ;send RESPONSE_I2C_ERROR to client
428     mov RS232_VALUE, TEMP1
429     rcall RS232_SEND_BYTE
430     mov RS232_VALUE, I2C_ERRORCODE         ;send errorcode to client
431     rcall RS232_SEND_BYTE
432
433   FUNCTION_I2C_GET_DISTANCE_END:
434 ret
435
436 ;This routine is executed when an unknown request arrives from the client.
437 FUNCTION_UNKNOWN:
438   ldi TEMP1, RESPONSE_REQUEST_UNKNOWN   ;send RESPONSE_REQUEST_UNKNOWN back
439   mov RS232_VALUE, TEMP1                ;to client
440   call RS232_SEND_BYTE
441 ret
442
443 ;This routine initializes the sleep-mode of the microcontroller
444 SLEEP_INIT:
445   in TEMP1, MCUCR
446   andi TEMP1, 0x0F                      ;delete sleep-configuration-bits
447   ori TEMP1, (1<<SE)|(0<<SM0)           ;enable sleep-mode an set it to idle
448   out MCUCR, TEMP1
449 ret
450
451 ;This routine sets the Pins where the LEDs are attached to output
452 LED_INIT:
453   sbi DDRB, 7
454   sbi DDRB, 6
455   sbi DDRB, 5
456 ret
457
```

```
458 ;This routine initializes the external interrupts
459 INT_INIT:
460   cbi DDRD, PD2                    ;set the pins of the external interrupts
461   cbi DDRD, PD3                    ;to input
462   in TEMP1, MCUCR
463   ori TEMP1, (3<<ISC10)|(3<<ISC00)  ;configure external interrupts for initiation
464   out MCUCR, TEMP1                 ;on every rising edge
465 ret
```

Listing D.2: prog/MiniBot/Registers.asm

```
1  ;***************************************************************************************
2  ;*
3  ;*    Registers.asm
4  ;*
5  ;***************************************************************************************
6
7  ;temporary registers
8  .def TEMP1 = r16
9  .def TEMP2 = r17
10 .def TEMP3 = r18
11 .def TEMP4 = r19
12 .def TEMPL = r24
13 .def TEMPH = r25
14
15 ;RS232-register
16 .def RS232_VALUE = r11
17
18 ;ADC-registers
19 .def ADC_SOURCE = r8
20 .def ADC_VALUE_L = r9
21 .def ADC_VALUE_H = r10
22
23 ;PWM-registers
24 .def PWM_LEFT_VELOCITY = r0
25 .def PWM_RIGHT_VELOCITY = r1
26
27 ;I2C-registers
28 .def I2C_ADDRESS = r3
29 .def I2C_REGISTER = r4
30 .def I2C_VALUE = r5
31 .def I2C_VALUE_L = r6
32 .def I2C_VALUE_H = r7
33 .def I2C_ERRORCODE = r2
34
35 ;action-state register
36 .def ACTION_STATE = r12
37
```

```
38 ;interval-counter registers
39 .def INTERVAL_COUNTER_L = r26
40 .def INTERVAL_COUNTER_H = r27
```

Listing D.3: prog/MiniBot/Actions.asm

```
1  ;***************************************************************************
2  ;*
3  ;*    Actions.asm
4  ;*
5  ;***************************************************************************
6
7  ;action-codes
8  .equ FORWARD = 0x00
9  .equ BACKWARD = 0x01
10 .equ LEFT = 0x02
11 .equ RIGHT = 0x03
12
13 ;action-states
14 .equ ACTION_NONE = 0x00
15 .equ ACTION_MOVING = 0x01
16 .equ ACTION_MOVING_INTERVAL = 0x02
17 .equ ACTION_TURNING = 0x03
18 .equ ACTION_TURNING_INTERVAL = 0x04
```

Listing D.4: prog/MiniBot/Protocol.asm

```
1  ;***************************************************************************
2  ;*
3  ;*    Protocol.asm
4  ;*
5  ;***************************************************************************
6
7  ;protocol-request-codes
8  .equ REQUEST_TEST = 0x01
9  .equ REQUEST_ADC = 0x02
10 .equ REQUEST_PWM_SET_VELOCITY = 0x03
11 .equ REQUEST_PWM_START_ACTION = 0x04
12 .equ REQUEST_PWM_START_ACTION_INTERVAL = 0x05
13 .equ REQUEST_PWM_STOP_ACTION = 0x06
14 .equ REQUEST_I2C_START_MEASUREMENT = 0x07
15 .equ REQUEST_I2C_GET_DISTANCE = 0x08
16 .equ REQUEST_GET_STATE = 0x09
17
18 ;protocol-response-codes
19 .equ RESPONSE_REQUEST_SUCCESSFUL = 0x01
20 .equ RESPONSE_REQUEST_UNKNOWN = 0x02
21 .equ RESPONSE_ACTION_UNKNOWN = 0x03
```

```
22 .equ RESPONSE_I2C_ERROR = 0x04
```

Listing D.5: prog/MiniBot/RS232.asm

```
1  ;***************************************************************************************
2  ;*
3  ;*    RS232.asm
4  ;*
5  ;***************************************************************************************
6
7  ;RS232-settings
8  .equ RS232_BAUD = 9600
9  .equ RS232_UBRRVAL = CLOCK/(RS232_BAUD*16)-1
10
11 ;This routine initializes the RS232-interface.
12 RS232_INIT:
13   ldi TEMP1, LOW(RS232_UBRRVAL)       ;set RS232-baudrate
14   out UBRRL, TEMP1
15   ldi TEMP1, HIGH(RS232_UBRRVAL)
16   out UBRRH, TEMP1
17   ldi TEMP1, (1<<URSEL)|(3<<UCSZ0)    ;set frame-format 8-bit
18   out UCSRC, TEMP1
19   sbi UCSRB, RXCIE                    ;enable RS232-Interrupt
20   sbi UCSRB, RXEN                     ;enable RX
21   sbi UCSRB, TXEN                     ;enable TX
22 ret
23
24 ;This routine receives a byte on the RS232-interface and stores
25 ;it in RS232_VALUE.
26 RS232_RECEIVE_BYTE:
27   sbis UCSRA, RXC                     ;wait for byte
28   rjmp RS232_RECEIVE_BYTE
29   in RS232_VALUE, UDR                 ;store byte in RS232_VALUE
30 ret
31
32 ;This routine sends the byte in RS232_VALUE on the RS232-interface.
33 RS232_SEND_BYTE:
34   sbis UCSRA, UDRE                    ;wait until interface is ready
35   rjmp RS232_SEND_BYTE
36   out UDR, RS232_VALUE                ;send byte
37 ret
```

Listing D.6: prog/MiniBot/PWM.asm

```
1  ;***************************************************************************************
2  ;*
3  ;*    PWM.asm
4  ;*
```

```
5  ;***************************************************************************
6
7  ;This routine initializes Timer unit 1.
8  PWM_INIT:
9     ldi TEMP1, 0x00         ;clear output control registers
10    out OCR1AH, TEMP1
11    out OCR1AL, TEMP1
12    out OCR1BH, TEMP1
13    out OCR1BL, TEMP1
14    ldi TEMP1, 0b10100010   ;configure PWM
15    out TCCR1A, TEMP1
16    ldi TEMP1, 0b00011011
17    out TCCR1B, TEMP1
18    ldi TEMPH, 0x00         ;configure timer resolution
19    ldi TEMPL, 0xFF
20    out ICR1H, TEMPH
21    out ICR1L, TEMPL
22    sbi DDRB, PB0           ;set data direction to output
23    sbi DDRB, PB1
24    sbi DDRD, PD6
25    sbi DDRD, PD7
26 ret
27
28 ;This routine sets left velocity to PWM_LEFT_VELOCITY and
29 ;right velocity to PWM_RIGHT_VELOCITY.
30 PWM_SET_VELOCITY:
31    out OCR1AL, PWM_LEFT_VELOCITY
32    out OCR1BL, PWM_RIGHT_VELOCITY
33 ret
34
35 ;This routine sets the control bits for moving forward.
36 PWM_FORWARD:
37    sbi PORTB, PB0
38    cbi PORTB, PB1
39    sbi PORTD, PB6
40    cbi PORTD, PB7
41 ret
42
43 ;This routine sets the control bits for moving backward.
44 PWM_BACKWARD:
45    cbi PORTB, PB0
46    sbi PORTB, PB1
47    cbi PORTD, PB6
48    sbi PORTD, PB7
49 ret
50
51 ;This routine sets the control bits for turning left.
52 PWM_LEFT:
```

```
53    sbi PORTB, PB0
54    cbi PORTB, PB1
55    cbi PORTD, PB6
56    sbi PORTD, PB7
57  ret
58
59  ;This routine sets the control bits for turning right.
60  PWM_RIGHT:
61    cbi PORTB, PB0
62    sbi PORTB, PB1
63    sbi PORTD, PB6
64    cbi PORTD, PB7
65  ret
```

Listing D.7: prog/MiniBot/I2C.asm

```
1   ;*****************************************************************************
2   ;*
3   ;*    i2c.asm
4   ;*
5   ;*****************************************************************************
6
7   ;I2C-frequence = 10.1 kHz:
8   .equ I2C_BITRATE = 45
9   .equ I2C_PRESCALER = 2
10
11  ;I2C-Codes:
12  .equ I2C_START_SUCCESS = 0x08
13  .equ I2C_REPEATED_START_SUCCESS = 0x10
14  .equ I2C_WRITE_SLAW_SUCCESS = 0x18
15  .equ I2C_WRITE_SLAR_SUCCESS = 0x40
16  .equ I2C_WRITE_SUCCESS = 0x28
17  .equ I2C_READ_SUCCESS = 0x50
18  .equ I2C_READLAST_SUCCESS = 0x58
19  .equ I2C_NO_ERROR = 0x00
20
21  ;This routine initializes the I2C-unit.
22  I2C_INIT:
23    ldi TEMP1, I2C_BITRATE              ;set bitrate
24    out TWBR, TEMP1
25    ldi TEMP1, I2C_PRESCALER            ;set prescaler
26    out TWSR, TEMP1
27  ret
28
29  ;This routine stores the value of I2C_VALUE in register
30  ;I2C_REGISTER of device at address I2C_ADDRESS.
31  I2C_STORE_BYTE:
32    rcall I2C_START             ;send start condition
```

```
33    rcall I2C_WRITE_SLAW          ;configure device for write access
34    out TWDR, I2C_REGISTER        ;load register-address
35    rcall I2C_WRITE               ;send register-address
36    out TWDR, I2C_VALUE           ;load value
37    rcall I2C_WRITE               ;send value
38    rcall I2C_STOP                ;send stop condition
39    ldi TEMP1, I2C_NO_ERROR       ;load I2C_NO_ERROR in I2C_ERRORCODE
40    mov I2C_ERRORCODE, TEMP1      ;to indicate that no error occured
41  ret
42
43  ;This routine loads the byte from register I2C_REGISTER
44  ;of the device at address I2C_ADDRESS into I2C_VALUE.
45  I2C_LOAD_BYTE:
46    rcall I2C_START               ;send start condition
47    rcall I2C_WRITE_SLAW          ;configure device for write access
48    out TWDR, I2C_REGISTER        ;load register-address
49    rcall I2C_WRITE               ;send register-address
50    rcall I2C_REPEATED_START      ;send repeated start condition
51    rcall I2C_WRITE_SLAR          ;configure device for read access
52    rcall I2C_READLAST            ;load value without acknowledge
53    in I2C_VALUE, TWDR            ;save value in I2C_VALUE
54    rcall I2C_STOP                ;send stop condition
55    ldi TEMP1, I2C_NO_ERROR       ;load I2C_NO_ERROR in I2C_ERRORCODE
56    mov I2C_ERRORCODE, TEMP1      ;to indicate that no error occured
57  ret
58
59  ;This routine loads the word from register I2C_REGISTER
60  ;of the device at address I2C_ADDRESS into registerpair
61  ;I2C_VALUEH:I2C_VALUEL.
62  I2C_LOAD_WORD:
63    rcall I2C_START               ;send start condition
64    rcall I2C_WRITE_SLAW          ;configure device for write access
65    out TWDR, I2C_REGISTER        ;load register address
66    rcall I2C_WRITE               ;send register address
67    rcall I2C_REPEATED_START      ;send repeated start condition
68    rcall I2C_WRITE_SLAR          ;configure device for read access
69    rcall I2C_READ                ;load value with acknowledge
70    in I2C_VALUE_H, TWDR          ;save value I2C_VALUE_H
71    rcall I2C_READLAST            ;load value without acknowledge
72    in I2C_VALUE_L, TWDR          ;save value I2C_VALUE_L
73    rcall I2C_STOP                ;send stop condition
74    ldi TEMP1, I2C_NO_ERROR       ;load I2C_NO_ERROR in I2C_ERRORCODE
75    mov I2C_ERRORCODE, TEMP1      ;to indicate that no error occured
76  ret
77
78  ;This routine sends a start condition on I2C.
79  I2C_START:
80    ldi TEMP1, (1<<TWINT)|(1<<TWSTA)|(1<<TWEN) ;set bits in control register
```

```
81    out TWCR, TEMP1
82    WAIT_I2C_START_SENT:                    ;wait for completion
83      in TEMP1, TWCR
84      sbrs TEMP1, TWINT                     ;interrupt-flag indicates completion
85      rjmp WAIT_I2C_START_SENT              ;not yet completed
86    in TEMP1, TWSR                          ;load status register and clear the
87    andi TEMP1, 0xF8                        ;prescaler-configuration-bits
88    ldi TEMP2, I2C_START_SUCCESS            ;compare value of status register
89    cpse TEMP1, TEMP2                       ;with estimated value I2C_START_SUCCESS
90    rjmp I2C_ERROR                          ;an error occured
91  ret
92
93  ;The following routines have the same structure like I2C_START, except the bits
94  ;that are set in the control register and the value to which the status register is
95  ;compared to for finding out if an error has occured.
96
97  ;This routine sends a repeated start condition on I2C.
98  I2C_REPEATED_START:
99    ldi TEMP1, (1<<TWINT)|(1<<TWSTA)|(1<<TWEN)
100   out TWCR, TEMP1
101   WAIT_I2C_REPEATED_START_SENT:
102     in TEMP1, TWCR
103     sbrs TEMP1, TWINT
104     rjmp WAIT_I2C_REPEATED_START_SENT
105   in TEMP1, TWSR
106   andi TEMP1, 0xF8
107   ldi TEMP2, I2C_REPEATED_START_SUCCESS
108   cpse TEMP1, TEMP2
109   rjmp I2C_ERROR
110 ret
111
112 ;This routine sends a stop condition on I2C.
113 I2C_STOP:
114   ldi TEMP1, (1<<TWINT)|(1<<TWEN)|(1<<TWSTO)
115   out TWCR, TEMP1
116 ret
117
118 ;This routine sends a SLAW-frame for device at address I2C_ADDRESS on I2C.
119 I2C_WRITE_SLAW:
120   out TWDR, I2C_ADDRESS
121   ldi TEMP1, (1<<TWINT)|(1<<TWEN)
122   out TWCR, TEMP1
123   WAIT_I2C_SLAWBYTE_SENT:
124     in TEMP1, TWCR
125     sbrs TEMP1, TWINT
126     rjmp WAIT_I2C_SLAWBYTE_SENT
127   in TEMP1, TWSR
128   andi TEMP1, 0xF8
```

```
129    ldi TEMP2, I2C_WRITE_SLAW_SUCCESS
130    cpse TEMP1, TEMP2
131    rjmp I2C_ERROR
132  ret
133
134  ;This routine sends a SLAR-frame for device at address I2C_ADDRESS on I2C.
135  I2C_WRITE_SLAR:
136    mov TEMP1, I2C_ADDRESS
137    inc TEMP1
138    out TWDR, TEMP1
139    ldi TEMP1, (1<<TWINT)|(1<<TWEN)
140    out TWCR, TEMP1
141    WAIT_I2C_SLARBYTE_SENT:
142      in TEMP1, TWCR
143      sbrs TEMP1, TWINT
144      rjmp WAIT_I2C_SLARBYTE_SENT
145    in TEMP1, TWSR
146    andi TEMP1, 0xF8
147    ldi TEMP2, I2C_WRITE_SLAR_SUCCESS
148    cpse TEMP1, TEMP2
149    rjmp I2C_ERROR
150  ret
151
152  ;This routine sends the byte in TWDR on I2C.
153  I2C_WRITE:
154    ldi TEMP1, (1<<TWINT)|(1<<TWEN)
155    out TWCR, TEMP1
156    WAIT_I2C_BYTE_SENT:
157      in TEMP1, TWCR
158      sbrs TEMP1, TWINT
159      rjmp WAIT_I2C_BYTE_SENT
160    in TEMP1, TWSR
161    andi TEMP1, 0xF8
162    ldi TEMP2, I2C_WRITE_SUCCESS
163    cpse TEMP1, TEMP2
164    rjmp I2C_ERROR
165  ret
166
167  ;This routine receives a byte from I2C and stores it into TWDR.
168  I2C_READ:
169    ldi TEMP1, (1<<TWINT)|(1<<TWEA)|(1<<TWEN)
170    out TWCR, TEMP1
171    WAIT_I2C_BYTE_RECEIVED:
172      in TEMP1, TWCR
173      sbrs TEMP1, TWINT
174      rjmp WAIT_I2C_BYTE_RECEIVED
175    in TEMP1, TWSR
176    andi TEMP1, 0xF8
```

```
177    ldi TEMP2, I2C_READ_SUCCESS
178    cpse TEMP1, TEMP2
179    rjmp I2C_ERROR
180  ret
181
182  ;This routine receives the last byte from I2C and stores is into TWDR
183  ;(i.e. without ACK).
184  I2C_READLAST:
185    ldi TEMP1, (1<<TWINT)|(1<<TWEN)
186    out TWCR, TEMP1
187    WAIT_I2C_LASTBYTE_RECEIVED:
188      in TEMP1, TWCR
189      sbrs TEMP1, TWINT
190      rjmp WAIT_I2C_LASTBYTE_RECEIVED
191    in TEMP1, TWSR
192    andi TEMP1, 0xF8
193    ldi TEMP2, I2C_READLAST_SUCCESS
194    cpse TEMP1, TEMP2
195    rjmp I2C_ERROR
196  ret
197
198  ;This branch is executed if an error has occured, the errorcode is saved
199  ;and the I2C-communication is stopped.
200  I2C_ERROR:
201    mov I2C_ERRORCODE, TEMP1          ;save errorcode
202    rcall I2C_STOP                    ;send stopcondition
203    in zh, sph                        ;increase the stack-pointer by 2 - now, the
204    in zl, spl                        ;address on the stack is the return address
205    adiw zh:zl, 2                     ;of the calling routine
206    out sph, zh                       ;so on return the calling routine is aborted too
207    out spl, zl
208  ret
```

Listing D.8: prog/MiniBot/ADC.asm

```
1  ;*****************************************************************************
2  ;*
3  ;*    ADC.asm
4  ;*
5  ;*****************************************************************************
6
7  ;This routine initializes the A/D-unit.
8  ADC_INIT:
9    ldi TEMP1, 3<<REFS0                  ;select internal reference-voltage
10   out ADMUX, TEMP1
11   cbi DDRA, 1                          ;configure ports as input
12   cbi DDRA, 3
13 ret
```

```
14
15 ;This routine starts an A/D-conversion.
16 ADC_START:
17    in TEMP1, ADMUX                                  ;erase previous ADC-source
18    andi TEMP1, 0xE0
19    or TEMP1, ADC_SOURCE                             ;select new ADC-source
20    out ADMUX, TEMP1
21    sbi ADCSR, ADIF                                  ;flag löschen
22    ldi TEMP1, (1<<ADEN)|(7<<ADPS0)|(1<<ADSC)        ;start conversion
23    out ADCSRA, TEMP1
24   WAIT_ADC:                                         ;wait for conversion
25      sbis ADCSRA, ADIF
26      rjmp WAIT_ADC
27    in ADC_VALUE_L, ADCL                             ;store result in memory
28    in ADC_VALUE_H, ADCH
29 ret
```

Anhang E

Wertetabelle des Infrarot-Sensors

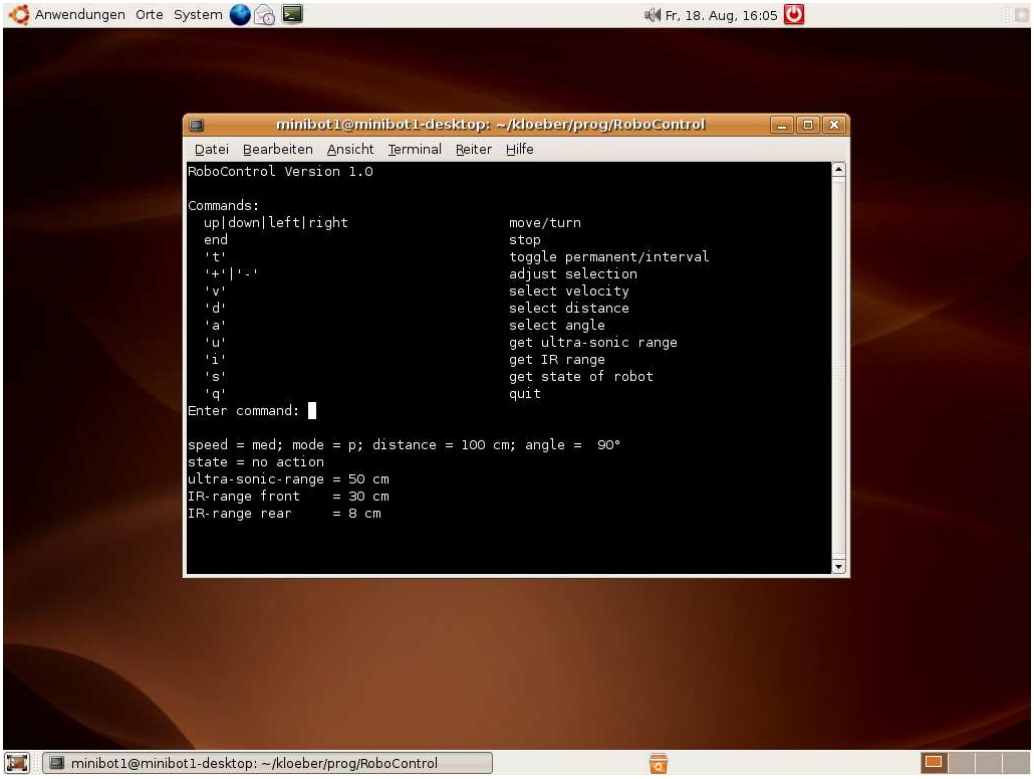
Entfernung [cm]	Digitalwert
4	1023
5	925
6	768
7	694
8	615
9	546
10	493
11	448
12	410
13	380
14	350
15	320
16	305
17	284

Entfernung [cm]	Digitalwert
18	267
19	250
20	234
21	219
22	207
23	196
24	187
25	179
26	172
27	165
28	157
29	149
30	147

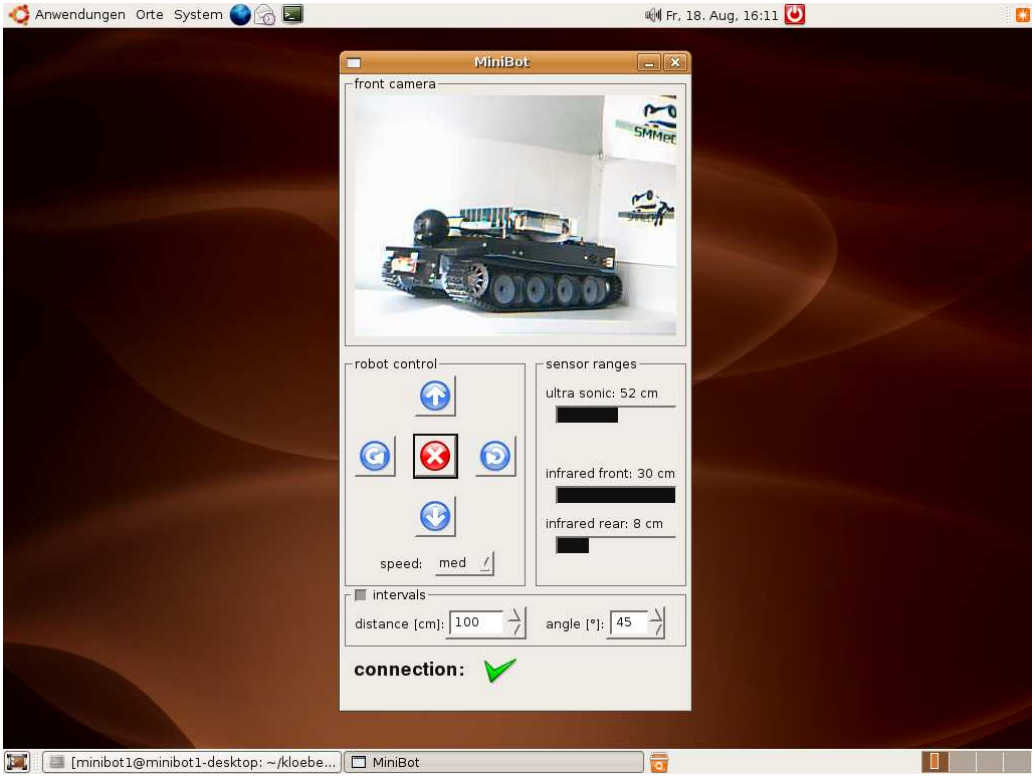
Anhang F

Screenshots

RoboControl



MiniBotGUI


Anwendungen Orte System
Fr, 18. Aug, 16:11
MiniBot
front camera
robot control
sensor ranges
ultra sonic: 52 cm
infrared front: 30 cm
infrared rear: 8 cm
speed: med
intervals
distance [cm]: 100
angle [°]: 45
connection:
[minibot1@minibot1-desktop: ~/kloebe...
MiniBot

Anhang G

Aufbau der Softwareentwicklungsumgebung

Programmierumgebung für den Mikrocontroller

Zur Programmierung des Mikrocontrollers in *AVR-Assembler* wurde das von *Atmel* herausgegebene *AVR Studio*[1] in der aktuellen Version 4.0 verwendet. Diese Entwicklungsumgebung ist frei erhältlich und speziell für die Programmierung von Atmels Mikrocontrollern ausgelegt. Neben einem Editor mit Syntaxhighlighting für AVR-Assembler enthält das AVR Studio einen Compiler zur Erzeugung des Binärcodes, der anschließend auf den Mikrocontroller übertragen werden muss.

Als nützlich erweist sich darüberhinaus der integrierte Simulator, mit dem sich die Abläufe auf dem Mikrocontroller simulieren lassen. Parallel zur Simulation können Registerinhalte, Pointer- oder Speicheradressen inspiziert werden. Dies erleichtert die Fehlersuche, da ein Debugging, der bereits auf den Mikrocontroller aufgespielten Software, nur schwer möglich ist (beispielsweise über Signale der LEDs).

Leider ist das AVR Studio nur für Windows verfügbar, wodurch ein ständiges Wechseln zwischen der Hauptprogrammierplattform Linux und Windows XP nötig wurde.

[1] *http://www.atmel.com/dyn/Products/tools_card.asp?tool_id=2725*

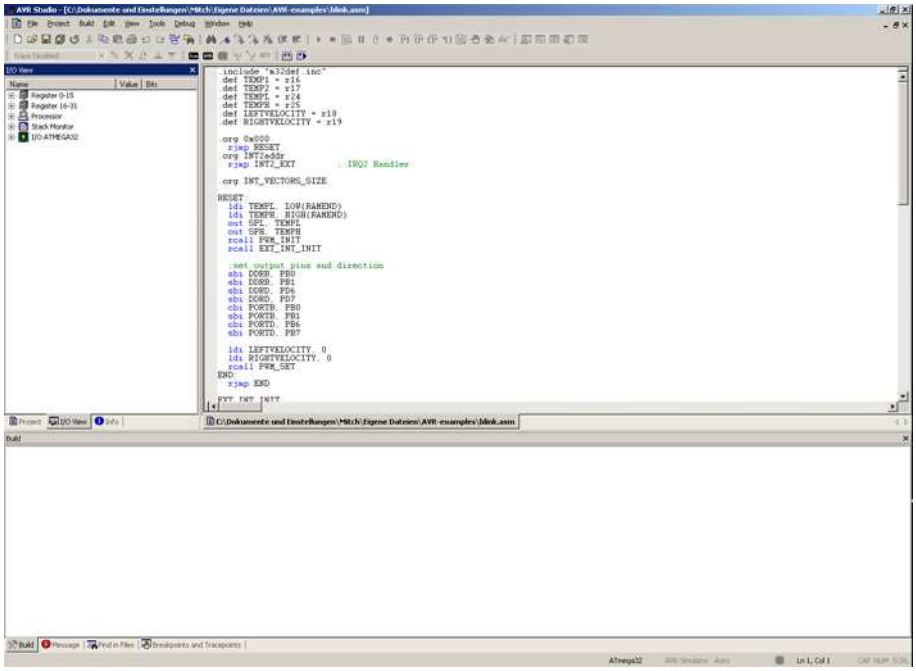


Bild G.1: Das AVR-Studio

Nach dem Programmstart muss ein neues Projekt angelegt werden. Anschließend wird die Art der Plattform, in unserem Fall der *AVR Simulator*, und der Typ des Mikrocontrollers, hier der *ATmega32*, gewählt.

Der Hauptbildschirm des Programms (siehe Abbildung G.1) gliedert sich in drei Teile. Im oberen linken Teilfenster können Informationen über das Projekt, der aktuelle Zustand des Simulators oder Informationen über den Mikrocontroller angezeigt werden. Das obere rechte Teilfenster stellt den Editor zur Eingabe des Assemblerquellcodes dar. Das untere Teilfenster zeigt Informationen über das erstellte Programm, Warnhinweise oder Fehlermeldungen beim Kompilieren und weitere Informationen an.

Wenn der erstellte Assemblercode korrekt ist, erzeugt der Compiler eine Binärdatei mit der Endung `.hex` im Projektordner. Diese enthält den Maschinencode, der nun auf den Mikrocontroller überspielt werden muss. Das Überspielen ist zwar auch mit Hilfe des AVR-Studios möglich, jedoch wird dafür ein spezieller Programmieradapter von Atmel benötigt, dessen Anschaffung unser Budget übertroffen hätte. Daher entwarfen wir einen eigenen Programmieradapter (siehe Kapitel 3.1.1), der das Programmieren des Mikrocontrollers über die serielle Schnittstelle ermöglichte.

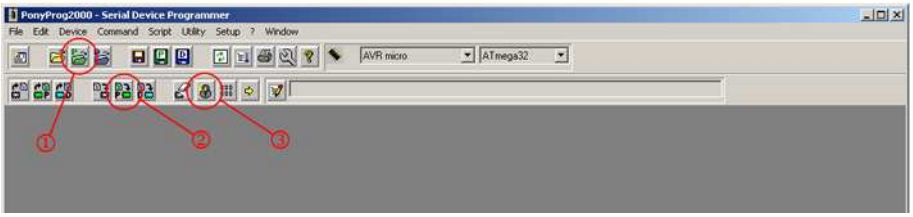


Bild G.2: Menüleiste von PonyProg 2000

Auf der beiliegenden CD befinden sich die Quellcode-Dateien und die erzeugte `.hex`-Datei im Verzeichnis `prog/MiniBot`. Die Projektdatei wurde nicht auf der CD gespeichert, da AVR Studio darin absolute Pfadangaben verwendet, wodurch sie auf einem anderen System unbrauchbar wäre. Eine neue Projektdatei kann jedoch leicht durch Erzeugen eines neuen Projekts und anschließendem Importieren der Quellcode-Dateien erzeugt werden.

Das Programm *PonyProg 2000*[2] erlaubt die Programmierung eines Mikrocontrollers über die serielle Schnittstelle und wurde von uns genutzt. Es ist ebenfalls frei erhältlich, allerdings existiert auch hier nur eine Windows-Version.

Bevor man die Programmierung starten kann, muss der zu beschreibende Typ des Mikrocontrollers eingestellt werden. Nachdem die Verbindung anschließend durch Auswahl des entsprechenden Menüpunktes kalibriert wurde, kann der Maschinencode in der vom AVR-Studio erzeugten `.hex`-Datei über *Open Program Memory* (siehe Punkt 1 der Abbildung G.2) geladen werden. Anschließend kann dieser mit *Write Program Memory* (siehe Punkt 2 der Abbildung G.2) auf den Mikrocontroller übertragen werden. Nach der Übertragung zeigt eine Dialogbox den Erfolg bzw. Misserfolg des Programmiervorgangs an.

Über den Menüpunkt *Security and Configuration Bits* (siehe Punkt 3 der Abbildung G.2) gelangt man zu den Einstellungen der Fuse-Bits. Die Einstellungen der Fuse-Bits können durch Auswahl von *Read* (siehe Punkt 2 der Abbildung G.3) aus dem Mikrocontrollers eingelesen und mittels *Write* (siehe Punkt 1 der Abbildung G.3) wieder auf den Mikrocontroller geschrieben werden. Ein Häkchen bedeutet hierbei, dass das Bit nicht gesetzt ist. Die Konfiguration der Fuse-Bits ist nötig, um die Taktquelle des Mikrocontrollers auf den externen Quarz mit einer Taktfrequenz von 14,7456 MHz einzustellen, da im Auslieferungszustand des Mikrocontrollers eine interne Taktquelle mit 1 MHz Taktfrequenz benutzt wird.

[2] *http://www.lancos.com/ppwin95.html*

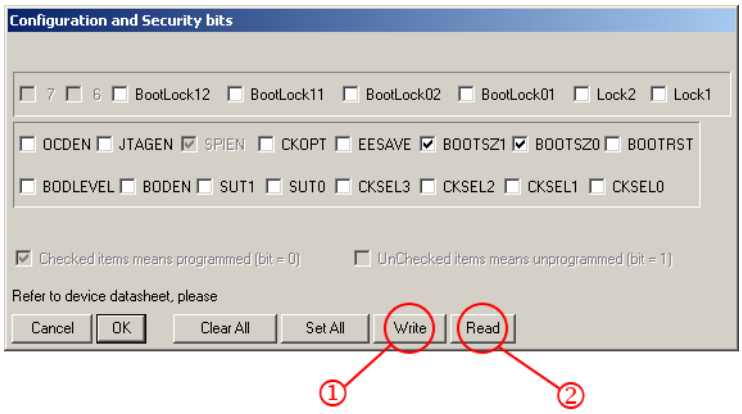


Bild G.3: Die Fuse-Bits-Einstellungen

Programmierumgebung für die Steuersoftware

Zur Programmierung der Steuersoftware für den Barebone-PC wurde die *Anjuta-IDE*[3] verwendet. Sie vereint alle Funktionen die zur Programmierung in C++ nötig sind und ist schlanker als beispielsweise *KDevelop*.

Zum Kompilieren wurde der *GNU C++-Compiler* in der Version 4.1.2 verwendet. Darüberhinaus wurde *make* zur Steuerung der Übersetzung benutzt. Der erstellte Quellcode befindet sich auf der CD in den Unterverzeichnissen von `prog/` und im Verzeichnis `/usr/src/MiniBot/` auf den MiniBots . Zur Kompilierung genügt der Aufruf von `make` in diesem Verzeichnis.

Da das Programm `MiniBotGUI` die Programmbibliothek `Qt` von `Trolltech` in der Version 4.1.4 verwendet, müssen die zugehörigen Bibliotheken, sowie die zu Qt gehörigen Tools (`moc`, `rcc`, usw.), wenn nicht schon vorhanden, nachinstalliert werden. Außerdem muss die `ncurses`-Bibliothek installiert werden, da diese von `RoboControl` benötigt wird.

Installation des Betriebssystems auf den MiniBots

Als Betriebssystem wurde *Ubuntu*[4] in der Version 6.06 auf den Barebone-PCs installiert. Folgende Benutzer wurden angelegt (mit zugehörigen Passwörtern):

[3] *http://anjuta.sourceforge.net/*
[4] *http://www.ubuntu.com/*

	MiniBot1	MiniBot2
Benutzername	`minibot1`	`minibot2`
Passwort	`minibot1`	`minibot2`
Root-Passwort	`root`	`root`

Nach der erfolgreichen Installation mussten jedoch einige Anpassungen gemacht werden:

Der Start verschiedener Anwendungen, die Qt benutzten, führte zu zwei Fehlermeldungen, aufgrund von nicht vorhandenen Komponenten, die in der Konfiguration des X-Servers jedoch auftauchten. Nachdem die entsprechenden Einträge in der für die Konfiguration zuständigen Datei `/etc/X11/xorg.conf` entfernt wurden, tauchten die Fehlermeldungen nicht mehr auf.

Außerdem waren die angezeigten Meldungen beim Start des Kernels nicht erkennbar. Dies liegt daran, dass die aktuellen Kernel-Versionen[5] keine zur integrierten Grafikkarte des Barebone-Motherboards kompatiblen Framebuffer-Treiber enthalten. Daher funktioniert ein Start von Ubuntu bei Benutzung des Framebuffers zur Darstellung der Konsole nicht richtig. Dies wurde gelöst, indem ein neuer Kernel kompiliert und auf den Start im Framebuffer-Modus verzichtet wurde. So werden alle angezeigten Meldungen im Textmodus ausgegeben.

Nach der Installation des eigentlichen Betriebssystems wurde die Umgebungsvariable `MINIBOT` durch einen Eintrag in `~/.bashrc` auf `MINIBOT_1` bzw. `MINIBOT_2` gesetzt. Dies ist nötig, damit beim Kompilieren die richtigen Konfigurationspassagen in der Datei `RoboSetup.h` benutzt werden, da sich die Konfigurationswerte bei beiden MiniBots unterscheiden.

Weitere Software

Inkscape

Zur Erstellung der Vektorgrafiken wurde das freie Zeichenprogramm *Inkscape*[6] benutzt. Es ermöglicht das Speichern im standardisierten Vektorformat *SVG*[7] und darüberhinaus den Export als Eingebettetes PostScript (*EPS*) zur weiteren Verwendung in Latex-Dokumenten.

[5] *http://www.kernel.org/*
[6] *http://www.inkscape.org/*
[7] *http://www.w3.org/TR/SVG/*

FindGraph

Zur Ermittlung der Approximierungsfunktion für die Umrechnung der Digitalwerte der Infrarotsensoren in die zugehörige Entfernung wurde die Trial-Version des Programms *FindGraph*[8] von *UNIPHIZ Lab*[9] benutzt. Zuerst wurden dort die Wertepaare als Punkte hinzugefügt. Dann wurde mit der Option *Fit Function* eine Approximierungsfunktion berechnet. Nach Auswahl des Funktiontyps $x = f(y)$ und einer hyperbolische Approximierung 3. Grades[Küh98] berechnete das Programm die gesuchte Funktion.

Photoshop und Gimp

Sowohl die Konstruktionszeichnungen und das Blockschaltbild, als auch die Bilder des Aufbaus wurden mit *Adobe Photoshop CS*[10] erstellt, bzw. überarbeitet. Da Photoshop zwar Bilder im *EPS*-Format exportieren kann, dabei aber leider keine *Bounding-Box* erstellt, wurde das freie Programm *Gimp*[11] verwendet, welches den Export der Bilder im *EPS*-Format inklusive der benötigten Bounding-Box erlaubt.

Lochmaster 3.0

Um das Layout der Lochrasterplatinen zu erstellen, wurde das Programm *Lochmaster 3.0*[12] von *Abacom*[13] verwendet. Es ist in einer kostenlosen Demoversion erhältlich, die zwar das Erstellen der Platinen erlaubt, aber eine eingeschränkte Bibliothek enthält und das Speichern der Entwürfe nicht ermöglicht. Die Bilder der Platinen wurden mittels *Screenshot* und Photoshop erstellt.

[8] *http://www.uniphiz.com/findgraph.htm*
[9] *http://www.uniphiz.com/*
[10] *http://www.adobe.com/de/products/photoshop*
[11] *http://www.gimp.org/*
[12] *http://www.abacom-online.de/html/lochmaster.html*
[13] http://www.abacom-online.de/default.html

MultiSIM 9

Die Schaltung der Spannungsanzeige wurde mit dem Programm *MultiSIM 9*[14] der Firma *Electronics Workbench*[15] erstellt und getestet. Dieses Programm kann man kostenlos 45-Tage in vollem Funktionsumfang nutzen und es enthält eine komplette Bibliothek an elektronischen Bauteilen.

[14] *http://www.electronicsworkbench.de/demo1.html*
[15] *http://www.electronicsworkbench.com/worldwide/germany/*

Literaturverzeichnis

[Amb05] Scott W. Ambler. *The Elements of UML 2.0 Style*. Cambridge University Press, Juli 2005.

[Bau04] Heinz-Josef Bauckholt. *Grundlagen und Bauelemente der Elektrotechnik*. Hanser Fachbuchverlag, Januar 2004.

[Bla06] Jasmin Blanchette. *C++ GUI Programmierung mit Qt 4*. Addison-Wesley, Oktober 2006.

[BN96] Jacqueline Proulx Farrell Bradford Nichols, Dick Buttlar. *Pthreads Programming: A POSIX Standard for Better Multiprocessing*. O'Reilly Media, September 1996.

[Cat05] John Catsoulis. *Designing Embedded Hardware*. O'Reilly Media, Mai 2005.

[CH97] Tracey Hughes Cameron Hughes. *Object-Oriented Multithreading Using C++*. John Wiley & Sons Ltd, August 1997.

[DP97] Carl Fenger Dominique Paret. *The I2C Bus from Theory to Practice*. John Wiley & Sons Ltd, Februar 1997.

[Fed04] Joachim Federau. *Operationsverstärker. Lehr- und Arbeitsbuch zu angewandten Grundschaltungen*. Vieweg, Dezember 2004.

[Gad00] Dhananjay Gadre. *Programming and Customizing the AVR Microcontroller*. McGraw-Hill/TAB Electronics, September 2000.

[Kai89] Burkhard Kainka. *Messen, Steuern und Regeln über die RS-232 Schnittstelle*. Franzis, 1989.

[Küh98] Ernst Kühner. *Grundlagen der Funktionalanalysis und Approximationstheorie*. Vandenhoeck & Ruprecht, Februar 1998.

[Mor02] John Morton. *AVR: An Introductory Course*. Newnes, September 2002.

[RHB06] Larry O'Cull Richard H. Barnett, Sarah Cox. *Embedded C Programming And The Atmel AVR*. Thomson Delmar Learning, Juni 2006.

[Sey88] Martin D. Seyer. *Complete Guide to Rs232 and Parallel Connections: A Step-By-Step Approach to Connecting Computers, Printers, Terminals, and Modems*. Prentice Hall, Juli 1988.

[Str86] John Strang. *Programming with curses*. O'Reilly Media, Januar 1986.