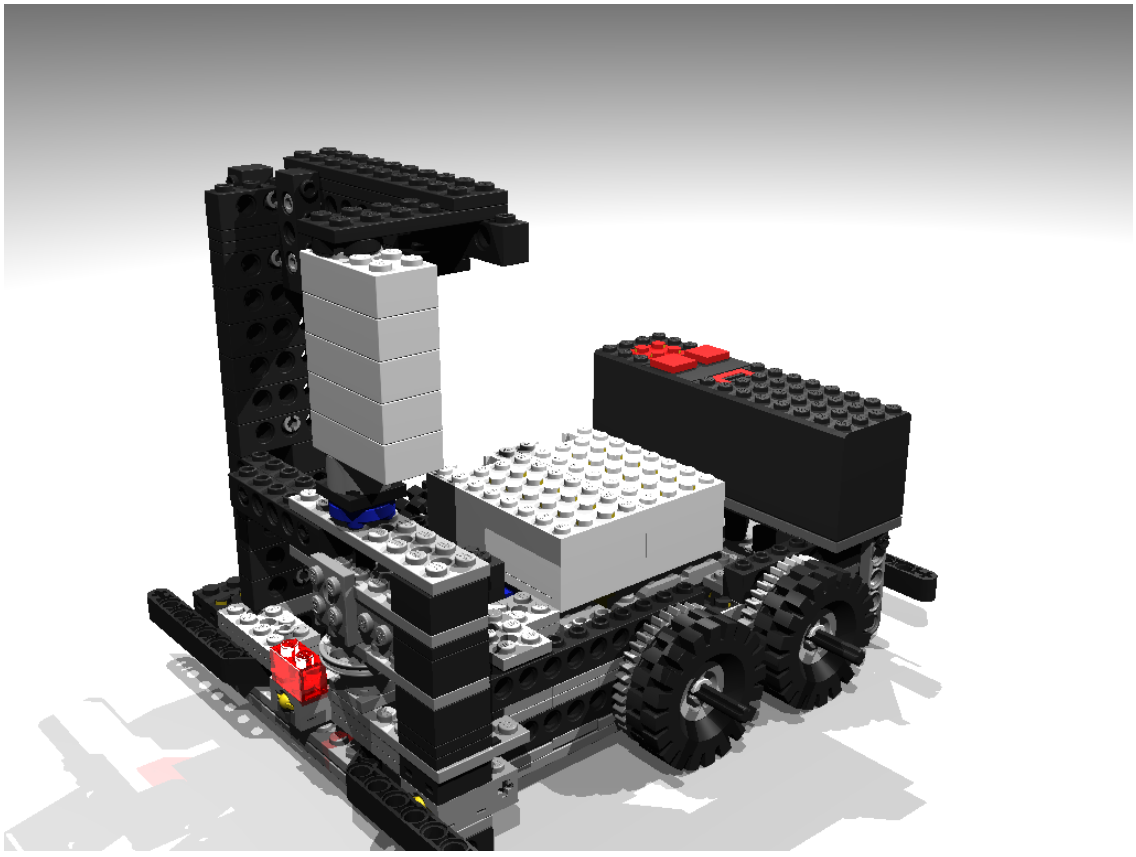
Endbericht

Projektgruppe 420

RoCK – Robot Construction Kit

Sommersemester 2003



Endbericht

Projektgruppe 420

ROCK – Robot Construction Kit

Sommersemester 2003

Teilnehmer:
Detlev Bartsch, Torsten Denno
Pedram Hadjian, Nico Karnatz
André Kernchen, Andreas Klapschus
Marcus Ladwig, Michael Patzer
Matthias Reck, Christoph Schlagbaum
Daniel Smolinski, Thorsten Wilmer

Betreuer:
Jens Wagner, Lars Wehmeyer

Universität Dortmund
Fachbereich Informatik
Lehrstuhl Informatik XII
Prof. Dr. Peter Marwedel

Inhaltsverzeichnis

Kapitel 1

Einführung – Das Ziel der Projektgruppe RoCK

Robot Construction Kit – Hard- und Software-Entwicklung für komplexe Kleinstroboter

Im Rahmen dieser Projektgruppe sollte von den Teilnehmern eine minimale Hardwareplattform entwickelt und gebaut werden, die in der Lage ist, einen aus LEGO-Bausteinen konstruierten Roboter zu steuern.

Roboter verfügen im Gegensatz zu Automaten über die Fähigkeit, komplexe Arbeitsvorgänge zu verrichten, die verschiedene Werkzeuge und/oder räumliche Beweglichkeit erfordern.

Einen eigenen Roboter zu bauen, setzte bis vor einiger Zeit erhebliche finanzielle Mittel und handwerkliche Fähigkeiten voraus. Das relativ hohe Einstiegsniveau resultiert vor allem aus den mechanischen Komponenten. Während elektronische Steuerungen in den letzten Jahren immer preiswerter und kleiner wurden, sind die mechanischen Teile noch immer relativ teuer und aufwendig zu beschaffen.

Während das Massachusetts Institute of Technology (MIT) bereits früh an eigenen Robotersteuerungen wie etwa dem „Cricket“ arbeitete, entwickelte LEGO selbst einen neuen Steuerbaustein, der sich nahtlos in die LEGO-Serie integrieren lässt: die RCX-Steuereinheit. Diese kompakte Einheit baut in Hard- und Software auf dem „Programmable Brick“ des MIT auf. Zusammen mit der MINDSTORMS-Reihe von LEGO entstand ein Roboterbaukasten mit einem kompletten Satz mechanischer Komponenten und einer einfachen graphischen Programmiersprache. Ergänzt wird dieses System durch eine Vielzahl von Aktoren und Sensoren.

Hauptmerkmal des von der Gruppe zu entwickelnden Gesamtsystems soll neben funktionierenden Sensor- und Aktoranschlüssen die Programmierbarkeit der Steuerung in der Hochsprache ANSI-C sein. Die Entwicklung eines Compilers, der die Programme in die Maschinensprache des verwendeten Controllers übersetzt, ist daher ein wesentlicher Bestandteil dieser PG. Bereits existierende Compiler für den ausgewählten Prozessor bieten keine ausreichende Unterstützung für den von uns anvisierten Anwendungsbereich.

Wunschsystem

Ein Wunschsystem weist folgende Eigenschaften auf:

- Volle Kompatibilität zu vorhandenen mechanischen und elektrischen Komponenten.
  Das Zielsystem sollte ins LEGO-Stecksystem passen und im elektrischen und logischen Interface kompatibel zu vorhandenen Sensoren und Aktoren sein.
- Modularer Aufbau
  Um eine geringe Größe der Steuereinheit zu ermöglichen, sollte der Energiespeicher vom Controller

getrennt werden. Hierbei bietet sich die Verwendung von einzeln erhältlichen LEGO-Batteriepaketen an. Die Anzahl der Ein- und Ausgänge der Zentraleinheit sollte über zusätzliche Baugruppen erweiterbar sein. Die Größe der gesamten Steuereinheit ist so klein wie möglich zu halten. Insgesamt sollte durch den modularen Aufbau ein leicht erweiterbares, billiges Einstiegssystem realisiert werden.

- Hochsprachenprogrammierbarkeit
  Um komplexe Algorithmen abarbeiten zu können, sollte die Zentraleinheit in einer Hochsprache programmierbar sein.

- Leistungsfähigkeit
  Das zu implementierende System sollte in der Lage sein, komplexe Algorithmen aus dem Bereich der Robotertechnik verarbeiten zu können. Dazu gehören etwa die Umrechnung von Raum- in Polarkoordinaten zur Justierung von Manipulatoren und die Verarbeitung von Sensorsignalen, wie die Frequenzanalyse der Impulsantwort eines Ultraschallsensors.

- Nachbaubarkeit
  Das gesamte Design ist von vornherein auf einfache Nachbaubarkeit auszulegen. Hierzu zählt die Verwendung von leicht erhältlichen Bauteilen sowie ein möglichst geringer Preis der Komponenten. Die Beschreibung der laufenden Arbeiten sollte frühzeitig im Internet veröffentlicht werden, um Synergieeffekte mit anderen Gruppen, die sich bereits mit einer ähnlichen Thematik auseinander gesetzt haben, zu ermöglichen.

Realisierung

Die Zentralsteuerung unterteilt sich in drei Komponenten:

- Zentraleinheit
  Wie beim Cricket fiel die Wahl auf einen 8-Bit System-On-Chip-Mikrocontroller. Im Cricket wurden Microchip-PIC-Mikrocontroller gewählt, da bereits alle Komponenten (mit Ausnahme von Leistungsschaltern) auf einem kleinen, preisgünstigen Schaltkreis zusammengefasst sind. Insbesondere analoge Eingänge sowie A/D-Wandler für die Sensoren sind bereits im Controller enthalten. In der Reihe der PIC-Prozessoren befinden sich geeignete Modelle, die neben ausreichender Leistungsfähigkeit für komplexe Algorithmen auch zu einem akzeptablen Preis erhältlich sind. Durch die hohe Integrationsdichte aller benötigten Komponenten sind kaum externe Bauteile nötig, so dass der Gesamtpreis für das System sehr niedrig ausfallen kann. Beispielapplikationen stehen in Form von Application-Notes des Schaltkreisherstellers Microchip zur Verfügung.

- Fan-in / Fan-out-Unit
  Um die üblicherweise geringe Anzahl von Ein- und Ausgängen der zentralen Steuereinheit zu kompensieren, soll es möglich sein, durch aufsteckbare Fan-in- und Fan-out-Units eine größere Anzahl von logischen Ports zur Verfügung zu stellen. Diese Einheiten arbeiten als Multiplexer, die den aktiven Sensoren und Motoren die benötigte Betriebsspannung zuleiten und die Sensordaten auslesen.

- Energiespeicher
  Um die Größe der zentralen Steuereinheit zu minimieren, soll die Stromversorgung nicht in demselben Gehäuse wie der Controller untergebracht werden. Auf diese Weise kann das Batteriepaket an einer zentralen, untenliegenden Position des Roboters platziert werden um den Schwerpunkt in die Tiefe zu verlagern und dadurch das Fahrverhalten zu stabilisieren. Die von Lego erhältlichen Batteriepakete bieten sich hierfür an, da sie sich am einfachsten einbauen lassen. Die Zentraleinheit mit ihren Bedienelementen wird hingegen an einer gut zugänglichen Position auf dem Gerät angebracht.

Minimalziel der Projektgruppe:

- Entwicklung einer minimalen Hardwareplattform, Anschluss von mindestens einem Sensor und Aktor sowie ein Funktionstest durch Assemblerprogrammierung
- Programmierung eines Compilers, der die Hochsprache ANSI-C in die Maschinensprache des verwendeten Mikrocontrollers überführt. Erfolgreiche Compilierung und Ausführung eines Testprogramms, welches Sensorwerte abfragt und Aktoren ansteuert.

Kapitel 2

Stand der Technik

In diesem Abschnitt wird beschrieben, in welchem Kontext sich die vorgestellte Lösung bewegt. Neben den von LEGO vertriebenen Bausätzen, Bauteilen und Elementen geht dieser Abschnitt vor allem auf verwandte Projekte wie beispielsweise NQC und LegOS, aber auch auf bereits vorhandene Compiler und Bibliotheken für das LEGO Mindstorms-System ein.

Der Leser soll hier zunächst ein Grundverständnis für die im weiteren Verlauf des Berichts verwendeten Begriffe gewinnen.

2.1 Hardware der RCX-Einheit

2.1.1 Vorläufer

Das Mindstorms-Design basiert auf schon vorher vorhandenen, bzw. simultan entwickelten, Ansätzen. Die beiden wichtigsten sind Brick und Cricket, die zu einem gewissen Teil Modell gestanden haben.

Der Brick

Der programmierbare Brick [MITc] ist ein Forschungsprojekt des MIT Media Laboratory [MITb] und damit ursprünglich nicht von LEGO initiiert. Dennoch ging die Weiterentwicklung im Jahre 1984 von LEGO aus, die einen programmierbaren LOGO-Baustein produzieren wollten, der LEGO-Motoren ansteuern kann. Dazu wurde der MIT-Brick in einen LEGO-Baustein eingebettet.

Benutzt wurde – im Gegensatz zur RCX-Einheit – ein Motorola 6811 Prozessor mit 32 Kilobyte RAM. Der Prozessor (Brick Model 120) kann vier Motoren oder Leuchten gleichzeitig ansteuern und Signale von sechs Sensoren verarbeiten. Ein Infrarot-Empfänger zur Programmierung ist bereits eingebaut, ein Sender kann angeschlossen werden. Ein zweizeiliges Display, ein Knopf und zwei Schalter komplettieren die Ausstattung. In frühen Versionen waren die I/O-Möglichkeiten des Bricks wesentlich beschränkter. Entwickelt wurde der Brick mit folgenden Zielen:

- universelle Einsetzbarkeit
- Vielseitigkeit der I/O
- Multi-Processing
- Kaskadierbarkeit

Dabei stand immer ein Punkt als Motivation im Vordergrund: Kinder sollen damit umgehen können, um spielend zu lernen.

Der Cricket

Der Cricket [MITa] war im Wesentlichen eine Weiterentwicklung des Bricks in deutlich kleinerem Format, aber dafür nicht so breit gefächert im Funktionsumfang. Auch diese Entwicklung geschah am MIT und ebenso wie beim Brick war hier Fred Martin [Mar] federführend. Daher liegen auch die Lizenzrechte bei ihm. Entwickelt wurde der Cricket ungefähr gleichzeitig mit der RCX, so dass es zu Synergieeffekten kam, von denen beide Produkte profitiert haben. Der augenscheinliche Unterschied des Cricket zum Brick liegt in der Größe, aber auch der Mikrocontroller und die Steuerungsmöglichkeiten differieren. So werden im Cricket Mikrocontroller aus der PIC-Familie eingesetzt und er bietet auch nur die Möglichkeit, zwei Sensoren sowie zwei Motoren anzusteuern.

Es ist eine Infrarotschnittstelle integriert sowie ein Miniaturlautsprecher, genau wie im „großen Bruder“. Der Cricket versteht if-, repeat- und loop-Befehle, verwaltet lokale und globale Variablen und beherrscht einfache Arithmetik mit 8-Bit Breite. Das ist aber nur der klassische Cricket. Darüber hinaus gibt es noch weitere Ausbaustufen, mit z. B. drei Sensoren und acht zweifarbigen LEDs, aber ohne Motor, oder ein Cricket mit 10-Bit A/D-Konverter und 16-Bit Arithmetik.

Beispielprojekte mit dem Cricket

Der Cricket wurde z. B. zum „Stackable“, einem manipulierbaren verteilten Display, ausgebaut. Dabei kann jeder Stackable individuell seine vier LEDs ansteuern und mit seinem jeweiligen Nachbar-Stackable kommunizieren. Damit erhält man ein beliebig erweiterbares und programmierbares Display, welches beispielsweise Börsenkurse anzeigen kann.

Weitere Informationen zu artverwandten Projekten findet man auf der Homepage des MIT unter dem Stichpunkt „Lifelong kindergarten“ [MITb].

Andere Steuereinheiten

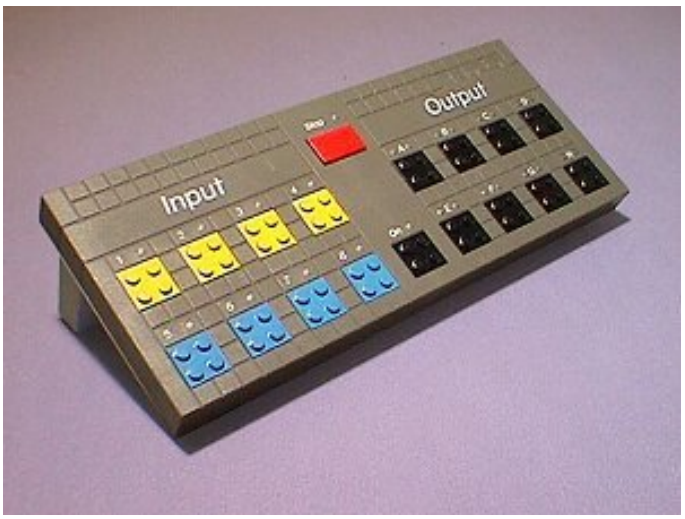


Abbildung 2.1: Die Dacta-Interface-Box von LEGO [CMA]

Vorreiter der Mindstorms waren auf jeden Fall der LEGO-CyberMaster und die Dacta-Interface-Box von

LEGO, die in Abb. 2.1 auf der vorherigen Seite zu sehen ist und die als stationärer (an die serielle Schnittstelle eines PCs angeschlossener) Vorfahre des Bricks gilt.

Das große Problem bei diesem Design ist natürlich die Verbindung zum PC, der hier noch die Aufgabe des Mikrocontrollers übernehmen muss. Dagegen arbeitet der CyberMaster schon per Funkübertragung und ermöglicht damit eine gewisse räumliche Unabhängigkeit. Trotzdem ist bei diesem System immer noch der PC die „denkende“ Einheit. Das hat sich erst mit dem Brick bzw. mit der RCX geändert.

2.1.2 Die RCX-Einheit

Die LEGO RCX-Einheit ist zentrale Komponente jedes Mindstorm Roboters. Seit der Markteinführung 1998 hat es von Version 1.0 bis 2.0 keine wesentlichen Änderungen bzgl. Hardware und Aussehen gegeben (siehe Abbildung 2.2).

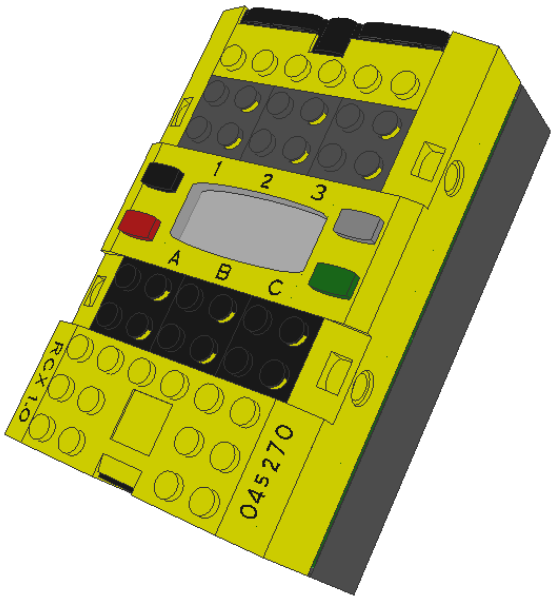


Abbildung 2.2: RCX-Einheit

Sie hat an der Oberseite drei Anschlüsse für Sensoren (1 bis 3) und drei Anschlüsse für Aktoren (A bis C). Ein Hitachi H8 Prozessor (16 Mhz) mit integriertem A/D-Wandler, serieller I/O und Timern ermöglicht die preiswerte Realisierung dieser Steuerungseinheit. Die integrierten 16 KB ROM beinhalten ein rudimentäres Betriebssystem. Für die Firmware und Benutzerprogramme stehen 32 KB statisches RAM zur Verfügung. Bei einem Batteriewechsel muss die RCX-Einheit daher immer wieder neu progrmmiert werden. Die kompakte Bauweise der RCX mit eingebauter Batterie und Display erschwert außerdem die Konstruktion von Robotern erheblich, da dort meist ein niedriger Schwerpunkt erwünscht ist, was sich aber selten mit der Displaysichtbarkeit und Erreichbarkeit der Taster zum Programmstart vereinbaren lässt.

Die Anzeige der RCX-Einheit: Auf ihr sieht man zunächst immer, ob gerade ein Programm läuft und wenn ja, welches. Außerdem kann man sich die Werte sämtlicher Ein- und Ausgänge anschauen. Welche das im Moment sind, kann man durch die Tasten links und rechts der Anzeige einstellen.

Der Pieper der RCX-Einheit: In jeder RCX-Einheit befindet sich ein mit dem PC-Lautsprecher vergleichbarer Tongeber, der durch Anwendersoftware angesprochen werden kann. Es können ihm auch Me-

lodien entlockt werden. Im RCX-Commando-Center gibt es bereits ein Tool dafür (siehe [Bra]).

Schnittstellen: Hinter der dunklen Blende an einer der Stirnseiten der RCX-Einheit verbirgt sich der Infrarot-Sender / Empfänger.

Nutzen kann man ihn zum einen mit der im separat zu erwerbenden „Ultimate Accessory Set“ enthaltenen Fernbedienung, ähnlich den bekannten Modellen für Unterhaltungselektronik. Die Reichweite beträgt etwa sechs Meter, steuern lassen sich damit unabhängig von einander die drei Ausgänge der RCX-Einheit, die Auswahl des gerade aktiven Programms sowie der eingebaute Signalgeber.

Zum anderen gehört zur Grundausstattung des Mindstormsbaukasten ein so genannter „IR-Tower” für die drahtlose Kommunikation zwischen der RCX-Einheit und einem PC. Aktuell liefert LEGO ein Modell zum Anschluss an die USB-Schnittstelle eines PCs, die ältere Version erfordete einen seriellen Port.

Diese Infrarot-Verbindung ermöglicht die Übertragung der Firmware zum RCX-Baustein, das Aufspielen selbst geschriebener Anwenderprogramme sowie die Fernsteuerung des Roboters, kurz: generell den Datenaustausch zwischen RCX und PC, sowie, wenn auch stark eingeschränkt, die Kommunikation zwischen mehreren RCX-Einheiten.

Die zu übertragenen Daten werden auf eine Trägerfrequenz von 38 kHz aufmoduliert und mit 2400 Bit pro Sekunde kodiert. Ein Bit ist folglich rund 417 μs lang. Nach dem Startbit kommen acht Datenbits, gefolgt von einem Stoppbit und einem Paritätsbit. Die Anzahl der gesetzen Bits ist immer ungerade (auch bekannt als „odd parity“). Ein Byte wird im so genannten „Little Endian“-Format übertragen, also mit führendem niederwertigsten Bit.

Das Signal ist unipolar, es gibt nur zwei Zustände. Das von LEGO gewählte Format wird auch als NRZ („non-return-to-zero“) bezeichnet, weil mehrere High-Pegel aufeinander folgen können, ohne dass dazwischen wieder der Low-Pegel erreicht wird. Die Erkennung dieser Signale erfordert deshalb ein exaktes Timing.

Übrigens entsprechen die technischen Daten der LEGO IR-Komponenten recht genau den Werten von handelsüblichen Fernbedienungen, so dass lernfähige Universalfernbedienungen oder Handheld-PCs zur Steuerung einer RCX-Einheit benutzt werden können. So verwundert es nicht weiter, dass bereits mehrere Selbstbauprojekte rund um die IR-Kommunikation existieren. Für einen Überblick empfiehlt sich [Nel].

2.1.3 Sensoren

Die RCX-Einheit verfügt über drei Sensoreingänge, hinter denen jeweils ein A/D-Wandler mit zehn Bit Auflösung steht. Dieser fragt die Eingangswerte alle 3 ms ab (Zeitmultiplexing) und stellt Spannungen von 0–5 V in internen Werten von 0–1023 (den sogenannten RAW-Values) dar. Die LEGO-Software übernimmt dann in Abhängigkeit von der Konfiguration des Sensors die Interpretation dieses Wertes. In der Firmware der RCX-Einheit ist die Auswertung von Berührungs-, Helligkeits-, Rotations- und Temperatursensoren bereits vorbereitet. Man kann sich sowohl die berechneten Werte als auch die RAW-Werte direkt auf dem Display der RCX-Einheit anzeigen lassen. Es war ursprünglich nicht vorgesehen, diese Rohdaten in Programmen zu verwenden.

Einige Enthusiasten entwickeln und bauen eigene Sensoren für das Mindstorms-System. Beispiele finden sich unter dem Stichwort „Homebrew Sensors“ [ROB] oder auf dieser Webseite [Nel]. Bei diesen Sensoren findet man sowohl passive (d. h. ohne eigene Stromversorgung) als auch aktive Sensoren und für eine Messgröße (z. B. Rotation) teilweise sehr verschiedene Ausführungen.

Darüber hinaus existiert eine MINDSTORM-Kamera in einem LEGO-Gehäuse. Dahinter verbirgt sich eine USB-Kamera. Diese kann daher auch nur in Verbindung mit einem PC benutzt werden.

Inzwischen vertreibt die Firma DCP Microdevelopments einen Adapter, welcher es ermöglicht, verschiedenste kommerzielle Sensoren (Lautstärke, Spannung, Feuchtigkeit, ...) an eine RCX-Einheit anzuschlie-

ßen (siehe [DCP]).

Der Tastsensor

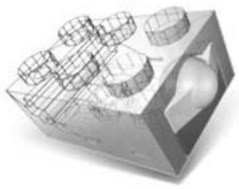


Abbildung 2.3: Tastsensor von LEGO

Der einfachste Sensor ist der Tastsensor. Er ist bereits im Basispaket zweimal enthalten. Wenn bei der RCX-Einheit „Tastsensor“ eingestellt ist, zieht ein Pull-Up-Widerstand (10 kΩ) die Ausgangsspannung auf 5 V. Durch die veränderten Widerstandswerte folgert die Steuerungseinheit auf die Messgröße. Bei Nichtbetätigung ist der Widerstand wie bei einem schaltenden Taster unendlich groß. Bei Betätigung hingegen ist der Widerstand variabel, aber in jedem Fall größer als ca. 500 Ω. Der Tastsensor ist auch der einzige Originalsensor, der nicht über eine eigene Anschlussleitung mit der Steuereinheit verbunden wird. Bei ihm muss man an den vorderen 2×2 Pins ein LEGO-Kabel anschließen.

Der Temperaturfühler

Abbildung 2.4: Temperatursensor von LEGO

Dieser Sensor ist ein temperaturveränderlicher Widerstand. Er befindet sich in einer Spitze, die aus einem 2×3 LEGO-Standardbaustein herausragt. Wenn er angeschlossen ist, wird er genauso wie der Tastsensor über einen 10 kΩ Widerstand mit 5 Volt verbunden und auch hier wird alle 3 ms eine Messung durchgeführt. Die RCX-Software rechnet die Widerstandswerte dann in Temperaturwerte von -20 bis +50 Grad Celsius um.

Der Lichtsensor

Da der Helligkeitssensor aktive Bauteile beinhaltet, erfordert er eine eigene Energieversorgung. Damit keine zusätzlichen Leitungen benötigt werden, hat man sich einen Schaltungstrick einfallen lassen: Die Betriebsspannung liegt nahezu permanent den Anschlüssen des Sensors an, wird jedoch alle 3 ms für 0,1 ms unterbrochen, um in dieser Zeitspanne den am Sensor anliegenden Spannungswert dem A/D-Wandler zuzuführen. In diesen 100 μs wird der Sensor über einem 10 kΩ Widerstand mit 5V̇ verbunden.
Der Lichtsensor ist im Basispaket der LEGO-Robotics enthalten. Er misst Helligkeitswerte von 0,6 bis 760 Lux. Diese werden von der RCX-Software auf einen Bereich von 0 bis 100 skaliert. Weil sich direkt

Abbildung 2.5: LEGO-Lichtsensor

	dunkel		mittelhell		hell	
Eingang	Sensor 1	Sensor 2	Sensor 1	Sensor 2	Sensor 1	Sensor 2
Nr. 1	28	21	50	46	91-96	91-95
Nr. 2	28	22	50	46	91-96	92-95
Nr. 3	28	21	50	46	91-96	91-95

Tabelle 2.1: Zwei Sensoren bei verschiedenen Lichtstärken an den Eingängen 1–3 der RCX

neben der Photozelle eine Leuchtdiode befindet, liegt der niedrigste erreichbare Wert im Bereich von 15 bis 20 (siehe Abbildung 2.6).

Dunkel steht hier für einen mit einer glatten, hellen Oberfläche abgedeckten Lichtsensor, mittelhell für normale Zimmerhelligkeit und hell für zusätzliche Beleuchtung durch eine LEGO-Lampe. Man sieht, dass zwischen den einzelnen Eingängen praktisch kein Unterschied besteht (siehe Tabelle 2.1.3) . Zwischen den beiden Lichtsensoren erkennt man aber große Differenzen und im oberen Bereich stellte sich kein statischer Wert ein, obwohl es sich um einen festen Versuchsaufbau handelt.

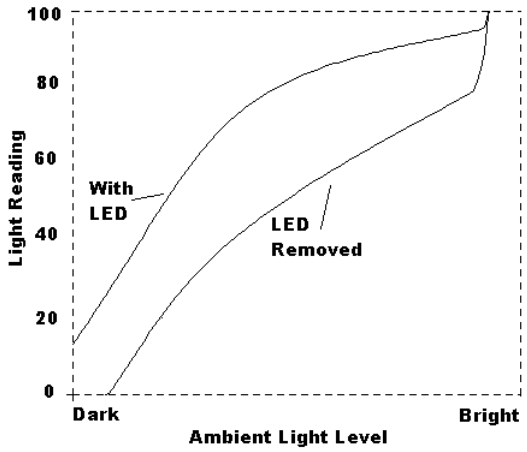


Abbildung 2.6: Helligkeitswert des Lichtsensors, mit und ohne LED

Sasuraibito [Gas] hat einen Lichtsensor geöffnet (Abbildung 2.7) und außerdem den Schaltplan abgenommen (Abbildung 2.8).

Die Bauteile Nr. 2, 3 und 10 in Abb. 2.7 dienen der Spannungsversorgung, Nr. 11 braucht nicht weiter beachtet zu werden, weil es sich um einen unbenutzten Operationsverstärker (OP) handelt. Eine konstante Stromversorgung für die LED (Nr. 5) und den Phototransistor (Nr. 7) von 7,5 mA wird durch die Bauteile

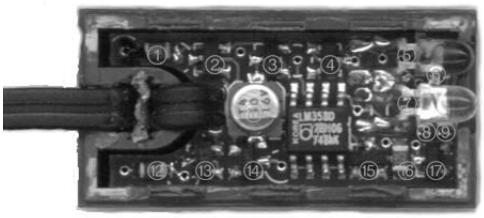


Abbildung 2.7: geöffneter LEGO-Lichtsensor

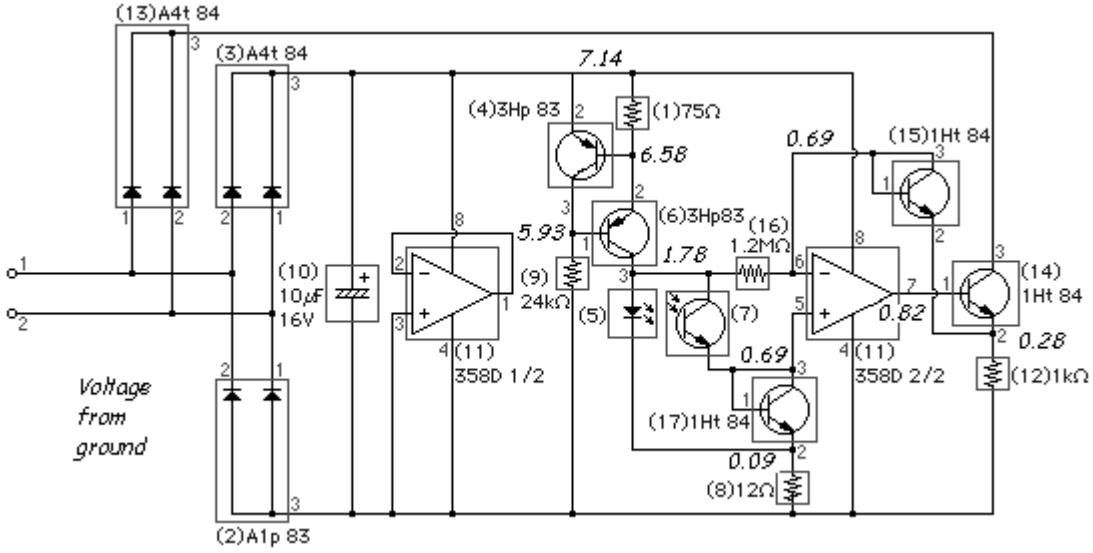


Abbildung 2.8: Innere Beschaltung des Lichtsensors

1, 4, 6 und 9 sichergestellt. Der OP, der den Transistor Nr. 14 ansteuert, kommt mit einer geringen äußeren Beschaltung aus (Nr. 15, 16, 17 und 8). Die Umgebungshelligkeit beeinflusst somit über den Phototransistor erst den OP und zu guter Letzt die Leitfähigkeit des Transistors Nr. 14. Während der 0,1 ms langen Messintervalle läuft die Schaltung über die in dem Kondensator gespeicherte Energie weiter und für die nun von außen anliegende Messspannung verhält sie sich wie ein normaler Widerstand. Man beachte dabei die beiden Dioden (Nr. 13), die dafür sorgen, dass das zweiadrige Anschlussstück verpolungssicher mit der RCX-Einheit verbunden werden kann.

Der Rotationssensor

Die innere Beschaltung des Rotationssensors (siehe Abb. 2.9) sorgt dafür, dass sich bei jeder vollen Drehung der Achse die Ausgangsspannung 16-mal ändert. Sie variiert dabei zwischen vier Werten. Anhand der Reihenfolge der Spannungen kann man auf die Drehrichtung schließen. Phillippe Hurban hat sich dankenswerterweise die Arbeit der Schaltungsanalyse gemacht (siehe: [Hur]). Er kommt zu dem Schluss, dass es sich um zwei Gabellichtschranken handelt, deren Widerstandswerte durch zwei in ihrer Bedeutung gewichtete Verstärkerschaltungen in vier Spannungen kodiert werden.

Abbildung 2.9: Rotationssensor von LEGO

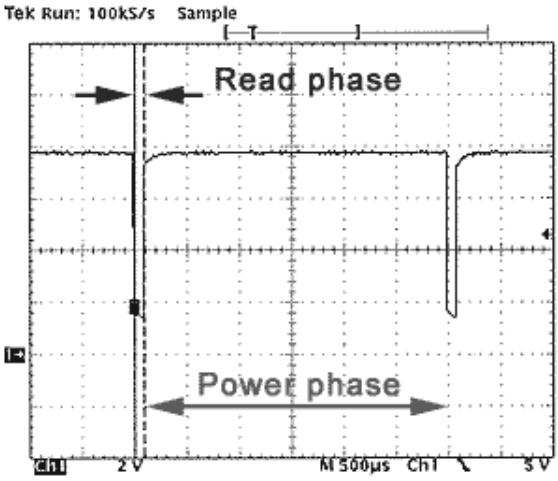


Abbildung 2.10: Oszillogramm von einer achtel Drehung

2.1.4 Aktoren

Eine LEGO RCX-Einheit hat drei Ausgänge (A bis C), die einzeln angesteuert werden können. Jeder Ausgang verfügt über drei Modi: Ein, Aus und Leerlauf. Die Leistung, die im Ein-Modus abgegeben wird, kann außerdem von Stärke 1 bis 8 variiert werden. Dies geschieht über eine Pulsbreitensteuerung mit einer Periodendauer von 8 ms. Eine 2 bedeutet dementsprechend eine Impulsdauer von 2 ms und eine Pausendauer von 6 ms. Die Leistungseinstellung kann in jedem Modus geändert werden; sie wirkt sich aber nur aus, wenn der Motor eingeschaltet wird. Der Unterschied zwischen Aus und Leerlauf ist, dass bei Aus die beiden Ausgangskontakte kurzgeschlossen werden und somit bei Motoren ein schneller Halt erzwungen wird, während bei Leerlauf die Kontakte offen bleiben. Man kann Aus mit einem stehenden Auto mit eingelegten Gang und Leerlauf mit einem Auto ohne eingelegten Gang vergleichen, die Handbremse ist dabei gelöst (vergl. [Knu99b]). Angeschlossen werden die Aktoren über dieselben zweiadrigen Verbindungsleitungen mit ihren 2×2 Verbindungsstücken, wie sie bei den Sensoren üblich sind. Bei einem Motor bewirkt ein verdrehtes Aufsetzen auch eine Laufrichtungsänderung.
Desweiteren verfügt die RCX-Einheit noch über ein LC-Display, sowie einen IR-Sender/Empfänger und einen Signalgeber.

Anleitungen zum Selbstbau von Aktoren gibt es viel weniger als bei Sensoren, weil man mit den Standardaktoren schon sehr viele Roboter realisieren kann.

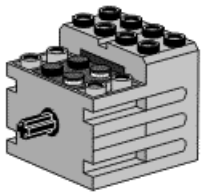


Abbildung 2.11: LEGO-Getriebemotor

Der LEGO-Motor

Im Gehäuse dieses Motors, der nicht nur bei LEGO-Mindstorms Verwendung findet, ist bereits ein Getriebe eingebaut. Er kann außerdem als Generator genutzt werden. Laut LEGO soll er einen Wirkungsgrad von 80 % haben und sehr gut für die Impulsbreitenansteuerung geeignet sein.

Hier die technischen Daten für den Motorbetrieb:

- 350 Umdrehungen in der Minute bei 9 V und Leerlauf
- 5 mA Leerlaufstrom
- 350 mA maximale Stromaufnahme (Stillstand durch Vollast)
- ab 1 V läuft der Motor an (ohne Last)

Für den Generatorbetrieb hat der Motor folgende Kennwerte

- maximal 350 Umdrehungen pro Minute (UpM)
- 9 Volt bei 350 UpM
- ca. 350 mA maximaler Kurzschlussstrom

Diese Werte stimmen, da es sich um nicht für den Dauerbetrieb geeignete Spielzeugmotoren handelt, später in der Praxis nur noch in der Größenordnung überein. Zu den Leerlaufdrehzahlen zweier Motoren hier eine Tabelle:

	Motor 1	Motor2
Upm Rechtslauf	307	327
Upm Linkslauf	301	329

Die LEGO-Lampe

Diese Lampe ist eine normale Glühbirne, die mit bis zu 9 Volt betrieben werden kann. Sie ist ausreichend hell, um den Lichtsensor ganz auszusteuern. Bei 9 V Nennspannung wurden 28,3 mA gemessen.

2.2 Software

2.2.1 Die Software der RCX-Einheit

Zuerst folgt ein kleiner Überblick über das Zusammenspiel der Softwarekomponenten auf der RCX-Einheit. Anschließend wird das Protokoll des Bootloaders zur Übertragung von Daten zwischen Zentraleinheit

und einem externen Partner kurz erklärt, um den Unterschied zum RoCK-Protokoll aufzuzeigen. Für detailliertere Angaben siehe [Prob]. Es handelt sich bei dieser Quelle um eine Analyse der Software der RCX-Einheit, die mittels „Reverse Engineering“ vorgenommen wurde.

Programmiertools der RCX-Einheit Zum Programmieren der RcX-Einheit stellt Lego eine auf die Zielgruppe abgestimmte graphische Programmiersprache zur Verfügung (siehe Abbildung 2.12). Aus dem graphischen Programm wird Byte-Code erzeugt, der von der RCX interpretiert wird.

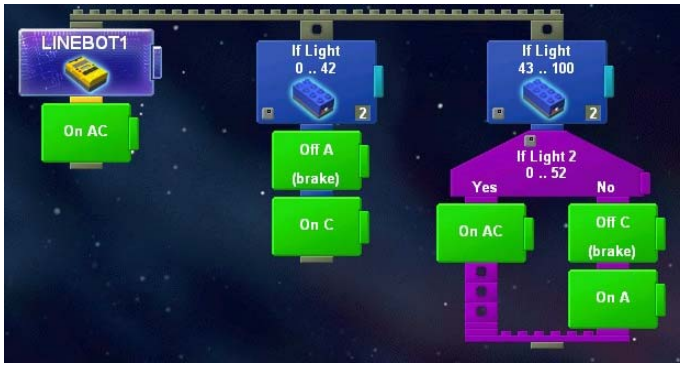


Abbildung 2.12: Screenshot eines mit LEGO-Software erstellten Programms

Als Alternative dazu hat sich NQC (Not Quite C) etabliert, welches sich in der Syntax an C anlehnt, aber längst nicht dessen Funktionsumfang hat.

Aufgabenverteilung bei der RCX-Einheit

Die Software der RCX-Einheit besteht aus einem Bootloader, der per Infrarot Daten empfangen kann, der Firmware und den Anwenderprogrammen (bis zu sechs Anwendungen sind möglich). Der Bootloader kann jegliche in der RCX-Einheit verwendete Hardware ansprechen und Daten erhalten, d. h. alle notwendigen Treiber sind bereits dort integriert. Dies ermöglicht sowohl visuelle als auch akustische Signalisierung, beispielsweise eines erfolgreichen Datentransfers, obwohl noch kein Betriebssystem (Firmware) eingespielt ist. Das Zusammenspiel von Bootloader und Firmware sieht wie folgt aus:
Die RCX-Einheit wird eingeschaltet und der Bootloader prüft anhand eines Bits, ob eine gültige Firmware vorhanden ist. Ist dies der Fall, wird die gesamte Hardware initialisiert und anschließend der Hauptthread der Firmware durch den Bootloader gestartet, der dann die sechs vorhandenen Handler der Reihe nach aufruft. Diese sechs Handler übernehmen daraufhin weitere Aufgaben, wie z. B. die Abfrage der Sensoren oder das Ansteuern von Aktoren.

- Handler 1: Sensoren
- Handler 2: Motoren (Aktoren)
- Handler 3: Schaltknöpfe und Anzeige
- Handler 4: Strom (Ein-/Aus-Schalter)
- Handler 5: Interpreter (für Anwenderprogramme und direkt vom PC bzw. der Fernbedienung übertragene Daten)
- Handler 6: Schaltet Bit 3 von Port 6 an/aus beim init/stop. (Der Zweck ist unbekannt, siehe auch [Prob])

Ist keine gültige Firmware vorhanden, wartet der Bootloader auf eingehende Daten, die mit dem nachfolgend beschriebenen Protokoll übertragen werden.

Das RCX-Protokoll

Das im Folgenden skizzierte Protokoll ist bereits im Bootloader implementiert und steht somit immer zur Verfügung, auch nach einem durch einen Batteriewechsel hervorgerufenen Stromausfall, bei dem die Firmware verloren geht. Empfangen werden die Daten über die Infrarot-Schnittstelle (siehe 2.1.2).

Das Paketformat: Ein Paket beginnt mit dem Header `55FF00`. Direkt danach folgt der Befehlscode, der aus einem Byte besteht, gefolgt von seinem Komplement. Daran schließt sich die Nachricht an, die von dem jeweiligen Befehl abhängig ist. Auf jedes einzelne Byte folgt die Übertragung seines Komplements. Zum Abschluss wird die Prüfsumme gesendet, ebenfalls mit Komplement.

Zusammengefasst sieht ein Paket also folgendermaßen aus (siehe auch [Prob]):
`55FF00'OPCODE'MESSAGE'CHECKSUM8`
Um Daten zur RCX-Einheit zu senden, wird nun dieses Paket übertragen und als Antwort wieder zurückgeschickt. Anschließend wird als Rückmeldung noch eine Nachricht übermittelt, die den entsprechenden Opcode aufweist und den Returncode als Message.

2.2.2 Schwächen der RCX / Mindstorms

Die LEGO-Programmiersoftware läuft ausschließlich unter Windows-Betriebssystemen ab Windows 98. Die Systemanforderungen werden mit einem Pentiumprozessor ab 233 MHz und 115 MB freier Festplattenkapazität angegeben. Erforderlich ist außerdem ein freier USB-Anschluss zur Ansteuerung des Infrarotsenders.

Zur Programmierung leistungsfähiger Steuerungen ist der RCX-Code (vgl. Kap. 2.2.1 auf Seite 20) von LEGO leider nicht geeignet, da der Code interpretiert wird.

Zusammenfassend lässt sich festhalten, dass die LEGO-Software für Windows auch und gerade für Menschen mit wenig Programmiererfahrung einen guten Einstieg in die Welt der Roboter bietet. Weitergehende Ansprüche erfüllt die Software allerdings nicht.

2.2.3 LegOS

LegOS (LEGO Operating System) ist eine speziell für die RCX-Einheit entwickelte Firmware in C und stellt somit eine Alternative zu NQC (siehe 2.2.4) oder der den MINDSTORM-Baukästen beiliegenden Entwicklungsumgebung dar.

Es handelt sich um ein kleines, auf den normalen GNU-Werkzeugen (binutils, gcc/g++) aufbauendes Betriebssystem, welches dynamisch Programme zuladen kann. Es ist sehr mächtig, aber auch recht schwer zu beherrschen, daher sind Vorkenntnisse der Sprache C dringend angeraten. LegOS bietet eine direkte Steuerung der Ein- und Ausgänge, des Displays, des Infrarotports und des RCX-Speichers. LegOS läuft nativ auf dem Prozessor der RCX.

Man benötigt in der Praxis ein Entwicklungswerkzeug bzw. den bevorzugten Editor und den Compiler für das Zielsystem – in diesem Falle egcs („Enhanced GCC“), den Open-Source-Nachfolger zu gcc. Auch unter Windows lässt sich mit LegOS arbeiten, sofern man mit der Cygwin-Software ein Linux-System emuliert. Auch ein Firmware-Uploader darf natürlich nicht fehlen.

Alternativ gibt es die interessante Möglichkeit, den Quellcode per Internet einzusenden und sich das compilierte Ergebnis zuschicken zu lassen. Mehr dazu unter [Nog].

Ein einfaches Beispiel

Wir setzen einen selbstgebauten Roboter voraus, bei dem an Aktor b der Antriebsmotor angeschlossen ist und an Sensor a ein Tastsensor. Nach Anzeige des Textes „Hello“ auf dem RCX-LC-Display wird der Roboter mit mittlerer Geschwindigkeit vorwärtsfahren, bis der Tastsensor gedrückt wird. Abschließend erscheint der aktuelle Ladezustand der Batterie auf dem RCX-Display.

```
#include 'conio.h'

int main(void) (
  cputs('Hello'); // ''Hello'' in die Anzeige
  lcd_refresh(); // und ausgeben
  delay(1000); // 1 Sekunde warten
  motor_a_dir(fwd); // Motor bewegt Roboter
  motor_a_speed(128); // mittlere Geschwindigkeit
  while (SENSOR_1>0x1000); // solange bis Taste
  motor_a_dir(off); // Motor aus
  cputs(BATTERY); // Batteriestand lesen
  lcd_refresh(); // und anzeigen
  delay(1000);
 return 0 ;
```

2.2.4 NQC

Idee und Möglichkeiten

Die Idee hinter der Entwicklung von NQC liegt in einem leistungsfähigen Ersatz der PC-Software, die mit dem Robot Construction Kit 1.5 ausgeliefert wird. NQC lehnt sich stark an die bekannte Programmiersprache C an, aber weil sie eben nicht das Gleiche ist wie C, heißt sie „not quite C“.

Wichtige Eigenschaften von NQC

Der NQC-Quellcode wird in Textdateien abgelegt, ähnlich wie bei C, C++ und Java. NQC compiliert diese Quelldateien in Bytecode und überträgt sie mittels des IR-Towers auf die RCX-Einheit.

NQC weist eine hohe Portabilität auf, weil hier direkt mit der IR-Einheit kommuniziert wird und der Programmierer nicht auf eine bestimmte Plattform angewiesen ist. NQC läuft sowohl auf MacOS, Linux und natürlich allen Windows-Derivaten. Benutzt man NQC unter Windows, so bietet sich die Verwendung des so genannten RCX Command Centers an, welches eine angenehme Programmierumgebung mit einem Editor (der Syntaxhighlighting für NQC unterstützt), Compilierung auf Knopfdruck und einfachem Upload bereitstellt, incl. Echtzeitsteuerung des RCX-Bausteins und weiteren nützlichen Eigenschaften.

NQC benutzt sogenannte Tasks zur Ausführung von Subroutinen, welche den z. B. aus JAVA bekannten Threads stark ähneln. Die Hauptroutine heißt main.

Für weitere Informationen über die vorgegebenen Routinen und Möglichkeiten von NQC siehe [Knu99a].

Beispielcode

Um die Unterschiede zwischen der LEGO-Oberfläche (vgl. 2.1.1 auf Seite 12) und NQC zu verdeutlichen, hier ein Code-Beispiel. Dabei handelt es sich lediglich um eine Demonstration der Funktionsweise von NQC, nicht um real erprobten Code. Der Kontext kann auf der Homepage von Dave Baum (siehe [Bau]) nachvollzogen und heruntergeladen werden.

```
#define TURNAROUND_TIME 425

int i;

task main() {
  // Arm limit sensor and grabber light sensor.
  SetSensor(SENSOR_3, SENSOR_LIGHT);
  SetPower(OUT_A + OUT_C, OUT_FULL);

  calibrate();
  i = 0;
  while (i < 5) {
    retrieve();
    i += 1;
  }
  Off(OUT_A + OUT_C);
}

#define NUMBER_OF_SAMPLES 10
int runningTotal;
int threshold;

sub calibrate() {
  // Take an average light reading.
  i = 0;
  runningTotal = 0;
  while (i < NUMBER_OF_SAMPLES) {
    runningTotal += SENSOR_3;
    Wait(10);
    i += 1;
  }
  threshold = runningTotal / NUMBER_OF_SAMPLES;
}

void grab() {
  // Run the motor until we hit the limit switch.
  OnFwd(OUT_A);
  until (SENSOR_3 == 100);
  // Back off from the switch.
  OnRev(OUT_A);
  until (SENSOR_3 != 100);
  Off(OUT_A);
}

void release() {
  // Run the motor until we hit the limit switch.
  OnRev(OUT_A);
  until (SENSOR_3 == 100);
  // Back off from the switch.
  OnFwd(OUT_A);
  until (SENSOR_3 != 100);
  Off(OUT_A);
}
```

Einschränkungen

Die NQC-Programmiersprache gibt Bytecode-Programme aus und ist somit an die Beschränkungen gebunden, die der Bytecode-Interpreter der Firmware auferlegt. Weil die originale RCX-Einheit vorausgesetzt wird, ist NQC somit an die Einschränkungen dieses Bauelements (z. B. drei Anschlüsse, fünf Programmspeicher) gebunden.

Verbesserungsmöglichkeiten

Auch die NQC-Programmiersprache ist den Beschränkungen durch den RCX-Baustein von LEGO unterworfen. Insbesondere schränkt den Programmierer die geringe Anzahl (jeweils drei) an Sensor- und Aktoranschlüssen und die Abhängigkeit von der verwendeteten Firmware in seiner Kreativität ein.

Wünschenswert wären an dieser Stelle einige der klassischen Hochsprachenelemente, die NQC leider nicht besitzt. Dazu gehören unter anderem

- mehr globale und lokale Variablen
- mehr Tasks
- Methoden mit Parameterübergabe und Rückgabewert
- Datenkapselung
- „echte“ Datenstrukturen

Der Nutzer sollte in die Lage versetzt werden, mit bekannten Mitteln (C, C++) eine leistungsfähige Robotersteuerung zu entwerfen, ohne – wie es hier der Fall ist – umfangreichen Einschränkungen unterworfen zu sein. Dem Entwurf und der Programmierung einer wirklich anspruchsvollen Robotersoftware liegt immer eine leistungsfähige Programmiersprache zugrunde.

2.2.5 Bestehende Compiler für die PIC-Familie

Eines der Hauptmerkmale der Anforderungen an die Projektgruppe sollte die Entwicklung eines Compilers sein, der ANSI-C Programme in die Maschinensprache des verwendeten Mikrocontrollers der PIC18-Familie übersetzt. Im Folgenden werden die wesentlichen Merkmale bereits existierender kommerzieller Compiler für die PIC-Prozessoren, die der Projektgruppe als Referenz-Compiler dienen sollen, aufgeführt. Der Sprachumfang des CCS C-Compilers ist ein früher C-Standard nach Kernighan und Ritchie. Demgegenüber unterstützt der HI-TECH Compiler den ANSI-C Standard.

	CCS - C V3.000	HI-TECH PICC V7.86
unterstützte Prozessoren	alle 12-Bit, 14-Bit Prozessoren und PIC18	alle 12-Bit, 14-Bit Prozessoren und PIC17, PIC18
MPLab integrierbar	ja	ja
RAM Allocation	Compiler weist Variable den verfügbaren Speicherbänken zu	Benutzer muss Speicherallokation über das Schlüsselwort BANK x regeln
Datentypen	Integer: 1 bit 8 bit signed/unsigned 16 bit signed/unsigned 32 bit signed/unsigned Floating point: 32 bit signed (Microchip Format)	Integer: 1 bit 8 bit signed/unsigned 16 bit signed/unsigned 32 bit signed/unsigned Floating point: 24 bit signed 32 bit signed (IEEE 754 Format)
Arraydimension	bis zu 5-dimensional	uneingeschränkt
Struct / Union	unterstützt	unterstützt
Pointer auf RAM	unterstützt, Bereich abhängig von Wahl des Pointers	Zugriff nur auf Daten der aktuellen Speicherbank
Pointer auf Funktionen	nicht unterstützt	unterstützt
Funktionsparameter	32	uneingeschränkt
Pointer auf Parameter	unterstützt	nicht unterstützt
Rekursion	nicht unterstützt	nicht unterstützt
Inline Assembly	unterstützt	mittels Makros
Standard Libraries	teilweise	teilweise
Interrupt Context Saving / Retrieval	unterstützt	unterstützt
separater Linker	integriert	benötigt

Als zusätzliche Besonderheiten der Implementierung des CCS Compilers lassen sich folgende Punkte aufzählen:

- Bibliotheken, die für alle PIC-Prozessoren nutzbar sind, darunter solche für die serielle Schnittstelle (RS 232), für Input / Output-Routinen, I^2C, Delays etc.
- CCS Compiler kann mit den Mathematikbibliotheken von Microchip eingesetzt werden
- formatierte printf ()-Ausgaben erlauben einfache Formatierung und Anzeige von Werten in dezimaler oder hexadezimaler Form
- Treiber für LCD-Module, Tastaturen, 24xx und 93xx serielle EEPROMs, RTCs, Touch Memories und A/D-Wandler werden als Quellcode ausgeliefert
- Hardwarefunktionen wie z. B. A/D, EEPROM, SSP, PSP etc. können über C-Funktionen angesprochen werden.
- Der CCS Compiler kann sowohl als integrierter Editor / Compiler, als auch als Kommandozeilen-Compiler eingesetzt werden und ist für Win9x/NT/2K erhältlich

Der HI-TECH Compiler erzeugt bei printf ()-Ausgaben im Gegensatz zum CCS sehr großen Code, gestaltet sich aber durch folgende zusätzliche Eigenschaften interessant:

- Multiplattformfähig – erhältlich für folgende Betriebssysteme: DOS, Win9x/NT/2K, Linux, UNIX

- unterstützt Bitfelder in Structures
- wird mit kompletter Laufzeitbibliothek im Source ausgeliefert

Kapitel 3

Problembeschreibung und Anforderungen

3.1 Minimalanforderungen

Wie bereits in Kapitel 1 auf Seite 11 beschrieben, ist unser Ziel der Entwurf eines kleinen Robotersteuerungssystems (ROCK). Schon beim ersten PG-Treffen war allen Teilnehmern klar, dass nicht nur das Minimalziel erreicht werden sollte. Einen Compiler, der eine Hochsprache in Assemblercode übersetzt, ist eine aufwendige Aufgabe, da aber weder die Abfrage eines Sensors oder das Steuern eines Motors aufwendig oder gar interessant ist, wurden die Anforderungen erweitert und zum größten Teil auch realisiert.

3.2 Erweiterung der Minimalanforderung

In Kapitel 1 auf Seite 9 wurde die RCX-Einheit von LEGO erwähnt. Deren starke Einschränkungen im Bereich Anschluss von Aktoren und Sensoren und der damit verbundenen Problematik, umfangreiche Aufgaben zu lösen, brachte dann die Idee für das neue Ziel: Ersetzen der RCX-Einheit von LEGO durch eine kleinere, modular aufgebaute und energetisch effizientere Einheit. Daraus ergaben sich folgende Aufgaben:

- Aussuchen und Beschaffen der benötigten Hardware (Prozessor, Kondensatoren, ...)
- Erstellen der einzelnen Komponenten des ROCK
- Compilerentwicklung
- Betriebssystem

Im Folgenden wird beschrieben, was bei einem Robotersteuerungssystem zu beachten ist.

3.3 Robotersteuerungssoftware

In diesem Abschnitt sollen die allgemeinen Merkmale einer Robotersteuerungssoftware (RSS) und deren Auswirkungen auf unser Projekt „Robot Construction Kit“ betrachtet werden.

Bequeme Bedienung

Die RSS sollte es erlauben, die Betriebsmodi des Roboters (zum Beispiel Ein- oder Ausschalten, Auswahl des zu startenden Programms oder interaktive Sensorabfragen) einfach und unabhängig vom PC zu wechseln. Dazu erscheint ein LC-Display (siehe Abschnitt 3.8 auf Seite 31) mit einfacher Menüführung sinnvoll.

Einfache – aber flexible – Programmierung

Es ist vorteilhaft, Robotersoftware am PC zu entwickeln, weil dieser über eine hohe Rechenleistung sowie vertraute, komfortable Eingabegeräte und ausgereifte Software zur Erleichterung von Entwicklung, Planung und Projektmanagement verfügt. Die RSS sollte hier am besten eine schnelle, drahtlose Möglichkeit der Übertragung auf den Roboter, z. B. per Infrarotschnittstelle, ermöglichen. Ein weiterer Vorteil der Infrarottechnik ist die galvanische Trennung.

Für den Anwender einer RSS ist es i. d. R. nicht von Interesse, sich eingehend mit der Hardware des Roboters zu befassen. Daher sollte der Roboter in einer Hochsprache programmierbar sein, wobei auch Treiber und Bibliotheken sowie eine API (Schnittstelle) mitgeliefert werden müssen.

Wegen der eingeschränkten Ressourcen des Roboters sollten die Anwendungen wenig Hauptspeicher verbrauchen und möglichst wenig Overhead beinhalten. Daher sollte kein Interpreter oder gar eine virtuelle Maschine verwendet werden, sondern die Maschinensprache des zugrunde liegenden Prozessors. Eine besondere Herausforderung bietet hierbei der Versuch, vorhandene Algorithmen auf den Roboter zu portieren.

Nebenläufigkeit

Schon die einfachsten Experimente mit LEGO-Robotern, wie z. B. das Folgen einer schwarzen Linie auf hellem Untergrund oder das Erkennen und Umfahren von Hindernissen, erfordern die simultane Abfrage mehrerer Sensoren und Ansteuerung von Aktoren. Verfügt die RSS über Multitasking, wird die Programmierung solcher Aufgaben stark vereinfacht.

Kurze Reaktionszeiten auf Sensoren und genaue Ansteuerung von Aktoren

Erstrebenswert ist eine schwache Echtzeitfähigkeit, d. h. Messungen und Steuerbefehle sollten innerhalb einer kurzen Zeitschranke verarbeitet werden. Nur so kann eine hinreichende Genauigkeit erreicht werden, wie sie die Navigation in unbekanntem Terrain oder das Greifen von Gegenständen erfordert.

3.4 Kompatibilität zu LEGO-Mindstorms

Aufgrund des Ziels, die RCX-Einheit von LEGO zu ersetzen, müssen wir LEGO-kompatibel sein, was nun näher erläutert wird.

Bauformen und Anschlüsse

Das ROCK-System soll in seinen Bauformen voll kompatibel zu LEGO werden. Durch den Einbau in entsprechend modifizierte LEGO-Gehäuse, z. B. von Batteriekästen, kann dieses Ziel einfach realisiert werden. Auch sollen alle Verbindungen bei den neuen Komponenten mit den zweiadrigen LEGO-Leitungen ausgeführt werden. Dies macht, neben der Verwendung von 9 V Betriebsspannung, auch einen Verpolungsschutz an jeder Schnittstelle erforderlich.

Betrieb von Aktoren

Aktoren sollen genau wie im LEGO-Mindstorms-System ansteuerbar sein. Bei Motoren kommt zusätzlich die Regelung der Laufrichtung hinzu. Zu einer Pulsweitenmodulation gibt es bei einer Gleichstromversorgung keine sinnvollen Alternativen. Hierbei sollten mindestens so viele Leistungsstufen wie bei LEGO vorhanden sein.

Betrieb von Sensoren

Alle passiven LEGO-Sensoren sollen unterstützt werden. Der Lichtsensor ist, da er seine Messwerte nur in Form von veränderlichen Widerständen kodiert, problemlos. Beim Rotationssensor ergeben sich aufgrund der Kodierung seines Messwertes in Widerstand und Zeit einige Probleme. Eine volle Unterstützung dieses Sensors wird aber trotzdem angestrebt. Zusätzlich ist geplant, weitere komplexere Sensoren, auch solche mit einer digitalen Schnittstelle, zu unterstützen. Der Roboter soll mit einem Sonar den Abstand zu Objekten messen können.

3.5 Modularität

Eine Robotersteuerung besteht in der Regel im Wesentlichen aus den folgenden Baugruppen:

- Zentrale Steuerung
- Leistungselektronik zur Motorsteuerung
- Ansteuerung der Sensoren
- Kommunikationsschnittstellen
- Ein- und Ausgabeschnittstellen

Die RCX-Einheit integriert zusätzlich zu diesen Baugruppen den Batterieraum in ihrem Gehäuse.

Ein Ziel besteht u. a. darin, die Modularität der Steuerung zu verbessern. Als Teil des Minimalziels soll das Batteriegehäuse außerhalb der zentralen Steuerung untergebracht werden. Insbesondere sind möglichst alle oben identifizierten Baugruppen in kleine, gut platzierbare Gehäuse einzubauen.

Zur Bereitstellung von Grundfunktionen soll sich die zentrale Steuerung auf eine begrenzte Anzahl an Sensorschnittstellen und Motorausgängen sowie Taster beschränken. Entsprechend ist für alle weiteren Funktionseinheiten, sowie zusätzliche Schnittstellen zu Motoren und Sensoren, die Realisierung durch kleine Baugruppen geplant. Ein Eindrahtkommunikationsbus übernimmt die Stromversorgung. Sofern nur zwei Leitungen verwendet werden, ist es möglich, alle Steckverbinder LEGO-kompatibel auszuführen. Hierbei können dann an einem Anschluss mehrere funktionale Einheiten betrieben werden. Möglich ist auf diese Art auch eine separate Stromversorgung bei großem Strombedarf einzelner Komponenten.

3.6 Betriebssystem

Eine wichtige Aufgabe des Betriebssystems (SOUL = **S**imple **O**perating system for **u**se with **L**EGO) ist es, den vorhandenen Bus (ROCKWIRE) und die daran angeschlossenen Geräte zu kontrollieren. Besonders ist darauf zu achten, dass die Latenzzeit über ROCKWIRE sehr gering ist, da sonst eine sinnvolle Steuerung eines Roboters nicht möglich ist. Weitere Anforderungen an SOUL ergeben sich aus dem Abschnitt 3.3 auf Seite 28.
Die Realisierung ist in Kapitel 4.5.1 auf Seite 76 beschrieben.

3.7 Kommunikation - ROCKWIRE

Das ROCKWIRE soll die Kommunikation zwischen der zentralen Steuereinheit (Master) und den anderen Baugruppen wie etwa LCD, Sensoren, Aktoren, ... (Slaves) ermöglichen. Es soll eine mastergesteuerte bidirektionale Kommunikation ermöglicht werden. Die Übertragungsgeschwindigkeit soll ausreichend kurze Latenzzeiten und hohe Datenübertragungsraten bieten, so dass die zentrale Steuereinheit auf alle Ereignisse angemessen und vor allem rechtzeitig reagieren kann.

Das Protokoll muss so gestaltet sein, dass es auch mit den beschränkten Ressourcen der PIC12 (siehe Abschnitt 4.4.2) Controller implementiert werden kann. Es ist weiter zu berücksichtigen, dass die PIC12 in der Regel nur ein RC-Glied als Taktgenerator besitzen.

Für Slaves mit geringem Stromverbrauch soll der kurzschlusssichere Bus auch die Stromversorgung übernehmen. Die Verbindung der Komponenten geschieht mittels Kabeln von LEGO, die sich einfach beliebig übereinander stecken lassen.

Das ROCKWIRE-Protokoll (siehe Abschnitt 4.3) ist die zentrale Idee zur Erfüllung der Anforderung, die beschränkte Anzahl an Ein- und Ausgängen üblicher Steuerungen zu überwinden.

3.8 Das LC-Display

In einen separaten LEGO-Baustein soll eine Flüssigkristallanzeige (LCD) integriert werden. Mit dieser Anzeige soll es unter anderem möglich sein, Status- und Fehlermeldungen auszugeben, den aktuellen Batteriestand abzufragen, sowie die momentanen Sensordaten anzuzeigen. Mit Hilfe von Drucktastern wird es ermöglicht, zwischen verschiedenen Betriebszuständen des LCDs zu wechseln.

3.9 Zusätzliche Sensoren

Die von LEGO erhältlichen Sensoren gestatten es Robotern nur beschränkt, ihre Umgebung effektiv und umfassend zu erforschen. Ein zentrales Problem der Drop-Zone-Mission ist aber, dass der Roboter ein Bild von seiner Umwelt hat, um seine Mission angemessen oder gar überhaupt zu erfüllen. Daher gehört der Bau von zusätzlichen Sensoren zu den weiterführenden Zielen dieser PG.

Ein wünschenswerter Sensor, der die Drop-Zone-Mission wesentlich vereinfacht, ist ein Sonarsensor. Der Roboter erhält damit die Möglichkeit, Hindernisse gezielt aufzuspüren oder sie zu umfahren, da ihre Position bekannt ist. Auch das Auffinden der Cola-Dose wird so sicher vereinfacht.

Das grundlegende Prinzip dabei ist, dass ein Ultraschallsender einen Impuls aussendet, die von einem Ultraschallempfänger aufgenommene Antwort im Frequenzbereich analysiert und so den Abstand zu Objekten misst.

Auf [Ben02] wird ein bildgebendes Sonar, das zwei PICs benutzt, beschrieben. Ferner erläutert [Wiz02], wie eine Abstandsmessung realisiert werden kann.

3.10 Der Aufbau des Gesamtsystems

In diesem Abschnitt wird die Anforderung beschrieben, wie ein Hochsprachenprogramm in dem System verarbeitet werden soll. Der Aufbau des Systems wird in Abbildung 3.1 auf der nächsten Seite in seiner Grundstruktur gezeigt. Der Quellcode wird mittels des Compilers in Assemblercode überführt und anschließend auf dem PC mit einem Assembler in Maschinencode übersetzt. Mit einem Kommunikationsprogramm wird dann der erzeugte Maschinencode über die RS232-Schnittstelle an die ROCK-Einheit

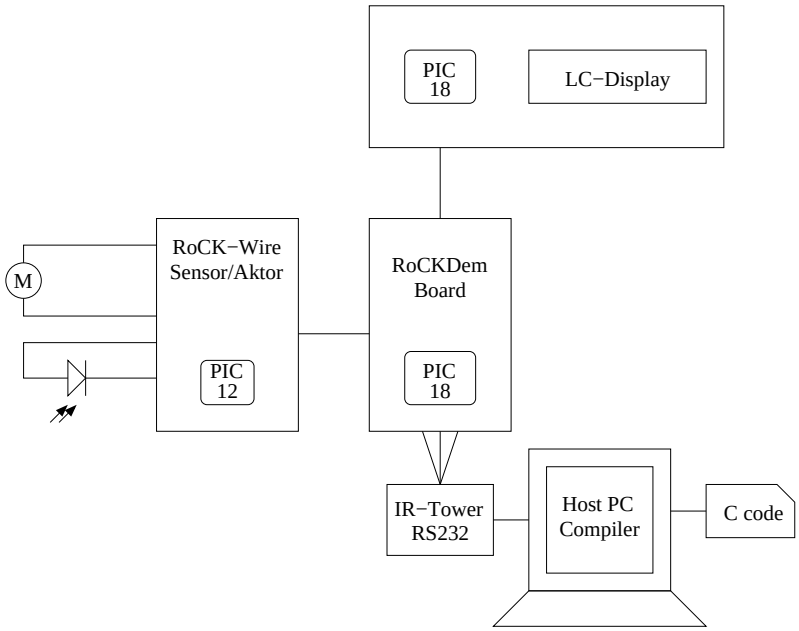


Abbildung 3.1: Komponenten des Robot Construction Kits

(auch Zentraleinheit oder RoCKDasBoard genannt) transferiert. Auf der ROCK-Einheit führt das Betriebssystem SOUL das übermittelte Programm aus. Das Programm kann dadurch auf per ROCKWIRE verbundene Komponenten zugreifen. Zwei Typen von ROCKWIRE-Geräten, der Sensor- und der Aktor-Slave sind links im Bild mit nur einem Symbol dargestellt. Oben erkennt man das LC-Display, welches ebenfalls über ROCKWIRE angeschlossen wird.

3.11 Der Compiler

Als eines der Minimalziele dieser Projektgruppe wird die erfolgreiche Übersetzung und Ausführung eines C-Programmes, welches Sensorwerte abfragt und Aktoren ansteuert, gefordert. Hierzu bietet der Lehrstuhl Informatik XII, der über langjährige Erfahrung mit der Entwicklung von Compilern für digitale Signalprozessoren verfügt, eine Reihe von Hilfsmitteln zur Entwicklung von ANSI-C Compilern an, die im Rahmen der PG zum Übersetzerbau verwendet werden. Als Compiler-Frontend wird der PG ein am Lehrstuhl XII entwickeltes ANSI-C Compilersystem zur Verfügung gestellt. Generell werden im Frontend sowohl Analysen des Sourcecodes durchgeführt, als auch eine Zwischendarstellung (Intermediate Representation) in Form von 3-Adress-Code (3-AC) generiert. Zwischendarstellungen, die das Austauschformat für Optimierungen und die Codegenerierung sind, werden maschinenunabhängig an das Backend übergeben. Die eigentliche Anforderung an die PG, war die Entwicklung eines Backends, welches anhand von Informationen über die Zielarchitektur maschinenabhängige Zwischendarstellungen (Low-Level Intermediate Representation) erzeugt, bei denen das zu übersetzende Programm durch Vorstufen von realen Maschineninstruktionen dargestellt wird. Die wesentlichen Aufgaben des Backends sind Codeselektion, Registerallokation sowie architekturspezifische Optimierungen auf erzeugtem Code. Somit ergeben sich die folgende Teilaufgaben:

- Erstellen einer Baumgrammatik, aus der ein Code-Generator-Generator einen Codeselektor erzeugt
- Implementieren einer Registerallokation, die an die spezielle Architektur des PIC-Prozessors angepasst ist
- Nach-Optimierung des erzeugten Codes

- Implementieren eines Backends, welches die komplette Übersetzung des 3-AC in Maschineninstruktionen durchführt

3.12 Benchmark: Drop-Zone-Mission

Die Drop-Zone-Mission ist eine aus den Aufgaben des Pathfinders [NAS] kreierte Mission, bei der es darum geht, einen Roboter zu konstruieren, der in der Lage ist, in einer ihm zunächst unbekannten und unbeschränkten Umgebung ein Objekt zu finden und aufzunehmen. Dieses so genannte Payload muss dann in eine Drop-Zone gebracht werden. Dabei hat der Roboter Hindernisse von Payloads zu unterscheiden und zu umfahren. In dieser allgemeinen Form ist die Mission sehr schwer zu lösen, da es z. B. keine Größenbeschränkung der Umgebung gibt.

Wir nutzen die nach Vorgaben von LEGO ein wenig vereinfachte Mission als Benchmark für unser ROCK. Das gesamte Gebiet, in dem der Roboter abgesetzt wird, hat einen weißen Untergrund und wird durch eine schwarze Umrandung begrenzt. Die Hindernisse (drei Stück) sind in ihrer Form, ihrer Farbe und auch in ihrem Material nahezu beliebig, es sollten jedoch keine leichten ferromagnetischen Gegenstände sein. Wir verwenden in unserer Präsentation kleine Blumentöpfe. Beim Payload (drei Stück) handelt es sich um eine Blechdose (z. B. eine Cola-Dose). Die Drop-Zone ist eine grüne Fläche, die der Größe eines DIN-A4-Blattes entspricht und sich in geringem Abstand von der Mitte einer der beiden kürzeren Strecken innerhalb der schwarzen Umrandung befindet.

Mit diesem Benchmark wollen wir unsere Zentraleinheit mit der aus dem LEGO-Mindstorms-Set vergleichen, mit dem Ziel diese Mission möglichst schneller und effizienter zu bewältigen.

Kapitel 4

Ergebnisse

4.1 Designfluss

4.1.1 Hardware

Zunächst wurden die Microcontroller PIC18F452 (Kapitel 4.4.1), PIC12F675 (Kapitel 4.4.2) sowie weitere Bauteile, wie z.B. MOSFETs, gekauft. Damit wurden einfache Versuchsschaltungen aufgebaut; die erste Schaltung war ein noch vorhandener PIC16, der zwei Leuchtdioden blinken lassen konnte.

Anschließend wurden grobe Schaltungsideen entwickelt und inkrementell in den Versuchsplatinen verwirklicht. Parallel wurden teile des PICDEM 2 Plus-Demonstration-Boards (Kapitel A.1.5) nachgebaut und Sensor- (Kapitel 4.2.3), Motor- (Kapitel 4.2.4), ROCKWIRE-Master- (Kapitel 4.3.2), ROCKWIRE-Slave- (Kapitel 4.3.2) sowie IR-Versuchsplatinen (Kapitel 2.1.2) mit Wirewrap-Technik aufgebaut. Für den Nachbau des PICDEM-Boards, aus dem nach mehreren Umbauten die Zentraleinheit entstanden ist, konnte kein einheitlicher Name gefunden werden: RoCKDasBoard oder RoCKDem-Board sowie Zentraleinheit sind synonym verwendet worden.

Die einzelnen Versuchsplatinen wurden nach ersten Tests zunächst mit Flachbandkabeln verbunden, um die Software integriert zu testen.

Nachdem die Integration der Versuchsplatinen abgeschlossen und jede Baugruppe getestet war, wurden die Schaltpläne verifiziert. Aus den Schaltplänen wurden die Platinenlayouts geroutet, die Platinen wurden von MV-PCB [M&] gefertigt.

Die Platinen wurden anschließend bestückt und in Betrieb genommen. Damit konnten zum ersten Mal alle Baugruppen gleichzeitig und in vollem Umfang getestet werden.

Parallel zur Bestückung der Platinen wurden verschiedene Roboterkonstruktionen erprobt.

4.1.2 Software

Das geplante Betriebssystem, das später SOUL (Kapitel 4.5.1) getauft wurde, musste parallel zur Hardware entwickelt werden. Hier kamen der Gruppe das PICDEM 2 Plus-Demonstration-Board (Anhang A.1.5) der Firma Microchip und die Application-Notes [Micd] zugute. Zu späteren Zeitpunkten sollte SOUL dann zunächst auf dem Evalution-Board (Kapitel 4.1.1) und letzendlich auf der fertigen Zentraleinheit (Kapitel 4.2.1) und dem LCD-ROCKWIRE-Slave (Kapitel 4.3.2) laufen.

Es wurde davon ausgegangen, dass sich die zu entwickelnden Hardware-Baugruppen, die mit SOUL ausgestatten werden sollten, in der Konfiguration der verwendten Pins stark unterscheiden, wobei aber auch

eine gewisse Ähnlichkeit durch die PIC-Architektur vorgegeben ist.

Da es sich bei dem PIC-Assembler um einen Makro-Assembler handelt, konnte über Defines und konditionelle Assemblerdirektiven erreicht werden, dass sich alle SOUL-Varianten aus einem Quellcode erzeugen lassen. (Abbildung 4.1)

```
; Which board? Only one should be defined!
;#define DEMOBOARD
;#define EVALUATIONBOARD
#define LCDPIC

; set the following define to specify the clock in MHz
; do not forget the radix
#define F_OSC_MHZ      .10

; I^2C Support: RoCKDem and LCD-Board are connected through I^2C
#ifdef  LCDPIC
        #define I2C_SLAVE
#else
        #define I2C_MASTER
#endif
```

Abbildung 4.1: Auszug aus define.inc

Der Bootloader und die Firmware-Flash-Funktionalität ist bei allen Versionen von SOUL und allen SOUL-betriebenen Baugruppen identisch, dadurch konnte die in JAVA geschriebene Flash-Applikation ROCK-Comm (Kapitel 4.5.2) stets unverändert bleiben.

Parallel zu allem wurde ein C-Compiler (Kapitel 4.6) entwickelt, dessen generierter Code während des gesamten Entwurfszyklus auf einem Simulator und schließlich auf der fertigen Hardware getestet wurde.

4.2 Realisierte Baugruppen

4.2.1 Zentraleinheit

Die Zentraleinheit entstand aus dem PICDEM 2 Plus-Demonstration-Board. Zunächst wurden vier Taster, vier Leuchtdioden und der Quarzschwingkreis übernommen. Anhand der festgelegten Pinbelegung wurden die anderen Komponenten angeschlossen.

Auf der Platine der Zentraleinheit wurden folgende Baugruppen untergebracht:

- Eine RS232-Schnittstelle
- Eine Sensorschaltung für drei Sensoren
- Zwei ROCKWIRE-Master
- Zwei H-Brücken zur Steuerung von zwei LEGO-Motoren
- einen ICSP-Port, der über einen Adapter angeschlossen werden kann
- Vier Leuchtdioden
- Ein Reset-Knopf
- Zwei Taster
- Spannungsversorgung und Oszillator

Später wurde auch der I^2C-Bus herausgeführt. Da das Sonar Spitzenströme von mehr als 100 mA benötigt, wurde der SMD-Linearregler 78L05 durch einen bedrahteten 7805 ersetzt.

Die bestückte Platine wurde in zwei LEGO 9V-Gehäuse eingepasst. Abbildung 4.2 zeigt einen Blick in das Gehäuse und die aufgebaute Platine.

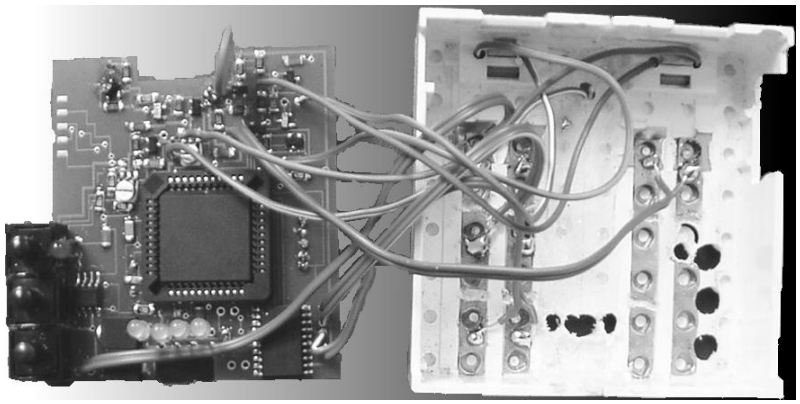


Abbildung 4.2: Die Zentraleinheit

Spannungsversorgung

Die Zentraleinheit benötigt für das ROCKWIRE, die Motoren und die H-Brücke 9 V und für die Logik-ICs 5 V. Die Spannungsversorgung erfolgt über Batterie.

Ein 9 V Batteriekasten wird über ein LEGO-Kabel an die Zentraleinheit angeschlossen. Diese Verbindung muss natürlich verpolungssicher sein. Da eine normale Graetz-Brücke einen zu hohen Spannungsabfall verursacht, wurde mit vier Schottky-Dioden mit sehr geringem Spannungsabfall die Verpolungssicherheit hergestellt.
Die 5 V Spannung wird über einen Linearregler 7805 erzeugt.

Das ICSP

Der PIC-Prozessor kann über einen ICSP-Port (In Circuit Serial Programming) mit dem ICD2 (Kapitel A.2.1 auf Seite 121) programmiert und debugt werden. Das ICD2 wird dazu über einem einen RJ12 Stecker und einer entsprechenden WESTERN-Buchse an das Board angeschlossen. Jedoch wurde aufgrund der Größe der WESTERN-Buchse darauf verzichtet, diese fest auf den Platinen zu montieren. Statt dessen wurden ihre Kontakte über eine kleine Steckerleiste aus der Schaltung herausgeführt, an die dann der Programmierstecker angeschlossen werden kann. Beim Letztgenannten handelt es sich um einen eigens angefertigten Adapter.

Für ICSP werden folgende Leitungen benötigt:

- Programmierspannung +12 V (MCLR invertiert)
- Betriebsspannung +5 V
- Masse
- Daten
- Clock

Die Pinbelegung für den PIC18F452 des RoCK-Bards ist der Tabelle 4.1 und dem Schaltplan D.2 auf Seite 138 zu entnehmen, bzw. für den PIC18F452 des LCD-Boards, der Tabelle 4.1 und dem Schaltplan D.1 auf Seite 137.

Pinbelegung des RoCK-PIC

Das RoCK-Board verfügt über drei Sensoreingänge, die an den A/D-Wandler angeschlossen werden mußten und einen einzelnen Sensorausgang, der beliebig gewählt werden konnte. Die zwei Aktoren benötigten jeweils drei Ausgänge, davon jeweils einen für Richtung und für Stop sowie einen Ausgang für die Steuerung der Motorleistung. Für diese wurde die PIC Pulsweitenmodulation benutzt. Die RoCKWire-Anschlüsse benötigen jeweils einen Eingang mit der Eigenschaft eines Interrupt on change und zwei Ausgänge, die jedoch beliebig gewählt werden konnten.

Der PIC18F452 bietet zudem weitere Funktionen, die von uns verwendet wurden, z. B. Anschlüsse für In-Circuit Debugging und In-Curcuit Serial Programming mit dem ICD2 und MPLAB (siehe A.2.1 auf Seite 121). Außerdem bietet der PIC18F452 eine Schnittstelle für die serielle Kommunikation: USART. Es wird lediglich noch ein RS232-Baustein benötigt, der den -12 V Pegel für RS232 erzeugt. Der I^2C Anschluss kann z. B. für die Ansteuerung eines Sonarmoduls (siehe 4.4.6 auf Seite 74) verwendet werden. Die Belegung für zwei Bedienungstaster wurde von dem PICDEM 2 Board übernommen (siehe A.1.5 auf Seite 120).

Diese Tabelle listet alle Pins des PIC18F452 auf. In der Spalte „Belegung“ findet man die von uns gewählte Anschlussbelegung für das RoCK-Board, zusammen mit der von diesem Pin erfüllten Funktion. In der Spalte „PIN-Funktionen“ findet man die zusätzlichen Funktionen, die an diesem Pin zur Verfügung stehen. Die Pin-Nummer bezieht sich auf die Nummer des jeweiligen Pins in dem von uns benutzten PLCC-Gehäuse.

Tabelle 4.1: Pinbelegung des PIC18F452 auf dem RoCK-Board

PIN-Name	PIN-Nr.	Belegung	PIN-Funktionen
RA0	3	Sensor 1 Input: SI1	Analog input 0
RA1	4	Sensor 2 Input: SI2	Analog input 1
RA2	5	–	Analog input 2 A/D reference voltage (LOW)
RA3	6	–	Analog input 3 A/D reference voltage (HIGH)
RA4	7	Switch 1	Timer0 external clock input
RA5	8	Sensor 3 Input: SI3	Analog input 4 SPI slave select Low voltage detect input
RB0	36	Switch 2	External interrupt 0
RB1	37	Aktor 2 Richtung: R2	External interrupt 1
RB2	38	Aktor 1 Richtung: R1	External interrupt 2
RB3	39	Aktor 1 Stop: S1	CCP 2
RB4	41	RoCKWire 1 Input: RI1	Interrupt on change
RB5	42	RoCKWire 2 Input: RI2 (LCD)	Interrupt on change Low voltage programming enable
RB6	43	ICSP Clock	In Circuit Debugger ICSP Clock
RB7	44	ICSP Data	In Circuit Debugger ICSP Data
RC0	16	Aktor 2 Stop: S2	Timer1 oscillator output
wird fortgesetzt			

Fortsetzung			
PIN-Name	**PIN-Nr.**	**Belegung**	**PIN-Funktionen**
			Timer1/Timer3 external clock input
RC1	18	Aktor 1 PWM: P1	Timer1 oscillator input CCP2
RC2	19	Aktor 2 PWM: P2	CCP1
RC3	20	Sonar	sync. serial clock for SPI input/output sync. serial clock for I^2C input/output
RC4	25	Sonar	SPI Data in I^2C I/O
RC5	26	CTS	SPI Data out
RC6	27	TX	USART asynchronous transmit USART synchronous clock
RC7	29	RX	USART asynchronous receive USART synchronous data
RD0	21	LED 1	PSP data port
RD1	22	LED 2	PSP data port
RD2	23	LED 3	PSP data port
RD3	24	LED 4	PSP data port
RD4	30	RoCKWire 1 Output: RO1	PSP data port
RD5	31	RoCkWire 1 Power: RP1	PSP data port
RD6	32	RoCKWire 2 Output: RO2	PSP data port
RD7	33	RoCKwire 2 Power: RP2	PSP data port
RE0	9	Sensor 1 Output: SO1	Analog input 5 PSP read control
RE1	10	–	Analog Input 6 PSP write control
RE2	11	–	Analog Input 7 PSP chip select control

4.2.2 Das LCD-Board

Im LCD-Slave steuert ein PIC18 die eingebauten Controlle des LCD (siehe Abschnitt 4.4.5). An den PIC18 wurde eine ROCKWIRE-Slave Baugruppe angeschlossen.

Die bestückte Platine wurde in zwei LEGO-9V Gehäuse eingepaßt. Dazu wurde ein Gehäuse halbiert und an das andere geklebt, so dass das Gehäuse eine Grundfläche von 4×12 Noppen hat. Abbildung 4.3 zeigt die bestückte Platine.

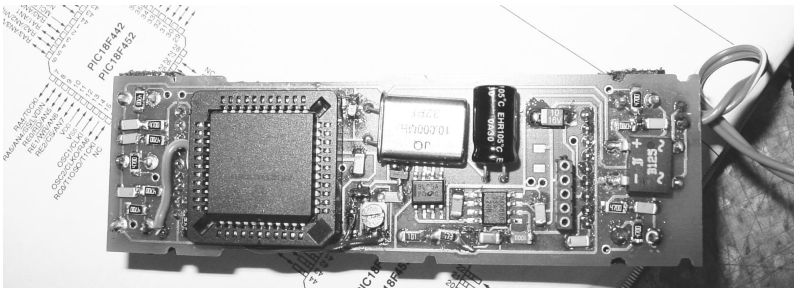


Abbildung 4.3: Die aufgebaute LCD-Platine

Der LCD-Treiber (siehe Abschnitt 4.5.3) kann das Display ansteuern. Über das ROCKWIRE können Kom-

mandos zur Darstellung übertragen werden.

Spannungsversorgung

Das LCD-Board wird über das ROCKWIRE mit Spannung versorgt, dieses liefert 9 V, ein Linearregler reduziert diese auf die benötigten 5 V. Aufgrund der Hintergrundbeleuchtung des LCDs nimmt das LCD-Board viel Leistung auf. Während der Kommunikation zwischen ROCK-Board und LCD-Board liegt jedoch auf dem ROCKWIRE keine Versorgungsspannung an, (siehe 4.3 auf Seite 47). Um während dieser Zeit die Versorgung des LCD-Boards sicher zu stellen, wurde ein großer Kondensator verwendet.

Die benötigte negative Kontrastspannung für das LCD wird mit Hilfe eines MAX850 ICs [Max] erzeugt.

Pinbelegung des LCD-PIC

Auf dem LCD-Board wurde ein EA DIP122-5NLED Display verwendet (siehe 4.4.5 auf Seite 69). Für die Ansteuerung des LCDs wurden 13 Pins verwendet. Da die Interaktion des Roboters mit dem Anwender über das LCD und den LCD-PIC erfolgen soll, wurden dafür, neben dem Reset-Taster, vier weitere Bedienungstasten vorgesehen. Außerdem wurden Pins für den ROCKWIRE-Slave Anschluß und ICSP benötigt. Siehe dazu auch die Pinbelegung des ROCK-Boards (4.2.1 auf Seite 37).

Tabelle 4.2: Pinbelegung des PIC18F452 auf dem LCDBoard

PIN-Name	PIN-Nr.	Belegung	PIN-Funktionen
RA0	3	–	Analog Input 0
RA1	4	–	Analog Input 1
RA2	5	Switch 1	Analog input 2 A/D reference voltage (LOW)
RA3	6	Switch 2	Analog input 3 A/D reference voltage (HIGH)
RA4	7	Switch 3	Timer0 external clock
RA5	8	Switch 4	Analog input 4 SPI slave select Low voltage detect input
RB0	36	–	External Interrupt 0
RB1	37	–	External Interrupt 1
RB2	38	–	External Interrupt 2
RB3	39	–	CCP2
RB4	41	RoCKWire Output: RO	Interrupt on change
RB5	42	RoCKWire Input: RI	Interrupt on change Low voltage programming enable
RB6	43	ICSP Clock	In Circuit Debugger ICSP Clock
RB7	44	ICSP Data	In Circuit Debugger ICSP Data
RC0	16	LCD A0	Timer1 oscillator output Timer1/Timer3 external clock input
RC1	18	LCD Reset	Timer1 oscillator input CCP2
RC2	19	–	CCP1
RC3	20	–	sync. serial clock for SPI input/output
wird fortgesetzt			

Fortsetzung

PIN-Name	PIN-Nr.	Belegung	PIN-Funktionen
			sync. serial clock for I^2C input/output
RC4	25	–	SPI Data in I^2C I/O
RC5	26	–	SPI data out
RC6	27	–	USART asynchronous transmit USART synchronous clock
RC7	29	–	USART asynchronous receive USART synchronous data
RD0	21	LCD D0	PSP data port
RD1	22	LCD D1	PSP data port
RD2	23	LCD D2	PSP data port
RD3	24	LCD D3	PSP data port
RD4	30	LCD D4	PSP data port
RD5	31	LCD D5	PSP data port
RD6	32	LCD D6	PSP data port
RD7	33	LCD D7	PSP data port
RE0	9	LCD R/W	Analog input 5 PSP read control
RE1	10	LCD E1	Analog Input 6 PSP write control
RE2	11	LCD E2	Analog Input 7 PSP chip select control

4.2.3 LEGO-Sensor-Schnittstellen

Die RoCK-Steuerung bietet die Möglichkeit, LEGO-Sensoren zu benutzen. Aus diesem Grund entstand ein Sensor-Hub, welcher bis zu drei Sensoren mittels eines PIC12 ausliest und über eine ROCKWIRE-Slave-Schnittstelle verfügt.

Elektrische Eigenschaften der LEGO-Sensoren

Zu den elektrischen Eigenschaften der LEGO-Sensoren gibt es umfangreiches Material im WWW, so beispielsweise auf [Gas]. Auf einige Besonderheiten, die in die Designziele für die Sensorschaltung einflossen, soll hier kurz eingegangen werden.

- Am A/D-Wandler des PIC darf keine zu große Spannung anliegen (V_{dd} + 0,5 V)
- Der Sensor wird über einen 220 Ω-Widerstand mit 8 V verbunden, wenn nicht gemessen wird
- Der Sensor wird zur Messung über einen 10 kΩ -Pull-up-Widerstand mit 5 V verbunden

Die Hardware für den RoCKSensor-Hub

Im Folgenden wird der Schaltplan in Abbildung D.7 auf Seite 143 näher beschrieben.

Damit am Eingang eines PIC nie mehr als 5 V anliegen, wird sein Eingang mit einer 4,7 V Zenerdiode (D1 bis D3) gegen Masse verbunden. Da bei größeren Spannungen ein Strom durch die Zenerdiode fließen würde, muss dieser begrenzt werden. Experimentell wurden 4,7 kΩ Vorwiderstände (R1 bis R3) gefunden,

die einen vertretbaren Strom (hier 0,7 mA) erlauben und die nur leicht außerhalb der Empfehlung von Microchip liegen. In [Mic97] §22.3 werden maximal 10 kΩ angegeben, in [Mic02a] §7.2.1 werden maximal 2,5 kΩ als Vorwiderstand empfohlen.

Der A/D-Wandler der PICs benutzt „successive“ Approximation, um die Ladespannung des Messkondensators digital zu wandeln. Eine Messung besteht aus zwei Teilen: Laden des Kondensators und Umwandeln der Spannung. Die Ladung des Kondensators beginnt, sobald der Eingang gewählt wird (ADCON0:3-2). Die Dauer dieses Ladevorgangs wird maßgeblich von den Vorwiderständen bestimmt. Da 4,7 kΩ verwendet wurden, ergibt sich eine Ladezeit von 4,6 μs. Diese Zeit muß der PIC warten. Die eigentliche Wandlung wird vom PIC automatisch durchgeführt. Am einfachsten ist es, das dezidierte RC-Glied zum Timing zu verwenden.

Die beiden anderen Anforderungen an die Schaltung werden ähnlich dem ROCKWIRE-Master gelöst. In jedem Strang sind 220 Ω-Widerstände eingefügt (R8 bis R10). Die P-Kanal-MOSFETs (T3 bis T5) überbrücken die 10 kΩ Widerstände, die die Sensoren zum Messen mit 5 V verbinden. Die zuerst eingesetzte Diode D4, die den Linerarregler schützen sollte, wurde entfernt, weil D4 nicht leitet und es zum Übersprechen der Kanäle kam.

Da die Sensoren gleichzeitig mit Strom versorgt werden können und auch der Messmodus aktiviert werden kann, ohne dass der Sensor ausgelesen werden muss, zieht ein 1 kΩ-Widerstand die Gates von T3 bis T5 auf Vcc und der N-Kanal-MOSFET T2 kann sie gemeinsam auf GND ziehen, so dass alle Gate-Source Spannungen der T3 bis T5 ca. -8 V betragen. Das Gate von T2 wird wieder durch einen 47kΩ-Widerstand vor elektrostatischer Entladung geschützt.

Der Rest der Schaltung wird durch die ROCKWIRE-Slave-Schaltung (siehe Abschnitt 4.3.2) realisiert. Für die Schaltung wurde eine Platine gefertigt.

Die gesamte Schaltung ist in Abbildung D.7 dargestellt.

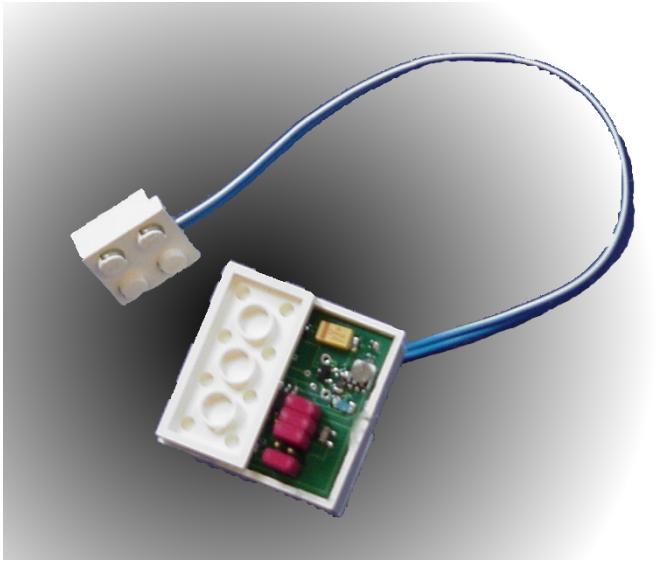


Abbildung 4.4: aufgebauter ROCKWIRE-Sensor

Abbildung 4.4 zeigt den ROCKWIRE-Sensor. Die Platine passt in zwei 2×4 Bausteine voller Höhe, sofern der Programmierr-Jumper nur einmal aufgelötet wird, um den PIC zu programmieren und dann wieder entfernt wird. Der Jumper war auch für Debugzwecke sehr hilfreich.

Sensoren auslesen mit den PIC

Das Auslesen der Sensoren ist dank der eingebauten A/D-Wandler sehr einfach. Im Programmbeispiel werden zunächst die Richtungen der Pins und die Verbindung zu den A/D-Wandlern konfiguriert. In der Measure-Methode findet die eigentliche Messung statt. Zunächst wird die 8 V-Spannungsversorgung abgeschaltet. Damit die Sensoren Zeit zum Reagieren haben, werden einige NOPs eingefügt. Danach wird der A/D-Wandler gestartet und auf das Ergebnis gewartet. In ADRESH können danach die obersten acht Bit des Messergebnisses abgelesen werden.

Damit der Kondensator jeweils einen definierten Ladungswert hat und ein Übersprechen weiter verhindert wird, kann im PIC18 ein nicht benutzter A/D-Eingang als digitaler Ausgang geschaltet werden und dieser nach erfolgter Messung selektiert werden.

Auswertung des Rotationssensors

Die Auswertung komplizierter Sensoren, z. B. des Rotationssensors oder des Lichtsensors, erfordert weitere Berechnungs- und Auswertungsfunktionen. Der Rotationssensor hat vier Zustände, die durch vier Messwerte repräsentiert werden. Je nachdem, welcher Zustand als nächstes erscheint, kann ein Zähler inkrementiert oder dekrementiert werden. Es reicht dabei aus, nur die obersten vier Bit zu betrachten.

Das Verhalten des Programms kann durch folgenden Automaten dargestellt werden (siehe Abbildung 4.5). Der aktuelle Zustand wird in `last` gespeichert. Der Zähler wird in `step` vorgehalten und ist der Messwert des Rotationssensors. Nach jeder Messung wird der Automat einmal aktualisiert. Die Zustände werden in den Konstanten `A` bis `D` gespeichert.

Dieser Automat wird für jeden der drei Sensoreingänge implementiert, vor die Variablen und Konstanten wurde dazu `s1` bis `s3` geschrieben und der Code dupliziert. In der Implementierung werden Sprungtabellen verwendet, um schnell in den aktuellen Zustand zu springen. Dort wird getestet, welcher der Fälle aufgetreten ist. Dabei wird nur auf die Fälle geprüft in denen eine Aktualisierung statt finden muss.

Der Fall, dass ein Zustand übersprungen wird, wurde ignoriert, da dies ein Fehlerfall ist, wenn die Abtastfrequenz zu klein oder die Geschwindigkeit des Rotationssensors zu hoch ist. Außerdem kann dann die Drehrichtung nicht mehr festgestellt werden.

Den Konstanten `A` bis `D` müssen nur einmal die real gemessenen Werte zugewiesen werden. Dies geschieht, indem der Rotationssensor in eine Richtung gedreht wird, die vier verschiedenen Werte können dann den Konstanten `A` bis `D` zugeiwesen werden. Die Zustände `A` bis `D` werden, da eine Sprungtabelle verwendet wird, mit 0 bis 3 codiert.

Sensoren am ROCKWIRE

Der Sensorhub kann über das ROCKWIRE benutzt werden, das Protokoll lautet wie folgt: Der Master sendet an die Adresse des Hubs eine Befehlsnummer, die Antwort enthält das jeweilige Ergebnis. Da nicht auf eine aktuelle Messung gewartet werden kann, mißt der Hub alle 3 ms, die gespeicherten Ergebnisse werden dann auf Anfrage zurückgeliefert. Die möglichen Befehle und Ihre Codierung sind in der Tabelle 4.3 auf der nächsten Seite angegeben.

4.2.4 Ansteuerung von Aktoren

Hardwareprototyp

Da die vorhandene H-Brücke drei Eingangssignale braucht, aber an einem PIC12 abzüglich der Pins, die das ROCKWIRE benötigt, nur noch vier Pins frei sind, wurde der dritte Eingang der H-Brücken durch je ein NAND-Gatter erzeugt.

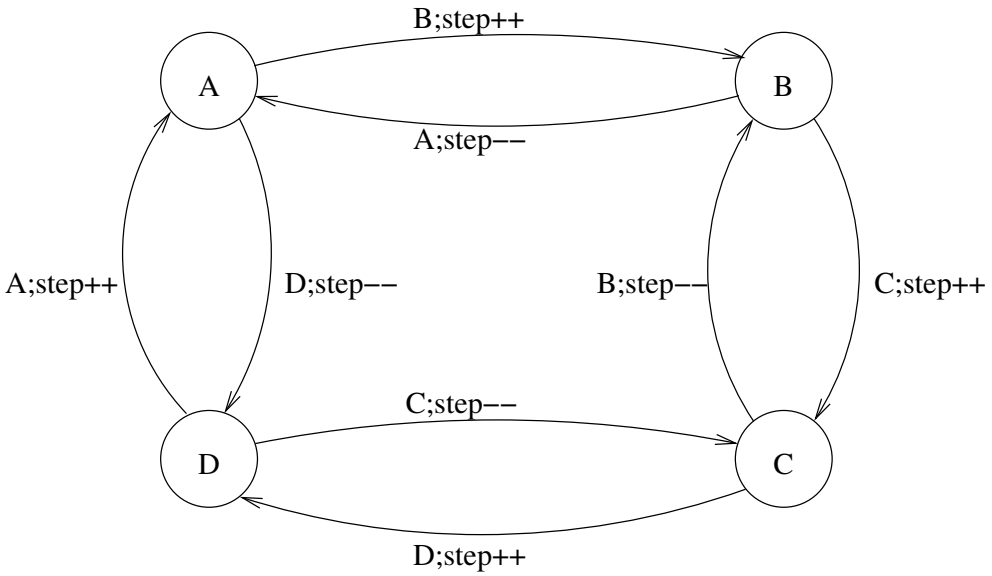


Abbildung 4.5: Auswertung Automat für den Rotationsensor

Nummer	Befehl	Rückgabe
0	Reset	0
1,2	Lese alle RAW	3 und drei Byte Werte der A/D Wandler
3	Lese alle Ticks	3 und drei Stepwerte
4,5	Lese Sensor 1 RAW	1 und Wert von Sensor 1
6	Lese Ticks des Sensor 1	1 und s1step
7	Schreibe Ticks des Sensor 1	0
8,9	Lese Sensor 2 RAW	1 und Wert von Sensor 2
10	Lese Ticks des Sensor 2	1 und s2step
11	Schreibe Ticks des Sensor 2	0
12,13	Lese Sensor 3 RAW	1 und Wert von Sensor 3
14	Lese Ticks des Sensor 3	1 und s3step
15	Schreibe Ticks des Sensor 3	0

Tabelle 4.3: ROCKWIRE Kommandos des Sensor Hub

Pin am PIC12		Eingang H-Brücke			Motor
GP 0 (Motor 1) GP 4 (Motor 2)	GP 1 (Motor 1) GP 5 (Motor 2)	B1 und A1 B1 und A1	B2 und A2 B2 und A2	ENA und ENB ENA und ENB	
L	L	L	L	H	Stop
L	H	L	H	H	Linkslauf
H	L	H	L	H	Rechtslauf
H	H	H	H	L	Leerlauf

Tabelle 4.4: Bedeutungen der Signale am PIC und der H-Brücke

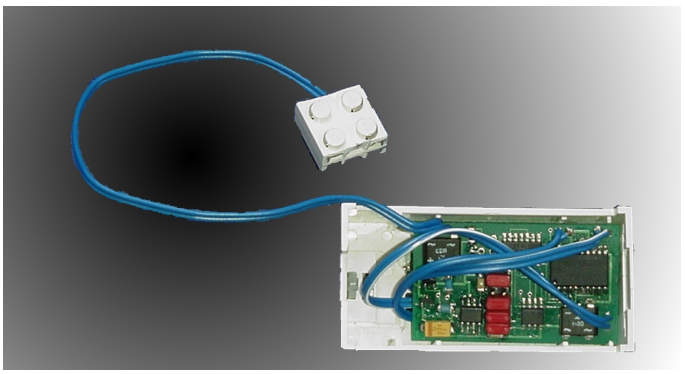


Abbildung 4.6: Der Motor-Slave in einem 9 V-Gehäuse

Die Tabelle 4.4 zeigt, wie aus den zwei PIC-Ausgangssignalen die drei Eingangssignale der H-Brücke erzeugt werden müssen und wie die zwei Ausgänge des PICs belegt sind. Es ist leicht zu sehen, dass die Spalte „ENA und ENB“ genau von dem Ausgang eines NAND-Gatters erhalten wird.

Ein zusätzliches NAND-Gatter in Form eines 7400-Bausteins, ließ sich noch im vorgesehenen Gehäuse integrieren. Ein mögliches Problem ist die Signallaufzeit dieses Gatters, die ein Schaltungshazard vor der H-Brücke erzeugt. Messungen an der konkreten Schaltung haben ergeben, dass durch das schnelle Schalten des 7400 keine derartigen Probleme auftreten.

Hardware

Auf dem Platinenlayout wurde noch eine Kontrolldiode der H-Brücke vorgesehen, um eventuell später hiermit neue Software für die PWM auf grobe Fehler zu testen, von dieser Möglichkeit wurde kein Gebrauch gemacht.

Der fertige Motor-Slave ist in ein 9 V-Gehäuse eingepasst. Abbildung 4.6 zeigt die aufgebaute Platine im Gehäuse.

Software zur Motoransteuerung

Der PIC12 simuliert eine PWM, indem ein Zähler regelmäßig die Motoren entsprechend des Status (`speicherm1` `speicherm2`) einschaltet. Die jeweiligen Motorzähler werden in jedem Takt dekrementiert und bei 0 wird der Motor ausgeschaltet.

```
;Dieses Codefragment realisiert die PWM-Steuerung am PIC12
;in speicherm1 und speicherm2 sind die Werte welche Richtung,
;Pulsweite oder haltezustand bestimmen abgelegt
```

Bit:	7	6	5	4	3	2	1	0
Bedeutung:	unbenutzt		Richtung	Halt/Stop	Geschwindigkeit			

Tabelle 4.5: Die Belegung der einzelnen Bits von `speicherm1` und `speicherm2`.

```
main2   movlw   0x10              ; hauptzaehler
        movwf   hauptzaehler      ; initialisieren
        movfw   speicherm1        ; Zähler 1 setzen
        andlw   0x0f
        movwf   zaehlerm1
        movfw   speicherm2        ; Zähler 2 setzen
        andlw   0x0f
        movwf   zaehlerm2

        incf    zaehlerm1         ; 0=keine Fahrt.
        incf    zaehlerm2

        clrf    latch
        btfsc   speicherm1 , 4    ; halt 1 testen
        bsf     latch  , 1        ; event. setzen
        btfsc   speicherm1 , 5    ; richtung 1 testen
        bsf     latch  , 0        ; event. setzen
        btfsc   speicherm2 , 4    ; halt 2 testen
        bsf     latch  , 5        ; event. setzen
        btfsc   speicherm2 , 5    ; richtung 2 testen
        bsf     latch  , 4        ; event. setzen


loop    movfw   latch
        andlw   0x33              ; rockwire ausmaskieren
        movwf   GPIO              ; Ausgänge setzen
                                  ; schreibt in ausgabe

        decfsz  zaehlerm1         ; ersten Motorzähler --
        goto    m1ok
        bsf     latch, 0          ; Motor 1 halten
        bsf     latch, 1
m1ok    decfsz  zaehlerm2         ; zweiten Motorzähler --
        goto    m2ok
        bsf     latch, 4          ; Motor 2 halten
        bsf     latch, 5

m2ok    WaitAtLeast 120


        decfsz  hauptzaehler      ; gesamtzähler --
        goto    loop
        goto    main2
```

Die Geschwindigkeit kann hier in einem Wert von 0000b bis 1111b gewählt werden. Das bedeutet, dass insgesamt 16 Geschwindigkeiten unterstützt werden.

4.2.5 Infrarot

Ziel der IR-Kommunikation ist es, den drahtlosen Datenaustausch zwischen unserer RoCK-Einheit und einem Rechner oder einer Fernbedienung, letztendlich auch zwischen mehreren Steuereinheiten untereinander, zu ermöglichen.

Ohne Anschluss des oft lästigen Kabels sollen alle Funktionen des Roboters ausführbar sein und Software, sei es nun ein Anwenderprogramm oder das Betriebssystem, auf den Mikrocontroller übertragbar sein.

Die IR-Hardware

Abbildung 4.7 veranschaulicht die Datenübertragung zwischen einem PC mit angeschlossenem IR-Tower und der RCX-Einheit. Der Computer sendet ein Byte, hier im Beispiel ein „d“ (hexadezimal 64, binär 01000110), welches von dem IR-Receiver empfangen und auf dem Digitaloszilloskop visualisiert wird.

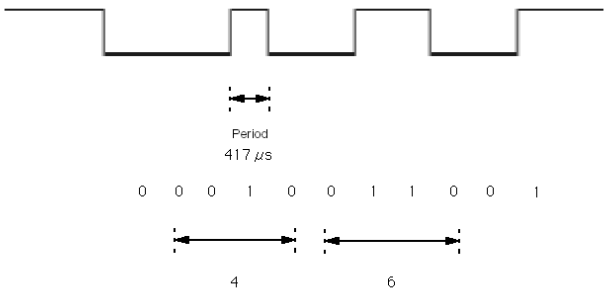


Abbildung 4.7: Ein Byte auf dem Digitaloszilloskop

Abbildung D.8 auf Seite 144 zeigt die entworfene Schaltung. Links des PIC12s befindet sich die ROCKWIRE-Slave-Schaltung, rechts davon die Sende- und Empfangsbaugruppe.

IR-Leuchtdioden werden mit relativ großen gepulsten Strömen betrieben. Möglich sind bis zu 3,5 A für 10 μs. Ein PIC12 kann solche Ströme nicht liefern, hier sind nur max. 25 mA möglich. Es wird daher ein N-Kanal-MOSFET verwendet, um die Diode mit Masse zu verbinden. Ein Vorwiderstand mit 47 Ω begrenzt den Strom auf ca. 65 mA. Dieser geringe Strom erlaubt zwar nur eine sehr begrenzte Reichweite, ermöglicht aber das Senden auch während der Zeitspanne, in der das ROCKWIRE Daten überträgt.

Die IR-Software

Für den PIC12 wurde eine Software geschrieben, die IR-Signale empfangen, die Parität korrekt auswerten und das Byte an einem Debug-Pin ausgeben kann.

Der Empfang funktioniert nach derselben Idee wie die entsprechende Routine beim ROCKWIRE. Es wird mit Interrupt-on-Change auf eine Änderung am Eingang (IRI) gewartet. Der IR-Empfänger liefert ein demoduliertes Signal, so dass davon ausgegangen werden kann, dass alle 417 μs das nächste Bit am Eingang anliegt.

Zunächst wird ein Startbit übertragen, es kann also nach 208 μs geprüft werden, ob noch eine 0 am Eingang anliegt. Dann folgen die acht Datenbits, wobei das LSB zuerst übertragen wird. Dies geschieht mit einer einfachen Zählschleife.

In den Code sind passende Verzögerungsschleifen eingefügt, so dass der PIC alle 417 μs den Eingang auswertet und das übertragene Byte zusammensetzt. Zum Test der Parität werden ankommende Einsen gezählt. Sind acht Bit übertragen, wird die Parität kontrolliert – sie muss ungerade sein.

Der IR-Empfänger sorgt für eine unterschiedliche Verzögerung des Ausgangssignals je nach Signalflanke. Der Wechsel von einer 0 zu einer 1 wird um ca. 120 μs und der Übergang von einer 1 zu einer 0 sogar um 220 μs verzögert. Daher besteht die Notwendigkeit, jeden Bitwechsel separat zu behandeln und bei 01- und 10-Übergängen verschiedene Signale zu kreieren, um die Latenzzeit zu kaschieren.

Nach systematischer Parametersuche für das Timing konnten mit einer Wire-Wrap-Schaltung Zeichen zum IR-Tower übertragen werden, die endgültige Platine wurde nicht mehr vollständig aufgebaut, siehe dazu 5.1 auf Seite 113.

4.3 ROCKWIRE

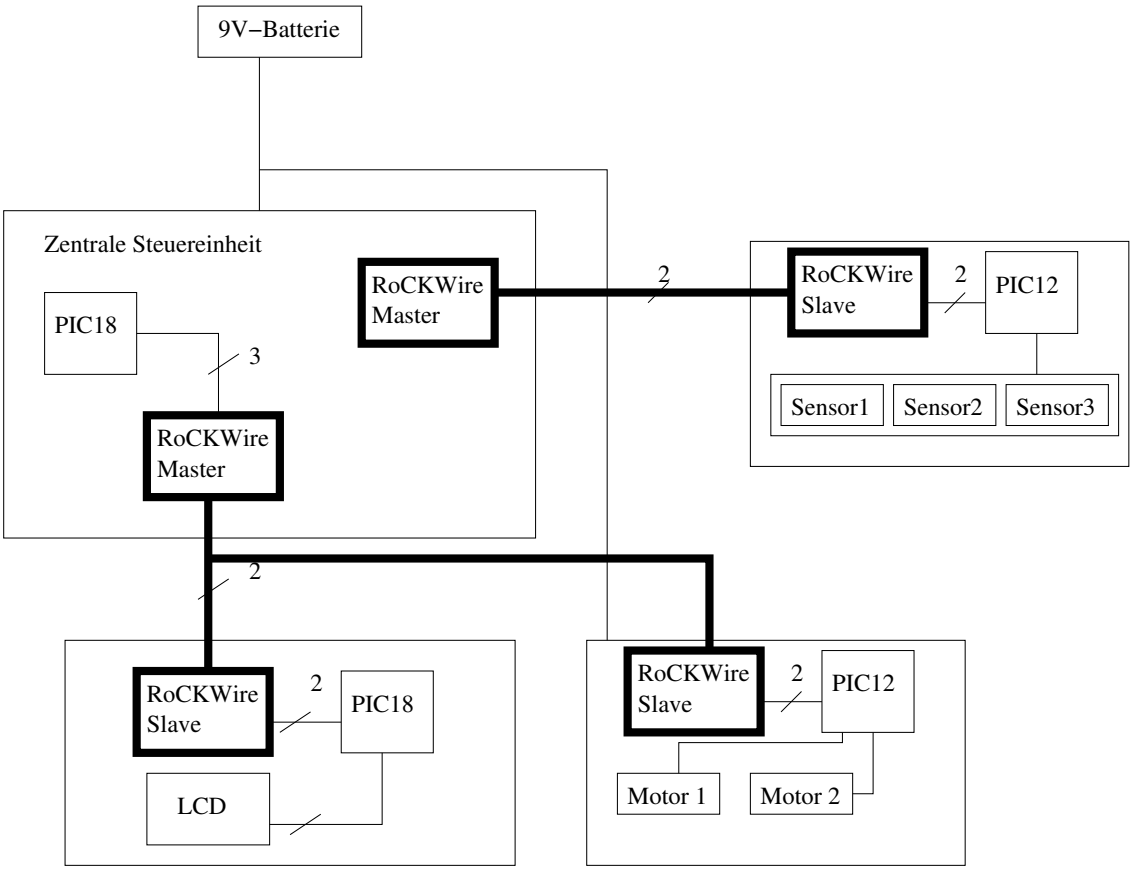


Abbildung 4.8: Blockdiagramm der Steuerung — mit hervorgehobenen ROCKWIRE-Komponenten

Das ROCKWIRE ist ein Zweidrahtbus zur bidirektionalen Kommunikation zwischen den Mikrocontrollern der Slaves und dem PIC18 der RoCK-Steuerung. Über das ROCKWIRE sollen Komponenten mit geringem Strombedarf wie der Sensorhub direkt mit Energie versorgt werden. Die Verwendung des ROCKWIRE ist im Blockdiagramm in Abbildung 4.8 dargestellt. Es ist sogar möglich den Strombedarf des LCD über das ROCKWIRE zu decken.

Es existieren andere Ansätze in diese Richtung, z. B. das 1-Wire von Maxim [MIP]. Das 1-Wire-Master-

Protokoll ist zwar Public Domain – einen Slave zu reimplementieren ist jedoch verboten. Dies ist jedoch für uns notwendig, da die von Maxim erhältlichen diskreten Bausteine einen Fan-out von eins haben. Außerdem sind die verwendeten Spannungen nicht LEGO-kompatibel. Die Slaves können nicht über das 1-Wire mit Strom versorgt werden. Dieses Protokoll wurde daher nicht weiter betrachtet.

Das ROCKWIRE ist eine Eigenentwicklung der PG, zugeschnitten auf die speziellen Anforderungen der Steuerung:

- LEGO-Kompatiblität: Eine 9 V Spannung muss zur Stromversorgung am ROCKWIRE anliegen. Aufgrund von Spannungsabfällen über dem Gleichrichter im ROCKWIRE-Master werden höchstens ca. 8,6 V am ROCKWIRE anliegen.
- Verwendung von LEGO-Steckverbindern.
- Einfache Implementierbarkeit: Die PIC12-Hardware muss das Protokoll auch implementieren können
- Die Slaves müssen ohne nennenswerten Energieverbrauch senden können
- Paralleler Betrieb mehrerer Slaves an einem Master-Port

4.3.1 ROCKWIRE-Protokoll

Die Applikationsebene

Die Applikation muss prüfen, ob eine Übertragung erfolgreich war und sie gegebenenfalls wiederholen. Unsinnige Werte sind abzufangen. Nur der Master kann feststellen, ob die Kommunikation erfolgreich war, indem die Antwort auf Plausibilität geprüft wird. Master und Slave werten während der Übertragung das Paritätsbit aus.

Die Protokollebene

Um mehrere Geräte an einem ROCKWIRE-Port betreiben zu können, verwenden wir ein Master-Slave-Protokoll. Der Master hat die Kontrolle über das ROCKWIRE und Slaves senden nur auf Anforderung des Masters. Protokollfehler werden durch ausbleibende Antworten erkannt.

Eine ROCKWIRE–Kommunikation hat folgende Form:

- Der Master sendet das Startbyte (0xdb).
- Der Master sendet die Adresse des angesprochenen Slaves.
- Der Master sendet die Anzahl der nun folgenden Bytes.
- Der Master sendet nun die von der Applikation gewünschten Bytes aus dem Buffer.
- Der Master sendet mindestens ein Pollbit.
- Der Slave liefert die Anzahl der Bytes der Antwort.
- Der Master sendet entsprechend weitere Pollbits
- Der Slave antwortet auf jedes Pollbit mit einem Byte, das die Applikation in einen Buffer geschrieben hat.

Die Byteebene

Diese Ebene verarbeitet Worte zu acht Bit. Die Hardware wird zum Senden und Empfangen in dieser Ebene angesprochen.

Damit der Beginn eines Bytes erkannt werden kann, wird ein Startbit benutzt. Mit einem Paritätsbit können Einbitfehler in einzelnen Bytes erkannt werden. Die darüber liegenden Ebenen werden über diese Fehler informiert, z. B. durch Ändern des Status.

Viele Konzepte der Software konnten für ROCKWIRE-Master und ROCKWIRE-Slave gemeinsam genutzt werden, da beide beim Senden und Empfangen vor dieselben Aufgaben gestellt werden. Eine Wiederverwendung des Source-Codes war nur für Busteilnehmer mit demselben Prozessor möglich.

Die Signalebene

Das ROCKWIRE besitzt drei Zustände:

Spannungsversorgung Das ROCKWIRE liefert die Versorgungsspannung für die Slaves. Es liegen 8,6 V an, die nur im Fehlerfall kurzgeschlossen werden dürfen.

Daten-HIGH Das ROCKWIRE wird dazu über den Pull-Up-Widerstand (R3 des Masters) mit 8,6 V verbunden.

Daten-LOW Das ROCKWIRE wird mit Masse verbunden. Es fließt ein Strom, der durch den Pull-Up-Widerstand begrenzt wird.

Die Kodierung der Bits

An die Kodierung der Bits stellen sich die folgenden Anforderungen:

- Es müssen drei Fälle kodierbar sein: 0, 1 und das Startbit.
- Die PIC12 müssen in der Lage sein, zu erkennen, wann ein Bit beginnt oder endet.
- Die Kodierung darf nur die ROCKWIRE-Zustände Daten-LOW und Daten-HIGH verwenden.

Wir haben uns entschieden, zur Kodierung die Zeit des Daten-LOW Zustands zu wählen.

Jeder Fall wird wie folgt kodiert:

- Jeder Fall beginnt mit einer fallenden Flanke.
- Da drei Fälle zu kodieren sind, werden drei verschiedene Signallängen gewählt. Die Dauer von Daten-LOW nach einer fallenden Flanke ist ausschlaggebend. Die endgültigen Zeiten sind nun wie folgt:

  1 Sehr kurz LOW (Master: 7μs, Slave: $t < 13\mu$s)

  0 Mittellang LOW (Master: 25μs, Slave: $t > 13\mu$s)

  Startbit Sehr lang LOW (52μs)

Ein typischer Signalverlauf ist in Abbildung 4.9 dargestellt. Im unteren Teil der Abbildung geben die Positionen von 0 und 1 den Zeitpunkt an, zu dem der Empfänger den Pegel des ROCKWIRE abtastet.

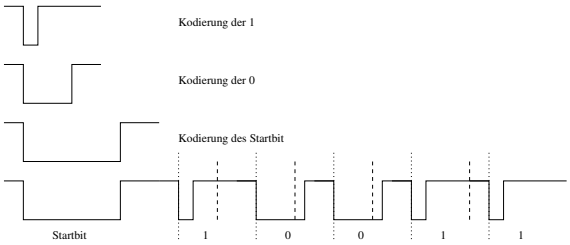


Abbildung 4.9: Oben: Kodierung von 1,0 und Startbit auf dem ROCKWIRE. Unten: Übetragung von 10011

Um das Timing festzulegen, wurden ROCKWIRE-Master- und Slave-Programme für den PIC12 geschrieben. Die Zeitmessung wird implizit über den Systemtakt des PIC12 vorgenommen. Daher müssen Befehlssequenzen definierter Länge in Maschinensprache kodiert werden. Zur Vereinfachung haben wir dies zunächst nur auf dem PIC12 vorgenommen.

In Abbildung 4.10 ist das reale Timing der Übertragung eines Bytes (1010 0101) mit den Prototypen dargestellt. In Tabelle 4.7 sieht man die Beschreibung der einzelnen Phasen. Tabelle 4.6 zeigt das Timing des Protokolls. Wir haben festgelegt, dass das MSB zuerst übertragen wird.

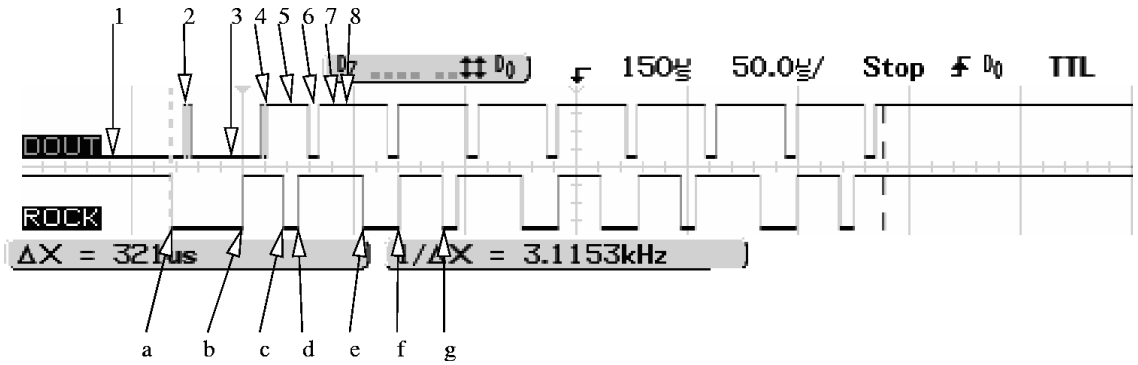


Abbildung 4.10: Das Timing des ROCKWIRE Prototypen. DOUT: Debugoutput des ROCKWIRE-Slave. ROCK: Das Signal am Eingang des ROCKWIRE-Slave.

Beg.	End.	Dauer	Beschreibung
a	b	32,4 μs	Startbit
b	c	17,6 μs	Pause nach dem Startbit
c	d	6,4 μs	Länge von LOW für 1
d	e	29,6 μs	Pause nach einer 1
c	e	36 μs	Länge einer 1
e	f	16,8 μs	Länge von LOW für 0
f	g	19,2 μs	Pause nach einer 0
e	g	36 μs	Länge einer 0

Tabelle 4.6: Das Timing des ROCKWIRE

Nr.	Phasen des ROCKWIRE-Slave
1	Warten auf das Startbit
2	Prüfen, ob LOW lang genug
3	LOW ist lang genug für Startbit
4	Ende der Startbit-Verarbeitung und Warten auf die fallende Flanke
5	Fallende Flanke erkannt
6	Auswertung des ROCKWIRE, HIGH entspricht log. 1
7	Warten auf HIGH
8	Weiter mit dem nächsten Bit

Tabelle 4.7: Die Phasen des ROCKWIRE-Slave

4.3.2 ROCKWIRE-Schaltung

Die ROCKWIRE-Schaltung stellt das Interface vom PIC12 oder PIC18 zum ROCKWIRE her. Da am ROCKWIRE bis zu 8,6 V anliegen, kann die MPU nicht direkt mit dem ROCKWIRE verbunden werden.

Es wird zunächst der ROCKWIRE-Slave vorgestellt, da die Schaltung im Aufbau einfacher gehalten ist. Einzelne Baugruppen werden im Master wiederverwendet.

Die ROCKWIRE-Slave Schaltung

hat folgende Aufgaben:

- Verpolungsschutz durch einen Gleichrichter; die Ausrichtung des Steckers ist egal.
- Gewinnung der Versorgungsspannung für die Slave-Schaltung.
- Spannungsstabilisierung auf 5 V für den PIC.
- Den Spannungspegel des ROCKWIRE für den MPU-Eingang auf 5 V begrenzen.
- Das ROCKWIRE bei Bedarf mit Masse verbinden, also eine Pegeländerung auf dem Bus veranlassen, wobei wenig Strom für den Schaltvorgang verbraucht werden soll.

Diese Aufgaben lassen sich im Schaltplan (siehe Abbildung D.6) zum ROCKWIRE-Slave wieder erkennen. Die Funktionsweise wird im Folgenden beschrieben

Spannungsversorgung

Die Spannung des ROCKWIRE wird durch eine Schottky-Graetzbrücke gleichgerichtet. Die Schottky-Diode D5 verhindert, dass C1 über den Spannungsteiler (R11/R12) oder das ROCKWIRE entladen wird. C1 glättet die Eingangsspannung des Linearreglers, sie beträgt ca. 8 V.

Die Versorgungsspannung der Gesamteinheit (Roboter) beträgt im günstigsten Fall 9 V. Sie kann während des Betriebs abhängig von der Last und dem Ladezustand des Energiespeichers variieren. Dann fallen 0,4 V im Gleichrichter des ROCKWIRE-Master, 0,4 V im Gleichrichter des ROCKWIRE-Slave und 0,2 V über D5 ab.

Der Kondensator C1 kann eine Energie von 0,7 mWs speichern, d. h. es könnten ca. 100 mA bei 8 V für 0,8 ms geliefert werden. Der Linearregler liefert die 5 V Versorgungsspannung für den Mikrocontroller. Bei einer Dimensionierung von C1 = 470 μF kann das Display inkl. MPU über das ROCKWIRE parasitär mit Energie versorgt werden.

Der ROCKWIRE-Master muss dafür Sorge tragen, dass das ROCKWIRE oft und lange genug im Stromversorgungszustand ist, damit C1 geladen werden kann. Ist dies nicht gegeben, so wird die MPU ab einer Spannung von ca. 3 V über C1 nicht mehr arbeiten.

ROCKWIRE auslesen

Der Spannungsteiler aus R11 und R12, beide 22 kΩ , teilt die Eingangsspannung. Auf den Platinen wurde alternativ zu R12 die Bestückung mit einem 47kΩ SMD-Trimpoti vorgesehen. Wir gehen davon aus, dass man ab 5 Slaves einen ROCKWIRE-Hub benötigt. Das Trimpoti erlaubt eine Feinjustage, wenn viele Slaves angeschlossen werden. Bei Daten-HIGH liegen 4 V am Eingang RI des PIC an. Bei Daten-LOW liegen 0 V am Eingang RI des ROCKWIRE-Slave PIC und 0,2 V am Eingang RI des ROCKWIRE-Master PIC an, da dieser keinen Gleichrichter zum Bus besitzt, wie aus der Schaltung in Abbildung D.3 leicht erkennbar ist.

ROCKWIRE mit Masse verbinden

Der N-Kanal Anreicherungstyp (FDN335N) mit der Bezeichnung T1 kann vom PIC direkt über den Ausgang RO angesteuert werden. Die Ausgangsspannung von 5 V reicht aus, um den MOS-FET voll durchzuschalten. Die entstehende Verbindung kann die Spannung des ROCKWIRE aufgrund der Spannungsabfälle über den Dioden des Gleichrichters nur bis auf höchstens 0,4 V absenken. Bei allen Slaves liegen aber ebenfalls 0 V an RI an, da ihre Gleichrichter denselben Spannungsabfall hervorrufen. Der Master wird 0,2 V korrekt als Daten-LOW interpretieren.

R13 wiederum dient zum Schutz des MOS-FET. Es darf höchstens eine Gate-Source-Spannung von 8 V anliegen Diese Spannung wird über R13 abgeleitet. Außerdem schützt er den empfindlichen Gate-Eingang vor elektrostatischer Entladung.

Die ROCKWIRE-Master Schaltung

Zunächst war geplant, den ROCKWIRE-Masteranschluss auch kompatibel zum LEGO-Sensor auszulegen – davon wurde aber Abstand genommen, weil dazu weitere Bauelemente notwendig gewesen wären.

ROCKWIRE-Master und Slave sind einander sehr ähnlich. Der N-Kanal-MOS-FET und Spannungsteiler finden sich in seinen Baugruppen. Der Teil zur Gewinnung der Versorgungsspannung kann entfallen. Der ROCKWIRE-Master hat die Aufgabe, das ROCKWIRE mit Strom zu versorgen. Dazu muss ein neuer Schaltungsteil entworfen werden. Die ROCKWIRE-Masterschaltung ist in Abbildung D.3 auf Seite 139 dargestellt. In der Zentraleinheit wurden zwei Master integriert. Hier wird exemplarisch die obere Schaltung um Master 1 beschrieben.

Das ROCKWIRE hat zwei Modi: Stromversorgung und Datenaustausch. Zur Stromversorgung muss der Pull-Up-Widerstand R17 (Abbildung D.3) überbrückt werden. Außerdem muss der Widerstand R17 geeignet dimensioniert werden. Der Wert von R17 ergibt sich aus folgenden gegensätzlichen Anforderungen:

- Er muss groß genug sein, damit, wenn ein Teilnehmer das ROCKWIRE mit Masse verbindet, kein zu großer Strom fließt und die Batterien somit zu schnell entladen würden.
- Er darf nicht zu groß sein, denn R17 bildet mit den Spannungsteilern (R11/R12) der ROCKWIRE-Slaves einen Spannungsteiler. Die maximale Spannung U_{RH} während Daten-HIGH auf dem ROCKWIRE ist um so größer, je kleiner R17 ist. Eine möglichst große und auch bei mehreren angeschlossenen Slaves gleichbleibende Spannung ist wichtig, da sonst die Spannungsteiler der ROCKWIRE-Slaves nicht mehr richtig dimensioniert sind.

Empirisch wurde 1 kΩ für R17 ermittelt. U_{RH} läßt sich näherungsweise unter Vernachlässigung der restlichen Schaltung der ROCKWIRE-Slaves durch folgende Formel in Abhängigkeit von der Anzahl n der

angeschlossenen ROCKWIRE-Slaves bestimmen:

$$\begin{aligned} U_{RH} &= 8,6V \cdot \frac{\frac{44k\Omega}{n}}{\frac{44k\Omega}{n} + 1k\Omega} \\ &= \frac{8,6V \cdot 44k\Omega}{44k\Omega + n} \end{aligned}$$

Zunächst begrenzte der Drahtwiderstand R22 mit 10Ω den Strom, der über das ROCKWIRE fließen kann, auf 860 mA. Auf den Platinen wurde der Drahtwiderstand durch einen PTC [RE] ersetzt, dieser (C 975) hat bis zu einem Strom von ca. 1 A einen sehr kleinen Widerstand ($< 1\Omega$) steigt dann aber rasch an, so dass die Kondensatoren der Slaves schnell geladen werden können, eine Kurzschlußsicherung aber gegeben ist.

R17 wird zur Stromversorgung des ROCKWIRE mit dem P-Kanal-MOS-FET T2 überbrückt. T2 wird mit dem N-Kanal-MOS-FET T3 gesteuert. Das ROCKWIRE liefert Strom für die ROCKWIRE-Slaves, wenn der Ausgang RP des PIC 1 ist.

4.3.3 ROCKWIRE-Software

Zunächst wurden prototypische ROCKWIRE-Sende- und Empfangsroutinen für den PIC12 implementiert. Eine Interruptbehandlung für jedes einzelne Bit kommt auf dem PIC12 nicht in Frage, weil für das Hauptprogramm nur 5 Takte verbleiben würden.

Senden mit dem ROCKWIRE

Eine Hilfsroutine SendBit sendet ein einzelnes Bit über das ROCKWIRE:

1. RO=1, das ROCKWIRE mit Masse verbinden, Daten-LOW
2. Warte 7 μs
3. Wenn eine 1 gesendet werden soll, d.h. das 7. Bit von A ist 1, dann RO=0, Verbindung aufheben, Daten-HIGH
4. Warte 8 μs
5. RO=0, Verbindung aufheben, Daten-HIGH
6. Warte 17 μs

Zum Senden eines Bytes wird der folgende Algorithmus verwendet, das zu sende Byte steht in A:

1. BitCount=0
2. SendStartBit
3. SendBit
4. Rotiere A nach links
5. BitCount um eins erhöhen
6. Wenn BitCount nicht gleich 7 ist, dann weiter mit 3.
7. RP=1, das ROCKWIRE mit Strom versorgen (nur Master)

Die Hilfsroutine SendStartBit ist äußerst einfach:

1. RP=0, Sicherstellen, dass das ROCKWIRE nicht direkt mit Strom versorgt wird. Dies kann für Slaves entfallen.
2. RO=1, das ROCKWIRE mit Masse verbinden, Daten-LOW
3. Warte 25 μs
4. RO=0, Verbindung aufheben, Daten-HIGH
5. Warte 13 μs

Das Warten zum Schluss von SendStartBit (5.) und SendBit (6.) gibt den anderen ROCKWIRE-Teilnehmern Zeit, die Verarbeitung des Bits zu beenden. Diese Zeit kann je nach Bedarf einfach verlängert oder verkürzt werden.

Empfangen mit dem ROCKWIRE

Das Empfangen eines Bytes ist etwas komplexer. Es wird ein Interupt-On-Change des PIC12 genutzt, um eine Änderung des Eingangs RI zu erkennen. In der Interrupt Routine wird WREG und STATUS gesichert und geprüft, ob es sich um ein Interrupt-On-Change handelt, dann wird folgendes Programm ausgeführt:

1. StartBitTest
2. BitCount=0
3. Rotiere RockIn um 1 nach links.
4. Schreibe eine 0 in Bit 0 von RockIn
5. Warte auf fallende Flanke
6. Warte 7 μs
7. Wenn RI=1 dann schreibe eine 1 in Bit 0 von RockIn
8. Erhöhe BitCount um 1
9. Warte solange bis RI=1 ist
10. Wenn BitCount nicht gleich 7 ist weiter mit 3.
11. Es liegt das Ergebnis in RockIn, verlasse den Interrupt.

Die Hilfsroutine StartBitTest pollt RI. Wird RI innerhalb von 20 μs nach der fallenden Flanke 1, so handelt es sich nicht um ein Startbit. Die Interruptserviceroutine wird sofort verlassen, die Kommunikation ist gestört. Es wird bis zum nächsten Byte dauern, bis der Slave wieder Bytes empfangen kann. Nach 20 μs muss es sich um ein Startbit gehandelt haben und es wird auf RI=1 gewartet.

Der Interrupt darf nicht verzögert werden, da sonst die StartBitTest Methode ein StartBit nicht richtig erkennt. Die ROCKWIRE-Slaves sollten auf die Verwendung weiterer Interrupts möglichst verzichten.

Parität des ROCKWIRE

Die Parität des ROCKWIRE ist für den Master gerade und für die Slaves ungerade. Dies hat zwei Vorteile:

- auf dem Logik Analyzer kann einfach zwischen Master und Slave unterschieden werden,
- kein Slave wird ein 0xdb eines Slaves als Beginn eines Pakets interpretieren.

Für den Fall, dass Bitfehler auftreten, bedeutet dies für die anderen Teilnehmer nur dann ein Problem, wenn weitere Bytes auch passen und das Protokoll der Protokollebene zufällig eingehalten wird – dies ist sehr unwahrscheinlich.

ROCKWIRE Master auf dem PIC18

Die Implementierung des ROCKWIRE-Treibers für SOUL (siehe auch Abschnitt 4.5.1 auf Seite 84) wurde nach der Prototyp-Entwicklung des ROCKWIREs vorgenommen. Dennoch konnte die vorhandene Implementierung des ROCKWIRE-Masters für den PIC12 nicht übernommen werden. Dies lag vor allem daran, dass der PIC18 unter SOUL mit 40 MHz und somit 10-mal schneller läuft als der im Prototypen verwendete PIC12.

Um den Treiber gut in SOUL zu integrieren und keine Rechenzeit unnötig zu verschwenden, indem man das Timing über *NOP*s realisiert, wurde die Nutzung von Interrupts angestrebt. Dazu mussten zwei Interrupt-Quellen verwendet werden: ein 16-bit Timer und der Interrupt-On-Change für das Eingangssignal. Es zeigte sich, dass das verwendete Timing – es liegt im Bereich von Mikrosekunden, während ein Befehl 100 ns zur Abarbeitung benötigt – nur realisiert werden kann, wenn der ROCKWIRE-Treiber nicht von anderen Interrupts unterbrochen werden kann. Dies geht nur, wenn alle anderen Interrupts eine niedrigere Priorität haben als die beiden ROCKWIRE-Interrupts. Dies wurde in SOUL auch realisiert, aber während der Scheduler den Stack kopiert, müssen alle Interrupts abgestellt werden. Auch lässt sich ein Übertragungsfehler auf dem ROCKWIRE erkennen und das als fehlerhafte verworfene Paket wiederholt senden.

Durch die Verwendung des Timer-Interrupts konnte das Timing sehr flexibel implementiert werden, indem der Timer immer nur über Defines parametrisiert wird. Dies ermöglichte es, in der Integrationsphase das Timing des Masters einfach an das Timing der Slaves anzupassen.

Die Interrupt-Routine des ROCKWIRE-Treibers wird durch die Funktion *RWSendPacket()* aufgerufen. Der Treiber führt dann die nötigen Aktionen aus und springt aus der Interrupt-Routine zurück. Beim Auftreten des entsprechenden Interrupts wird wieder in die Interrupt-Routine gesprungen und anhand einiger Flags der aktuelle Zustand festgestellt und somit die nächste Aktion durchgeführt. Es wird somit eine Zustandsmaschine realisiert. Bei der anschließenden Realisierung des ROCKWIRE-Slaves für den PIC18 des LCDs wurde das Design des Treibers übernommen, es wurden aber zwei Sprungtabellen benutzt, wodurch sich Fehler in der Implementierung einfacher korrigieren lassen. Als Zustände werden zum einen die Art des gerade übertragenen Bytes – Startbyte, Adresse, Länge, Datenbyte, dabei wird jeweils auch unterschieden in welche Richtung die Übertragung stattfindet – und auch der Zustand auf Bitübertragungsebene – Start- und Pollbit, Datenbit und Paritätsbit – unterschieden. Für den Slave ergeben sich 13 Zustände auf Byteebene und insgesamt zehn Zustände für die Bitübertragung. Dabei müssen auch die Zustände für die fallende und steigende Flanke der Bits unterschieden werden. Zu den Zuständen auf Byteebene kommen noch verschiedene Pseudo-Zustände hinzu, z. B. um die Daten aus dem Puffer zu lesen.

Um die korrekte Funktion bei einem Fehlverhalten eines Slaves zu gewährleisten, wird beim Empfangen von Daten nicht nur auf eine Flanke auf dem ROCKWIRE gewartet, sondern zusätzlich über den Timer ein Timeout vorgegeben, auf dessen Eintreten mit einem Abbruch der Übertragung des gesamten Pakets reagiert wird.

Die Nutzung bei eigenen Applikationen (insbesondere der Drop-Zone-Mission) zeigt, dass die ROCKWIRE-Kommunikation sehr stabil und zuverlässig funktioniert.

ROCKWIRE Applikationen auf dem PIC12

Die ROCKWIRE–Hubs für Motoren und Sensoren wurden integriert. Für die Integration des IR–Hub blieb keine Zeit.

Der ROCKWIRE–Slave Treiber und die Applikation wurden zunächst getrennt entwickelt und getestet. So kann die gleiche Quelltext-Datei (pic12.asm/rockw.asm) für den ROCKWIRE–Slave Treiber in allen Hubs verwendet werden.

Da *RWProcess* aus dem Interrupt zur Behandlung der ROCKWIRE Kommunikation aufgerufen wird, ändert die Funktion nur den Status des Hauptprogramms und erzeugt die Antwort. Die Interrupt Routine kann dann die Antwort an den Master senden.

4.4 Beschreibung der verwendeten Bauteile

4.4.1 Der PIC18F452

Der PIC18F452 gehört zur Familie der Enhanced-Range PICmicro Mikrocontroller (MPU) der Firma Microchip [Micd]. Dieser 8-Bit Mikrocontroller hat einen 16-Bit RISC-Befehlssatz, der 75 Instruktionen umfasst.

Der PIC18F452 hat 32 KB internen Programm-Speicher in FLASH-Technik, dazu 1536 Byte Daten-Speicher und 256 Byte EEPROM. Es gibt 34 I/O-Pins und acht 10-bit Analog-Digital-Wandler (ADC).

Im Gegensatz zu Desktop-Mikroprozessoren nutzen die PIC18-MPUs die Harvard-Architektur, haben also getrennte Adressräume für Programm- und Datenspeicher. Dies wird auch genutzt, um den Programmspeicher mit einer nicht flüchtigen Speicherart (ROM, EPROM, OTP, FLASH) zu realisieren. Beim PIC18F452 wird FLASH als Programmspeicher genutzt[1]. Auch wurde die Datenbreite des Programmspeichers im Gegensatz zum Datenspeicher auf 16 Bit erhöht, so dass die meisten Instruktionen auf einmal gelesen werden können. Der Adressraum der PIC18 für den Programmspeicher ist 2 MB (2^{21}) groß.

Das FLASH-ROM kann mit Schreibschutz versehen werden („Code-Protection“).

Je nach FLASH-Typ sind zum Programmieren und Löschen unterschiedliche Spannungen nötig. Die neueren FLASH-PICs erzeugen die dazu nötigen Spannungen aus der Versorgungsspannung und vereinfachen damit die Schaltung.

Bei den PIC18FXX2 ist sogenanntes In-System-Programming oder In-Circuit-Serial-Programming (ICSP) möglich, d. h. mit nur vier Pins (davon zwei zur Spannungsversorgung) kann man den Mikrocontroller neu programmieren. Durch eine entsprechende Schaltung muss der Entwickler dafür sorgen, dass die daran zusätzlich angeschlossenen Geräte durch das Programmieren nicht in ihrer Funktion gestört werden. Der Schaltungsaufwand für die beiden Pins ist bedeutend niedriger als für der Aufwand für die parallele Programmierung mit mehr als 8 Pins.

Zusätzlich sind die PIC18FXX2 noch dazu fähig, ihr eigenes FLASH-ROM mittels der Instruktion *TBLWD* zu beschreiben. Damit werden Bootloader möglich, die das eigentliche Programm über eine geeignete Schnittstelle downloaden und sich damit updaten. Dabei ist zu beachten, dass das Schreiben des FLASH-Speichers in Blöcken von 8 Byte vorgenommen werden muss und das Löschen des FLASH in Zeilen von 64 Byte. (Siehe auch Kapitel 4.5.1 auf Seite 77.)

Der Datenspeicher ist in SRAM-Technik realisiert und hat eine Datenbreite von 8 Bit. Die maximale Größe des RAM beträgt 4 KB (2^{12}). In diesen Adressraum sind aber auch die Special Function Register (SFR) eingeblendet. Der Datenspeicher, der zur freien Benutzung zur Verfügung steht, wird General Purpose Register (GPR) genannt. Die Größe des RAM in den Datenblättern bezeichnet immer die Anzahl der General Purpose Register (GPR), die frei benutzt werden können. Da die meisten Instruktionen nur 8 Adressbits

[1] Das F in PICxxFxxx steht für die FLASH-Variante des Prozessors.

als Argument nutzen, ist der Speicher in Bänke von 256 Byte aufgeteilt. Die oberen 4 Adressbits stehen im Bank Select Register (BSR) . Zusätzlich wurde eine Access Bank eingeführt, die aus der unteren Hälfte von Bank 0 (0x000:0x07F) und der oberen Hälfte von Bank 16 (0xF80:0xFFF) besteht. In letztere sind die SFR eingeblendet. Die Instruktionen haben als Argument nicht nur 8 Adressbits sondern auch noch das Access Bit, das angibt, ob die Instruktion das BSR oder die Access Bank nutzt. Dadurch wird das Problem der häufigen Bankwechsel, das es bei bei den älteren PICs gab, wesentlich abgemildert.

Zusätzlich ist noch ein EEPROM für Daten vorhanden, das aber nicht in den Adressraum eingeblendet ist und wie jedes Hardware-Modul nur über spezielle Register angesprochen werden kann. In diesem EEPROM kann man Daten persistent abspeichern. Der besondere Vorteil gegenüber dem FLASH besteht darin, dass es sich einfacher beschreiben lässt und typischerweise eine höhere Anzahl von Schreibzugriffen zulässt. Auch das EEPROM lässt sich per DataProtection vor versehentlichem Überschreiben schützen.

Alle Instruktionen nutzen direkte Adressierung. Eine indirekte Adressierung lässt sich durch die File Select Register (FSR) erreichen, von denen insgesamt drei vorhanden sind. Diese lassen sich so nutzen, dass bei einem Zugriff automatisch die Adresse inkrementiert oder dekrementiert wird. Die Benutzung gestaltet sich recht einfach, es gibt für jedes FSR 2 SFRs, in die die zu dereferenzierende Adresse geschrieben wird. Danach kann man mit anderen SFRs auf diese Adresse zugreifen.

Viele der Instruktionen arbeiten mit dem Working Register (W) als Akkumulator. Bei den PIC18 ist dieses Register zusätzlich noch als WREG im RAM eingeblendet. Damit gibt es bei den PIC18 nicht mehr die Unterscheidung zwischen einen Registersatz, der einen zusätzlichen Speicherbereich darstellt und dem RAM. Aus diesem Grund werden auch die einzelnen Speicherwörter (8 Bit) im RAM als File Register bezeichnet, die sich noch einmal in SFR und GPR aufteilen. Dabei lassen sich die GPRs als Register eines homogenen Registersatzes klassifizieren. Die SFRs fallen aus verständlichen Gründen nicht unter diese Einteilung.

Neben der Auswahl, ob eine Adresse sich auf die Access Bank bezieht, lässt sich bei vielen Instruktionen über das Destination Bit angegeben, ob das Ergebnis in das WREG oder in das angegebene File Register zurückgeschrieben werden soll.

Die MPU nutzt eine 2-stufige Pipeline, so dass pro Zyklus meistens eine Instruktion abgearbeitet wird. Die Instruktionen, die 2 Wörter lang sind (*LFSR*, *MOVFF*, *GOTO*, *CALL*), brauchen einen Zyklus länger, da pro Zyklus nur ein Programm-Wort gelesen wird. Auch alle Instruktionen die den Program Counter (PC) verändern, benötigen einen Zyklus mehr zur Abarbeitung, da in der ersten Stufe der Pipeline die Instruktion gelesen und erst in der zweiten Stufe ausgeführt wird. Durch das Pipelining wird dabei aber bereits die folgende Instruktion eingelesen, deren Ausführung durch den Sprungbefehl aber hinfällig geworden ist und durch ein Flush (NOP) ersetzt werden muss. (Siehe auch Abbildung 4.11 auf der nächsten Seite.) Der PC liegt im normalen Adressraum und kann dadurch nicht nur durch die klassischen Sprungbefehle verändert werden, sondern auch durch viele andere Instruktionen, z. B. auch durch eine Addition bei einem computed GOTO. Da der PC 21 Bit breit ist, wurde er auf drei Register aufgeteilt, aufgrund des WORD-Alignments ist das Least Significant Bit immer 0. Die oberen 2 Byte des PCs sind als Latch-Register realisiert, d. h. es existiert nur ein direkter Zugriff auf das untere Byte des PCs (*PCL*). Beim Lesen des *PCL* wird der Inhalt des PC in die Latch-Register geschrieben, beim Schreiben des *PCL* wird umgekehrt der Inhalt der beiden Latch-Register auf einmal in den PC transferiert.

Als Sprungbefehle existieren zum einen relative Sprungbefehle mit begrenzter Reichweite (je nach Instruktion kann bis zu ± 128 bzw. ± 1024 Instruktionen weit gesprungen werden) und zum anderen zwei absolute Sprungbefehle (*GOTO* und *CALL*), die als 2-WORD-Instruktionen eine 21-bit Adresse[2] als Argument haben und so zu jeder Adresse im gesamten Adressraum springen können. Die bedingten Sprungbefehle sind alle relativ und werten die Flags des STATUS Registers aus, das durch vorherige Instruktionen gesetzt wurde. Welche Instruktion welche Flags setzt, ist dem Datenblatt [Mic02b] zu entnehmen. Als Erbe von älteren PIC-Familien hat der PIC18 auch noch sogenannte konditionelle Skip-Befehle, die jeweils die

[2] Genau genommen nutzen sie nur eine 20-bit Adresse. Da das Least Significant Bit des Program Counter immer 0 ist, wird dieses nicht explizit in der Instruktion codiert.

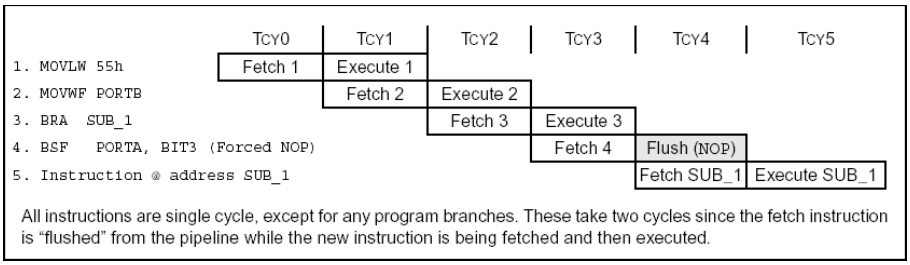


Abbildung 4.11: Pipelining bei Sprungbefehlen

folgende Instruktion überspringen (skippen). Dies funktioniert auch bei 2-WORD-Instruktionen.

Der Program Counter (PC) wird bei einem *CALL* oder dem Auftreten eines Interrupts auf einem 31-stufigen Stack gesichert. Durch ein *RETURN* wird er wieder vom Stack geholt. Dieser Stack ist durch Software manipulierbar – außerdem kann ein Interrupt generiert werden, wenn der Stack voll oder leer ist. Dies kann man zum Beispiel nutzen, um eine Stack-Verwaltung zu implementieren, die eine größere Rekursionstiefe unterstützt. Vor allem der Scheduler eines Multitasking-fähigen Betriebssytems nutzt die Möglichkeit, den Stack zu verändern (siehe Abschnitt 4.5.1 auf Seite 80).

Für die drei wichtigsten Register (WREG, STATUS und BSR) existiert ein Fast Register Stack, der allerdings nur eine einzige Stufe hat. Das Sichern der Register kann man mit einem *CALL FAST* veranlassen, beim Funktionsaufruf durch Interrupts wird auch der Fast Register Stack benutzt. Weil der Stack nur einstufig ist, kann man ihn nur für High Priority Interrupts sinnvoll verwenden, da in den anderen Fällen immer ein High-Priority Interrupt auftreten kann, bei dessen Aufruf der Fast Register Stack überschrieben wird. Das Wiederherstellen der Register wird durch ein *RETURN FAST* erreicht.

Die PIC18-MPUs haben einen Multiplizierer, der in nur einem Zyklus zwei 8-Bit-Werte miteinander multipliziert. Die Faktoren können entweder beide im RAM stehen oder einer im Working Register. Das 16-Bit Ergebnis wird in zwei speziellen Registern (PRODH:PRODL) gespeichert. Division und der Umgang mit nicht ganzen Zahlen muss in Software realisiert werden.

Zu Energiesparzwecken hat der PIC18 noch die *SLEEP* Instruktion implementiert, die den ganzen Prozessor bis zum Auftreten des nächsten Interrupts bzw. bis zum Ablaufen des Watchdog-Timers anhält.

Hardwarekomponenten des PIC18F452

Hier erfolgt nur eine kurze Einführung in die einzelnen Hardwarekomponenten. Dies soll nicht als Referenz dienen, sondern nur die vorhandene Funktionalität darstellen. Dem Datenblatt können weitere Informationen entnommen werden [Mic02b].

Oszillator: Der Oszillator dient als Taktquelle, der die Taktfrequenz des Mikrocontrollers festlegt. Die PIC18FXX2 sind für Frequenzen von 0–40 MHz ausgelegt. Jeweils vier Takte sind ein Zyklus, in dem eine Instruktion ausgeführt wird. Somit erreicht man mit diesem Mikrocontroller bis zu 10 MIPS.

Es sind acht verschiedene Oszillatortypen möglich. Der Typ wird mit dem Schreiben des Programms in den Mikrocontroller festgelegt: LP (Low Power Crystal), XT (Crystal/Resonator), HS (High Speed Crystal/Resonator), HS + PLL (High Speed Crystal/Resonator with PLL enabled), RC (External Resistor/Capacitor), RCIO (External Resistor/Capacitor with I/O pin enabled), EC (External Clock), ECIO (External Clock with I/O pin enabled). Die Varianten mit Phase Locked Loop (PLL) vervierfachen den Takt des verwendeten Quarzes.

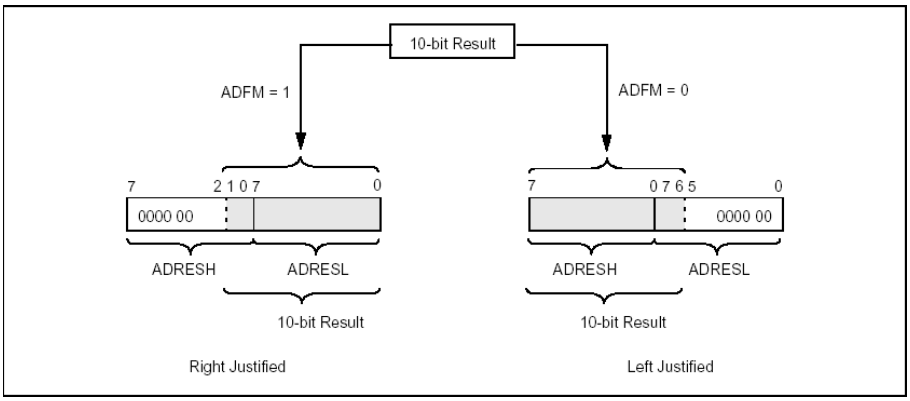


Abbildung 4.12: 10-Bit Ausrichtung des ADC-Ergebnisses

I/O-Ports: Der PIC18F452 besitzt 34 Tri-State Ein-/Ausgänge auf die 5 Ports PORTA-PORTE verteilt. Einzelne Pins sind mit anderen Funktionen gemultiplext (z. B. analoge Eingänge, PSP, USART) und ihre Nutzung ist nur alternativ zu den anderen Funktionen möglich. Alle I/O-Pins lassen sich wahlweise als Eingänge oder Ausgänge nutzen.

Die Eingänge sind entweder TTL oder Schmitt-Trigger kompatibel. Welcher Eingänge welche Charakteristik haben, kann dem Datenblatt [Mic02b] entnommen werden.

Die I/O-Ports werden jeweils durch 3 SFRs gesteuert, deren Bits jeweils einen Pin steuern:

TRISx Mit dem data direction register kann man konfigurieren, welcher Pin ein Eingang oder Ausgang ist. Wenn ein Pin als Eingang konfiguriert wurde, befindet sich dessen Ausgangstreiber im High-Impedance Zustand.

PORTx Das Port-Register enthält den Zustand der Eingänge, man kann es auch zum Schreiben benutzen.

LATx Ins Latch-Register geschriebene Werte werden gehalten und am Ausgang angelegt. Soll dieser geschriebene Wert nochmals gelesen werden (z. B. bei Read-Modify-Write Befehlen), sollte man dazu das Latch-Register verwenden und nicht das Port-Register, weil letzteres bspw. durch als Eingang konfigurierte Pins beeinflusst werden kann.

Hinweis: Durch das Aktivieren einiger Features, wie z. B. der RS232-Unterstützung, können die jeweilig dazu vorgesehen Pins nicht mehr als Allzweck-IO-Pins verwendet werden.

Analog-Digital-Konverter: Die acht Kanäle des Analog-Digital-Konverters (ADC) arbeiten mit 10-Bit Auflösung. Sie nutzen entweder die Versorgungsspanunng oder eine externe Spannungsquelle als Referenzspannung. Durch die Verwendung eines internen RC-Oszillators kann der ADC auch im SLEEP-Mode arbeiten, wenn alle anderen Oszillatoren abgeschaltet sind.

Weil die AD-Konvertierung langsam ist, kann bei deren Beendigung ein Interrupt ausgelöst werden, so dass das Programm während der Konvertierung weiter arbeiten kann.

Das 10-Bit Ergebnis wird in zwei SFR abgelegt, die wahlweise links oder rechts ausgerichtet sind. Wenn nur 8-Bit Auflösung benötigt werden, kann das Ergebnis links ausgerichtet werden, so dass man sehr einfach die unteren 2 Bits ignorieren kann ohne selbst shiften zu müssen. (Abbildung 4.12) Der PIC18 unterstützt keine Shift-Operation, diese kann durch die Rotate-Operation aber nachgebildet werden.

Das CCP2-Modul (siehe auf der nächsten Seite) kann mit dessen Special-Event-Trigger den ADC starten.

Timer / Counter: Es gibt vier Timer, die sich wahlweise als Timer oder Counter nutzen lassen.

Timer Ein Timer zählt die einzelne Takte des Oszillators.

Counter Ein Counter zählt fallende oder steigende Flanken an einem Eingang.

Die Timer/Counter sind 8- oder 16-Bit breit und können einen Prescaler nutzen, mit dem man nicht jeden einzelnen Takt oder Flanke zählt, sondern nur jede 2., 4. usw. Dadurch kann man die maximale Zeitdauer oder Anzahl der Flanken bis zum Überlauf in einem weitem Bereich einstellen.

Bei einem Überlauf des entsprechenden Registers kann ein Interrupt generiert werden.

Bei den 16-Bit Timer/Counter befindet sich das obere Byte in einem Latch-Register, das seinen Wert erhält, wenn das Low-Byte gelesen wird und dann nicht mehr verändert wird. Dadurch kann man sicher sein, dass High- und Low-Byte zusammengehören, auch wenn man sie nicht auf einmal auslesen kann.

Watchdog: Als Sicherheitsfunktion besitzt der PIC18F452 einen Watchdog, der, wenn er beim Programmieren aktiviert wurde, den PIC zurücksetzt, wenn das Programm nicht regelmäßig die *CLRWDT*-Instruktion aufruft.

Realisiert ist der Watchdog durch einen Timer mit internem RC-Oszillator, der auch im SLEEP-Mode arbeitet. Sobald der Watchdog-Timer einen Overflow hat, wird der Mikrocontroller entweder zurückgesetzt oder, falls er sich im SLEEP-Mode befand, aufgeweckt. Um dies zu verhindern muss regelmäßig die *CLRWDT*-Instruktion aufgerufen werden, die den Timer auf 0 setzt. Typischerweise muss das alle 18 ms (min. 7 ms) stattfinden, wenn man den Postscaler auf 1:1 gestellt hat. Mittels des Postscalers lassen sich diese Zeiten bis zum 128-fachen verändern. Da ein RC-Oszillator verwendet wird, dessen genaue Frequenz stark durch die Temperatur beeinflusst wird und auch großen Schwankungen in der Herstellung unterliegt, lässt sich nicht genau spezifizieren, welchen Takt der Watchdog-Timer hat.

Der Watchdog kann nicht garantieren, das ein Programm nicht abstürzt, aber dieses Risiko vermindern. Es bietet sich an, in Polling-Schleifen, in denen man darauf wartet, dass ein Hardware-Modul eine Funktion ausführt (z. B. Empfangen von Daten per USART), nicht *CLRWDT* aufgerufen wird, so dass für solche Schleifen automatisch eine Abbruch-Bedingung existiert, die dann allerdings direkt den Mikrocontroller neu startet. Im Programm kann man diese Art von Reset erkennen und geeignet behandeln.

CCP (Capture, Compare, Pulse Width Modulation): Es sind zwei CCP-Module vorhanden, die jeweils eine von drei Funktionen ausführen können.

Capture Im Capture-Modus wird der Wert eines Timers bei einer bestimmten Flanke eines Eingangs gespeichert, man kann so die Zeit bis zu dieser Flanke messen. Durch einen Prescaler kann man nur jede n-te Flanke erfassen. Durch den Capture-Vorgang kann ein Interrupt ausgelöst werden.

Compare Im Compare-Modus wird ein Ausgang entweder gesetzt, gelöscht oder verändert, wenn ein Timer einen bestimmten Wert hat. Zusätzlich kann dadurch ein Special Trigger Event ausgelöst werden, der den Timer zurücksetzt oder eine A/D-Konvertierung starten. Diesen Modus kann man zur Erzeugung eines periodischen Rechtecksignals nutzen oder bei Benutzung des Special Event Triggers ein analoges Signal mit einer bestimmten Frequenz samplen lassen.

Pulse Width Modulation In diesem Modus kann man ein pulsweitenmoduliertes Signal entsprechend Abbildung 4.13 erzeugen. Ein solches Signal hat eine programmierbare Perioden- und Pulsdauer. Dazu wird der Timer2 benutzt, der dann nicht mehr zur Verfügung steht. Diese Art von Signalen kann man zur Ansteuerung von Summern mit variabler Frequenz oder zur Ansteuerung von Gleichstrommotoren (wie den LEGO-Motoren aus Abschnitt 2.1.4 auf Seite 20) nutzen. Durch das Nachlaufen und die Trägheit des Rotors, dreht sich dieser mit einer Geschwindigkeit, die von der Pulsdauer abhängig ist. Die Periodendauer muss klein genug sein, damit die einzelnen Schaltvorgänge nicht bemerkt werden.

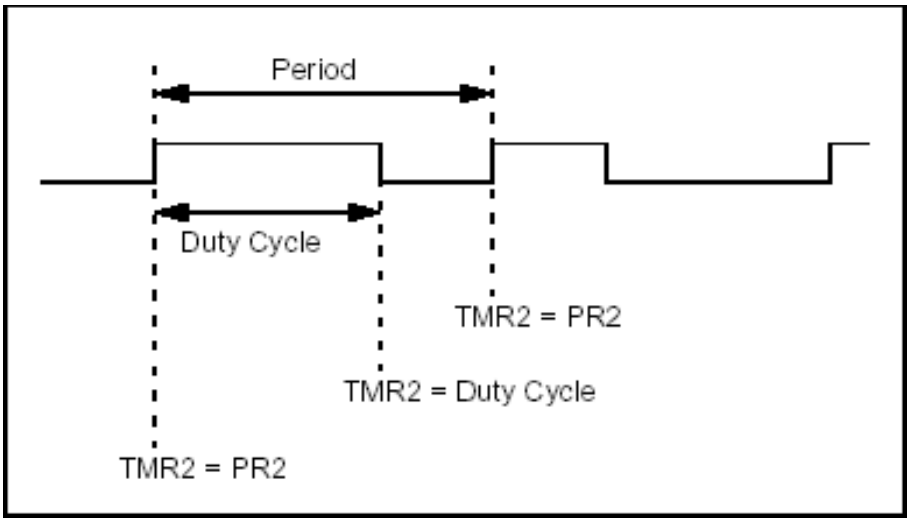


Abbildung 4.13: Pulsweitenmoduliertes Signal

MSSP (Master Synchronous Serial Port):

SPI Der MSSP implementiert zum einen ein Serial Peripheral Interface (SPI), eine viel genutzte serielle Schnittstelle von seriellen EPROMs, ADCs, DACs und anderen Chips. Mit Einschränkungen lässt sich auch der MicroWire-Standard nutzen, der die gleiche Funktion erfüllt.

I^2C Das MSSP implementiert ebenfalls die I^2C-Modi full master mode und den slave mode.

I^2C (Inter-Integrated Circuit) ist ein 2-Draht-Bus (plus Masse-Verbindung) mit einem Master (der Multimaster-Mode wird auch unterstützt) und mehreren Slaves, der zur Kommunikation innerhalb eines Systems genutzt wird. Da nur zwei Leitungen gebraucht werden, ist die Verwendung dieser Schnittstelle einfach und billig. Es existiert eine große Anzahl von Komponenten, die diesen Standard nutzen.

USART / SCI: Der USART (Addressable Universal Synchronous Asynchronous Receiver Transmitter) wird auch als SCI (Serial Communications Interface) bezeichnet.

Der USART unterstützt die folgenden Modi:

Synchronous Mode Eine halb-duplex synchrone Verbindung zwischen verschiedenen Peripherie-Geräten. Es wird sowohl der Master- als auch der Slave-Modus unterstützt.

Asynchronous Mode Dies dürfte die häufigste Verwendung des USART sein, da so eine asynchrone voll-duplex Verbindung nach dem RS232/RS485 Standards (mit Ausnahme der elektrischen Pegel, dafür muss dann ein Pegelumwandler (z. B. MAX232) eingesetzt werden) realisiert werden kann. So kann eine serielle Verbindung mit einem PC realisiert werden.

Der USART hat einen integrierten Baud-Raten-Generator und auch einen 1 Byte Receive-Buffer. Bei jedem empfangenen Byte und bei Sendebereitschaft kann ein Interrupt ausgelöst werden.

LVD (Low Voltage Detect): Die Low Voltage Detection kann benutzt werden, um bei einem batteriebetriebenen Gerät das Absinken auf eine konfigurierbare kritische Spannung zu detektieren um dann in der Interruptroutine noch die letzten Vorkehrungen (Daten im EEPROM speichern, Peripheriegeräte in einen

sicheren Zustand bringen oder einfach nur eine Warnung an den Benutzer auszugeben) vorzunehmen, bevor die Spannung endgültig zu niedrig ist, um den weiteren Betrieb des System zu gewährleisten.

Resetgeneratoren / Delays: Der PIC18F452 beinhaltet mehrere Komponenten, die den Schaltungsaufwand für einen sicheren stabilen Betrieb erheblich vermindern.

Zum einen ist ein Power-up Reset (POR) enthalten, der einen Power-up Timer beinhaltet, der dafür sorgt, dass sich die Spannungsversorgung erst stabilisieren kann, bevor der Mikrocontroller sein Programm ausführt. Zusätzlich wird noch ein Oscillator start-up Timer benutzt, da die Oszillatoren eine gewisse Zeit benötigen, um einen regelmäßiges Signal mit ausreichender Amplitude zu erzeugen.

Ein Brown-out Reset (BOR) ist zuschaltbar, um bei einem Absinken der Versorgungsspannung auf ein Niveau, bei dem der Mikrocontroller nicht mehr richtig funktioniert, den PIC zurück zusetzen. Bevor der Mikrocontroller neu gestartet wird, wird darauf gewartet, dass die Versorgungsspannung eine gewisse Zeit wieder im spezifizierten Bereich ist.

Durch diese Komponenten ist normalerweise kein externer Resetgenerator nötig.

Interrupts: Die Interrupts der PIC18 wurden gegenüber den älteren PICs stark verbessert. So wurde eine 2-Level Priorisierung eingeführt und es existiert der Fast Register Stack.

Alle Interruptquellen sind einzeln konfigurierbar, so dass man für jede Interruptquelle bestimmen kann, ob und mit welcher Priorität sie einen Interrupt auslösen darf. Für zeitkritische Abschnitte (z. B. beim Flashen) im Programm kann man auch alle Interrupts auf einmal abschalten.

Sobald ein Interrupt gefeuert wird, wird ein *CALL FAST* ausgeführt. Bei Low Priority Interrupts wird dabei an die Adresse 0x0018, bei High Priority Interrupts an die Adresse 0x0008 gesprungen. Dabei wird jeweils das Interrupt-Enable-Flag für die eigene und die niedrigere Priorität gelöscht, so dass die Low Priority Interrupts nur noch durch High Priority Interrupts unterbrochen werden können. Die Interruptroutine muss dann mit Hilfe der Interrupt-Flags in verschiedenen SFRs bestimmen, welche Quelle den Interrupt ausgelöst hat. Das entsprechende Flag muss dann gelöscht werden, damit dieser Interrupt nicht noch einmal gefeuert wird. Am Ende der Interruptroutine muss diese mit *RETFIE* verlassen werden.

Durch die Verwendung des Fast Register Stacks werden automatisch die wichtigen SFRs WREG, STATUS und BSR gesichert. Wenn die Interruptroutine dann mit *RETFIE FAST* verlassen wird, werden diese Register automatisch wieder zurückgeschrieben. Siehe auch auf Seite 58.

Kritikpunkte des PIC18F452

Leider muss man aber auch feststellen, dass es einige Probleme mit diesem Mikrocontroller gibt. So wurde im Verlauf des Projektes festgestellt, dass der Mikrocontroller einige Fehler enthält. Microchip pflegt auf seiner Homepage [Micd] das entsprechende Errata der Referenz. Diese Liste enthält zu jedem Fehler die Beschreibung der Umstände, die den Fehler hervorrufen und auch Workarounds dagegen, sowie die Angabe, ab wann die Prozessoren den Fehler nicht mehr enthalten. Diese Auflistung gibt aber leider nicht die exakten Umstände an, die zu den Fehlern führen und vor allem wird nicht der eigentliche Fehler beschrieben. Dadurch ist es nicht möglich, an einer Symptomatik, die man am eigenen System feststellt, herauszufinden, ob er durch einen der angegebenen Fehler verursacht wird. Auch die Angabe der Workarounds ist nicht immer befriedigend, da teilweise empfohlen wird, eine bestimmte Funktionalität nicht mehr zu nutzen, wenn man diese aber unbedingt benötigt, so ist man darauf angewiesen, zu hoffen, dass das eigene System diesen Fehler nicht hervorruft. Diese Ungenauigkeit des Errata ist wahrscheinlich auch darauf zurückzuführen, dass Microchip diese Fehler noch untersucht und somit noch nicht alle gewünschten Angaben hat.

Letztendlich verhinderten diese Fehler aber nicht die Realisierung, so dass sie nur die Implementierung der Software verzögerten.

Fazit

Da die oben genannten Eigenschaften alle Anforderungen an den Mikrocontroller einer Robotersteuerung abdecken, wurde der PIC18F452 ausgewählt. Einzig der Datenspeicher ist so klein, das man sich bei der Entwicklung der Software um den Speicherverbrauch besondere Gedanken machen muss. Dies schränkt aber nicht die Funktionalität ein, da bei den verfügbaren und geplanten Sensoren keine großen Datenmengen anfallen sollten.

Dieser Mikrocontroller zeichnet sich außerdem durch seine gute Verfügbarkeit auch bei Einzelhändlern und sein gutes Preis-/Leistungsverhältnis aus.

4.4.2 Der PIC12F675

Der von uns verwendete Prozessor PIC12F675 ist für seinen Preis und seine kleine Baugröße ein sehr leistungsfähiger Mikrocontroller. In Millionen von Computermäusen verrichtet er, bzw. ein verwandter Baustein, seine Dienste. Da der PIC18 von uns in der Zentraleinheit eingesetzt wird, ist es auch logisch auf einen Baustein, der in vielen Bereichen ähnlich (wenn auch erheblich leistungsschwächer) ist, zurückzugreifen. Hier eine Auflistung der für uns wichtigen Eigenschaften:

- Bis zu 20 MHz Taktfrequenz (externer Oszilator)
- Betrieb mit internem RC-Oszilator mit 4 MHz (Genauigkeitseinbußen)
- 1024*14 Bit Flashspeicher für das Programm
- 64 Byte Arbeitsspeicher
- 128 Byte EEPROM
- 6 der 8 Pins sind weitgehend freiprogrammierbar
- Interruptfähig (konfigurierbar)
- Interner A/D-Wandler mit 10 Bit Auflösung.
- Zwei Timer

Zusammenfassend und vereinfachend kann man sagen, dass der PIC 12 ein achtbeiniger Baustein ist, von dem sechs Pins vom Entwickler frei konfigurierbar sind. Damit ergeben sich folgende vorteilhafte Eigenschaften für das ROCK:

- Der Speicher ist ausreichend dimensioniert
- Der Mikrocontroller kann für alle derzeit geplanten Sensor-/Aktorbaugruppen verwendet werden
- Der A/D-Wandler ist besonders für die Sensorabfrage von Bedeutung
- Bei Verwendung des internen Taktgenerators können alle 6 Ein-/Ausgangspins benutzt werden

Die genauen Details zum PIC12 können dem Datenblatt [Mic02a] entnommen werden. PIC12 und PIC18 werden in Tabelle 4.8 gegenübergestellt. Bei diesem Projekt hat sich der PIC12 bewährt. Das Bauteil ist robust und es gab während der PG nur einen einzigen Ausfall aufgrund von mechanischer Überbeanspruchung. Lediglich eine größere Anzahl an Pins wäre wünschenswert, um die Peripherie-Schaltung einfacher gestalten zu können.

	PIC12F675	PIC18F452
Befehle	35	75
RAM	64 Byte	1536 Byte
Flash-ROM	1024 $\times$14 Worte	16384$\times$ 16 Worte
EEPROM	128 Byte	256 Byte
Taktfrequenz	bis 20 MHz	bis 40 MHz
Internes RC-Glied	Ja	Nein
I/O-Ports	6	34
Pins	8	44
Timer	2	4
A/D-Pins	4	8
Schnittstellen	-	MSSP, USART, I^2C, PSP
Interrupt Vektoren	1	2
Interrupt Quellen	7	18
Hardware Stack	8	31
Stromaufnahme	$<$ 1,0 mA	12 mA

Tabelle 4.8: PIC12F675 [Mic02a] und PIC18F452 [Mic02b] im Vergleich

4.4.3 MOSFETs

MOSFETs (Metal-Oxide Semiconductor Field-Effect Transistor) sind spannungsgesteuerte Transistoren. Es gibt sie in zwei Polaritäten:

- N-Typen – positive Gate-Source-Spannung,
- P-Typen – negative Gate-Source-Spannung

Die N- und P-Typen können jeweils unterschiedliche Gates besitzen:

- Anreicherungstypen (Enhancement) – selbstsperrend,
- Verarmungstypen (Depletion) – selbstleitend.

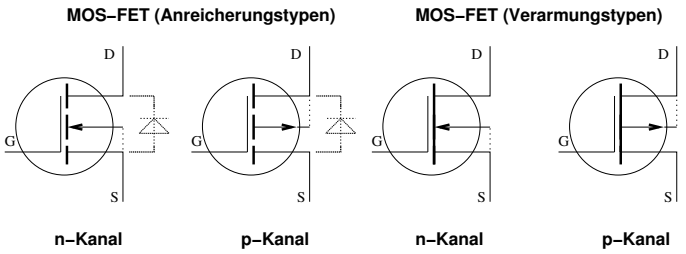


Abbildung 4.14: Schaltsymbolübersicht

Ein MOSFET besitzt (mindestens) drei Anschlüsse: Gate (G), Source (S) und Drain (D).

Zwischen Source und Drain kann ein Strom fließen. Bei Typen, die keine Diode besitzen (vgl. Abbildung 4.14 rechts oder das Datenblatt des Herstellers) kann der Strom in beide Richtungen fließen. Wenn zwischen Source und Drain eine Diode existiert (z. B. FDN335N, Abb. 4.14 links), kann in eine Richtung immer ein Strom fließen (z. B. von Source nach Drain), und in die andere Richtung (bspw. von Drain nach Source) fließt der Strom nur, wenn eine ausreichende Spannung zwischen Gate und Source anliegt.

Kenngröße	Wert
max. Strom	1,7 A
Einschaltzeit	32 ns
Ausschaltzeit	30 ns
minimale Schaltspannung (U_{GS})	1,5 V

Tabelle 4.9: wesentliche Eigenschaften des N-Kanal MOSFETs FDN335N (Quelle: [Faib])

Im sperrenden Zustand ist die Drain-Source-Strecke hochohmig, im leitenden Zustand ist der Widerstand kleiner als 1 Ω. Der Vorteil gegenüber Bipolartransistoren besteht darin, dass fast keine Spannung abfällt und zum Schalten nur eine Spannung erforderlich ist und fast kein Strom in das Gate fließt. Es wurden die MOSFET-Typen FDN335N und FDN304P ausgewählt, da sie einen geringen Widerstand im leitenden Zustand aufweisen und einem großen maximalen Strom standhalten. Unter den in SMD-Bauform lieferbaren MOSFETs sind die preisgünstigsten. Die Abbildungen 4.15 und 4.19 zeigen die Schaltsymbole. Die Source-Drain- Widerstände sind über den Drain-Strom bei verschiedenen Gate-Spannungen in den Abbildungen 4.16 und 4.20 dargestellt. Weitere wesentliche Eigenschaften der verwendeten MOSFETs sind in den Tabellen 4.9 und 4.10 wiedergegeben. (Quelle: [Faib] und [Faia])

Die MOSFETs sind äußerst empfindlich. Da höchstens 8 V zwischen Source und Gate anliegen dürfen, werden sie schon von geringer elektrostatischer Aufladung zerstört.

Da die Schaltspannung zwischen Gate und Source anliegen muss, sollte der Sourceanschluss auf einem nicht veränderlichen Potential liegen, damit die Schaltung einfacher wird.

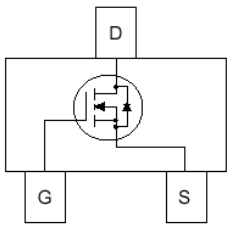


Abbildung 4.15: Schaltsymbol des FDN335N im Gehäuse (Quelle: [Faib])

Der N-Kanal MOSFET FDN335N ist selbstsperrend. Da N-Kanal MOSFETs ihre Source an Masse haben, kann die positive Gate-Source-Schaltspannung direkt vom Ausgang eines PICs geliefert werden.

Bei kleinen Schaltspannungen ($\leq$ 2 V) ist zu beachten, dass der Widerstand zwischen Drain und Source des MOSFET abhängig vom zu schaltenden Strom ist, da der MOSFET nicht voll durchgeschaltet hat.

Für P-Kanal MOSFETs, hier der FDN304P, gilt das Gleiche wie für die N-Kanal-Typen. Er ist auch selbstsperrend, aber wesentlich langsamer. Der Source-Anschluss sollte bei P-Kanal-Typen auf einem positiven Potential liegen, damit am Gate einfach eine relativ dazu negative Spannung angelegt werden kann.

Ist der Source-Anschluss des P-Kanal-MOSFETs mit 5 V verbunden, kann das Gate direkt mit dem Ausgang des PICs verbunden werden.

Soll der P-Kanal-MOSFET im 9 V-Strang schalten, dann ist der Source-Anschluss mit 9 V verbunden. Die Source-Gate-Spannung sollte zwischen 0 und -8 V liegen. Wird der PIC direkt mit dem Gate verbunden, liegt die Source-Gate-Spannung zwischen -4 V und -8 V. Es muss also eine kleine Ansteuerungsschaltung entworfen werden:

- Die einfache, in Abbildung 4.17 dargestellte, Schaltung benutzt einen Spannungsteiler.

  Es wird ein Spannungsteiler verwendet, so dass am Gate des MOSFETs auch weniger als -1 V anliegen. Dies ist der Fall bei $\frac{100k}{100k+470k\Omega} \cdot -4V = -0,7V$. Bei log. 0 liegen $\frac{100k}{100k+470k\Omega} \cdot -9V = -1,6V$

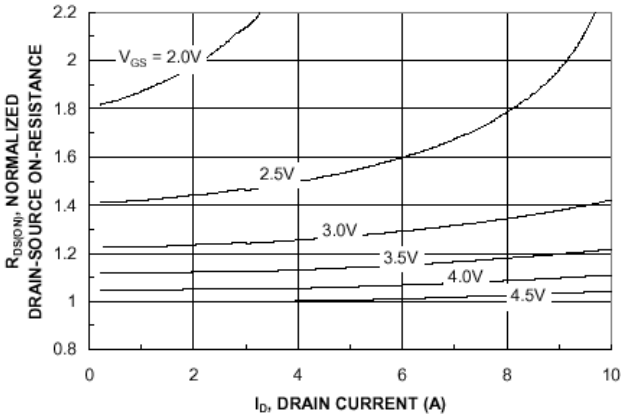


Abbildung 4.16: Widerstand des FDN335N über dem geschalteten Strom bei diversen Schaltspannungen (Quelle: [Faib])

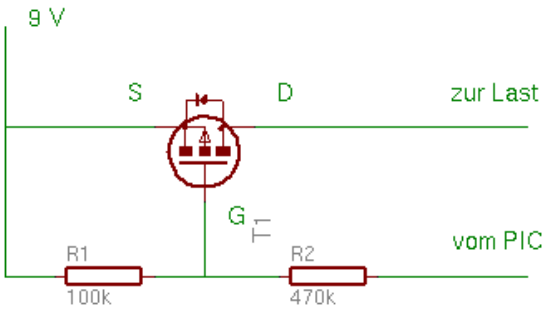


Abbildung 4.17: P-MOSFET mit Spannungsteiler schalten

an. Der MOSFET schaltet also nicht voll durch, wie man aus dem Diagramm in Abbildung 4.20 leicht entnehmen kann.

Der Vorteil ist die Einfachheit der Schaltung und die Tatsache, dass der MOSFET bei Auftreten von großen Strömen von selbst sperrt.

Ein wesentlicher Nachteil hingegen ist die Schaltzeit, die der MOSFET braucht. Bei 100 kΩ benötigt der MOSFET 595 μs zum Sperren. Mehr als 500 Hz bei einem Tastverhältnis von 1:1 sind nicht praktikabel. Der Grund ist darin zu suchen, dass das Gate eine nicht zu vernachlässigende Kapazität ist und somit ein Tiefpassfilter aufgebaut wurde.

Ein weiterer Nachteil ist natürlich, dass der MOSFET nicht richtig durchschaltet.

Eine andere Dimensionierung der Widerstände z. B. 1 kΩ und 4,7 kΩ würde auch dieser Schaltung eine größere Arbeitsfrequenz erlauben, eine brauchbare Flankensteilheit wäre aber auch dann nicht gegeben.

Noch kleinere Widerstände sind leider nicht praktikabel, weil sie einen zu großen Ruhestrom hervorrufen würden. Die Schaltung wäre nicht mehr sinnvoll mit einer Batterie zu betreiben.

- Eine kompliziertere Schaltung benutzt einen weiteren N-MOSFET (siehe Abb. 4.18), der nur wenig mehr Fläche auf der Platine als ein Widerstand benötigt.

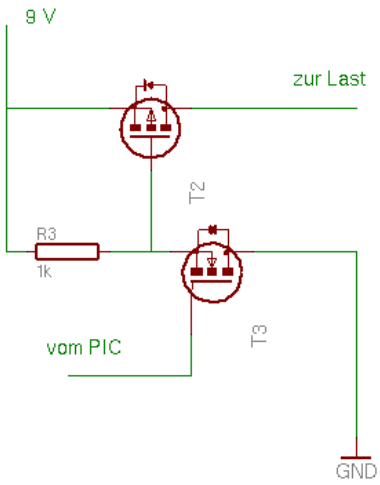


Abbildung 4.18: P-MOSFET mit N-MOSFET durchschalten

Wenn T3 sperrt, zieht der 1 kΩ-Widerstand das Gate von T2 auf 9 V und T2 sperrt ebenfalls.

Wenn T3 leitet, zieht er das Gate von T2 auf Masse; es liegen zwischen Gate und Source an T2 -9 V an, und T2 schaltet voll durch. Durch R3 fließen dabei 9 mA .

Um T2 leitend zu schalten, benötigt die Schaltung fast keine Zeit ($< 1\mu$s). Um T2 vom leitenden in den sperrenden Zustand zu schalten, wird wegen des 1 kΩ Widerstandes sehr viel Zeit benötigt. Ein Experiment hat ergeben, dass, wenn das Gate von T3 ab einer Frequenz von 47 kHz bei einem Tastverhältnis von 1:1 betrieben wird, T2 nur noch bis auf 0,9 V sperrt. Es stellte sich heraus, dass dieses Schaltverhalten für unsere Zwecke ausreichend ist. Wir verwenden nun diese Schaltungsvariante.

4.4.4 H-Brücke

Eine H-Brücke ist ein Baustein, der mittels vier Schalttransistoren (heute meist MOSFETs) bei einer Eingangsspannung einen Gleichstrommotor in beide Richtungen laufen lassen kann. Ordnet man die Schalttransistoren auf den Kanten eines Rechtecks an und zeichnet in der Mitte die Last (Motor) ein, sieht das

Kenngröße	Wert
max. Strom	2,4 A
Einschaltzeit	54 ns
min. Ausschaltzeit	104 ns
minimale Schaltspannung (U_{GS})	1,5 V

Tabelle 4.10: wesentliche Eigenschaften des P-Kanal MOSFETs FDN304P (Quelle: [Faia])

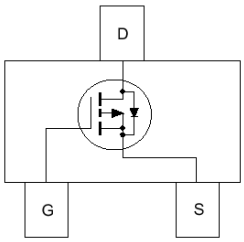


Abbildung 4.19: Schaltsymbol des FDN304P im Gehäuse (Quelle: [Faia])

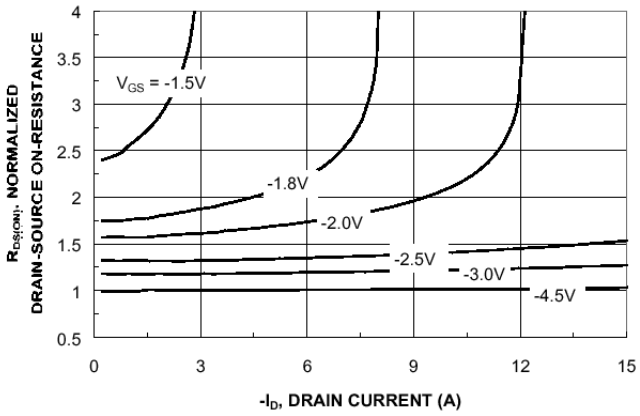


Abbildung 4.20: Widerstand des FDN304P über dem geschalteten Strom bei diversen Schaltspannungen (Quelle: [Faia])

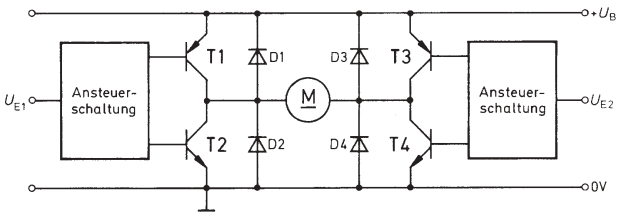


Abbildung 4.21: Prinzip einer H-Brücke aus [Kam92]

Schaltbild aus wie ein „H“. Die Zeichnung 4.21 verdeutlicht dies, wobei hier Bipolartransistoren als Schaltelemente eingezeichnet sind.

Die von uns ausgewählte H-Brücke HIP 4020 verfügt über folgende für uns wichtige Eigenschaften:

- 0,5 A maximaler Strom
- 15 V Betriebsspannung
- Schutzdioden für die Schalttransistoren
- innere Schutzbeschaltung mit optionaler Alarmmeldung durch Diode
- Ansteuerung über TTL-Logikpegel

Außerdem ist durch getaktetes Schalten eine Leistungsregulierung möglich – mehr zu diesem Thema findet sich im Abschnitt 4.4.1. Abgesehen vom Preis, der sogar über dem unseres PIC18-Prozessors liegt, hat dieser Baustein für uns nur positive Eigenschaften. Da dieser Baustein nur in SMD-Footprint vorliegt, musste eine kleine Adapterplatine angefertigt werden, um die Realisierung des Versuchsboards zu ermöglichen. In [sem] können alle weiteren technischen Details nachgelesen werden.

4.4.5 Der LCD-Baustein

In diesem Kapitel wird allgemein die Funktionsweise eines Dotmatrix LCD-Controllers erläutert. Für die PG420 wird das EA DIP122-5NLED Display der Firma Electronic Assembly genutzt. Dieses Display wird über den SED1520-Controller der Firma EPSON angesprochen.

Dot-Matrix-Displays gibt es in verschiedenen Größen, wie z. B. einzeilig, zweizeilig oder vierzeilig und mit 8 bis 40 Zeichen pro Zeile. Jedes Zeichen wird dabei in unserem Fall in einer Matrix aus 5x8 Punkten dargestellt.

Mit dem SED1520 Controller ist es möglich, über die „Segment outputs“ 61 Spalten und über die „Common outputs“ 16 Zeilen zu adressieren. Bei einer 5×8 Matrix pro Zeichen ergeben sich 2 Zeilen mit jeweils 12 Zeichen. Das eingebaute RAM, in dem die Zeichen gespeichert werden können, ist in diesem Fall 2560 Bits groß.

Es können auch mehrere solcher Controller zusammengeschlossen und im Master/Slave Modus betrieben werden, um die Größe des Displays zu verdoppeln. Um die Zeilenlänge zu erweitern gibt es auch die Möglichkeit, den Baustein durch den SED1521 zu erweitern, der die ansteuerbaren Spaltenleitungen um weitere 80 erweitert.

Interne und externe Organisation

Dank des Controllers HD44780 [Lin] der Firma Hitachi, der sich als Standard für derartige Controller durchgesetzt hat, ist die Ansteuerung solcher Displays weitgehend einheitlich.

Interface und Pin-Belegung

Hier wird zunächst als Referenz die Steckerbelegung eines Displays mit Hitachi-Controller erläutert (siehe 4.4.5). Das DIP122-5NLED [Elea] hat nur kleine Abweichungen, welche etwas später im Text erläutert werden. Die 1- und 2-zeiligen Displays (sowie 4-zeilige Displays mit bis zu 20 Zeichen pro Zeile) besitzen einen Reihe von 14 oder 16 Lötkontakten, bzw. eine einreihige Stiftleiste mit 14 oder 16 Kontakten. Die Kontakte 15 und 16 sind für den Anschluß der Hintergrundbeleuchtung (soweit vorhanden) vorgesehen. Sofern ein hintergrundbeleuchtetes Display nur 14 Kontakte am Anschluss hat, so sind die Beleuchtungsanschlüsse an einer anderen Stelle des Displays zu finden. Die Pins 1 bis 14 sind in der Regel identisch belegt.

Pin	Symbol	Pegel	Beschreibung
1	Vss	Masse	Masse GND
2	Vdd	+5V	Betriebsspannung +5V
3	Vo	0 .. 1,5V (-2..-5V)	Displayspannung (Kontrast)
4	RS	H/L	Register Select
5	R/W	H/L	H:Read / L:Write
6	E	H	Enable
7	D0	H/L	Datenleitung 0 (LSB)
8	D1	H/L	Datenleitung 1
9	D2	H/L	Datenleitung 2
10	D3	H/L	Datenleitung 3
11	D4	H/L	Datenleitung 4
12	D5	H/L	Datenleitung 5
13	D6	H/L	Datenleitung 6
14	D7	H/L	Datenleitung 7 (MSB)

Die Ansteuerung läuft hierbei folgendermaßen:

- Die Daten bzw. der Befehl mit Parametern werden auf den Bus an den Pins 7 bis 14 angelegt.
- Pin 4 wird im Falle eines zu schickenden Befehls HIGH, im Falle zu schickender Daten auf LOW gesetzt und Pin 5 bei einem Lesezugriff auf HIGH, bei einem Schreibzugriff auf LOW gesetzt.
- Nun wird der Enable Pin auf HIGH und wieder auf LOW gezogen. Die steigende Flanke veranlasst den Controller, die Pins 4 und 5 zu überprüfen und die Art des bevorstehenden Transfers festzustellen. Die darauffolgende fallende Flanke veranlasst den Controller, die auf dem Bus liegenden Daten zu lesen oder im Falle eines Lesezugriffs seitens der steuernden MCU Daten auf den Bus zu legen. Im letzteren Fall, also bei Daten, die von der MCU gelesen werden sollen, liegen diese mit einem Takt Verzögerung am Bus an.

Verdrahtung

Der Kreuzungspunkt zwischen Zeilenleitung und Spaltenleitung beschreibt hier ein Pixel, das auf den Arbeitspeicher abgebildet wird. Jedes Bit im RAM repräsentiert ein Pixel, das im Falle einer 1 an- und im Falle einer 0 ausgeschaltet wird. 5 Bytes im RAM bezeichnen also ein Zeichen auf dem Display. In Abblidung 4.22 auf der nächsten Seite ist eine Ansteuerung durch den HD44780 in einem Diagramm dargestellt.

Jedoch muss die interne Organisation von Zeilenleitungen (Common outputs) und Spaltenleitungen (Segment outputs) nicht notwendigerweise die Zeilen- und Spaltenorganisation auf dem Display

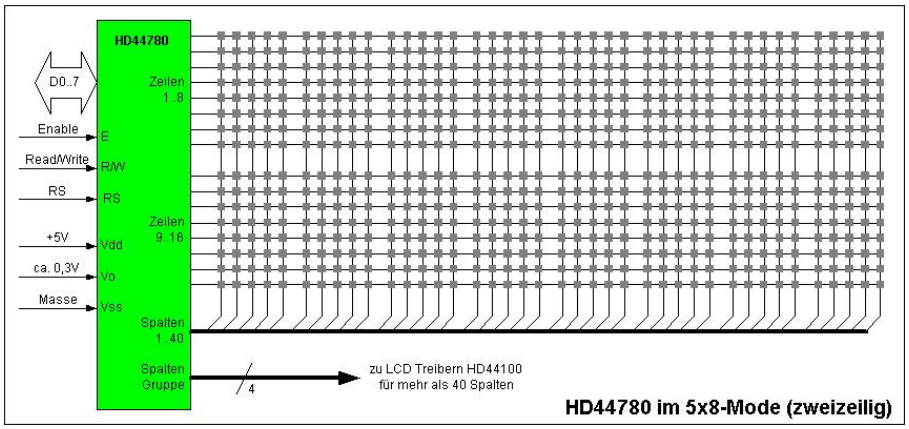


Abbildung 4.22: Ansteuerung eines dot-matrix Displays durch den HD44780 Controller

widerspiegeln. Um z. B. ein längeres einzeiliges Display herzustellen, wird nicht notwendigerweise ein Erweiterungsbaustein wie der SED1521 oder der HD44100 eingesetzt, sondern die beiden Zeilen des Controllers werden einfach hintereinander angeordnet. Hieraus ergibt sich das Problem, dass das Ansprechen der richtigen Zeile nicht immer auf dem offensichtlichen Weg erfolgt.

In dem von uns eingesetzten EA DIP122-5NLED arbeiten zwei SED1520 Controller, die mit Hilfe der Erweiterungsbausteine jeweils vier untereinander angeordnete Zeilen ansprechen (entnommen aus der Dokumentation [Eleb] [Elea]).

1x16 Display

Wird für ein Display mit 1x16 Zeichen z. B. der Controller von Hitachi verwendet, so muss dieser um weitere Spalten erweitert werden. Dies kann auf zwei Arten geschehen: entweder werden die zwei Zeilen hintereinander angeordnet und das Display muss trotz einer Displayzeile zweizeilig initialisiert werden (ein sog. 8+8 Display), oder der Controller wird erweitert und das Display muss nur einzeilig initialisiert werden. Einen Hinweis kann die Rückseite des Displays geben. Ist dort nur ein Controller wie in Abb. 4.23 sichtbar, handelt es sich wahrscheinlich um ein 8+8 Display. Ist dagegen noch zusätzlich ein zweiter Baustein erkennbar, kann es sich um ein echtes 1x16 Display handeln.

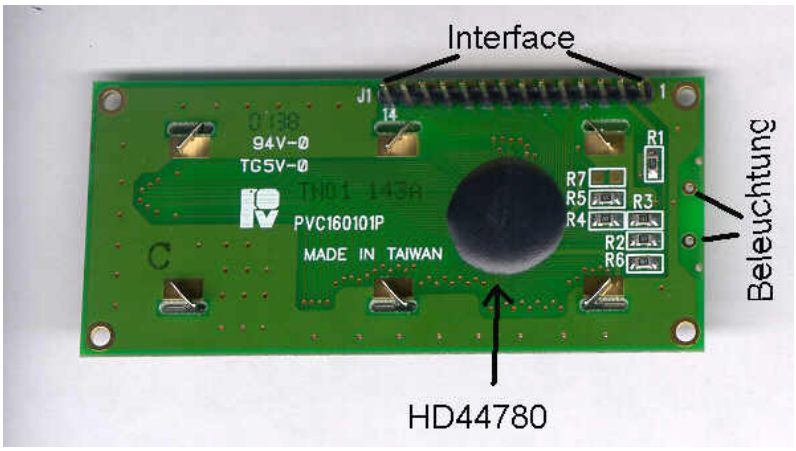


Abbildung 4.23: Die Rückseite eines 1x16 Displays

4x20 Display

Ein solches LCD kann mit nur einem Controller und zwei Erweiterungsbausteinen aufgebaut sein. Hier werden zunächst die zwei Zeilen auf insgesamt 40 Zeichen erweitert, logisch in der Mitte getrennt und untereinander angeordnet. Hier ergibt sich das Problem, dass die Zeichen 21 bis 40 der ersten Zeile des Controllers in die dritte Zeile des Displays abgebildet werden. Entsprechend werden Zeichen 21 bis 40 der zweiten Zeile des Controllers in die vierte Zeile des Displays abgebildet.

Das 4x24 dot-matrix Display EA DIP122-5NLED

Anschluß der Kontrastspannung

Es wird eine Spannung zur Einstellung des Displaykontrastes benötigt. Es gibt zwei unterschiedliche Displaysorten, die auch unterschiedliche Kontrast-Spannungen benötigen:

Standard-Displays (Temperaturbereich: 0°C ...50°C) benötigen eine Konstrastspannung zwischen 0V und 1,5V. Großdisplays und Displays für hohe Umgebungstemperaturen (Temperaturbereich: -20°C . . . 70°C) benötigen aber oft eine Spannung von -2V . . . -5V, was ihren Einsatz kompliziert. Bei vielen „no-name“-Displays hat man das Problem, dass nicht gekennzeichnet ist, ob es sich um ein Display für hohe Temperaturen handelt. Speist man es nicht mit einer negativen Kontrastspannung, so ist auf dem Display kein Zeichen erkennbar.

Abb. 4.24 verdeutlicht den Anschluß des Displays und die Erzeugung der Kontrast-Spannung für Standard-Displays. Durch eine Änderung der Spannung läßt sich der Kontrast und der optimale Blickwinkel zum Display verändern.

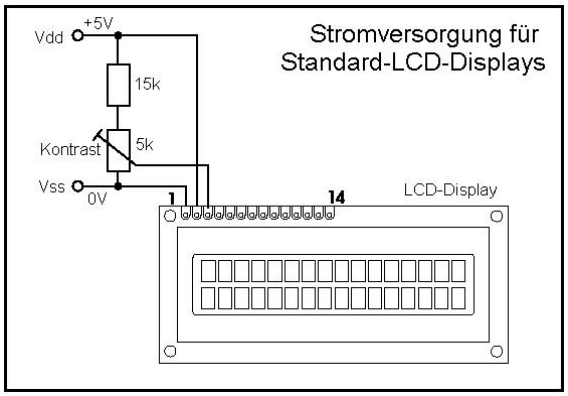


Abbildung 4.24: Beschaltung mit negativer Kontrastspannung

Interface und Ansteuerung

Mit Hitachi Controllern ist es üblich, bei vierzeiligen Displays mit mehr als 20 Zeichen pro Zeile je einen Controller zwei Zeilen steuern zu lassen. Bei dem vorliegenden Display ist je ein SED1520-Controller, der ohne Erweiterungsbaustein auch mehr Spalten als der HD44780 ansprechen kann, für vier Zeilen einer Hälfte „zuständig“.

Das Interface des eingesetzten Displays ist mit 2×9 Pins zweireihig angeordnet, mit zwei seperaten Enable-Pins für jeweils einen Controller. Die restlichen Pins sind parallel an beide Controller angeschlossen.

Interface und Pin-Belegung

für die linke Steckerleiste gilt folgende Belegung:

Pin	Symbol	Pegel	Beschreibung
1	Vss	L	Masse GND
2	Vdd	H	Betriebsspannung +5V
3	Vo	0 . . . 1,5V (-2 . . . -5V)	Displayspannung (Kontrast)
4	A0	H/L	Register Select
5	R/W	H/L	H:Read / L:Write
6	E1	H	Enable linke Displayhälfte
7	D0	H/L	Datenleitung 0 (LSB)
8	D1	H/L	Datenleitung 1
9	D2	H/L	Datenleitung 2

und für die rechte Hälfte gilt:

Pin	Symbol	Pegel	Beschreibung
10	D3	H/L	Datenleitung 3
11	D4	H/L	Datenleitung 4
12	D5	H/L	Datenleitung 5
13	D6	H/L	Datenleitung 6
14	D7	H/L	Datenleitung 7 (MSB)
15	E2	H	Enable rechte Displayhälfte
16	RES	L	Reset
17	A	-	LED-Beleuchtung +
18	C	-	LED-Beleuchtung -

Hier wird auch deutlich, dass man sich bei der Pin-Belegung an die Belegung der HD44780 Displays mit einem Controller gehalten hat. Lediglich der Pin zum Umschalten zwischen Daten und Befehlen wurde hier A0 genannt. Zusätzlich ist noch ein zweiter Enable-Pin und ein Reset-in hinzugekommen. Die Pins für die Hintergrundbeleuchtung sind bei HD44780 Displays, falls vorhanden, auf Pin 15 und 16. Beim DIP-5NLED sind diese auf Pin 17 und 18 verschoben.

Die Ansteuerung erfolgt wie oben beim Hitachi Controller beschrieben. Es muss nur der gewünschte EnablePin (für die rechte oder linke Hälfte des Displays) angesteuert werden, wobei darauf geachtet werden sollte, dass zunächst einmal die Art der Daten (mit Setzen/Löschen des A0 Bits) „gewählt“ bzw. die Art des Zugriffs (R/W) festgelegt werden sollte, bevor eines der Enable Signale gesendet wird (E0/E1). Wird das E0 Bit gesetzt, hat dies noch keine Auswirkung, außer dass man sich für eine der beiden Displayhälften entschieden hat. Erst das Löschen des Bits bewirkt ein Lesen der Daten, die am Bus anliegen.

Busy Flag

Werden interne Operationen vom SED1520 ausgeführt, ist das Display nicht ansteuerbar. Um nicht Gefahr zu laufen, aufgrund einer zu schnellen Abfolge der Befehlssequenz, Befehle zu „verlieren“, kann das sog. „Busy-Waiting“ realisiert werden. Während der Operationsphase ist der einzige Befehl auf den das LCD reagiert, der „Status-Read“ Befehl. Dieser nutzt Pin D7 als Ausgabe, um Bereitschaft zu signalisieren.

Eine andere Vorgehensweise ist es, zwischen dem Senden der Befehle immer die maximal mögliche Zeit abzuwarten, was natürlich nicht so effektiv wie das „Busy-Waiting“ ist. Ist jedoch der SED1520 Baustein höher getaktet als der ansteuernde Microcontroller, so müssen keine Vorkehrungen getroffen werden, um die Bereitschaft des LCD sicherzustellen. Der PIC18 ist mit 40 MHz getaktet langsam genug, dass der PIC nicht warten muss und auch das Buy-Flag nicht ausgewertet werden muss.

Display Data RAM

In diesen Speicher werden die darzustellenden Bilddaten gespeichert. Ein Bit im Speicher repräsentiert ein Pixel auf dem Display. Die Displayhälfte wird durch das schon bekannte A0 Signal gewählt.

Auf jeder Hälfte findet man die Aufteilung wie in Abb. 4.25 vor. Die Seite („Page“) wird durch das entsprechende Schreiben einer Seitenzahl in das 2 Bit „Page Register“ angesprochen. Das „Column Address Counter“ Register muss nun die Nummer der zu beschreibenden Spalte enthalten. Nach dem Beschreiben wird diese automatisch erhöht. Dieses Verhalten kann mit dem „Read-Modify-Write“ Befehl geändert werden. Wird der Modus geändert, so wird die Spaltennummer nicht automatisch erhöht. Sinnvoll ist dieser Modus, sobald der aktuelle Inhalt zuerst gelesen werden muss, bevor ein neuer hineingeschrieben wird. Beispielsweise möchte man bei einem blinkenden Cursor, den Status des aktuellen Bits negieren. Dieser Modus wird erst wieder beendet, sobald ein „End“ Befehl gesendet wird.

Scrollen

Ein weiterer, einfach zu implementierender Effekt ist das Scrollen des Bildschirminhalts. Das „Display Data RAM“ ist größer als das darstellbare Gebiet auf dem Display. Dies bedeutet, dass mehr Inhalte gespeichert werden, als auf dem Display dargestellt werden kann. Zum Scrollen verwendet man das sog. „Display Start Line“ Register. Hier wird einfach die Zeilennummer hineingespeichert, welche der ersten Zeile auf dem Display entsprechen soll. Inkrementiert man diese Zahl um einen kleinen, konstanten Wert, scrollt der Inhalt auf dem Display entsprechend.

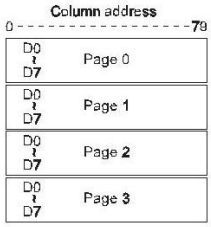


Abbildung 4.25: Die logische Aufteilung des Displays

4.4.6 Der Ultraschallsensor SRF08

Der Ultraschallsensor SRF08 [Liv] von Devantech Ltd (Robot Electronics) misst die Entfernung zu Objekten im Bereich zwischen 3 cm und 6 m. Abbildung 4.26 zeigt das Sonar von vorn und hinten. Es wird der I^2C-Bus genutzt um den einfachen und standardisierten Anschluss von bis zu 16 SRF08 zu ermöglichen. Zusätzlich zur Entfernungsmessung misst das Sensormodul auch noch die Helligkeit.

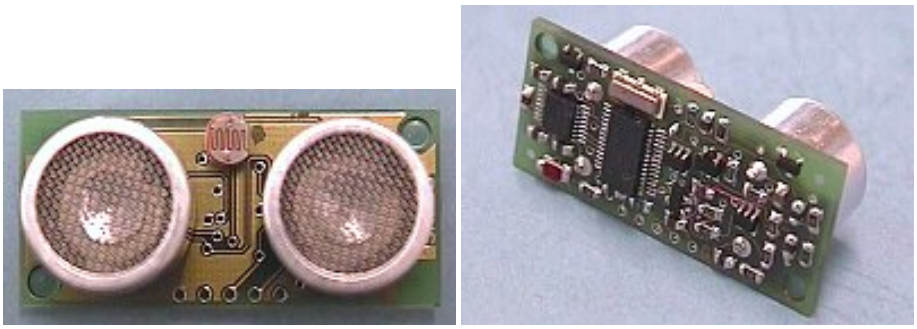


Abbildung 4.26: Sonar (SRF08) von vorn (links) und hinten (rechts)

Die Ansteuerung des Sensormoduls über das I^2C ist dabei der Ansteuerung der bekannten 24xx EEPROM Serie nachempfunden. Es wird immer erst eine Adresse für das zu lesende/schreibende Register gesendet und der nächste Befehl liest oder schreibt den Inhalt des entsprechenden Registers. Die Bedeutung der einzelnen Register ist der Spezifikation [Liv] zu entnehmen. Eine Messung wird eingeleitet, indem man

an Register 0 einen bestimmten Wert schreibt, der die Messung startet. Dabei kann man auswählen, ob man später die Messergebnisse in μs Laufzeit des Schalls oder umgerechnet in Zentimeter oder Zoll, lesen möchte. Beim Start einer Entfernungsmessung wird auch der Lichtsensor aktiviert und kann danach ausgelesen werden. Bis zum Ende der Entfernungsmessung (ca. 65 ms) kann auf das Sensormodul nicht zugegriffen werden. Es ist möglich, den Messbereich einzuschränken und somit die Entfernungsmessung zu beschleunigen, allerdings kann dies leicht zu Fehlmessungen führen, so dass von dieser Möglichkeit kein Gebrauch gemacht wurde. Zur Ansteuerung des Sensors in SOUL siehe Abschnitt 4.5.1 auf Seite 82.

Laut Spezifikation können bis zu 17 Echos gemessen werden, doch bereits erste Tests ergaben, dass nur der Wert für das erste Echo ein verwertbares Ergebnis lieferte. Der Grund für dieses von der Referenz abweichende Verhalten konnte nicht gefunden werden. Auch wurde bei den Tests festgestellt, das der Öffnungswinkel des Ultraschallsensors mit 90° sehr groß ist. Dadurch wird es schwer einzelne Objekte mit dem Sensor anzupeilen, da es wahrscheinlich ist, dass sich im erfassten Volumen des Sensors mehrere Objekte befinden. Somit ist die Auswertung der Messergebnisse (siehe Abschnitt 4.7.3 auf Seite 109) mit geeigneten Algorithmen vorzunehmen. Auch ist zu beachten, das die Ultraschallwellen unter Umständen von unerwarteten Gegenständen reflektiert werden. Bei der von uns durchgeführten Drop-Zone-Mission, die auf mehreren Tischen aufgebaut war, gab es anfangs Reflexionen von dem Metallrahmen unter der Tischplatte. Diese Fehlmessungen konnten aber durch ein erhöhtes Platzieren des Sensors behoben werden.

4.4.7 IR-Bauteile

IR-Bauteile

Die IR-Diode: Als IR-Sendediode findet ein preiswertes Standardbauteil, eine GaAs-IR-Lumineszenzdiode des Typs LD 271 [Sie], Verwendung.

Dieses Modell zeichnet sich unter anderem durch seine hohe Zuverlässigkeit und Impulsbelastbarkeit aus.

Von den LEDs, die sichtbares Licht emitieren, unterscheiden sich IR-Dioden durch ihren geringeren Wirkungsgrad, der für eine vergleichbare Leuchtstärke einen höheren Durchflussstrom erfordert. Mit steigender Stromstärke muss jedoch die Einschaltzeit verringert werden, um das Bauteil nicht zu überlasten. So empfiehlt der Hersteller, einen Strom von 100 mA nicht länger als 20 ms anliegen zu lassen, soll die Stromstärke gar 1 A betragen, muss die Leuchtdauer auf 100 μs verkürzt werden. Ein Dauerstrom von 2 mA stellt hingegen kein Problem dar.

Der IR-Empfänger: Zum Empfang der IR-Signale dient ein kompakter TSOP 1838 von Vishay Telefunken. Dieses sogenannte „Photomodul für PCM Fernbedienungssyteme“ vereint mehrere Baugruppen in einem dreipoligen Gehäuse. Dazu gehört der eigentliche Empfänger samt eines Vorverstärkers und Filter, die sowohl gegen Störlicht schützen als auch für ein von Störimpulsen freies stabiles Ausgangssignal sorgen.

Den inneren Aufbau des TSOP verdeutlich die Abbildung 4.27, die dem Datenblatt [Vis] entnommen wurde.

Ein weiterer Vorteil dieses Moduls ist, dass es fast vollständig ohne zusätzliche externe Bauteile auskommt. Lediglich ein Konsensator und zwei Widerstände werden benötigt, und das auch nur, um das Ausgangssignal von Spannungsschwankungen in der Betriebsspannung zu entkoppeln. Der Ausgang kann dadurch direkt mit dem PIC12, der das IR-Signal dekodiert, verbunden werden.

Den simplen Aufbau und Einsatz einer solchen Schaltung zeigt Abbildung 4.28, welche ebenso aus dem Datenblatt stammt.

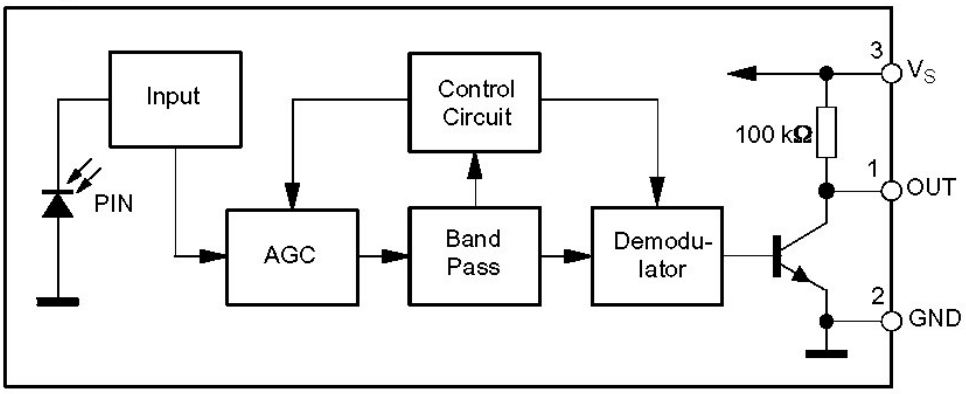


Abbildung 4.27: Blockdiagramm der TSOP 18xx-Familie

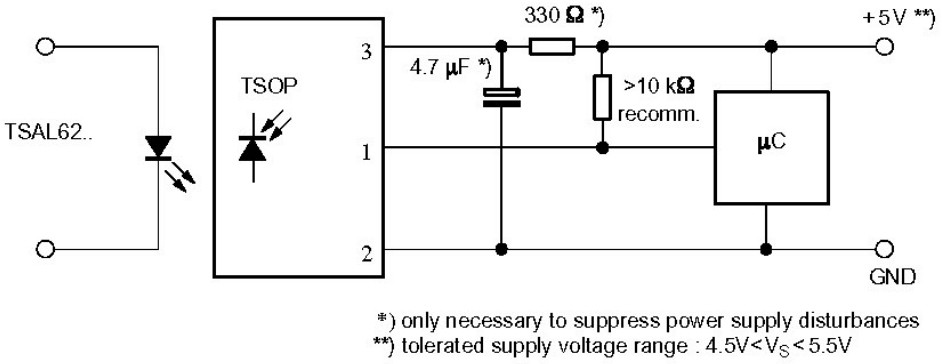


Abbildung 4.28: externe Beschaltung der TSOP 18xx-Familie

4.5 Die ROCK -Software

4.5.1 Anatomie von SOUL

Als Betriebssystem für ROCK wurde SOUL implementiert. Dieses Betriebssystem soll die Ziele aus Abschnitt 3.3 auf Seite 28 verwirklichen.

Wie bereits in Abschnitt 4.1 auf Seite 35 angedeutet, wird bei der Entwicklung darauf geachtet, möglichst flexibel zu sein. So soll SOUL auf mindestens drei jeweils leicht verschiedenen Zielsystemen (PICDem-Board, ROCKBoard und LCDBoard, außerdem auch dem ROCKEvalBoard) laufen. Die jeweils nötigen Hardware-Unterschiede werden durch konditionelle Assemblierung getrennt implementiert. Auch Parameter von SOUL sollten, wenn möglich, schnell veränderbar sein, so dass man noch leicht Anpassungen vornehmen kann. So ist zum Beispiel die Frequenz des Oszillators, die Anzahl der Tasks, die Anzahl der Synchronisationsobjekte und die Größen der verschiedenen Buffer per Define festgelegt und SOUL kann in deren gültigen Wertebereich (bei dessen Überschreitung die Assemblierung mit einem Fehler abbricht) ohne weitere Änderung übersetzt werden.

SOUL ist in mehrere Module aufgeteilt, die sich der Abbildung 4.29 auf der nächsten Seite entnehmen lassen und nachfolgend erklärt werden. Einzelne Module können von der Assemblierung per Define aus *define.inc* ausgeschlossen werden und belegen somit keinen Platz im knappen Speicher.

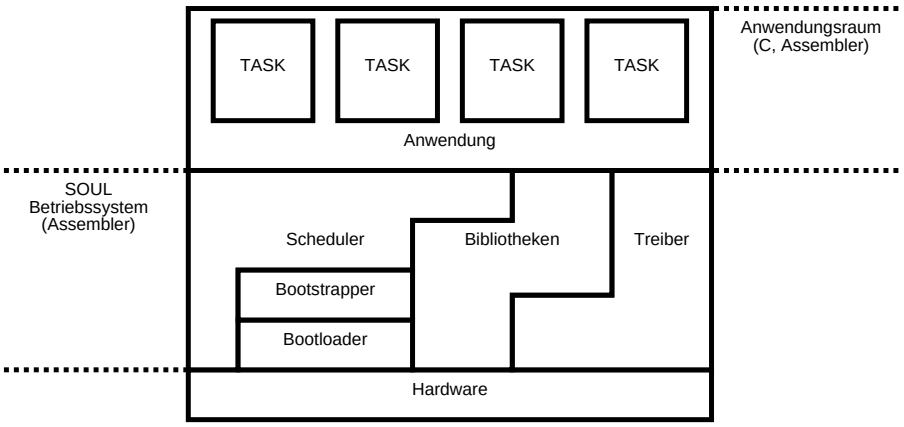


Abbildung 4.29: Aufbau von SOUL

Bootloader

Unser Bootloader besitzt (im Gegensatz zu dem der RCX-Einheit) nur eine absolut notwendige Basisfunktionalität. Alle darüber hinausgehenden Aufgaben werden vom Betriebssystem (SOUL) erledigt. Das bedeutet, der Bootloader kann erkennen, ob eine gültige Version von SOUL vorliegt und diese starten (dabei gibt er die Kontrolle komplett an SOUL ab), andernfalls erwartet er, dass Daten über die RS232-Schnittstelle vom PC eingespielt werden.

Das Bootloaderprotokoll der ROCK-Einheit: Die Syntax für die Übertragung der Daten wird im Folgenden vorgestellt.

Syntax des Protokolls: Die Syntax des ROCK-Protokolls sieht wie folgt aus:

```
<STX><STX><data><CHKSUM><ETX>
```

Der `data` Bereich variiert, je nachdem, was übertragen werden soll. Die Syntax des `data` Bereichs sieht dabei so:

```
<COMMAND><DLEN><ADDRL><ADDRH><ADDRU><DATA>
```

Die Symbole des Protokolls werden in der Tabelle 4.11 auf der nächsten Seite erläutert.

Erläuterung anhand eines Beispiels:

Im folgenden Beispiel (Tabelle 4.12 auf der nächsten Seite) werden acht Byte an die Addresse 0x200 in den Speicher des PIC18 geschrieben und anschließend noch einmal gelesen, um das Schreiben zu verifizieren.

Hier wird schon ersichtlich, dass `DLEN` je nach Command eine andere Bedeutung hat, die u. a. im Folgenden erläutert wird.

Zur Zeit exisieren die folgenden acht Kommandos:

- 0x00: Read Version
  Senden:

Symbol (Wert)		Erklärung
STX (0xF)		Kennzeichnet den Beginn eines Paketes (wenn es zweimal vorkommt)
COMMAND		Das Kommando, das mit dieser Transmission ausgeführt werden soll
DLEN		Länge der Daten, die zu dem Kommando gehören
ADDR		Die Zieladdresse der Daten (24Bit)
DATA		Die eigentlichen Daten (bis zu 255 bytes)
CHKSUM		Das 8-bit 2er Komplement der Summe von LEN und DATA
ETX (0x4)		Kennzeichnet das Ende eines Paketes
DLE (0x5)		Kennzeichnet ein gestopftes Byte

Tabelle 4.11: Definition der Symbole des Bootloaderprotokolls

Beschreibung	2xSTX	CMD	DLEN	ADDR L/H/U	DATA	CS	ETX
Schreiben	0F 0F	02	01	00 02 00	16 EF 01 F0 FF FF FF FF	09	04
Lesen	0F 0F	01	08	00 02 00		F5	04
Antwort	0F 0F	01	08	00 02 00	16 EF 01 F0 FF FF FF FF	03	04

Tabelle 4.12: Schreiben und Lesen mit dem ROCK-Protokoll

`<STX><STX>[<0x00> 0x01]<0xFF><ETX>`
Für diesen Befehl ist nur das Kommando wichtig. 0x01 könnte auch durch einen anderen Wert ausgetauscht werden – dabei muss dann die Prüfsumme angepasst werden.
Empfangen:
`<STX><STX>[<0x00> 0x02 <VERL><VERH>]<CHKSUM><ETX>`
Empfangen werden zwei Byte – die Minor-(VERL) und Major-(VERH) Versionsnummer.

- 0x01: Read Program Memory
  Senden:
  `<STX><STX>[<0x01><DATALEN><ADDRL><ADDRH><ADDRU>]<CHKSUM><ETX>`
  Hier gibt DATALEN die Anzahl in Bytes an, die aus dem Speicher der angegebenen Adresse gelesen werden soll.
  Empfangen:
  `<STX><STX>`
  `[<0x01><DATALEN><ADDRL><ADDRH><ADDRU><DATA>]`
  `<CHKSUM><ETX>`
  DATALEN, ADDRL und ADDRH wie in der Anfrage und in DATA stehen die gelesenen Daten des entsprechenden Speicherbereichs.

- 0x02: Write Program Memory
  Senden:
  `<STX><STX>`
  `[<0x02><DATALEN><ADDRL><ADDRH><ADDRU><DATA>]`
  `<CHKSUM><ETX>`
  In diesem Fall bedeutet DATALEN die Anzahl der Blöcke, die in den Speicher geschrieben werden sollen. Ein Block entspricht 8 Byte.

- 0x03: Erase Program Memory
  Senden:
  `<STX><STX>[<0x03><DATALEN><ADDRL><ADDRH><ADDRU>]<CHKSUM><ETX>`
  Beim Löschen gibt DATALEN die Anzahl der zu löschenden Zeilen an. Eine Zeile entspricht 64 Bytes.
  Empfangen:
  `<STX><STX>[<0x03>]<CHKSUM><ETX>`

Gibt an, dass der letzte Löschbefehl korrekt abgearbeitet wurde. Keine Antwort weist auf einen Fehler hin.

- 0x04: Read EE Memory
  Senden:
  `<STX><STX>[<0x04><DATALEN><ADDRL><ADDRH><0x00>]<CHKSUM><ETX>`
  Wie beim Lesen des Programmspeichers bedeutet DATALEN die Anzahl der zu lesenden Bytes.
  Empfangen:
  `<STX><STX>`
  `[<0x04><DATALEN><ADDRL><ADDRH><0x00><DATA>]`
  `<CHKSUM><ETX>`
  DATALEN, ADDRL und ADDRH wie in der Anfrage und in DATA stehen die gelesenen Daten des entsprechenden Speicherbereichs.

- 0x05: Write EE Memory
  Senden:
  `<STX><STX>`
  `[<0x05><DATALEN><ADDRL><ADDRH><0x00><DATA>]`
  `<CHKSUM><ETX>`
  Hier entspricht DATALEN der Anzahl der Bytes, die geschrieben werden sollen.
  Empfangen:
  `<STX><STX>[<0x05>]<CHKSUM><ETX>` Gibt an, dass der letzte Befehl korrekt abgearbeitet wurde. Keine Antwort weist auf einen Fehler hin.

- 0x06: Read Configuration Memory
  Analog zu Read Program Memory

- 0x07: Write Configuration Memory
  Analog zu Write Program Memory

Zu beachten ist dabei noch, dass man zwar Blöcke von 8 Byte schreiben kann, diese jedoch zuvor löschen muss. Der Grund ist, dass der zu beschreibende Speicher auf 1 stehen muss, denn beim Schreiben kann nur eine 1 in eine 0 geändert werden, nicht umgekehrt. Das hat zur Folge, dass auch wenn nur 8 Bytes geschrieben werden sollen, trotzdem 64 Byte gelöscht werden müssen (eine Zeile). Siehe auch Abschnitt 4.4.1 auf Seite 56.

Bootstrapper

Der Bootstrapper ist die Initialisierungsroutine von SOUL. Sie hat die Aufgabe, das Betriebssystem zu starten. Zu diesem Zweck muss natürlich auch die Hardware konfiguriert werden, denn auch die Treiber definieren Init-Funktionen, die dann vom Bootstrapper aufgerufen werden.

Tasks

SOUL ist als preemptives Multitasking-Betriebssystem ausgelegt, das quasi-parallel mehrere unabhängige Tasks ausführt. Dazu wird den Tasks nacheinander Rechenzeit zugeteilt, um sie ihnen nach kurzer Zeit wieder zu entziehen.

Wenn ein Task darauf wartet, dass ein bestimmtes Ereignis eintritt oder der Task auf exklusiven Zugriff auf eine Resource wartet, so wird die Wartezeit nicht vergeudet, sondern vom Scheduler an andere Tasks abgegeben und der wartende Task wird blockiert.

Normalerweise wird der User sein Programm in einem oder mehreren Tasks implementieren.

Von SOUL wird ein SystemTask definiert, der sogenannte IdleTask. Seine Existenzberechtigung liegt darin, dass theoretisch alle anderen Tasks blockiert sein können, z. B. weil sie alle Zugriff auf die gleiche Critical-Section verlangen, und der Scheduler dann keinem Task mehr Rechenzeit zuordnen kann. Dies ist deswegen problematisch, da in diesem Fall keine Interrupts mehr bearbeitet werden könnten und somit das System in einen Dead-Lock geriete.

Scheduler

Der Scheduler ist das Herz des Betriebssytems, da er als einziger Teil von SOUL regelmäßig aktiviert wird und dann den Context Switch durchführt. Dabei wird der aktuelle Task durch einen anderen, nicht blockierten Task ersetzt. Von allen Tasks wird der Context im Datenspeicher gesichert.

Zum Kontext eines Tasks gehört der Return-Stack, in dem die Rücksprungadressen der aktuell aufgerufenen Funktionen gespeichert sind sowie das Working Register (WREG), das STATUS Register, das Bank Select Register (BSR) und die Register PRODL, PRODH, TBLPTRL, TBLPTRH, TBLPTRU, TABLAT, PCLATH, PCLATU und die File Select Register (insgesamt sechs Register). Da eine gemeinsame Sicherung dieser Register zu viel Speicher und Zeit benötigen würde, wurden einige Optimierungen implementiert: Es wird jeweils nur ein Teil des 31-stufigen Return-Stacks für jeden Task reserviert. Außerdem werden die Register, die das UPPER-Byte von Adressen im Programmspeicher enthalten, ignoriert, da unser PIC18F452 nur $32K = 2^{15}$ Byte Programmspeicher hat; man also immer mit 15-bit Adressen auskommt. Dies spart alleine für den größten Teil des Kontext, den Return-Stack, ein Drittel des benötigten Speichers. Zusätzlich befinden sich im Return-Stack jeweils mehrere Tasks, die erst gemeinsam alle 31 Level des Return-Stack vollständig belegen. Für die Tasks, die nicht im letzten Teil des Return-Stacks liegen, wird nicht der Return-Stack im Speicher gesichert, wodurch wiederum Speicher eingespart wurde. Nur der letzte Teil des Return-Stacks wird von mehreren Tasks abwechselnd genutzt, so dass für diese ihr jeweiliger Teil des Return-Stacks gesichert und wiederhergestellt werden muss (siehe auch Abbildung 4.30).

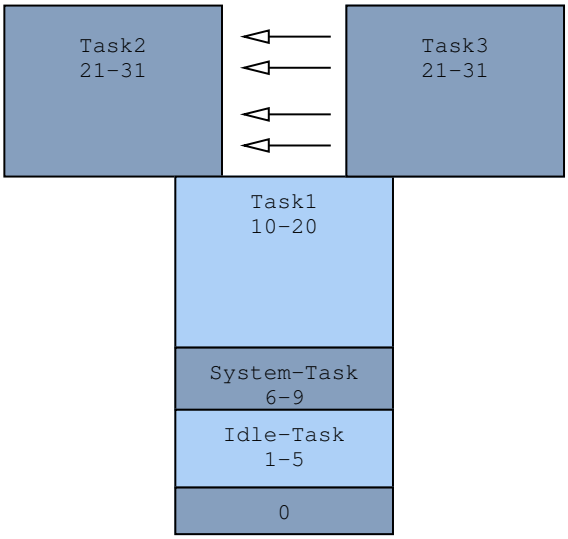


Abbildung 4.30: Die Stacknutzung von SOUL

Der Scheduler wird durch den TIMER0-Overflow-Interrupt ausgelöst, der in periodischen Abständen (ca. 10 ms) auftritt. Als erstes wird dann der aktuelle Task gesichert, danach wird der nächste auszuführende Task bestimmt, wobei blockierte Tasks (siehe Seite 81) ignoriert werden. Die Tasks bilden dabei eine zyklische Sequenz, die immer in der gleichen Reihenfolge abgearbeitet wird. Statische oder dynamische Priorisierung wird nicht unterstützt, da davon auszugehen ist, dass nie sehr viele Tasks mit unterschiedlichen Anforderungen (d. h. unterschiedlichen Prioritäten) aktiv sind – es ist sogar davon auszugehen, dass zur Laufzeit die meisten Tasks blockiert sind. Bei einem Programm zur Steuerung eines Roboters müssen

alle Tasks auf die Hardware zugreifen, dies aber natürlich untereinander synchronisieren. Deshalb ist davon auszugehen, dass meistens neben dem IdleTask nur ein weiterer Task nicht blockiert ist, komplizierte Scheduling-Algorithmen also nicht sinnvoll wären.

Dieser einfache, aber vor allem auch effiziente, Round-Robin-Scheduling-Algorithmus [GL02, Seite 99] garantiert außerdem, dass die Rechenzeit fair vergeben wird und auch Effekte wie Starvation (siehe [GL02, Seite 44] und [SLPRS, Seite 35ff.]) vermieden werden. Ein Dead-Lock durch zwei Tasks, die jeweils auf den anderen warten, kann nicht ausgeschlossen werden, es liegt in der Verantwortung der Softwareentwickler, diesen Fall zu verhindern.

Der Scheduler kann blockierte Tasks mit Hilfe des Arrays *TaskSyncInfo* erkennen, indem für jeden Task der aktuelle Zustand vermerkt ist. Der Index in diesem Array identifiziert den Task. Ein Task im Zustand *ready* oder *running* hat den Wert 0xff, alle anderen Werte kennzeichnen einen blockierten Task.

Dabei haben die Werte ungleich 0xff auch eine besondere Bedeutung. Wenn auf ein Synchronisationsobjekt (siehe Seite 81) mehrere Task warten, so muss für jedes Synchronisationsobjekt eine Liste der wartenden Task verwaltet werden. Wenn man aber die Einschränkung akzeptiert, dass ein Task immer nur auf ein Synchronisationsobjekt warten kann, dann kann die Warteschlangen aller Synchronisationsobjekte auf das Array *TaskSyncInfo* abbilden: Der Wert zu jeden Task gibt an, welcher Task in der Warteschlange der Nachfolger ist, der Wert 0xfe terminiert die Warteliste für ein Synchronisationsobjekt. Somit wird für die Warteschlange eines Synchronisationsobjektes nur ein Byte benötigt, das den ersten wartenden Task bzw. 0xfe für die leere Liste, angibt.

Ein Beispiel:

```
TaskSyncInfo = {0xff, 0xff, 0xfe, 4, 2, 6, 0xfe, 0xff}
```

Zusammen mit der Information, dass auf zwei Synchronisationsobjekte gewartet wird und Task 3 bzw. Task 5 am Anfang der jeweiligen Schlange stehen, kann man schließen, das die Tasks 0, 1 und 7 *ready* bzw. *running* sind und die Warteschlange für das erste Synchronisationsobjekt aus den Tasks 3, 4 und 2 besteht und die Tasks 5 und 6 auf das zweite Synchronisationsobjekt warten.

Bibliotheken

Zur Benutzung von SOUL existieren mehrere Bibliotheken, die Funktionen für verschiedene Anwendungsfälle bereitstellen. Nähere Angaben machen die entsprechenden Include-Dateien. Abbildung 4.31 zeigt exemplarisch, wie das Sonar in C angesprochen wird.

Synchronisationsfunktionen: Eine besondere Stellung nehmen die Synchronisations-Funktionen ein, die es ermöglichen, ein Programm als System mehrerer Tasks zu implementieren.

Es wurde eine Funktion *void Wait(PRODH nWaitHigh, PRODL nWaitLow)* realisiert, die als Parameter einen 16-bit unsigned Integer (aufgeteilt in HIGH- und LOW-Byte) übergeben bekommt, der angibt, wieviele Millisekunden der aktuelle Task unterbrochen werden soll. Mit Hilfe eines Timers wird erreicht, dass keine Rechenzeit unnötig verbraucht wird, während der Task unterbrochen ist. Es können mehrere Tasks gleichzeitig warten.

Es werden als Synchronisationsobjekte CriticalSections (auch binäre Semaphoren genannt) und Events unterstützt. Dabei können jeweils mehrere Tasks auf ein Synchronisationsobjekt warten. Aufgrund der gewählten einfachen (und effizienten) Implementierung ist es aber nicht möglich, das ein Task auf mehrere Synchronisationsobjekte gleichzeitig wartet. Die Funktionalität, auf mehrere Synchronisationsobjekte zu warten, hätte wesentlich mehr Aufwand (Speicher und Rechenzeit) gekostet und es ist den Autoren auch kein üblicher Anwendungsfall für eine Robotersteuerung bekannt, die diese Funktionalität benötigt. Mit mäßigem Aufwand kann der Nutzer auch mit der bestehenden Funktionalität das fehlende *WaitForMultipleObjects( )* mit kleinen Einschränkungen implementieren.

CriticalSection: Es existieren die Funktionen *void EnterCriticalSection(WREG nCritSec)* und *void LeaveCriticalSection(WREG nCritSec)*, denen man jeweils im Working Register die zu nutzende Critical Section übergibt. EnterCriticalSection sollte aufgerufen werden, wenn man einen exklusiven Zugriff auf eine Ressource (z. B. einen gemeinsamen Speicherbereich, Hardware) benötigt. Wenn ein anderer Task vorher bereits diese Funktion aufgerufen hat, so ist der aktuelle Task blockiert, bis der andere Task das entsprechende *LeaveCriticalSection* ausführt. Es gibt noch die Funktion *bool IsInCriticalSection(WREG nCritSec)*, mit der man ohne zu blockieren testen kann, ob sich dieser oder ein anderer Task in einer bestimmten Critical Section befindet.

Events: Ein Event kann entweder signalisiert oder nicht signalisiert sein. Es steht die Funktion *void WaitForEvent(WREG nEvent)* zur Verfügung, mit der ein Task solange blockiert wird, bis das angegebene Event signalisiert wird. Wenn dies beim Aufruf der Funktion bereits der Fall war, so kehrt diese Funktion sofort zum Aufrufer zurück. Zur Statusänderung von Events gibt es die Funktionen *void ResetEvent(WREG nEvent)*, die das Event in den nicht signalisierten Zustand versetzt sowie *void SetEvent(WREG nEvent)* und *void PulseEvent(WREG nEvent)*, die alle wartenden blockierten Tasks wieder deblockieren und im Fall von *SetEvent* das Event in den signalisierten Status versetzen. Bei *PulseEvent* verbleibt das Event im nicht signalisierten Status. Auch hier existiert eine Funktion *bool GetEvent(WREG nEvent)*, mit der man den Status eines Events nicht blockierend abfragen kann.

LCD-Ansteuerung: Auf dem LCDBoard wird auch eine Version von SOUL laufen, die aber keine User-Programme ausführen wird, sondern über ROCKWIRE Anweisungen von der ROCK-Einheit entgegen nehmen wird und diese dann ausführt. Dabei wird die LCD-Bibliothek verwendet, die einen einfachen Zugriff auf das LCD ermöglicht. Näheres siehe Abschnitt 4.5.3 auf Seite 87.

Motor: Zur Zeit ist bereits für die Motorausgänge direkt an der ROCK-Einheit die Motoransteuerung implementiert. Dabei wird die Pulsweitenmodulation des PIC18 genutzt. Die Funktions-Schnittstelle ist so entwickelt, dass man diese Funktion auch für Motoren nutzen kann, die über das ROCKWIRE angesteuert werden.

Es gibt nur ein einzelnes Makro *bool MotorSetMode(RockWireAdr, Motor, Mode, Speed)*. Dieses bekommt als Parameter die die ROCKWIRE-Adresse (*auch:* Loopback-Adr der ROCK-Einheit), die zu setzenden Motoren (mittels bitweisem ODER können mehrere angegeben werden), den Modus (FLOAT, STOP, FORWARD oder REVERSE) und die Geschwindigkeit übergeben. Die Rückgabe gibt an, ob der Vorgang erfolgreich war. Als Parameter sollten immer die Defines aus dem Include-File benutzt werden.

SRF08: Die Nutzung des Ultraschallsensors SRF08 (siehe Seite 74) wird über nur fünf Funktionen ermöglicht. Als erstes ruft der Nutzer die Funktion *void SRFActivateRangeCM()* auf, die die Entfernungsmessung startet. Während der Messung (Dauer: 65 ms) kann nicht auf den Sensor zugegriffen werden, so dass sinnvollerweise ein Aufruf von *Wait(0x00, 0x41)* erfolgt. Danach kann mit *int SRFGetRangeLOW()* und *int SRFGetRangeHIGH()* die Entfernung des am nächsten befindlichen Objektes abgefragt werden. Dabei ist zu beachten, dass der Sensor einen vorzeichenlosen 16-Bit Wert liefert.

Mit der Funktion *int SRFGetLight()* kann das Ergebnis des Lichtsensors des SRF08-Moduls abgefragt werden. Mit *int SRFGetSWRev()* kann die Software-Revision des Moduls herausgefunden werden. Ist diese 0xff, dann ist das Modul noch mit der Entfernungsmessung beschäftigt, wahrscheinlich weil nach *SRFActivateRangeCM()* nicht lange genug gewartet wurde.

Tools: Zusätzlich gibt es noch das Makro *swptsk*, das die Zeitscheibe sofort an den nächsten Task abgibt, indem der Interrupt gesetzt wird, der den Scheduler auslöst.

Das Makro *ActiveWait(microSeconds)* wartet mindestens die angegebene Zeit. Dabei wird die Frequenz des Oszillators berücksichtigt. Diese Funktion stellt während des Wartens ihre Rechenzeit aber nicht anderen

```
SRFActivateRangeCM();
Wait(0,90);
if(SRFGetRangeHIGH(0)==0)
   {
     if ((cm=SRFGetRangeLOW(0))< 0)
           cm = AREA_WIDTH;
     else if(cm > SIGHT_DISTANCE)
           cm=AREA_WIDTH;
   }
else
       cm = AREA_WIDTH;
```

Abbildung 4.31: Beispiel der Nutzung einer SOUL-Bibliothek (Sonarabfrage in C)

Tasks zur Verfügung, sondern verschwendet diese mit „busy waiting“. Dieses Makro sollte, wenn immer möglich, durch die Funktion *Wait( )* (siehe Seite 81) ersetzt werden.

Treiber

Für SOUL wurden auch mehrere Treiber implementiert, die die Schnittstelle zwischen User und Hardware bilden. Aus Sicht des Users unterscheiden sie sich nicht von Bibliotheken, da sie für den User auch nur Funktionen als Schnittstelle zur Verfügung stellen. Aus Sicht von SOUL gibt es aber den eklatanten Unterschied, dass Treiber Interrupt-Unterstützung benötigen und somit auch eine Interrupt-Routine implementieren, die von SOUL aufgerufen wird.

RS232: Die RS232-Schnittstelle wird unterstützt, indem innerhalb des Treibers zwei Ring-Buffer existieren: einer zum Senden und einer zum Empfangen. Wenn man senden möchte, muss man erst den RS232-Transmit-Buffer „locken“, um dann entweder byte-weise oder gleich einen ganzen null-terminierten String aus dem Programmspeicher auf einmal in den Transmit-Buffer zu schreiben. Danach sollte man das Lock entfernen. Währenddessen wird durch eine Interrupt-Routine der Inhalt des Buffers versendet.

Das Empfangen läuft ähnlich ab; der Zugriff auf den RS232-Receive-Buffer muss auch hier durch die Lock-Funktionen geschützt werden. Dann kann der Buffer byte-weise ausgelesen werden.

Die Sende- und Empfangsfunktionen blockieren, wenn ihre Puffer voll bzw. leer sind. Deshalb gibt es noch weitere Funktionen, die den Zustand der beiden Ring-Buffer zurückgeben. Auch Fehlerzustände können so abgefragt werden.

I^2C: Die Schnittstelle des I^2C-Treibers ist der Schnittstelle der RS232 nachempfunden. Der größte Unterschied zu dieser liegt jedoch darin, dass I^2C eine Master-/Slave-Verbindung ist, bei der jede Kommunikation vom Master ausgeht.

Da diese Schnittstelle als vorläufiger Ersatz für die ROCKWIRE-Verbindung benutzt werden sollte, wurden noch weitere Anforderungen an die Benutzung der Schnittstelle gestellt. Auf der Schnittstelle wird deshalb automatisch das ROCKWIRE-eigene Byte-Stuffing durchgeführt, obwohl dies auf dem I^2C-Bus eigentlich nicht nötig wäre. Dazu wird jedes Datenbyte mit dem Wert *0xdb* verdoppelt wird, um es auf dem ROCKWIRE von dem ersten Byte eines neuen Pakets zu unterscheiden.

Der Master (ROCKBoard) muss:

- Erst den Transmit-Buffer (TX-Buffer) mit dem Request-Packet füllen (z. B. mit *I2CWriteChar( )* oder *I2CWriteStringFLASH_m( )*).
- Dann mit *I2CSendPacket()* den Request zum Slave schicken.

- Wenn die Funktion zurückkehrt, steht im Receive-Buffer (RC-Buffer) das Response-Packet.
- Und nun die Antwort aus dem RC-Buffer lesen (z. B. mit *I2CReadChar()*).

Der Slave (LCDBoard) muss folgende Aufgaben erfüllen:

- Mit *WaitForEvent(I2C_RC_NOTEMPTY_EVENT)* darauf warten, dass ein Request-Packet eingetroffen ist – wobei sich die EOT-Erkennung bei den Slaves nicht ganz einfach gestaltet. Dies muss aus dem Kontext geschlossen werden.
- Den RC-Buffer mit *I2CReadChar()* auslesen und interpretieren.
- Den vorherigen Schritt solange wiederholen, bis das Request-Paket zu Ende ist (Achtung: *I2CRead Char()* ist blockierend; die Ende-Erkennung muss also funktionieren!)
- Jetzt den TX-Buffer mit der Response füllen und dabei auf jeden Fall das erste Byte (Paketlänge) angeben, so dass der Master erkennen kann, wie lang das ROCKWIRE -Paket ist.
- Während oder nach diesem Vorgang sollte dann der Request ausgeführt werden (z.B. LCD-Ausgabe).

Wenn die Bibliothek zur Ansteuerung des Ultraschallsensors SRF08 verwendet wird, dann darf dieser Treiber nicht verwendet werden.

ROCKWIRE: Der ROCKWIRE-Treiber hat eine ähnliche Schnittstelle wie der I^2C-Treiber, um genau zu sein, wurden die beiden Schnittstellen so ähnlich entworfen, da der I^2C-Treiber als vorläufiger Ersatz für das ROCKWIRE implementiert wurde.

Als erstes muss der Nutzer mit der Funktion *RWLock()* sich den exklusiven Zugriff auf das ROCKWIRE sichern. Danach wird z. B. mit *RWWriteChar(char)* das Request-Paket in den Puffer geschrieben. Dabei ist das Protokoll aus Abschnitt 4.3.1 auf Seite 48 zu beachten. Es muss also zuerst *RW_ESC_CHAR* (0xdb), dann die Adresse des Slaves, gefolgt von der Anzahl der Daten-Bytes und schließlich die eigentlichen Daten in den Puffer geschrieben werden. Wenn keine Daten gesendet werden sollen, so muss das Längen-Byte gleich Null sein. Nach diesen Vorbereitungen wird das eigentliche Senden mit *RWSendPacket()* eingeleitet. Diese Funktion startet den Sendevorgang des ROCKWIRE-Treibers und wartet, bis die Antwort des Slaves empfangen wurde. Falls der Slave nicht reagiert, so wird durch einen Timeout der weitere Programmablauf sichergestellt. Nun kann mit *RWTransmitError()* und *RWReceivedError()* der Erfolg des Sende- und Empfangsvorgangs überprüft werden. Mit der Funktion *RWReadChar()* kann der Puffer byteweise ausgelesen werden und somit die Antwort des Slaves verarbeitet werden. Dabei sollte darauf geachtet werden, nur im Puffer vorhandene Bytes auszulesen. Dies lässt sich mit der Funktion *RWReceivedBufferEmpty()* überprüfen. Zum Schluss muss wieder das ROCKWIRE für andere Tasks mit *RWUnlock()* freigegeben werden.

Der Treiber implementiert alle anderen Aspekte des ROCKWIRE-Protokolls, der Nutzer muss also nicht explizit das Byte-Stuffing (Ersetzen von 0xdb (dem Startbyte) durch zwei 0xdb) durchführen. Auch die Stromversorgung über das ROCKWIRE für die Slaves ist immer gewährleistet.

4.5.2 ROCKCOMM

Das Protokoll des Bootloaders wurde den ersten Teilen der Application-Note AN851 [Mica] von Microchip nachempfunden, um auf ein schon existierendes Programm (P1618QP.exe, beschrieben in [Mica]) zur Übertragung von Daten zur ROCK-Einheit zurückgreifen zu können. Dieses Programm arbeitete leider nicht besonders stabil. Also wurde eine eigene Anwendung entwickelt, die zum einen stabil läuft und bei der auch die Erweiterungen der Gruppe berücksichtigt werden konnten, wie zum Beispiel die USB-Unterstützung oder ein verändertes Protokoll.

Die Programmoberfläche besteht aus einer Toolbar, in der die Befehle, die das Protokoll unterstützt, direkt ausgeführt werden können. In der Statuszeile werden die aktuellen Verbindungsdaten angezeigt und darüber die aktuelle Einstellung für die Betrachtung von eingehenden Daten. Diese Daten werden in den Fenstern auf dem Desktop der Applikation dargestellt (siehe Abbildung 4.32).

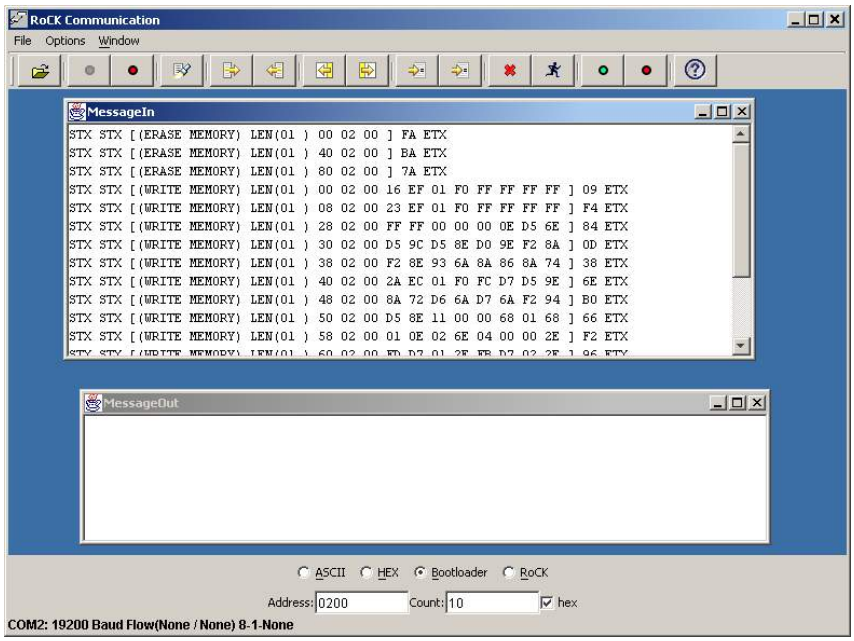


Abbildung 4.32: Screenshot von ROCKComm

Leistungsumfang von ROCKComm

- Kommunikation mit dem Bootloader der ROCK-Einheit über
  - RS232
  - USB
- Verwendung als Terminal
  - Seriell
  - Parallel
  - USB
  - Socket
- Aufruf sämtlicher durch das Protokoll bereitgestellter Funktionen
  - Read Version
  - Read Program Memory
  - Write Program Memory
  - Erase Program Memory
  - Read EE Memory
  - Write EE Memory
  - Read Configuration Memory

  - Write Configuration Memory
  - Enable SOUL

- Anzahl der pro „Write Program Memory“ übertragenen Blöcke kann konfiguriert werden.
- Empfangen von Daten von SOUL
- Empfangene Daten können als
  - ASCII
  - HEX
  - Bootloader Protokoll
  - SOUL Protokoll

  interpretiert werden.
- Speichern/Laden verschiedener Konfigurationen

Einstellmöglichkeiten (Konfiguration)

Der wichtigste Konfigurationsaspekt ist der des Kommunikationskanals und seiner Einstellungen wie Baudrate oder Flusskontrolle. Diese Daten können in dem Menü *Options/Configuration/Port Configuration* eingestellt werden. Unter dem Tab *General* verbergen sich die Einstellungsmöglichkeiten für die zu übertragenden Blöcke pro Schreibvorgang. Die automatische Verifizierung der geschriebenen Daten kann man hier deaktivieren und einen Konsolenbefehl konfigurieren, der in der Toolbar aktiviert werden kann.

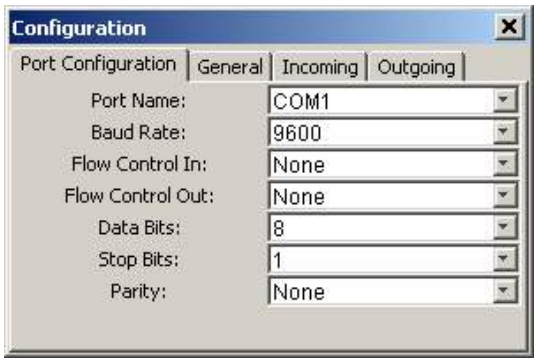


Abbildung 4.33: Kommunikationsparameter von ROCKComm

Installationshinweise

ROCKComm läuft auf den Plattformen Win32, Solaris und Linux. Diese Einschränkungen existieren auf Grund der benötigten Hardwaretreiber für RS232 und USB, die zur Zeit nur für diese Plattformen mit der entsprechenden Schnittstelle existieren.

Es wird ein Java Runtime Environment benötigt. Getestet haben wir mit den Versionen 1.3 und 1.4, wobei keine Probleme auftraten. Die beiden bereits erwähnten Kommunikationspakete bestehen jeweils aus einem Jar-Archiv und einer Bibliothek. Das Jar-Archiv muss in das lib-Verzeichnis und die Bibliothek in das bin-Verzeichnis der Java-Installation entpackt werden. Das Paket JavaComm beinhaltet noch eine weitere Datei javax.comm.properties - diese muss in dasselbe Verzeichnis installiert werden wie das Jar-Archiv. Nun kann ROCKComm mittels des folgenden Aufrufs gestartet werden:

```
java -cp <JAVAHOME>/lib/usbcomm.jar;<JAVAHOME>/lib/comm.jar;
RoCKComm.jar RoCK.RoCKComm
```

4.5.3 Programmierung des LCD-Bausteins

Die LCD-Ansteuerung kommt ohne Interrupt-Routine aus, wird also in der SOUL-Nomenklatur nicht als Treiber, sondern als Bibliothek bezeichnet. Zunächst wurde für fast alle Steuer-Codes des LCD-Controllers (siehe Hardware Beschreibung 4.4.5) ein low-level Unterprogramm geschrieben. In der Nomenklatur erhielten diese Routinen das Suffix *impl*. Die aufwendige Parameterübergabe wird durch Makros, bei denen das Suffix weggelassen wurde, vereinfacht. (Abbildung 4.34)

Befehl	Parameter	Kurzbeschreibung
set_lcd_in	–	Befehl als Eingabe definieren
set_lcd_out	–	Befehl als Ausgabe definieren
lcd_dsp_on	WREG=[0..1]	Linke oder Rechte LCD-Hälfte einschalten
lcd_write	WREG	Datenpins belegen
lcd_static	WREG=[0..1]	Statisch-Modus aktivieren
lcd_commit	–	Vorprogrammierten Befehl ausführen
lcd_set_ctrl	macro x	Controller Auswählen
lcd_sel_page	macro x	LCD-Seite auswählen (eine Seite ist ein acht Pixel breiter Streifen einer LCD-Seite)
lcd_sel_col	macro x	Spalte auswählen
lcd_set_startline	macro l=[0..15]	Hardware-Scrolling
lcd_erase_half	–	Löschen einer Display-Hälfte
lcd_erase	–	Löschen beider Hälften
lcd_print_eight_pixel	macro c	Acht Pixel untereinander ausgeben
lcd_print_char	WREG	Zeichen mit ASCII-Code aus WREG ausgeben
lcd_set_picture_start	macro x	Table-Register auf Addr. x setzen
lcd_fs_picture_here	macro –	Fullscreen-Bild ausgeben
lcd_fs_picture	macro x	Mischung aus den beiden obigen Kommandos

Abbildung 4.34: Übersicht der LCD-Bibliotheks-Funktionen

Die zwei vorhandenen LCD-Controller werden durch die selben Unterprogramme angesprochen, wobei man im Treiber ein Flag setzen kann, welcher gerade adressiert werden soll.

Mit dieser soliden Basis wurde weiterführende Funktionalität, wie die Ausgabe von Zeichen und Grafik implementiert. Der Zeichensatz konnte von der Projektgruppe MPlay3 [Proa] übernommen werden; er lag als C-Konstanten-Definition vor und wurde in eine PIC-Assembler-Konstanten-Definition konvertiert.

Grafiken können am PC gezeichnet (z.B. mit dem freien Bildbearbeitungs-Programm GIMP) und als Portable Network Graphics Format gespeichert werden. Ein kleines in C geschriebenes Programm wandelt diese dann in Include-Dateien um, die man in SOUL einbinden kann. (Abbildung 4.35 auf der nächsten Seite)

4.6 Der Compiler

Ein Ziel dieser Projektgruppe besteht in der Erstellung eines ANSI-C-Compilers [AVA86] für den Microchip PIC18FXX2-Prozessor. Die Realisierung kann grob in sechs Abschnitte eingeteilt werden. Zunächst

Abbildung 4.35: Das Start-Logo unter Linux und unter SOUL

musste das Frontend (Lance A.3.1 auf Seite 126) an die Charakteristiken des PIC18 angepasst werden. Danach galt es, eine Baumgrammatik für den Code-Generator-Generator *Olive++* (vgl. A.3.2 auf Seite 127) zu erstellen, um dann in einem weiteren Schritt den von Olive erstellten Code-Generator in ein Backend-Interface einzubetten. Daraufhin wurde die LLIR (vgl. A.3.3 auf Seite 129) als Datenstruktur für die Intermediate Representation eingeführt und eine Registerallokation auf Basis des im Buch von Appel [App98] beschriebenen Algorithmus implementiert. Schliesslich vervollständigt die Peephole-Optimierung unsere Realisierung des Compilers. Diese Entwicklungsstufen werden im Folgenden näher erläutert und durch Beschreibungen der in diesem Prozess entwickelten Hilfsmittel ergänzt.

4.6.1 Anpassung von LANCE-Konfigurationen

LANCE [Leu] ist ein am Lehrstuhl XII entwickeltes ANSI-C Compiler Frontend, welches das Ziel hat, die Erstellung von Compilern für neue Zielprozessoren zu erleichtern. Dabei werden sowohl Analysen des Source-Codes durchgeführt als auch eine Zwischendarstellung generiert. Desweiteren können ebenfalls Graph-Datenstrukturen zur Analyse bzw. Weiterverwendung im vcg-Format erzeugt werden. Nicht bedient werden von LANCE die Erzeugung des Assemblercodes, maschinenspezifische Optimierungen und die Assemblierung. Im Folgenden sollen nun die Konfiguration von LANCE und die Einbettung in die Entwicklungsumgebung dargestellt werden. Hierzu wird die notwendige Anpassung an die Zielprozessorarchitektur mittels der config-Datei erläutert. Diese Parameter werden in der config-Datei spezifiziert und müssen für die jeweilige Zielprozessoren angepasst werden. An dieser Stelle werden dementsprechend ausreichende Kenntnisse über den jeweils zu programmierenden Prozessor vorausgesetzt. Die benötigten Daten wurden im wesentlichen aus dem PICMICRO 18C MCU Family Reference Manual [Mic02b] bezogen. Die volle Pfadangabe des configuration files, welches von LANCE benutzt werden soll muss hierzu mittels der Umgebungsvariable `LANCE2_CONFIG="Pfadangabe"` gesetzt werden. Kommentare werden in der config-Datei durch eine "//" Sequenz eingeleitet. Eine nicht vollständige Angabe der Parameter wird durch eine Warnung gemeldet, in diesem Fall werden die jeweiligen default-Werte verwendet. Es folgt eine Erläuterung der Parameter der config-Datei.

MAX_FRONTEND_ERROR (default value: 5)
Maximale Anzahl der Fehler die durch das Frontend ausgegeben, weitere Fehler werden ignoriert.

KEYWORD_INLINE (default value: 0)
Boolescher Wert, der angibt, ob das Frontend das nicht-standardisierte Schlüsselwort `INLINE` akzeptiert. Dieses ist für die zukünftige Implementierung von function inlining gedacht.

KEYWORD_INTERRUPT (default value: 0)
Boolescher Wert, der angibt, ob das Frontend das nicht-standardisierte Schlüsselwort `INTERRUPT` in Funktionsdefinitionen erlaubt.

KEYWORD_TYPEOF (default value: 0)
Boolescher Wert, der angibt, ob das Frontend das nicht-standardisierte Schlüsselwort `TYPEOF` zur Defintionen von Typen erlaubt.
REJECT_FLOATS (default value: 0)

Boolescher Wert, der angibt, ob die Typen `float, double` und `longdouble` vom Frontend akzeptiert werden.

REJECT_LONG (default value:0)
siehe **REJECT_FLOATS**, hier bezugehmend auf den Typen long integer.

HIGH_CONST_CSE (default value: 127)
Grössere positive Integer Konstanten als die angegebene Zahl werden als Kandidaten für die CSE (common subexpression elimination) gehandelt. Dieser Wert ergibt sich aus unten deklarierten Bit-Breiten und den dadurch grösst bzw. kleinst darstellbaren Zahlen.

LOW_CONST_CSE (default value: -128)
Analog zur Erläuterung zu HIGH_CONST_CSE, hier bezugnehmend auf die kleinst darstellbare negative Integer Konstante.
BITWIDTH_ADDRESSABLE (default value: 8)
Dieser Parameter gibt die Länge der minimal addressierbaren Speichereinheit in Bits an.

BITWIDTH_POINTER (default value: 32)
Dieser Parameter gibt die Bit-Breite eines Objekts vom Type Pointer an.

BITWIDTH_CHAR (default value: 8)
Maximaler Bit-Breite eines Objektes vom Type Char. Folgende Parameter werden analog behandelt: **BITWIDTH_SHORT_INT, BITWIDTH_INT, BITWIDTH_LONG_INT, BITWIDTH_FLOAT, BITWIDTH_DOUBLE, BITWIDTH_LONG_DOUBLE**.
Die jeweiligen default values sind typenabhängig.

ALIGN_POINTER (default value: 1)
Parameter für das Speicheralignment des angegebenen Typs in Byte. Folgende Parameter werden analog behandelt: **ALIGN_CHAR, ALIGN_SHORT_INT ,ALIGN_INT, ALIGN_LONG_INT, ALIGN_FLOAT, ALIGN_DOUBLE, ALIGN_LONG_DOUBLE**.
Die jeweiligen default values sind typenabhängig.

Die konkrete Realisierung der Datei config.pic, die mit Kommentaren versehen wurde, sieht dann wie folgt aus.

```
MAX_FRONTEND_ERROR = 10
KEYWORD_INLINE = 0
KEYWORD_INTERRUPT = 0
KEYWORD_TYPEOF = 0
REJECT_FLOATS = 1
// Da zunächst der floating type für den Compiler nicht vorgesehen ist,
   wurde dieser Wert auf true gesetzt
REJECT_LONG = 1
// siehe REJECT_FLOATS, hier bezugehmend auf den Typen long integer
HIGH_CONST_CSE = 127
LOW_CONST_CSE = -128
BITWIDTH_ADDRESSABLE =  8
BITWIDTH_POINTER =  12
// Da der PiC18 über das 12-Bit breite FSR (file select register) den
   Speicher addressiert wurde dieser Wert angepasst
BITWIDTH_CHAR =  8
BITWIDTH_SHORT_INT =  8
BITWIDTH_INT =  8
BITWIDTH_LONG_INT = 8
BITWIDTH_FLOAT = 8
BITWIDTH_DOUBLE = 8
BITWIDTH_LONG_DOUBLE =  8
```

```
ALIGN_POINTER = 1
ALIGN_CHAR = 1
ALIGN_SHORT_INT = 1
ALIGN_INT = 1
ALIGN_LONG_INT = 1
ALIGN_FLOAT = 1
ALIGN_DOUBLE = 1
ALIGN_LONG_DOUBLE = 1
```

Mit Hilfe von Umgebungsvariablen kann LANCE den Präprozessor des GNU C Compilers (gcc) verwenden. Der hierzu notwendige Eintrag `LANCE2_CPP="gcc -E"` muss ergänzend vorgenommen werden.

4.6.2 Erstellung einer PIC-Baumgrammatik für OLIVE++

Der Codegenerator des Compilers basiert auf der Überdeckung von Datenflussbäumen (siehe Abbildung 4.36), die vom Frontend (LANCE) in der Intermediate Representation (IR) erstellt werden. Ein Codegenerator-Generator wie *Olive* (vgl. Kap. A.3.2 auf Seite 127) erzeugt dann aus einer Baumgrammatik den Codegenerator.
Im Folgenden wird nun am Beispiel beschrieben, wie man Grammatikregeln für *Olive* aufbaut und eine Überdeckung der Datenflussbäume ermöglicht. Dabei handelt es sich um konkrete Regeln aus der Baumgrammatik für den PIC18-Codegenerator.
Ein kurzes C-Programm dient hier zu Demonstrationszwecken:

```
void main ()
{
 int n,i;
 n = 25;
 if (n < 50) {
    i = 1 + n;
    n++;
 }
}
```

Durch den `compile`-Befehl von LANCE erhält man den C-Code in 3-Adress-Code (vgl. Kap. A.3.1 auf Seite 126). Diese Darstellung kann jetzt Funktion für Funktion, Baum für Baum durchlaufen werden und die einzelnen Bäume mit ihren Operationen und Operanden in einer Preorder-Reihenfolge (vgl. Kap. 4.6.5 auf Seite 100) ausgegeben werden:

```
(cs_WRITE [IR stm 1: 'n_4 = 25;']
 (cs_INTCONST [IR exp 1: '25' C type: int  ]))

(cs_CJUMP [IR stm 4: 'if (t5) goto LL1;']
 (cs_LOGNOT [IR exp 11: '!t1' C type: int  ]
  (cs_LESS [IR exp 8: 'n_4 < 50' C type: int  ]
   (cs_READ [IR exp 7: 'n_4' C type: int  ])
   (cs_INTCONST [IR exp 2: '50' C type: int  ]))))
```

Dies sind zwei der vier generierten Bäume, welche, mit etwas Übung, recht leicht erkannt werden können. Dennoch ist eine Visualisierung der Graphen für die Analyse wesentlich vorteilhafter. Zu diesem Zweck wandelt der *Treeparser* (vgl. Kap. 4.6.5 auf Seite 100) diese textuelle Beschreibung in ein graphisches Format um, das z. B. mit *Xvcg* als gezeichneter Baum dargestellt werden kann. Die Abbildungen 4.36 und 4.37 auf der nächsten Seite zeigen die vier in diesem Beispiel zu überdeckenden Bäume. Die Erfahrung lehrt, dass es sinnvoll ist, möglichst große Bäume zu erhalten, für die dann bessere (z. B. in Form

von weniger Befehlen) Überdeckungen gefunden werden können. Bei der Implementierung des Backends kann durch den Befehl `DFTForceSymToMem(sym)` ein jeweiliges Rückschreiben aller Variablen in den Hauptspeicher erzwungen werden, was die Datenflussbäume (DataFlowTrees / DFT) künstlich zerteilt.

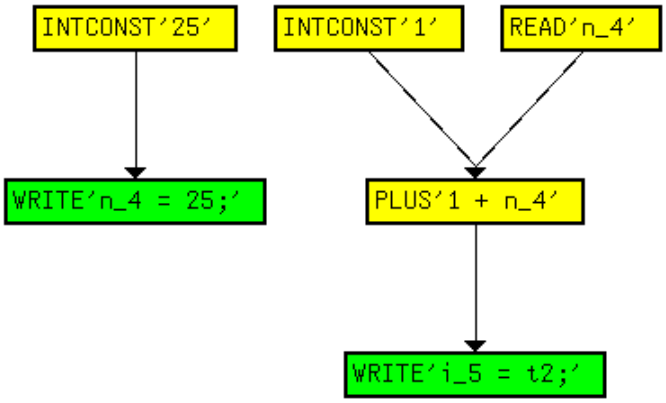


Abbildung 4.36: Screenshot der Datenflussbäume des Beispielprogramms (I)

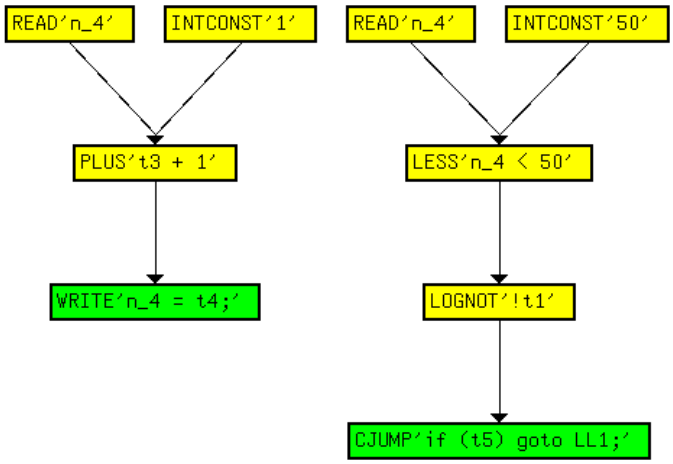


Abbildung 4.37: Screenshot der Datenflussbäume des Beispielprogramms (II)

Ausgehend von diesen Informationen ist es nun möglich, die entsprechenden Regeln für die Baumgrammatik zu erstellen. Dazu fängt man am besten mit der Wurzel an. Wenn wir zunächst den ersten Baum in Abbildung 4.36 betrachten, so braucht man eine `cs_WRITE`-Regel, die ein Datum aus der `cs_INTCONST`-Regel in der Variable `n4` abspeichert. Hierzu könnte jetzt eine Regel formuliert werden, indem man die WRITE-Operation direkt in Abhängigkeit von der INTCONST-Operation stellt. Damit wäre allerdings auch nur dieser konkrete Fall abgedeckt. Eine solche Regel sähe folgendermaßen aus:

```
stm: cs_WRITE(cs_INTCONST)
{  Hier die Kostenfunktion }
=
{  Hier der C-Code, der die entsprechenden
   Assembler-Befehle erzeugt
}
```

Da es aber ineffizient wäre, für jede mögliche Kombination solcher Operationen eigene Regeln zu schreiben, ist es sinnvoll, eigene Nichtterminale zu definieren, die z. B. das Register darstellen, in dem die Re-

sultate der Teilbäume übergeben werden. So hat man in diesem Beispiel durch das Nichtterminal ACC (steht für Akkumulator oder hier WREG) die Möglichkeit, eine Regel zu schreiben, die beliebige Integer-Konstanten in das WREG (Working REGister) lädt und dann in einer zweiten Regel das Schreiben des WREGs in eine Speicherzelle zu formulieren. Solche Regeln finden sich dann auch in der Baumgrammatik des Compilers:

```
acc: cs_INTCONST
{  $cost[0].cost = 1; }
=
{ char s[50];
  sprintf(s,"%d",CS_EXPVAL($1));
  fprintf(asmfile,"\n\tMOVLW .%s",s);
};

stm: cs_WRITE(acc)
{  $cost[0].cost = 1 + $cost[2].cost; }
=
{ char* v1;
  v1 = $action[2]();
  fprintf(asmfile,"\n\tMOVWF %s,0",CS_LHSSYM($1)->Name());
};
```

Zur Erläuterung: Ein Wurzel-Statement (`stm`), repräsentiert durch eine `cs_WRITE`-Operation, speichert Werte mit dem Befehl `MOVWF dest,0` aus dem WREG (`acc`) an eine bestimmte Speicherstelle `dest`, die in der IR als links vom Zuweisungsoperator stehendes Symbol (`CS_LHSSYM($1)`) übernommen werden kann. Dabei „kostet“ diese Operation einen Befehlszyklus (`$cost[0].cost = 1`) plus die Kosten der Operation (`+ $cost[2].cost`), die vorher das Datum in das WREG lädt. Um auch sicherzustellen, dass das Datum vorher auch im WREG steht, muss vor dem Aufruf des `MOVWF`-Befehls erst der Teilbaum ausgeführt werden, der in das Nichtterminal `acc` schreibt (`v1 = $action[2]()`).
Das geschieht in der Regel `acc: cs_INTCONST`, in dem einfach der angegebene Wert (Expression Value / `CS_EXPVAL($1)`) mit dem `MOVLW`-Befehl als Literal in das WREG geladen wird.
Man muss dabei darauf achten, dass die Nummern in den Kosten-Bezeichnern (z. B. `$cost[0]`), den Action-Aufrufen und den Zugriffen auf Baumsymbole (z. B. `CS_LHSSYM($1)`) in der Deklaration (bzw. ersten Zeile der Regel) auf die richtige Position zeigen. Dabei kann man die Position durch Abzählen bestimmen: `stm` = 0, `cs_WRITE` = 1, `acc` = 2.
Der generierte Code für diesen Baum lautet dann:

```
; Comment: IR stm: n_4 = 25;
; Comment: line 6 in "test.c": n = 25;
        MOVLW .25
        MOVWF n_4,0
```

Um das Verfahren zur Erstellung von Grammatikregeln weiter zu vertiefen, kann man als etwas komplexeres Beispiel noch den zweiten Baum von Abbildung 4.37 auf der vorherigen Seite anführen, der einen bedingten Sprung (durch die IF-Anweisung im C-Code) beschreibt. An dieser Stelle war es nötig, weitere Nichtterminale einzuführen, die die Sprungbedingungen in Form von Statusflags im PIC18 darstellen. Das ist eine Folge der Aufteilung eines IF-Befehls in einen Teil, in dem die Sprungbedingung (LESS) berechnet wird (z.B. durch Subtraktion der beiden zu vergleichenden Operanden voneinander und anschliessender Betrachtung des Negativ-Flags) und einen zweiten Teil, der diese Bedingung logisch invertiert (durch ein LOGNOT) und dann entsprechend springt (CJUMP).
Die dafür eingefügten Nichtterminale `ncj, nncj, zcj, nzcj` stellen dann die vier Möglichkeiten aus Zero- (`zcj` - zero comparison jump / `n(ot)zcj`) oder Negativ-Flag (`ncj` - negative comparison jump / `n(ot)ncj`) dar. Dabei beeinträchtigen unterschiedliche Vergleiche entsprechend unterschiedliche Flags, wie z. B. `cs_LESS`, das auf `ncj` Einfluss nimmt. Bei einem Vergleich, der durch Subtraktion von a

und b überprüft ob a kleiner als b ist, wird beispielsweise das Negativ-Flag gesetzt, wenn a kleiner b ist. Um also diesen Baum durch die Grammatik zu erkennen sind folgende Regeln nötig:

```
acc: cs_READ
{  $cost[0].cost = 1; }=
{ char s[50];
  sprintf(s,"%s",CS_EXPSYM($1)->Name());
  fprintf(asmfile,"\n\tMOVF %s,0,0",s)};

ncj: cs_LESS(acc,cs_INTCONST)
{  $cost[0].cost = 2 + $cost[2].cost; }=
{ char* v;
  v = $action[2]();
  fprintf(asmfile,"\n\tSUBLW .%d",CS_EXPVAL($3));
  fprintf(asmfile,"\n\tSUBLW .0")};

nncj: cs_LOGNOT(ncj)
{  $cost[0].cost = $cost[2].cost; }=
{ char* v;
  v = $action[2]()};

stm: cs_CJUMP(nncj)
{  $cost[0].cost = 1 + $cost[2].cost; }=
{ char* v;
  v = $action[2]();
  fprintf(asmfile,"\n\tBNN %s",CS_LABEL($1))};
```

Diese Regeln müsen nicht unbedingt in dieser Reihenfolge auch in der Grammatik stehen. Die Anordnung ist hier nur so gewählt, weil man dadurch die Baumstruktur noch erkennen kann (durch Lesen von unten (Wurzel) nach oben (Blätter)). Dabei kann man sehen, dass sich das Invertieren des Flags einfach nur durch einen Wechsel des Nichtterminals zum Gegenteil auswirkt und daher keine Assembler-Befehle nötig sind. Ausserdem kann man dadurch die Reihenfolge des erzeugten Sourcecodes erkennen:

```
; Comment: IR stm: if (t5) goto LL1;
; Comment: line 7 in "test.c": if (n < 50) {
        MOVF n_4,0,0
        SUBLW .50
        SUBLW .0
        BNN LL1
```

Das Label `LL1` wird schon durch das Frontend generiert und hinter die Befehle (Bäume) gesetzt, die ausgeführt werden müssen, wenn die Bedingung erfüllt ist, um diese Befehle zu Überspringen.

Bisher wird ein Grossteil der möglichen Operationen (vgl. Kap. A.3.2 auf Seite 127) in beliebigen Kombinationen für 8-Bit-Integer-Zahlen vom Codegenerator unterstützt. Ausnahmen bilden bisher: Division, Modulo, Shift und Structcopy, die aber bisher noch nicht endgültig von der späteren Implementierung ausgeschlossen wurden (vgl. Kap. 5.3 auf Seite 114).
Auch Optimierungen des späteren Assemblercodes durch spezielle Grammatikregeln, die dann geringere Kosten verursachen, sind schon in recht grossem Umfang vorhanden. Solche Optimierungen, wie z. B. die Ersetzung zweier Assemblerbefehle durch nur einen beim Inkrementieren eines Wertes im Speicher, werden durch den Befehlssatz des PIC18 (hier: `INCF`) angeboten und sind oft leicht durch spezielle Regeln realisiert.
Dennoch war es teilweise nötig, wenn Regeln von nicht-kommutativen Operationen zwei Operanden verlangten, die beide im WREG stehen sollten – da arithmetische Operationen (wie bei einer 1-Adress-

Maschine) nur über den Akkumulator möglich sind – einen der beiden Operanden in die Accessbank auszulagern.
Als allgemeine Grundlage zur Erstellung von Grammatikregeln sollte so vorgegangen werden, dass diese zunächst so generell einsetzbar wie möglich formuliert werden, d. h. mit denen man zunächst jeden möglichen Baum überdecken kann, um dann später „bessere“ (optimalere) Regeln zu ergänzen. Auch die Anzahl der Operationen, die in der Wurzel eines Baumes stehen können, sind stark begrenzt (`cs_WRITE`, `cs_WRITEARG`, `cs_LABEL`, `cs_STORE`, `cs_VOIDRETURN`, `cs_RETURN`, `cs_JUMP` und `cs_CJUMP`). In der PIC-Baumgrammatik sind das alle Regeln, die auf ein Nichtterminal `stm` (Statement) abbilden.

Implementierung eines Software-Stacks

In den Grammatikregeln wurde ebenfalls die Konstruktion eines Software-Stacks implementiert, mit dessen Hilfe die Übergabe von Funktionsparametern realisiert werden kann. Hierzu wurden folgende Regeln genutzt:

- cs_CALL
- cs_PASSARG
- cs_READARG
- cs_WRITEARG

Grundlage für die Benutzung des Stacks ist es, einen Bereich in der benutzten Bank dafür zu reservieren. Dies geschieht in der Funktion `c_0`, wo mit dem Befehl `LFSR 2,SP` der Zeiger auf die erste freie Stackposition im File Select Register 2 gesichert wird. Die Größe des Stacks wird durch den ASMParser (siehe 4.6.5) festgelegt; die Defaultgröße ist 32 Byte. Für jeden Funktionsaufruf wird die Regel `cs_CALL` gematcht:

```
acc: cs_CALL(acc,args)
{
  $cost[0].cost = 10 + $cost[2].cost + $cost[3].cost;
}
=
{
  int v1;
  char* v2;
  EmitMOVFF(NewPR(PHREG_FSR2L),NewPR(PHREG_FSR0L));
  EmitMOVFF(NewPR(PHREG_FSR2H),NewPR(PHREG_FSR0H));
  v2 = $action[2]();
  v1 = $action[3]();
  EmitMOVFF(NewPR(PHREG_FSR0L),NewPR(PHREG_FSR2L));
  EmitMOVFF(NewPR(PHREG_FSR0H),NewPR(PHREG_FSR2H));
  EmitCALL(v2,0);

  for (int i=1;i<=v1;i++)
        EmitMOVF(NewPR(PHREG_POSTDEC2),1,0);
};
```

Zunächst wird die Sicherung des FSR2 durchgeführt. Dies wurde notwendig, um Parameter an andere Funktionen weiterzugeben, die in der aufgerufenen Funktion benutzt werden.

Der Actionteil des acc-Parameters sorgt nun dafür, dass der Name der Funktion bekannt wird; der Actionteil des args-Parameters erlaubt es, rekursiv die Parameter der Funktion zu übergeben. Dies geschieht in der Regel `cs_PASSARG`:

```
args: cs_PASSARG(args,acc)
{
  $cost[0].cost = 1 + $cost[2].cost + $cost[3].cost;
}
=
{
  int v1;
  $action[3]();
  EmitMOVWF(NewPR(PHREG_POSTINC0),0);
  v1 = $action[2]();
  return 1+v1;
};
```

Hier wird zu Beginn das zu übergebene Argument in das WREG geladen (durch `$action[3]()`) und dann auf den Stack gelegt. Dabei konnte eine Eigenheit des PIC18 genutzt werden: Das Special Function Register `POSTINC0` erlaubt einen Zugriff auf den Wert an der im entsprechenden FSR gespeicherten Adresse, führt darüberhinaus aber auch noch ein Inkrement der dort gespeicherten Adresse durch. Dies erlaubt daher ein Ablegen und Anpassen des Stackpointers in einem Befehl. Über den Aufruf von `$action[2]()` wird nun der – wenn vorhanden – nächste Parameter in gleicher Weise abgearbeitet. Falls kein weiterer Parameter übergeben wird, wird die Regel `cs_NOARG` gematcht.

Anschließend wird in der CALL-Regel der alte Wert des FSR zurückgeschrieben und wieder auf den Ursprungswert zurückgesetzt. Dies wird ermöglicht, da in jeder PASSARG-Regel ein Zähler erhöht wird, der insgesamt dann der Anzahl der Parameter entspricht.

Werden innerhalb einer aufgerufenen Funktion die Parameter gelesen, dann wird die Regel `cs_READARG` gematcht:

```
acc: cs_READARG
{

  $cost[0].cost = 2;
}
=
{

  EmitMOVLW(-CS_ARGNUM($1));
  EmitMOVF(NewPR(PHREG_PLUSW2),0,0);
};
```

Auch hier kann man ein spezielles Register ausnutzen. `PLUSW0` liefert den Wert an der Adresse, die durch das FSR0 angegeben wird, und erhöht diese Adresse um den Wert, der sich im WREG befindet. Mit `CS_ARGNUM($1)` kann die Nummer des gerade betrachteten Parameters bestimmt werden; durch das negative Vorzeichen erhält man also genau den Offset dieses Parameters im Stack nach unten gesehen.

Analog verhält sich das Schreiben in ein Funktionsargument, mit der Einschränkung, dass der neue Wert zunächst zwischengespeichert werden muss, bevor er in die entsprechende Stelle im Stack geschrieben wird.

4.6.3 Erzeugung eines Backend-Interfaces

Um den von OLIVE erzeugten C-Code für die Codeselektion auch nutzen zu können, wurde ein Backendfile erzeugt, das die bereitgestellten Routinen benutzt. Hier werden die von LANCE generierten Datenstrukturen (bzw. das zugehörige File) durchlaufen und ein Assemblerfile für die Ausgabe bereitgestellt.

Zunächst müssen die Symboltabellen durchlaufen werden, um alle für die benutzten Variablen nötigen Linkerinformationen einzubinden. In unserem Fall werden hier die auftretenden Variablen für die Simulation in MPLAB aufbereitet und unter den notwendigen Prozessorinformationen und Includefiles eingefügt. Ein typischer Header sieht etwa folgendermaßen aus:

```
        #include<define.inc>
        PROCESSOR PROC_NAME
        #include PROC_INCLUDE
        radix dec
        UDATA_ACS
v_0     RES 1
v_1     RES 1
v_2     RES 1
v_3     RES 1
SP      RES 32
```

Nun werden alle Funktionen der IR – und darin enthaltene Basisblöcke – traversiert und für jeden einzelnen Datenflussbaum in einem Basisblock wird der Codeselektor aufgerufen. Dies geschieht einfach durch den Aufruf `burm_label(DFT)` – wenn DFT der aktuell betrachtete Datenflussbaum ist. Das eigentliche Durchlaufen des Programms ist einfach über ineinander geschachtelte Schleifen zu realisieren. Beispielhaft:

```
   for (F = IR->FirstFunction(); F; F = IR->NextFunction())
      for (BB = (BasicBlock*)BBList->First(),i = 1; BB;
         BB = (BasicBlock*)BBList->Next(), i++)
         {
            dftlist = DM->DFTList(BB);
            for (DFT = (LANCEDataFlowTree*)dftlist->First(),j = 1; DFT;
                 DFT = (LANCEDataFlowTree*)dftlist->Next(), j++)
                  {
      burm_label(DFT)
                  }
         }
```

Die ausgewählten Regeln der Grammatik sorgen dann für das Einfügen des korrekten Codes der überdeckten Bäume. Abschließend musste lediglich das PIC-Schlüsselwort `END` als letzter Befehl der `main`-Funktion eingesetzt werden, bevor das generierte Assemblerfile geschlossen werden kann.

4.6.4 Überführung der Komponenten in die LLIR

Um die LLIR (vgl. A.3.3 auf Seite 129) benutzen zu können, müssen zunächst einige Daten bereitgestellt werden. Am wichtigsten ist zunächst die Beschreibung des Befehlssatzes des Zielprozessors. Dies geschieht über ein Dateienpaar (im Folgenden unsere Versionen „proc.c“ und „proc.h“), welches die Befehle, die virtuellen und physikalischen Registernamen und die Operatoren enthält. Die Definition der Register in „proc.h“ kann zum Beispiel folgendermaßen aussehen (man beachte, dass die Präfixe PHREG_ und VREG_ vordefiniert sind und nicht geändert werden dürfen):

```
    // Accumulator
    #define PHREG_WREG              "WREG"
```

```
// Multiply registers:
#define PHREG_PRODH             "PRODH"
#define PHREG_PRODL             "PRODL"
// local variables
#define VREG_R "v_"
```

Der zweite wichtige Abschnitt besteht aus der Auflistung der Maschinenbefehle. Er besteht aus einem ENUM-Typ, bei dem alle Einträge das Präfix INS_ haben müssen. In unserer Datei stellt sich dies wie folgt dar:

```
enum InstructionCode
  {
  INS_ADDLW,
  INS_ADDWF,
  INS_ADDWFC,
  INS_ANDLW,
  INS_ANDWF,
  .
  .
  .
  INS_TSTFSZ,
  INS_XORLW,
  INS_XORWF,
  INS_LAST
  // Dummy instruction containing the
  // number of instruction codes
```

Desweiteren enthält die Datei noch diverse PRAGMAs (Compilerbefehle), die von uns jedoch bisher unberührt geblieben sind. Die Datei „proc.c“ enthält, dem ENUM entspechend, nun die Kostentabelle der Instruktionen. Der unten angegebene Auszug aus unserer Tabelle zeigt die beispielhafte Belegung der Kosten durch die Anzahl der Zyklen, die die jeweilige Instruktion benötigt.

```
static int CostTable[] = {
// Statically cost base values:
  1, // INS_ADDLW       add 8-bit literal to WREG
  1, // INS_ADDWF       add WREG to f
  1, // INS_ADDWFC      add WREG and carry bit to f
  1, // INS_ANDLW       AND literal with WREG
  2, // INS_GOTO        unconditional jump
  .
  .
  .
  1, // INS_XORLW       exclusive OR literal wiht WREG
  1, // INS_XORWF       exclusive OR WREG with f
};
```

Der LLIR-Parser wird aus dem File „llir3lex.lex“ erstellt und muss auch angepasst werden. Er enthält Abschnitte, in denen man sogenannte „Unmovable Statements“ (in unserem Fall alle Sprungbefehle) und implizite Registerparameter definiert. Letztere geben an, bei welchen Operationen ein bestimmtes Register immer ein Operand ist. Die Produktregister (PRODL und PRODH) beinhalten zum Beispiel immer das Resultat einer Multiplikation; eine entsprechende Angabe im File sähe folgendermaßen aus:

```
LLIR_Operation::createImplicitRegParam(INS_MULLW, PHREG_PRODL,
```

```
USAGE_DEF);
   LLIR_Operation::createImplicitRegParam(INS_MULLW, PHREG_PRODH,
USAGE_DEF);
```

Wie man sieht, kann gleich die entsprechende Usageinformation mit angegeben werden (hier: DEF). Von dieser Angabe haben wir jedoch wieder abgesehen, da die Produktregister natürlich nicht alloziert werden können. Für andere Architekturen mag es allerdings Befehle geben, die allgemeine Register als festen Parameter beinhalten und für die solche Definitionen sinnvoll oder gar nötig sind.

Anpassung des Backendfiles für die LLIR

Das eigentliche Backendsourcefile muss ebenso angepasst werden. Neben der offensichtlichen Einbindung der LLIR-Headerdatei werden nun alle im Programmdurchlauf erkannten Funktionen und Basisblöcke zur LLIR-Datenstruktur hinzugefügt. Man durchläuft hierzu einfach wie vorher die Liste der Funktionen und fügt jede nun mit der Befehlssequenz

```
llirBB=new LLIR_BB(block.c_str());
llirFun=new LLIR_Function(<Funktionsname>,"}\n", llirBB);
llir.InsertFunction(llirFun);
```

hinzu, wobei llir unsere LLIR-Instanz und llirBB der zu erzeugende erste Basisblock der Funktion ist. Analog durchläuft man nun die Basisblöcke der von LANCE aufgebauten Datenstruktur für diese Funktion und ruft für die Befehle den Codeselektor auf. Um auch dort die korrekten Befehle einzufügen, fügt man in das Backendfile noch einige Vereinfachungen ein. Damit man in der Grammatik komplexere Aufrufe einspart, kapselt man diverse Funktionsaufrufe. Die Funktion CreateIns erhält einen Instruction-Code (wie in der proc.h-Datei definiert) und mehrere (oder keine) Parameter. Beispielhaft im Folgenden die Version mit zwei Parametern:

```
void CreateIns(InstructionCode mnemonic, LLIR_Parameter *p1,
         LLIR_Parameter *p2)
{
    LLIR_Operation *o=CreateOp(mnemonic);
    o->AddParameter(p1);
    o->AddParameter(p2);
    LLIR_Instruction *i=new LLIR_Instruction();
    i->InsertOp(o);
    llirBB->InsertIns(i, llirIns);
    llirIns=i;
}
```

Zunächst wird eine neue LLIR-Operation mit den übergebenen Parametern erzeugt und die entsprechende Instruktion in den aktuellen Basisblock eingefügt. Die Funktion InsertIns (wie auch InsertFunction und InsertBB) sorgt dafür, dass die Verkettung der einzelnen Befehle (Funktionen/ Blöcke) vorgenommen wird. Über die LLIR kann so zum Beispiel die erste oder letzte Instruktion eines Blocks erreicht werden. Zur weiteren Vereinfachung werden nun für jeden Befehl (und alle seine Varianten) Emitfunktionen angelegt, die im Codeselektor aufgerufen werden. Da dort mit der Klasse LLIR_Register gearbeitet wird, wird der Aufwand weiter verringert. Beispielhaft für den Befehl MOVF mit einem Parameter (Register) sieht die Emitfunktion folgendermaßen aus:

```
void EmitMOVF(LLIR_Register *reg)
{
  CreateIns(INS_MOVF, new LLIR_Parameter(reg,USAGE_DEFUSE,false));
}
```

Hier wird gleich der LLIR_Parameter-Konstruktor mit drei Parametern benutzt, um automatisch die Usageinformationen korrekt anzugeben. In anderen Fällen können sich diese Informationen jedoch in Abhängigkeit einzelner Parameter ändern. Hier können dann Fallunterscheidungen genutzt werden, um verschiedene Versionen der Benutzung des Registers abzudecken.

Ebenfalls im Backend wird die Behandlung von globalen Variablen durchgeführt. Hierzu wurde die Funktion `c_0` konstruiert, in der Initialisierungen vorgenommen werden. Die von LANCE angelegte Symboltabelle `symtab` enthält alle globalen Variablen und ihre Initialisierungswerte – sofern vorhanden. Die Befehle für `c_0` werden dann wie folgt konstruiert:

```
for (sym=symtab->FirstObject();sym;sym=symtab->NextObject())
{
  if(sym->Type()->Class()!=IRTYPE_FUNCTION)
  {
    LLIR_Register* tmp = NewGVR(sym,VREG_R);
    SetGVR(sym,tmp);
    if (sym->InitExp())
    {
      EmitMOVLW(sym->InitExp()->Value());
      EmitMOVWF(tmp,bsr);
    };
  };
};
EmitRETURN(0);
```

Die globalen Variablen werden dabei in einer separaten Hashtabelle mittels der Befehlsfolge `NewGVR()`, `SetGVR()` abgelegt. Die oben auftretende Variable `bsr` kann über den Compiler gesetzt werden, um entweder die Access Bank des PIC oder eine andere durch das Bank Select Register spezifizierte Speicherbank zu benutzen.

Anpassung der Baumgrammatik

In der Baumgrammatik selbst können nun die Actionparts entsprechend angepasst werden. Folgendes Beispiel zeigt die Write-Regel eines WREG-Inhalts in der ursprünglichen Fassung:

```
stm: cs_WRITE(acc)
{
  $cost[0].cost = 1 + $cost[2].cost;
}
=
{
  char* v1;
  v1 = $action[2]();
  fprintf(asmfile,"\n\tMOVWF %s,0",CS_LHSSYM($1)->Name());
};
```

Zusätzlich zu der Ersetzung des Ausgabebefehls `printf` durch die Emitfunktion sind noch einige kleinere Änderungen vorzunehmen. Zunächst wird geprüft, ob das Zielregister bereits in der Hashtabelle vorhanden ist:

```
                LLIR_Register *tmp;
                tmp = GetVR(CS_LHSSYM($1));
```

Sollte dies nicht der Fall sein, wird ein neues (virtuelles) Register erzeugt und in der Hashtabelle eingetragen.

```
if (!tmp)
{
  tmp=NewVR(VREG_R);
          SetVR(CS_LHSSYM($1),tmp);
        }
```

Anschließend kann der Befehl – nach Abarbeitung der Unterbäume – eingefügt werden.

```
        $action[2]();
        EmitMOVWF(tmp,bsr);
```

In der LLIR-Version der Grammatik kann ebenfalls der Fall eines lesenden Zugriffs auf eine vorher unbeschriebene Variable einfach über die Kostenfunktion abgefangen werden. Beispiel:

```
        acc: cs_PLUS(cs_READ,acc)
        {
           LLIR_Register *tmp;
           tmp = GetVR(CS_EXPSYM($2));
           if(!tmp) $cost[0].cost = INFINITY;
           else $cost[0].cost = 1 + $cost[3].cost;
         };
```

Da nun alle Register in der Hashtabelle abgelegt werden, kann man überprüfen, ob das zu lesende Register enthalten ist. Sollte dies nicht der Fall sein, werden die Kosten der Regel auf unendlich gesetzt, was bedeutet, dass diese Regel nie ausgewählt wird.

4.6.5 Erstellte Hilfsmittel

Im Entwicklungsprozess unseres Compilers war es nötig – und auch sinnvoll –, einige Tools zu erstellen, die z. B. beim Testen sehr hilfreich waren. Es handelt sich bei diesen Tools um zwei relativ einfache Perl-Programme:

1. **Der *Treeparser***, arbeitet auf den Informationen, die man von Lance über die zu überdeckenden Bäume erhält. Diese Informationen liegen in einer Preorder-Reihenfolge vor, so dass in jeder Zeile ein Knoten inklusive der auszuführenden Operation ausgegeben wird. Wurzeln stehen in dieser Darstellung linksbündig, ihre Söhne werden um ein Leerzeichen eingerückt und deren Söhne wiederum ein Leerzeichen mehr, und so weiter. Der *Treeparser* läuft über die Knoten und erstellt für jeden Knoten einen Eintrag in einem VCG-kompatiblen [San] Format in einer Datei. In einem zweiten Durchlauf durch die Bäume werden dann anhand der Einrückungen die Kanten in die Datei eingefügt. VCG ist dabei eine einfache Beschreibungssprache für Graphen, die dann anhand von speziellen Layout-Algorithmen von Programmen wie z. B. Xvcg gelesen und graphisch dargestellt werden können. Ein Knoten in VCG ist dann eine einfache Zeile nach folgendem Beispiel:

   ```
    node: { title : "2" label: "WRITE 'i_5 = t1;' "
   color: green}
   ```

Dieses Beispiel beschreibt einen grünen Knoten mit dem eindeutigen Titel „2“ und als (später sichtbares) Label wurde die Operation WRITE eingefügt. Entsprechend werden dann Kanten mit Start- und Zielknoten generiert:

```
edge: {sourcename: "3" targetname: "2" color: black}
```

Im Backend wurde dann die Option aufgenommen, die Baumstruktur in eine Datei zu schreiben, die dann vom *Treeparser* in das VCG-Format umgewandelt wird. Anhand dieser Visualisierungen konnte man dann leicht erkennen, welche Bäume wie in der Grammatik überdeckt werden müssen, bzw. welche Regeln falsch oder noch fehlend waren.

2. **Der *ASM-Parser*** formatiert die Ausgabe des Assemblercodes unseres Backends in eine zum MPLab kompatible Form um. Dazu werden Header ergänzt, Klammern um Operanden entfernt und eine globale Einsprungmarke eingefügt.

4.6.6 Registerallokation und Peephole-Optimierung

Wichtige Phasen des Backendlaufes sind die Registerallokation (die Zuordnung der als in unendlicher Zahl angenommenen virtuellen Register zu den endlich vielen physikalischen Registern), sowie die Code-Optimierungen. Gelegentlich wird die Registerallokation auch als Optimierung aufgefasst, sofern der generierte Code ohne sie schon korrekt und ausführbar wäre.

Registerallokation

Der Ausgangspunkt für die Registerallokation ist der so genannte Interferenzgraph.

Def.: *Der Interferenzgraph G = (V,E) ist ein ungerichteter Graph mit der Knotenmenge V (der Menge der virtuellen Register) und der Kantenmenge* $E \subseteq V \times V. \exists\, e = (v,w) \in E : v,w \in V \Leftrightarrow live(v) \cap live(w) \neq \emptyset$

Dabei ist `live(n)` die Menge aller Programmpunkte, zu denen `n` einen Wert hat, der an einem späteren Programmpunkt noch benötigt wird. Offensichtlich können also zwei virtuelle Register nicht auf dasselbe physikalische Register abgebildet werden, wenn sie adjazent sind.

Die in der Literatur beschriebene Vorgehensweise für die Allokation basiert auf dem Färben des Interferenzgraphen. Da dieses Problem NP-hart ist, wurden Heuristiken entwickelt, die gute Ergebnisse erzielen können; in unserem Fall wurde auf einen Algorithmus aus dem Buch von Appel [App98] zurückgegriffen. Hierbei wird von einer Konstanten `k` ausgegangen, die die Anzahl der physikalisch vorhandenen Register bezeichnet.

Zunächst werden zwei Listen angelegt – die Simplify List, die Knoten enthält deren Grad kleiner als `k` ist, und die Spill List, welche die anderen Knoten enthält. Solange erstere Liste nicht leer ist, wird die Funktion `simplify` aufgerufen. Dort werden Knoten aus der Simplify List sowie aus dem Interferenzgraphen entfernt und auf den `Select Stack` gelegt. Sollte die Simplify List leer sein, wird die Funktion `SelectSpill` aufgerufen, sofern die Spill List nicht ebenfalls leer ist. In SelectSpill wird ein Knoten aus der Spill List entfernt und hinten an die Simplify List angehängt. Dies entspricht der Bezeichnung dieses Knotens als potentieller Kandidat für ein `Spillen` (Auslagern von Werten im Hauptspeicher). Durch das Verkleinern des Graphen in der Funktion Simplify ist also gesichert, dass irgendwann sowohl die Spill List als auch die Simplify List leer sind.

Nun werden den Registern Farben zugewiesen. Dazu wird ein Knoten vom Select Stack entfernt und alle Farben – repräsentiert durch Zahlen – für unzulässig erklärt, wenn eine Kante im Interferenzgraphen zu einem Knoten existiert, der bereits diese Farbe zugewiesen bekommen hat. Die erste mögliche Farbe wird dann diesem Knoten zugewiesen. Für weitere Details siehe [App98].

Zur Umformung in der LLIR wird nun noch einmal das Programm durchlaufen. Für jedes Register wird nun in einer Datenstruktur nach dem ersten anderen Register mit derselben Farbe gesucht. Wird eines gefunden, dann wird das getestete Register durch dieses ersetzt. In der LLIR sieht das folgendermaßen aus:

```
for(mapit help=colorMap.begin();help!=colorMap.end();help++)
{
  if((*help).second == j)
  {
    if (strcmp((*help).first->GetName(),test->GetName()) == 0) break;
    LLIR_UsageType ut = ins->GetOp(0)->GetParam(i)->GetUsageType();
    if(fun->FindRegister((*help).first->GetName()))
    {
      ins->GetOp(0)->AddParameter
      (new LLIR_Parameter(fun->FindRegister
           ((*help).first->GetName()),ut,false),i);
      ins->GetOp(0)->DeleteParameter(ins->GetOp(0)->GetParam(i+1));
      break;
    };
  };
};
```

Dabei ist `test` das aktuell zu färbende Register, `j` die zugewiesene Farbe und `i` die Nummer des Parameters im aktuellen Befehl, der durch das Register `test` ersetzt wird.

Peephole-Optimierungen

Die Integration von Compiler-Optimierungen zur Verbesserung der Codegüte werden, sofern diese Optimierungen nicht bereits auf der Intermediate-Representation durchgeführt werden können, in das Compiler-Backend verlagert. Hierbei werden in der Literatur verschiedene Ansätze beschrieben, die in ihrer Anwendbarkeit allerdings oftmals stark architekturabhängig sind. Eine Compiler-Optimierung, die für jedes Compiler-Backend durchgeführt werden kann, sind die Peephole-Optimierungen. Hierbei sollen ineffiziente Codesequenzen, die in der Codeselektion durch die Abbildung der Datenflussgraphen auf Bäume entstehen, ersetzt werden. Dazu werden bestimmte Folgen von Instruktionen in kürzere oder schnellere Instruktionen transformiert, um lokal begrenzte Codeverbesserungen durchzuführen. Häufig werden hierdurch überflüssiges Laden und Speichern eliminiert, desweiteren algebraische Transformationen und Vereinfachungen durchgeführt. Die Vorgehensweise besteht dann darin, nur einen kleinen Programmausschnitt zu betrachten (so genannte Peepholes), welcher zumeist nur Befehlsfolgen aus zwei bis drei Befehlen umfasst. Dieser Ausschnitt kann aber natürlich der jeweiligen Architektur bzw. dem erzeugten Code angepasst werden. Diese Form der Optimierung kann sowohl vor, als auch nach der Registerallokation durchgeführt werden. Peephole-Optimierungen werden in der Literatur oftmals als Nach-Optimierungen oder lokale Optimierungen bezeichnet, da diese Form der Optimierung nur auf Basisblock-Ebene durchgeführt werden kann. Die Problematik bei Peepholeoptimierungen besteht darin, Ansätze für solche Optimierungen zu finden – eine Studie des erzeugten Codes zu vielen Programmen ist daher notwendig.

Integration von Peephole-Optimierungen

Bis zum derzeitigen Stand wurden insgesamt zehn Peephole-Sequenzen gefunden, die jeweils durch günstigere bzw. kürzere Befehlssequenzen in der Nach-Optimieriung überdeckt werden können. Hierbei ist im Wesentlichen darauf zu achten, dass die Ersetzungssequenz die Programmkorrektheit bewahrt, insbesondere das Setzen von Status-Registern ist bei der Wahl von solchen Sequenzen zu berücksichtigen. So konnten z. B. bei der zu diesem Zeitpunkt aktuellen Implementierung der Drop-Zone-Mission 149 Instruktionen

von 4491 Instruktionen eingespart werden.

Die im Rahmen der Projektgruppe implementierte Realisierung der Peephole-Optimierung bietet folgende Vorteile:

- Effiziente Ersetzungsstrategie
- Betrachtung von Programmausschnitten beliebiger Länge
- Einfügen neuer Optimierungen durch den Benutzer

Die Grundlage für die Peephole-Optimierung stellt der sogenannte Peephole-Tree dar. Hierbei handelt es sich um einen entarteten Baum, der aus einer Wurzelliste besteht. In den jeweiligen Wurzelknoten stehen die Startinstruktionen von Sequenzen, die ersetzt werden können. Die Kinder der Wurzelknoten sind die Folgeinstruktionen einer solchen Sequenz. Diese Knoten können weiterhin Nachbarknoten haben, in denen Instruktionen mit gleichem Vorgängerbefehl stehen, dadurch wird der Speicherplatzbedarf des Baumes gering gehalten und die Laufzeit der Optimierungsphase reduziert. In den Blättern stehen die jeweiligen Ersetzungssequenzen. Der Baum wird dynamisch aufgebaut, indem die Peephole-Optimierungen aus einer Datei, der sogenannten picpeep.dat, ausgelesen werden. Dieses Vorgehen ermöglicht somit das Einfügen neuer Optimierungen durch den Benutzer, hierzu muss die picpeep.dat vom Benutzer angepasst werden. Auf die hierbei zu verwendene Syntax und die korrekte Verwendung der Parameter wird im Folgenden eingegangen. Die Grafik in Abb. 4.38 zeigt aussschnittsweise den Baum wie er durch die derzeitige pic-

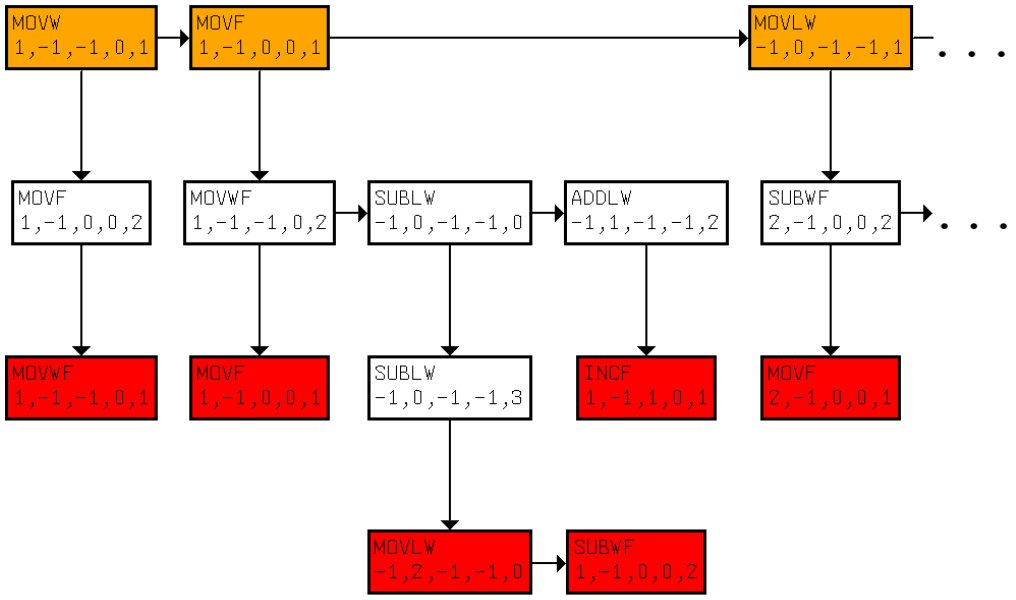


Abbildung 4.38: Peephole-Tree

peep.dat aufgebaut wird. Folgender Codeausschnitt und dessen Ersetzung werden in Abb. 4.39 dargerstellt.

```
MOVLW c                          MOVF X,0,a
SUBWF X,0,a          ->          SUBLW c
SUBLW 0
```

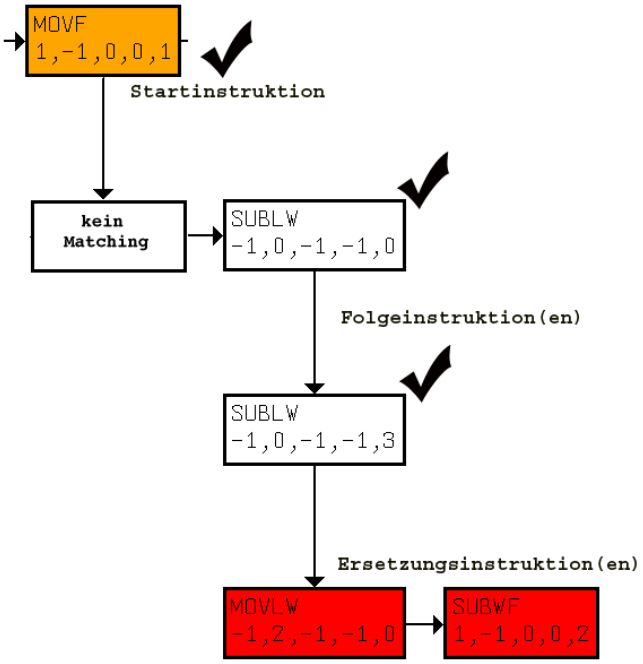


Abbildung 4.39: Ersetzungssequenz

Editieren der picpeep.dat

Wie bereits oben erwähnt, erlaubt die Peephole-Optimierung das Einfügen neuer Optimierungen durch den Benutzer. Hierzu muss die dazu bereits existierende picpeep.dat um die jeweils neu gefundenen Peepholes und deren Ersetzungen erweitert werden. Hierzu soll exemplarisch an einem Auszug der Datei die zu verwendende Syntax erläutert werden.

```
-,
41,1,-1,-1,0,1,
37,1,-1,0,0,2,
>,
41,1,-1,-1,0,1,
```

Eine jeweils neue Peephole-Sequenz beginnt nach der Zeichenfolge `'-,'`. Es folgen die einzelnen Befehle, eingeleitet durch den Instruktionscode des Befehls mit jeweiliger Angabe der Befehlsparameter. Auf die Zeichenfolge `'>,'` folgt dann die Ersetzungssequenz. Eine Übersicht über die Befehle mit zugehörigen Instruktionscodes liefert folgende Tabelle:

Mnemonic	Code	Mnemonic	Code	Mnemonic	Code
ADDLW	0	CPFSEQ	23	POP	46
ADDWF	1	CPFSGT	24	PUSH	47
ADDWFC	2	CPFSLT	25	RCALL	48
ANDLW	3	DAW	26	RESET	49
ANDWF	4	DECF	27	RETFIE	50
BC	5	DECFSZ	28	RETLW	51
BCF	6	DCFSNZ	29	RETURN	52
BN	7	GOTO	30	RLCF	53
BNC	8	INCF	31	RLNCF	54
BNN	9	INCFSZ	32	RRCF	55
BNOV	10	INFSNZ	33	RRNCF	56
BNZ	11	IORLW	34	SETF	57
BRA	12	IORWF	35	SLEEP	58
BSF	13	LFSR	36	SUBFWB	59
BTFSC	14	MOVF	37	SUBLW	60
BTFSS	15	MOVFF	38	SUBWF	61
BTG	16	MOVLB	39	SUBWFB	62
BOV	17	MOVLW	40	SWAPF	63
BZ	18	MOVWF	41	TBLRD	64
CALL	19	MULLW	42	TBLWT	65
CLRF	20	MULLWF	43	TSTFSZ	66
CLRWDT	21	NEGF	44	XORLW	67
COMF	22	NOP	45	XORWF	68

Die auf den Instruktionscode folgenden Parameter stehen für fileregister, literal, destination, accessbank und dem Hilfsparameter depth. In der folgenden Tabelle werden die jeweiligen Parameter mit Wertigkeit erläutert.

Parametername	Werte-Beschreibung
fileregister	setze -1, falls es sich um einen Literalbefehl handelt setze x, falls sich der Befehl auf das Fileregister des x-ten Befehls bezieht
literal	setze den Wert der im Befehl stehen soll Ausnahme bei Parameter depth=0, s. unten
destination	setze -1, falls der Parameter nicht im Befehl vorhanden oder beliebig ist setze 0, falls der Befehl ins WREG schreibt setze 1, falls der Befehl ins Fileregister x schreibt
accessbank	setze -1, falls Ziel beliebig ist setze 0, falls auf Accessbank zugegriffen wird setze 1, falls die Bank via BSR ausgewählt wird
depth	numeriert die Befehle der Sequenz durch setze 0, falls Literal beliebig

Dementsprechend lässt sich anhand des Beispiels

```
-,
40,-1,0,-1,-1,0,
61,2,-1,0,1,2,
60,-1,0,-1,-1,3,
>,
37,2,-1,0,1,1,
60,-1,1,-1,-1,0,
```

das durch das Backend erzeugte Peephole formulieren: Die Befehlssequenz

`MOVLW c` (wobei c beliebige Konstante)
`SUBWF X,0,1` (wobei X beliebiges Register, WREG-Befehl, schreibt in die BSR-Vorgabe)
`SUBLW 0`

kann ersetzt werden durch die Befehlssequenz

`MOVF X,0,1` (wobei X Register aus dem zweiten Befehl ist, WREG-Befehl, schreibt in die BSR-Vorgabe)
`SUBLW c` (wobei c Konstante aus dem ersten Befehl ist.)

4.7 Die Applikation Drop-Zone-Mission

Die Drop-Zone-Mission ist der Benchmark, mit dem gezeigt werden soll, dass der Bau von komplexen Robotern mit Hilfe der Lösungen von ROCK möglich ist. Die eigentliche Aufgabe, die es im Rahmen der Drop-Zone-Mission zu lösen gilt, ist in Abschnitt 4.7.2 auf Seite 109 beschrieben.

4.7.1 Der Roboter

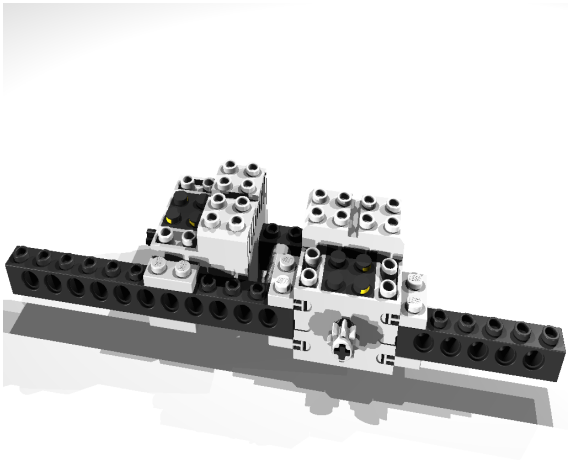


Abbildung 4.40: Eine Hälfte des Chassis

Um die Drop-Zone-Mission zu lösen, wurde ein Roboter in verschiedenen Evolutionsstufen erarbeitet. An den Roboter stellen sich folgende Anforderungen:

- Differentieller Antrieb, um einfach zu navigieren.
- Bumper an allen Ecken, um Hindernisse zu erkennen.
- Einen Mechanismus, um Coladosen und Hindernisse voneinander zu unterscheiden.
- Eine Vorrichtung, um Coladosen zu bewegen.
- Sensoren, um das Spielfeld, insbesondere die Drop-Zone, zu erkennen.
- Raum, um die Steuerung unterzubringen.

- Sonardrehvorrichtung.
- Stabilität der Konstruktion.

Diese Ziele wurden durch Aufbau mehrerer Roboter inkrementell erreicht. Die Kompaktheit des Roboters wird durch die Verwendung der seitlichen Befestigungsnuten der Motoren erreicht, eine Hälfte ist in Abbildung 4.40 zu sehen. Die Nutzung der Nuten bewirkt, dass der Motor ein Teil des Chassis bildet und so nicht unnötig Raum verwendet wird.

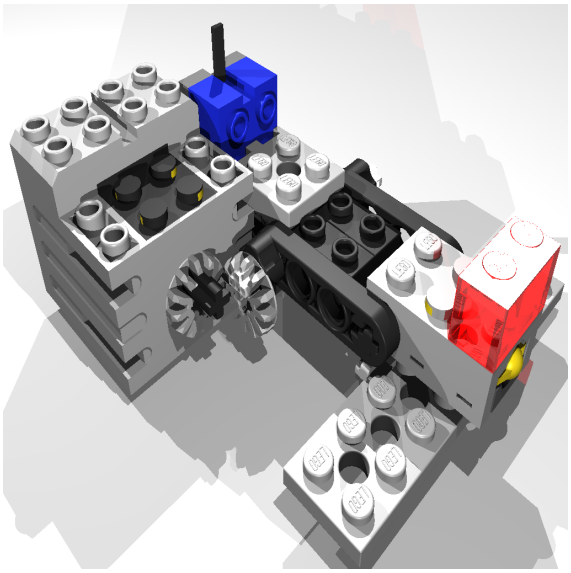


Abbildung 4.41: Der Magnetmechanismus

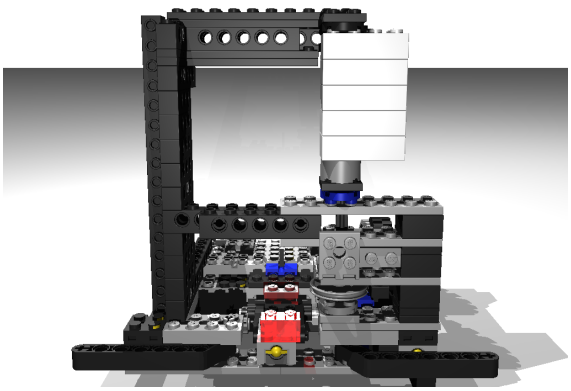


Abbildung 4.42: Der Sonaraufbau

Coladosen können von kleinen starken Dauermagneten so stark angezogen werden, dass der Roboter sie in die Drop-Zone ziehen kann. Da sich herausgestellt hat, dass Abwurfmechanismen, die die Dose bewegen, nicht Platz sparend aufzubauen sind, wurde ein anderer Ansatz verfolgt. Ein oberhalb des beweglichen Dauermagneten angebrachter Querholm verhindert, dass die Dose nach hinten kippen kann. Um die Coladose abzusetzen, wird der Magnet über eine einfache Klappvorrichtung lotrecht gedreht und von der Dose entfernt. Nachdem der Roboter ein Stück zurückgefahren ist, kann der Magnet wieder in die Ruheposition gebracht werden.

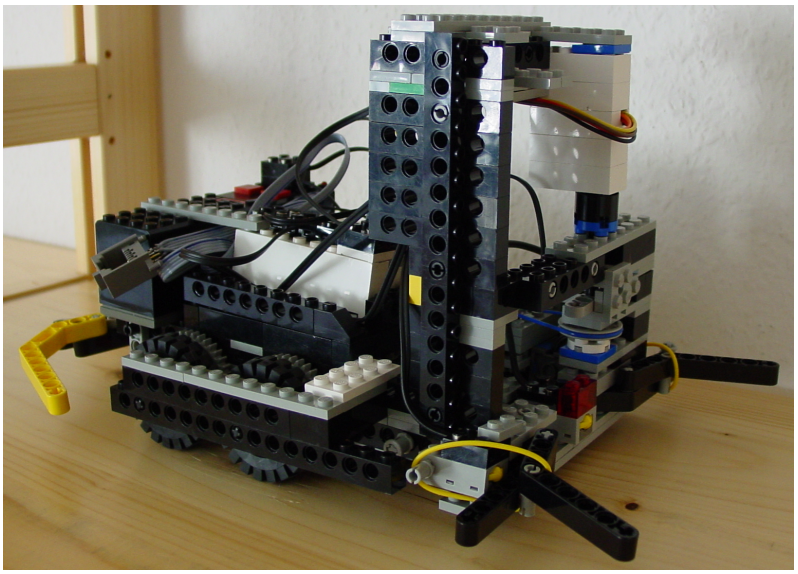


Abbildung 4.43: Foto des Roboters

Unter dem Magneten befindet sich ein Taster, der gedrückt wird, sobald eine Dose am Magneten hängt. Diese Teilkonstruktion ist in Abbildung 4.41 dargestellt.

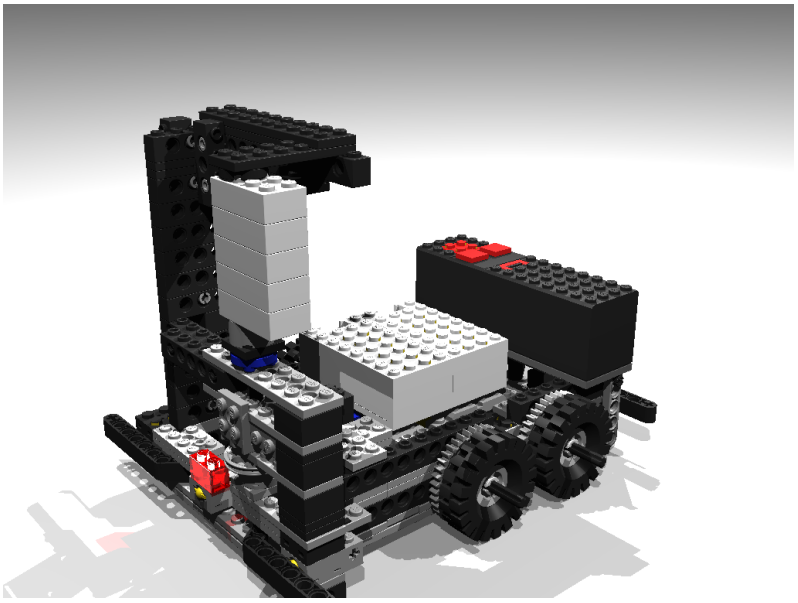


Abbildung 4.44: POV-Ray-Modell des Roboters

Das Sonar besitzt einen so großen Öffnungswinkel, dass bei ungünstiger Montage der Abstand zur Tischfläche gemessen wird und fälschlich als Objekt in einer Nähe von weniger als 50 cm erkannt wird. Daher ist es notwendig, dass das Sonar in einer größeren Höhe am Roboter befestigt wird.

Der Algorithmus zum Lösen der Drop-Zone fordert, dass das Sonar nach vorne und links gedreht werden kann, außerdem sollte das Sonar möglichst nah am Magneten angebracht sein, damit der Paralaxenfehler minimiert wird und der Roboter das Objekt nicht verfehlt. Dieser Aufbau ist in Abbildung 4.42 gezeigt.

Ist das Sonar wiederum zu nahe am Magneten, dann werden Teile des Roboters als Objekte aufgefasst,

obendrein wird der Sensor von der Dose teilweise verdeckt.

Zwischen den beiden Antriebsmotoren ist der Motorhub untergebracht. Am hinteren Teil des Roboters befinden sich weitere Bumper zur Erkennung von Hindernissen bei Rückwärtsfahrt. Über den Bumpern ist als Gegengewicht zum vorderen Aufbau der Batteriekasten aufgesetzt. Die wiederaufladbaren Batterien können so leicht gewechselt werden.

Über den Antriebsmotoren und dem Motorhub ist die ROCK-Einheit gut zugänglich aufgesteckt.

Ein Problem, welches sich auch durch einen modularen Aufbau nicht lösen lässt, ist, dass jeder Hub, Sensor und Motor mindestens ein Kabel benötigt, die alle entweder an einem Hub oder der Zentraleinheit enden, so dass ein nicht zu vermeidendes Kabelgewirr entsteht.

Abbildung 4.43 zeigt ein Foto des realen Roboters. Ein POV-Ray-Modell ist in Abbildung 4.44 zu sehen.

4.7.2 Der Algorithmus

Die Anwendung steuert einen Roboter, der mit vier Motoren und acht Sensoren ausgestattet ist. Zwei Motoren bilden den Antrieb und steuern je zwei Räder auf der linken bzw. auf der rechten Seite. Der dritte Motor bewegt den Abstoßmechanismus für das aufgenommene Payload und der Vierte dreht den Sonarsensor um je 90° nach links bzw. nach rechts. An den vier Ecken angebrachte Berührungssensoren sollen vor Hindernissen warnen. Unter dem Magneten befindet sich ein Berührungssensor, welcher ein Hindernis anzeigen soll. Eingesammelte Cola-Dosen halten diesen Taster bis zum Abwurf gedrückt. Weiterhin gibt es noch zwei Lichtsensoren, einer ist mittig und der andere links vorne angebracht. Beide sind nach unten ausgerichtet, um die Umrandung oder die Drop-Zone zu erkennen.
Die o. a. Aufgabe erledigt der Roboter durch die Anwendung wie im Folgenden beschrieben:

Nachdem der Roboter auf die Lichtverhältnisse kalibriert und auf einer beliebigen Position innerhalb der Umrandung abgestellt wurde, beginnt er nach dem Rand zu suchen. Sobald er die schwarze Linie erreicht hat, dreht sich der Roboter, um der Begrenzungslinie entgegen dem Uhrzeigersinn zu folgen. Während er der Umrandung folgt, scannt er mit Hilfe des Sonarsensors die links von ihm liegende Missionsfläche. Erkennt er ein Objekt 90° links von sich, dreht er sich darauf zu und fährt dorthin. Dort angekommen (mittlerer Berührungssensor ist gedrückt) prüft er, ob es sich um ein Hindernis oder ein Payload handelt, indem er ein Stück zurückfährt und erneut überprüft, ob der mittlere Sensor weiterhin gedrückt ist. Ist dies nicht der Fall, ist es ein Hindernis gewesen und er fährt zum Rand zurück, um das nächste Objekt zu suchen. Ist der Sensor jedoch immer noch gedrückt, so hat er ein Payload gefunden, das er nun in die Drop-Zone bringen wird. Dafür fährt er zur Linie zurück und folgt dieser bis zur nächsten Kurve. Von dort fährt der Roboter schräg über die Fläche, um so die grüne Drop-Zone zu finden und das Payload mit Hilfe des Abstoßmechanismus abzustellen. Wurde die Drop-Zone nicht gefunden, fährt er zur nächsten Kurve. Hat er die Drop-Zone erreicht, merkt er sich die Kurve, von der aus er gestartet ist und kann dadurch die Drop-Zone später schneller wiederfinden.
Für eine detaillierte Beschreibung siehe [PG4].

4.7.3 Die Sonarauswertung

Der Algorithmus soll Objekte innerhalb des Feldes erkennen. Erste Versuche eines Detektionsalgorithmuses liefen ins Leere, da zuviele Objekte erkannt wurden.

Um einen für dieses Problem passenden Algorithmus zu entwerfen, wurde das Programm zunächst so modifziert, dass keine Erkennung stattfindet und jeder Messwert in den EEPROM des PIC18 geschrieben wird. Da nur 256 Speicherplätze zur Verfügung stehen, wurde der Prozessor nach dem 256. Schreibvorgang angehalten.

So entstanden verschiedene Messkurven. Die Messkuve in Abbildung 4.45 zeigt die Rohdaten. Der Roboter ist dabei entlang der langen Seite des Spielfeldes gefahren. Die erste Messung wurde direkt nach Einbiegen

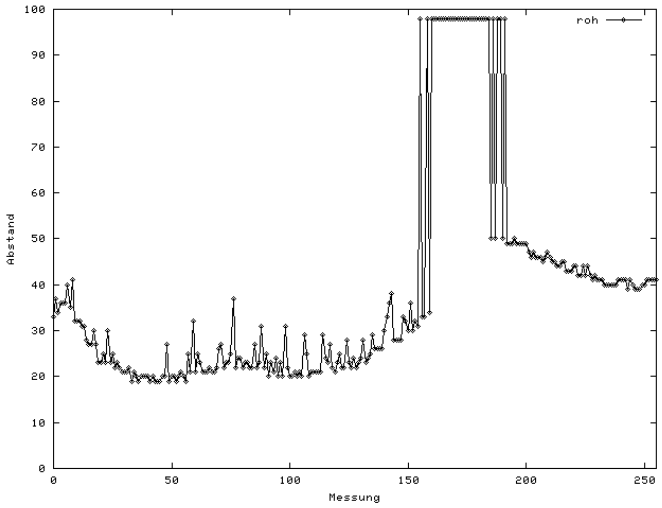


Abbildung 4.45: ungefilterte Messwerte des Sonars

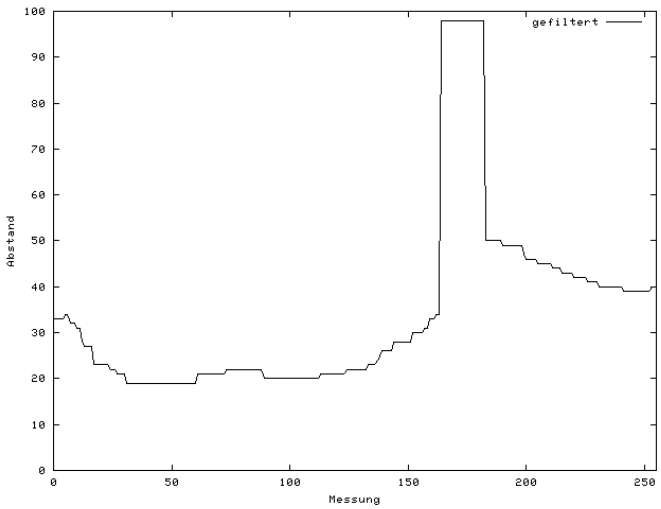


Abbildung 4.46: gefilterte Messwerte des Sonars

auf die Gerade getroffen. Die letzte wurde im letzten Drittel aufgenommen. In ca. 27 cm Entfernung von den beiden schwarzen Linien stand eine Cola-Dose, eine Wagenbreite weiter in ähnlicher Entfernung eine zweite. Auf dem letzten Teil des Spielfeldes stand eine dritte Dose.

Die Betrachtung der Rohdaten zeigt, dass die Messwerte sehr stark variieren, so dass eine größere Historie vorgehalten werden muß. Außerdem ist eine Glättung notwendig, damit die Objekte einfacher erkannt werden können. Mit Hilfe von MS-Excel wurden verschiedene Funktionen und Analysemethoden ausprobiert.

Das Diagramm aus Abbildung 4.46 zeigt die Daten, wie sie der Analyse zugeführt werden. Sie werden aus den Rohdaten gewonnen, indem das gleitende Minimum der letzten sieben Werte benutzt wird. Um die Messgenauigkeit zu erhöhen, wird vorher der Mittelwert zwischen jeweils benachbarten Messwerten gebildet.

Die Detektion benutzt die Erkenntnis, dass ein Objekt zunächst um `DEPTH` näher kommt und sich dann wieder vom weiterfahrenden Roboter entfernt. Um nicht zu viele Objekte zu erkennen, wird weiter gefordert, dass sich das Objekt auch wieder um `DEPTH` entfernt hat. Die Detektion wird dann ausgelöst, sobald eine Entfernungszunahme festgestellt wurde. Der gesamte Algorithmus ist in Abbildung 4.47 abgedruckt.

Wie oft die Entfernungen kleiner werden müssen, kann in `DEPTH` festgelegt werden. Die History ist ebenfalls variabel angelegt.

Für diesen Algorithmus wurde ein Unix-Programm geschrieben, das aus MPLAB exportierte Messreihen einliest und die erkannten Objekte meldet. So ist es möglich, viele Situationen aufzunehmen und die optimalen Werte für `DEPTH` (1) und `HISTORY` (7) zu bestimmen.

```
#define HISTORY 7
#define MAX 98
#define DEPTH 1
int histc=0;
int hist[HISTORY];
int last=0;
int now=MAX;
int status=DEPTH;

int minh(){
  int ret=MAX;
  int i=HISTORY-1;
  while(i>=0){
    if(hist[i]<ret)ret=hist[i];
    i--;
  }
  return ret;
}


int senseObject(pic a){
        int ret=0;
        hist[histc++]=a;
        if(histc==HISTORY)histc=0;

        now=minh();

        if(status>0){
                /*Find minimum*/
                if(now<last)status--;
        }
        if(status<0){
                /*Find maximum*/
                if(now>last)status--;
                if(status<=-DEPTH){
                   status=-DEPTH; /*Now we are up enough*/
                   if(now<last)status=DEPTH-1;
                }
        }
        if(status==0){
                /*Minimum can now  come,
                  if goes up this is the Object go for max*/
                if(now>last){
                        ret=1;   /*Found some thing!*/
                        status--;
                }
        }

        if(status>DEPTH)status=DEPTH;

        return ret;

}
```

Abbildung 4.47: Algorithmus zum Erkennen von Objekten

Kapitel 5

Ausblick

5.1 Kommunikation per Infrarot

Die Infrarotkommunikation wurde so weit implementiert, dass der PIC12 einzelne Bytes senden und empfangen kann. Die Zeit reichte leider nicht aus, um diese Softwarekomponenten an das ROCKWIRE anzupassen und zu testen. Es fehlt daher die Methode `RWProcess`. Sie müsste per Infrarot ankommende Pakete zwischenspeichern und auf Anfrage an den Master weiterleiten. Dieser schickt – entweder gleichzeitig oder in einem neuen Paket – die zu sendende Antwort.

Weiterhin fehlen noch einige Erweiterungen des ROCKComm-Programms (siehe Abschnitt 4.5.2) und ein modifizierter Bootloader, der die benötigten Treiber zu einem früheren Zeitpunkt in der Bootreihenfolge initialisiert, um sie rechtzeitig bereitzustellen.

Da die Datenübertragung per Infrarot sehr langsam ist, wie man bei der RCX-Einheit sieht, ist diese auf Dauer sicher kein akzeptabler Weg, um häufig benötigte oder aktualisierte Software auf den Master zu transferieren, denn eine Datenmenge von 32 kB benötigt dafür mehrere Minuten. Für die Verständigung mehrerer Roboter untereinander wäre es sicherlich ein wünschenswertes und interessantes Feature. Da die ROCK-Einheit aber auch so erweiterbar ist und per serieller Schnittstelle kommunizieren kann, ist die fehlende Infrarotübertragung verschmerzbar.

5.2 RoCK-Betriebssystem

An SOUL lassen sich noch einige Verbesserungen vornehmen:

- Implementierung des zweiten Anschlusses für einen ROCKWIRE-Master
- SOUL und die Programme getrennt voneinander übertragbar machen
- Bereitstellung eines Timers, der periodisch eine User-Funktion aufruft
- Möglichkeit, per Infrarot neue Programme zu übertragen
- Entwicklung eines Benutzerinterfaces auf dem LCD-PIC (Shell)
- Realisierung einer Debug-Konsole, um per RS232-Schnittstelle SOUL zu debuggen
- Verbesserung der Initialisierung der Tasks, so dass man einfach neue Tasks hinzufügen oder verändern kann

- Unterstützung für mehrere Programme im Speicher, von denen aber jeweils nur eines läuft; ein Programm besteht dabei wie gewohnt aus mehreren Tasks
- Nutzung des *SLEEP*-Modus des PIC18 im IdleTask, wenn alle anderen Tasks blockiert sind

5.3 Compiler

Der ROCK-Compiler erlaubt bisher leider noch nicht den vollen Umfang der Programmierung im ANSI-C-Standard. Auch die Optimierungsmöglichkeiten im Backend sind nur teilweise (z. B. Peepholes) implementiert. Der ANSI-C-Standard [Gig] beschreibt den Umfang und die Charakteristik der Programmiersprache C, denen der Compiler möglichst gerecht werden sollte. Es war nicht beabsichtigt – und in dem gesetzten zeitlichen Rahmen auch nicht möglich – C-Code im vollen Umfang dieses Standards in PIC18-Assembler zu übersetzen. Dennoch werden schon einige Merkmale von ANSI-C erkannt, wie z. B. der Datentyp „signed int“, ein Großteil des Befehlssatzes (mit den Ausnahmen Division und Modulo) oder eindimensionale Arrays, um nur einige zu nennen.
Für zukünftige Projekte ist eine Erweiterung des Funktionsumfangs, d. h. eine Erhöhung des Abdeckungsgrades des ANSI-C-Standards, erstrebenswert. Dazu könnte man noch folgende Fähigkeiten des C-Codes auf dem PIC18 implementieren:

- Zahlen vom Datentyp long und float (und entsprechender Arithmetik).
- Integration der bisher noch nicht realisierten Operationen (vgl. Kap. A.3.2 auf Seite 127) wie Division, Shift-Operationen oder Modulorechnung, die aufgrund der fehlenden Unterstützung durch den PIC-Befehlssatz noch nicht eingesetzt wurden.
- Arrays mit mehr als einer Dimension.
- Parameterübergabe von mehr als einem Pointer, bzw. generell von Datentypen mit mehr als acht Bit.
- Inline-Assembly (um z. B. optimierten Code einbetten zu können) oder Compiler-Known-Functions.

Abgesehen von den schon genannten Ergänzungen zum ROCK-Compiler gibt es natürlich noch weitere Optimierungsmöglichkeiten abseits des ANSI-C-Standards. Dazu zählt exemplarisch die Erkennung der Sprungweite im Compiler, um ggf. Branchbefehle mit ihrer begrenzten Sprungweite durch Gotobefehle zu ersetzen, wenn das Sprungziel nicht mehr vom Branch adressiert werden kann. Auch die Erkennung weiterer Peepholes erscheint lohnenswert.
Alle diese Features waren aufgrund des relativ kurzen Entwicklungszeitraums nicht mehr realisierbar, dennoch bietet der ROCK-Compiler schon deutlich mehr als nur die Basisfunktionalität zur Programmierung von PIC18-Prozessoren und insbesondere unserer ROCK-Einheit im Zusammenspiel mit dem Betriebssystem SOUL.

Anhang A

Hilfsmittel

A.1 Verwendete Hardware-Hilfsmittel

A.1.1 Debugging mit dem Mixed-Signal-Oszilloskop

Das Agilent 54641D 2+16-Kanal, 350 MHz Mixed-Signal-Oszilloskop (MSO) kann mit zwei analogen und 16 digitalen Kanälen detaillierte Signalanalysen durchführen und bietet die Möglichkeit, es wie einen Logik-Analyzer zu benutzen.

Der Anwender hat die Möglichkeit, die komplexen Zusammenhänge zwischen den Signalen von bis zu 18 Kanälen gleichzeitig anzusehen.

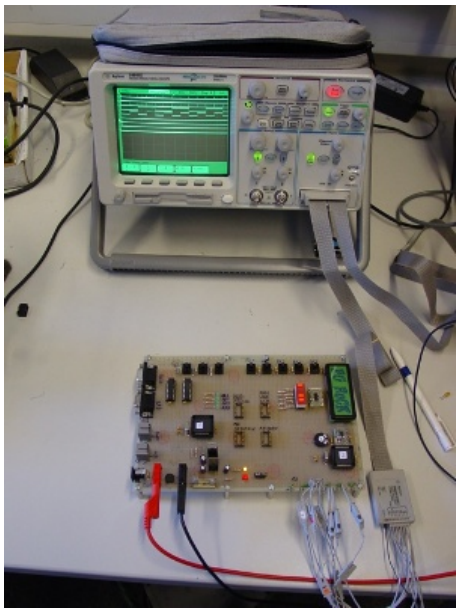


Abbildung A.1: Das Agilent 54641D-Oszilloskop mit dem Evaluation-Board [Agi]

Für die Zwecke dieser PG stellte das Mixed-Signal-Oszilloskop ein wichtiges Hilfsmittel zum Debuggen der elektrischen Schaltung sowohl auf dem Evaluation-Board wie auch für verschiedene Testaufbauten im Zusammenhang mit dem ROCKWIRE dar. Insbesondere bei der Fehlersuche mit der Displayansteuerung und der Timings für ROCKWIRE leistete das Gerät wertvolle Dienste, da mit seiner Hilfe die Datenports

am Prozessor (PIC12 bzw. PIC18) hinsichtlich des Timings und der Signaldauer einfach analysiert werden konnten.

Auch nach Aufbau der endgültigen SMD-Platinen war das Gerät in der Lage, wichtige Hinweise auf Probleme mit dem Oszillator zu liefern.

Die Abbildung A.2 zeigt die Anschlussklemmen des Oszilloskops in Verbindung mit den von uns herausgeführten Prozessorpins während einer typischen Debug-Sitzung am ursprünglich verwendeten ROCK-Evaluation-Board

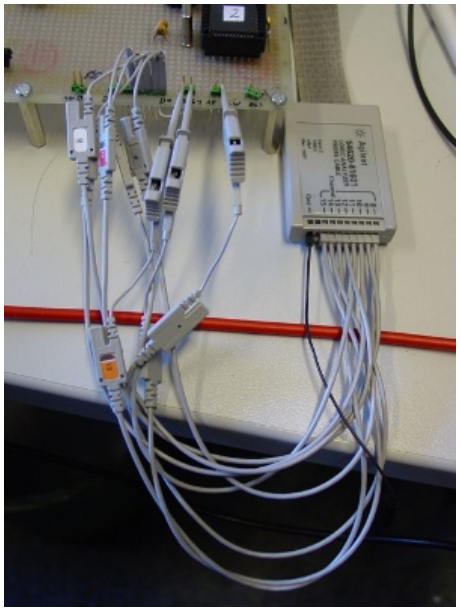


Abbildung A.2: Typische Debug-Sitzung und Anschlussklemmen auf dem Evaluation-Board

Nachfolgend eine Beschreibung der einzelnen Bedienelemente. Für eine nähere Erläuterung der Funktionsweise sei auf die Homepage des Herstellers Agilent [Agi] verwiesen, wo man unter anderem eine detailliertes (englisches) Handbuch findet.

Interessant ist insbesondere die Möglichkeit, die aktuelle Anzeige als Bitmap auf Diskette abzuspeichern, um sie dann ausdrucken oder weiterbearbeiten zu können. Von dieser Möglichkeit wurde im Rahmen des Endberichts und für den Abschlussvortrag durchaus rege Gebrauch gemacht.

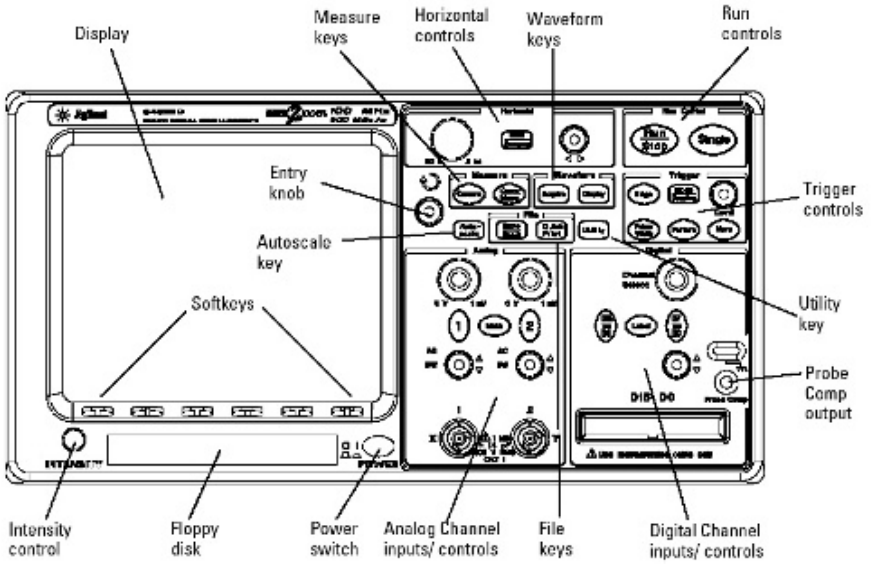


Abbildung A.3: Bedeutung der Bedienelemente des Oszilloskops. [Agi]

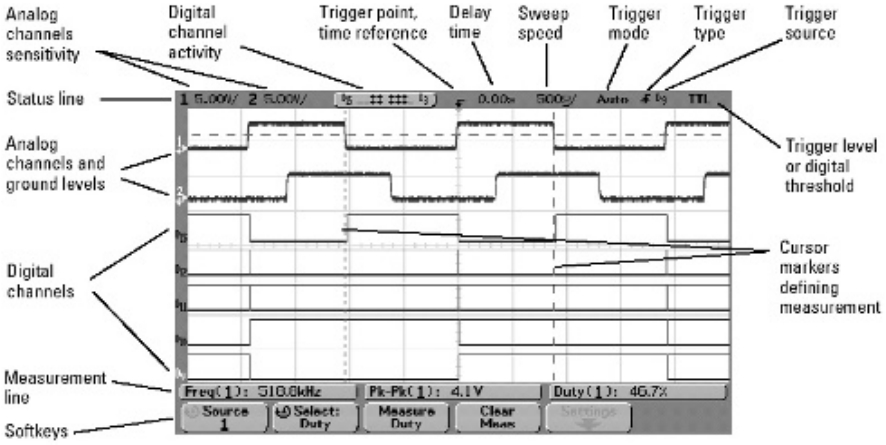


Abbildung A.4: Die einzelnen Bildschirmelemente. [Agi]

Auf dem Bildschirm erkennt man auf der einstellbaren Zeitskala deutlich das an den Dateneingängen anliegende Signal. Dieses kann nun hinsichtlich der Dauer und Periodizität der Taktflanken im Vergleich mit den entsprechenden Sollwerten des Herstellers bzw. der Vorüberlegung auf Papier analysiert werden.

A.1.2 Der Festspannungsregler

Ist in einer Schaltung ein genauer und stabiler Spannungspegel gefordert, so setzt man solche Spannungsregler ein. Bei Anwendungen mit ständiger Spannungsstabilisierung und geringer sowie stabiler Stromentnahme reicht eine einfache Schaltung mit Vorwiderstand und Z-Diode aus. Wenn allerdings größere Ströme, genauere und stabilere Spannungswerte gefordert werden, sind integrierte Festspannungsregler notwendig. Diese haben zusätzlich eine interne Strombegrenzung, die bei Überlastung und Kurzschluss einsetzt. Bei einem Kurzschluss regelt der Festspannungsregler seine Ausgangsspannung automatisch herunter. Wird der Kurzschluss aufgehoben, stabilisiert sich die Ausgangsspannung wieder auf ihren festen Wert. Eine thermische Schutzschaltung verhindert die Zerstörung des ICs durch Überhitzung. Die bekanntesten Bauteile dieser Art sind die 78xx-Serie für positive und die 79xx-Serie für negative Spannungen. Man legt also eine bestimmte Spannung am Eingang des Reglers an und erhält am Ausgang eine stabile niedrigere Spannung. Hierbei sollte die angelegte Spannung min. 3 V über der Ausgangsspannung liegen, da sonst die Verlustleistung zu hoch wäre, was zu einer größeren Wärmeentwicklung führen würde. Bei Low-Drop Typen reichen auch 0,4 V über der Ausgangsspannung. Die angelegte Spannung sollte aber höchstens 36 V betragen. Die Ausgangsspannung kann 5, 6, 8, 9, 12, 15, 18 oder 24 V groß sein. Die Beschaltung ist bei den drei Pins denkbar einfach. Aufmerksamkeit verdienen lediglich die Kondensatoren, die an Ein- und Ausgang angeschlossen werden sollten. Ihre Werte kann man den jeweiligen Datenblättern entnehmen. Eine Beispielbeschaltung ist in Abb. A.5 dargestellt.

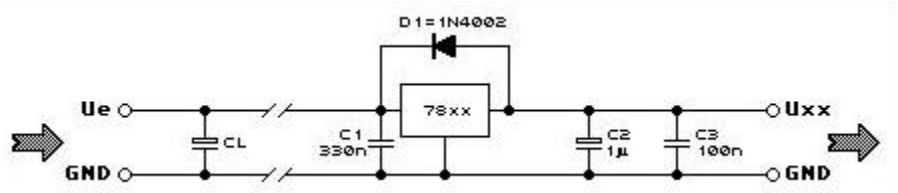


Abbildung A.5: Eine Beispielbeschaltung eines Linearreglers [Spr]

Die Rücklaufdiode

Die ebenfalls in der Schaltung eingezeichnete Diode D1 ist eine Vorsichtsmaßnahme. Es reicht schon aus, dass die Ausgangsspannung U_a für einen kurzen Moment größer als die Einganssspannung U_e ist, um den Regler zu zerstören, was wiederum zu einem Kurzschluss zwischen Ein- und Ausgang führen würde. Dann würde U_a der Spannung U_e entsprechen, was die Zerstörung der Schaltung zur Folge hätte.

In den Schaltungen der PG ist eine Rücklaufdiode nicht erforderlich, weil am Ausgang keine oder nur sehr kleine Kondensatoren vorgesehen sind.

Minimaler und maximaler Ausgangsstrom

Für einen einwandfreien Betrieb der Regelschaltung wird eine minimale Stromlast vorgeschrieben. Typisch sind hier 5 mA, im schlimmsten Fall können es auch 10 mA sein. Nur unter der Erfüllung dieser Voraussetzung sind die angegebenen Werte garantiert. Es gibt auch so genannte “Low-Power”-Linearregler. Diese haben eine geringere Anforderung an die minimale Stromlast.

Die Information, wieviel Strom das entsprechende Bauteil maximal liefert, ist ebenfalls dem Datenblatt zu entnehmen. Bei der 78xx- sowie der 79xx-Serie z. B. trägt dieser Wert 1 A. Es ist jedoch zu beachten, daß

diese Werte nur innerhalb eines maximalen Spannungsabfalls, der sogenannten “Dropoutspannung”, gültig sind.

A.1.3 Die Brücken-Gleichrichterschaltung

Die Brücken-Gleichrichterschaltung wird auch als Zweipuls-Brücken-Gleichrichterschaltung B2 bezeichnet. Diese dient dazu eine Wechselspannung in Gleichspannung umzuwandeln. Dasselbe könnte auch mit einer einfachen Einweg-Gleichrichterschaltung erreicht werden. Im zweiten Fall wird jedoch, je nach Ausrichtung der Diode, nur die positive Welle einer anliegenden Wechselspannung durchgelassen, wobei die erste Schaltung wie in Abb. A.6 die beiden Stromrichtungen in dieselbe Richtung durch die angeschlossene Schaltung laufen lässt.

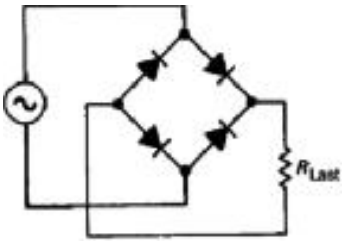


Abbildung A.6: Eine Brücken-Gleichrichterschaltung

Die Brückengleichrichter sind auch als ICs erhältlich (siehe Abb. A.7).

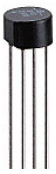


Abbildung A.7: Eine integrierte Schaltung des Brückengleichrichters

Bei unserem Board ist die Spannungsquelle eine Gleichspannung. Ein Brückengleichrichter kann jedoch auch der Verpolungssicherheit dienen.

A.1.4 Die Ladungspumpe

Mit Ladungspumpen kann man negative Spannungen erzeugen. Es gibt viele integrierte Schaltkreise, die als Ladungspumpe arbeiten. Am bekanntesten ist wohl der in Abb. A.8 dargestellte ICL7660. Dieser IC benötigt lediglich zwei externe Kondensatoren, um bei +5V Eingangsspannung -5V am Ausgang zu erzeugen.

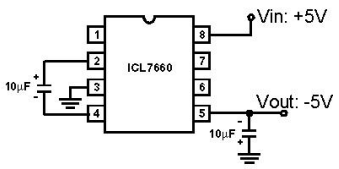


Abbildung A.8: Der icl7660 [Spr]

Die bricht unter Last allerdings zusammen (siehe Abb. A.9), aber bei 18 mA Last stehen immer noch -4 V zur Verfügung. Das dürfte für die meisten Anwendungen reichen. Um mehr Strom zu erzeugen, kann man mehrere dieser ICs parallel schalten.

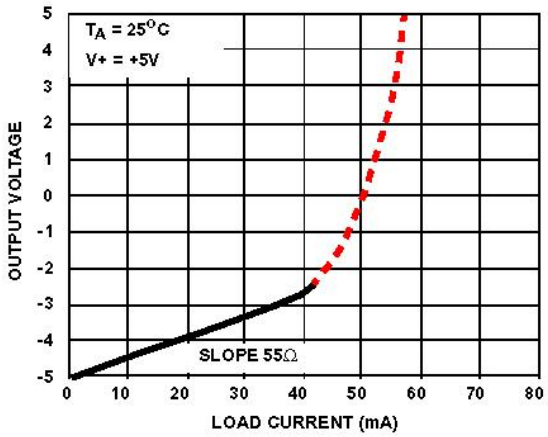


Abbildung A.9: Der icl7660 unter Laststrom [Spr]

A.1.5 Das PICDEM 2 Plus-Demonstration-Board

Das PICDEM 2 Plus [Micc] (Abbildung A.10) vereinfacht die Einarbeitung in die Entwicklung von Embedded-Systemen mit der PIC-Architektur sehr stark und bietet eine Plattform, eigene Designideen zu testen.

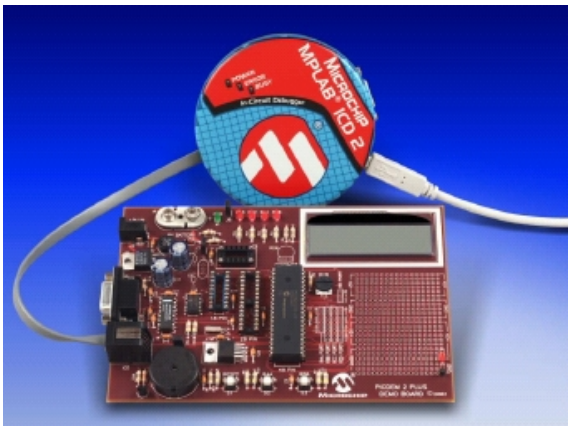


Abbildung A.10: Das PICDEM2 Plus und der ICD2 [Micc]

Im Lieferumfang ist ein Demonstrationsprogramm (inklusive Assembler-Quelltext) enthalten, das viele der Funktionen der MCU anschaulich zeigt. Zum Beispiel werden mit dem PIC18F452 eine Uhr und ein Temperatursensor realisiert, wobei die Daten auf einem LCD und auch per RS232 ausgegeben werden. Im Besonderen wird beispielsweise auch die hardware-unterstützte Pulsweiten-Modulation (PWM) anhand eines Piezo-Summers demonstriert. Das PICDEM 2 Plus kann zudem auch mit dem In-Circuit-Debugger ICD2 (Kapitel A.2.1 auf der nächsten Seite) eingesetzt werden.

Merkmale des PICDEM 2 Plus

- 2x16 LCD
- per PWM-Signal betriebener Piezo
- RS232-Schnittstelle
- Temperatursensor
- 4 LEDs
- 2 Tast-Schalter, Reset-Schalter
- PIC18F452 und PIC16F877 MCU (nur eine gleichzeitig)
- ICD-Anschluss
- Demo-Programme mit Quelltexten
- Prototyping-Bereich (von der PG nicht genutzt)
- Arbeitet mit 9 V-Batterie oder Netzteil

A.2 Entwicklungsumgebungen und -methoden

A.2.1 MPLAB / ICD2

Der In-Circuit-Debugger 2 (ICD2) [Micb] ermöglicht es, Programme zur Laufzeit in der Schaltung zu debuggen. Der PIC18 läßt sich mit diesem Gerät bequem in der mitgelieferten IDE MPLAB2 programmieren und debuggen, sofern es keine Zeitkritischen Abschnitte sind. Abbildung A.11 zeigt den Aufbau, der beim Entwickeln von SOUL verwendet wurde. Der ICD2 wurde an den USB eines PCs angeschlossen, auf dem das MPLAB genutzt wurde. Der ICD2 wurde wiederum per Western-Stecker mit einem Entwickler-Board verbunden.

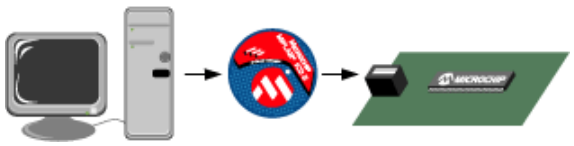


Abbildung A.11: Entwicklungsumgebung von SOUL [Micb]

Das Geheimnis des In-Circuit-Debuggens sind zwei dedizierte Leitungen bzw. Mikrocontroller Pins, die nur vom ICD2 genutzt werden (bei der PIC18F452 MCU handelt es sich um $\overline{MCLR}$ und Vpp). Dadurch werden zum einen ICSP (In Circuit Serial Programming) und zum anderen Debugging-Funktionen durch die proprietäre ICD2-Firmware realisiert. Die ICD2 Debugging-Funktionalität ist in die MCU eingebaut und wird durch das Einprogrammieren des Debug-Codes in den Ziel-Prozessor aktiviert. Hierbei sind einige Ressourcen im Debug-Modus der MCU nicht mehr für Anwendungen verfügbar. Dazu zählen eine Ebene im Stack, einige General Purpose Register (beim PIC18F452 Register 0x5f4-0x5ff) und ein kleiner Teil des Programmspeichers (beim PIC18F452 0x7dc0-0x7fff).

Die PG RoCK nutzte zunächst das Entwickler-Board PicDem 2 Plus von Microchip (siehe Kapitel A.1.5 auf der vorherigen Seite) in Verbindung mit dem ICD2. Bei der Entwicklung des RoCKDem-Boards wurde auf Kompatibilität zum PICDEM 2 Plus und ICD2 geachtet, so dass es auch möglich ist, das eigene System zu programmieren und zu debuggen.

Die Firmware des ICD2 basiert auf Flash-Technologie, die ein Upgrade ermöglicht, was von der PG RoCK beim Umstieg von dem nicht ganz ausgereiften MPLAB5 auf MPLAB6 auch genutzt wurde.

Der ICD2, der wie oben beschrieben per USB an den Entwickler-PC angeschlossen wird, ist eine intelligente Schnittstelle (oder Übersetzer) zwischen MPLAB und dem zu entwickelnden System, die es dem Entwickler erlaubt, den Verlauf der Ausführung des zu testenden Programmes und den aktuellen Speicherinhalt der MCU zu beobachten und zu verändern. Es kann ein einzelner Breakpoint gesetzt werden, um das Programm an dieser Stelle zu unterbrechen und zu tracen.

Zusätzlich steht noch ein Single Step Mode zur Verfügung, in dem man jeden Befehl einzeln ausführen

kann, dabei ist aber zu beachten, dass in diesem Modus keine Interrupts bearbeitet werden.

Weitere Informationen zu MPLAB und ICD2 befinden sich auf dem Internetauftritt von Microchip. [Micd]

A.2.2 Eagle 3.55

Einfach Anzuwendender Graphischer Layout Editor

Bei Eagle handelt es sich um einen graphischen Editor, zum Zeichnen von Schaltplänen und zur Erstellung eines Leiterplattenlayouts. Die Dateiformate von Eagle können von vielen Leiterplattenherstellern direkt verwendet werden.
Eagle besteht aus drei Teilen: Bauteilbibliotheken mit dem Bibliothekseditor, dem Schaltplan- und dem Layouteditor.
In Schaltplänen werden für die Bauteile Symbole verwendet, diese sollen vor allem die Lesbarkeit des Schaltplanes erhöhen. Daher sollten Symbole immer den Anforderungen des aktuellen Schaltplans angepasst werden.
Für den Leiterplattenentwurf müssen hingegen Abbildungen der realen Bauteile verwendet werden. Dieses wird durch die Bauteilbibliotheken erreicht, die die Symbole mit Packages verbinden. Ein Package ist die Abbildung eines Bauteilgehäuses. Dabei werden die Pins der Symbole mit den entsprechenden Pins bzw. Pads der Packages verbunden.
Eagle wird mit einer großen Anzahl von Bibliotheken geliefert, im Internet sind weitere Bibliotheken erhältlich, außerdem kann man selber Bibliotheken editieren. Es ist jedoch etwas kompliziert einzelne Bauteile aus mehreren Bibliotheken in eine einzige Bibliothek zu kopieren, um z. B. eine Arbeitsbibliothek mit allen benötigten Bauteilen zu erstellen. Dazu wird die Bibliothek über Export: Script in eine Textdatei exportiert, diese enthält die Symbole, Packages und Devices. Sie kann mit einem einfachen Editor bearbeitet werden, wobei man aber sehr sorgfältig vorgehen muß. Das editierte Bibliotheksscript kann dann in eine Bibliothek importiert werden.
Beim Entwurf eines Schaltplanes muß man zuerst die richtigen Bauteile auswählen. Der spätere Leiterplattenentwurf kann nur dann funktionieren, wenn die korrekten Packages benutzt wurden. Es kostet Zeit und Geld, wenn man eine Leiterplatte herstellt, auf welche die Bauteile, die vorgesehen sind, gar nicht passen.
Als nächstes zieht man die Leitungen bzw. Netze zwischen den einzelnen Symbolen, diese werden in Eagle Nets genannt. Jedes Netz bekommt in Eagle einen eindeutigen Namen. Um die Übersichtlichkeit zu steigern, sollte man möglichst selbsterklärende Namen für die Netze vergeben.
Ein Netz besteht aus seinem eindeutigen Namen und der Liste der Pins, die an diesem Netz angeschlossen sind. Dabei muss ein Netz in einem Schaltplan nicht durch einen Draht bzw. durchgehende Leitung beschrieben sein. Wenn zwei auf einem Schaltplan räumlich getrennte Netze den selben Namen haben, bilden sie in der internen Logik ein zusammenhängendes Netz, welches auf einer Leiterplatte ein Verbindung benötigt.
Es ist möglich, mehrere Netze auf dem Schaltplan zu einem Bus zusammenzufassen, logisch bleiben es aber einzelne Netze. Somit ist der Bus nur ein Mittel des Schaltplaneditors, der keinerlei Auswirkungen auf den Layouteditor hat.
Es passiert häufiger, dass man vermeintlich einen Pin an ein Netz angeschlossen hat, diese aber nicht verbunden sind. Das gleiche kann bei zwei Netzen geschehen, die man zu einem einzigen verbinden wollte.
Daher sollte regelmäßig ein Electrical Rule Check durchgeführt werden. Dieser überprüft z. B., ob mindestens zwei Pins an einem Netz angeschlossen sind, oder ob alle Pins, die Ein- oder Ausgangssignale benötigen, auch an Netze angeschlossen sind.
Wenn der Schaltplan fertiggestellt ist, kann aus diesem ein Board erzeugt werden. Beim Entwurf der Leiterplatte müssen nun die Bauteile bzw. Packages möglichst optimal auf der Platte platziert werden.
Der Layouteditor hat mehrere Layer, die jeweils verschiedene Funktionen erfüllen, z. B. gibt es Layer für die Umrisse der Platine und einen für Bohrungen, usw. Außerdem gibt es einen Layer für die Kupferbahnen auf der Oberseite und eine für die der Unterseite.
Dabei ist zu beachten, dass man alle Layer von oben betrachtet, d.h. man sieht durch die Platine von oben auf die Unterseite. Bauteile, die auf der Unterseite platziert werden sollen, müssen dazu gespiegelt werden.

Die noch nicht verbundenen Netze werden als Airwires in einem eigenen Layer dargestellt, hier ist es wichtig, dass die Netze im Schaltplan korrekt verbunden wurden.
Das optimale Platzieren der Bauteile und die optimale Verlegung der Kupferleiterbahnen, das sogenannte Routen, sind die eigentlich schwierigen und sehr zeitaufwendigen Arbeitsschritte.
Im Design Rule Check können Vorgaben, z. B. vom Platinenhersteller, bezüglich minimaler Leiterbahnbreite, minimaler Leiterbahnenabstand, Bohrdurchmesser, Lötstoplack usw. eingestellt werden, deren Einhaltung überprüft wird.
Eagle besitzt zwar auch einen Autorouter, der auf Basis einiger Vorgaben versucht, die Leiterbahnen zu verlegen, bei etwas komplexeren Leiterplatten wird aber selten ein zufriedenstellendes Ergebnis erzielt.
Nachdem das Leiterplattenlayout fertig ist, sollte ein Ausdruck davon gemacht werden, und die Bauteile darauf gelegt werden, um zu überprüfen, ob diese auch passen.
Eingeschränkte Demoversionen der neuesten Eagle Version sind im Internet auf der CadSoft Homepage [Cad] erhältlich.
Außerdem wird die Lektüre des Tutorial dringend empfohlen. Es wird zum einen mit der Eagle Software ausgeliefert, bzw. steht auf der CadSoft Homepage [Cad] zum Download bereit.

A.2.3 Eclipse

Eclipse ist eine Entwicklungsumgebung, die von der OSI (Open Source Initiative) zertifiziert wurde und unter der CPL (Common Public License) verfügbar ist. Eclipse ist in Java entwickelt und somit auf den verschiedensten Plattformen einsetzbar. Die Integration von CVS in Eclipse kommt uns zusätzlich entgegen, da wir uns für CVS als Dokumentenverwaltung entschieden haben. Durch die umfangreiche Hilfe ist der Umgang mit der Eclipse-Plattform auch für den ungeübten Entwickler schnell erlernbar. Der „Code Assistent“ und der „Parameter Hit“ tragen deutlich zur schnellen Entwicklung bei. Diese beiden Funktionen zeigen einem zu jedem Zeitpunkt die möglichen Methoden bzw. Variablen für eine Methode. Für häufig verwendete Fragmente wie Try / catch etc. gibt es Templates, die mit dem Code-Assistenten schnell eingefügt werden können. Diese Templates sind erweiterbar, zum Beispiel für die Dokumentation. Besonders hervorzuheben ist die „on the fly“ Kompilierung des Codes. Bei einem Fehler wird das entsprechende Wort farblich markiert und in einem separaten Fenster wird der Fehler erläutert und Korrekturvorschläge präsentiert. Ein Nachteil liegt darin, dass die Tastenkürzel für häufig verwendete Befehle nicht verändert bzw. nicht hinzugefügt werden können. Hier liegt noch ein gewisses Verbesserungspotential. Das Programm ist sehr stabil und hat auch keine Probleme, viele Instanzen des implementierten Programms zu starten (der JBuilder von Borland ist bei diesem Versuch häufig abgestürzt). Dieser Aspekt ist nicht zu vernachlässigen, da das PICDemBoard nicht immer zur Verfügung stand und somit die Anwendung mit sich selbst kommunizieren musste. Für eine detailliertere Beschreibung von Eclipse siehe [ecl].

Eclipse ist eine freie Entwicklungsumgebung, bei der es sich lohnt, die Zeit zur Einarbeitung zu investieren.

A.2.4 Code-Dokumentation

Die Beschreibung der Produkte/Anwendungen, die von uns entwickelt wurden, wollten wir nicht zweimal machen (Code und separate Beschreibung). Es bietet sich an deshalb an, die Dokumentation im Code abzulegen, denn es gibt einige Programme, die aus diesen Kommentaren eine Dokumentation erstellen können. Die Dokumentation ist auf der Archiv-CD (Kapitel C) zu finden. Im Folgenden werden zwei nützliche Werkzeuge zur Dokumentation beschrieben.

Doxygen

Mit Doxygen wird aus dem im Code implementierten Beschreibungen der einzelnen Methoden bzw. Klassen eine Dokumentation für die gesamte Anwendung generiert. Dabei müssen einige syntaktische Regeln eingehalten werden, siehe [Hee]. Doxygen ist eine weit verbreitete Anwendung, die schon in vielen Pro-

jekten Anwendung fand und ist somit entsprechend ausgereift. Hinzu kommt die Flexibilität in der endgültigen Darstellung des Dokuments. Man kann eine Darstellung als HTML generieren lassen, bei der dann u. a. die Kopf, Fuß und StyleSheets frei konfiguriert werden können. Andere Darstellungsformate, wie z. B. RTF, LaTeX, XML oder Man sind auch möglich. Einen Nachteil hat Doxygen jedoch, es kann nichts mit Assemblercode anfangen. Somit muss unser Assemblercode mit einer anderen Dokumentationserstellungssoftware bearbeitet werden.

Robodoc

Hierbei fiel die Wahl auf Robodoc, ein Programm, welches stetig weiter entwickelt wird und z. Z. in der Version 4.0 vorliegt. Es erzeugt eine akzeptable Dokumentation, die optisch und funktional jedoch nicht ganz an die Qualität von Doxygen heranreicht. Für eine detaillierte Beschreibung siehe [SKW].

A.2.5 Versionsverwaltung mit CVS

> CVS is an instrument for making sources dance to your tune. But you are the piper and composer. No instrument plays itself or writes it's own music.[Ced]

Bei der computergestützen Durchführung von Projekten ergibt sich das Problem, dass Änderungungen an Dateien in der Regel nicht rückgängig gemacht werden können. Also ist es notwendig, alte Versionen selbst zu verwalten. Dieses ist aus folgenden Gründen sehr wichtig:

- Anwender hat Probleme mit einer alten Version und will oder kann nicht die neueste benutzen oder kaufen.
- Entfernte Verfahren oder Merkmale des Projektes erweisen sich doch als besser
- Allgemeine Fehlersuche
- Vergleichen, um zu ermitteln, was sich im Laufe der Zeit am Projekt geändert hat

Das automatische Erstellen von (inkrementellen) Sicherheitskopien reicht hier nicht aus, da es in gewissen Zeitintervallen erfolgt und somit Änderungen verloren gehen können.

CVS unterstützt das gleichzeitige, entfernte Arbeiten an Dateien in Softwareprojekten. Einfache Änderungen, die parallel gemacht wurden, können automatisch zusammengeführt werden, d. h. non exclusive locking wird unterstützt. Wurde genau am selben Punkt gearbeitet, muss man die Änderungen manuell zusammenführen.

CVS kann Entwicklerkonferenzen nicht ersetzen. Bei punktuellen Problemen in eigenen Klassen schafft es der Entwickler meistens noch durch eigene Kraft, modul-übergreifende, strukturelle Probleme müssen weiterhin gemeinsam gelöst werden.

Auch sollte trotz CVS jedes Modul (verkörpert durch eine ganze Reihe von Dateien) weiterhin noch so etwas wie einen Verantwortlichen haben. Wenn wir Softwareentwicklung mal mit einem gepflegten Fünf-Sterne-Essen vergleichen, darf nicht jeder Koch die Soße abschmecken, dafür gibt es immer noch den Chef de Saucier. Kurzum, viele Köche verderben weiterhin den Brei. Dies ist auch der Grund, warum Opensource-Projekte (viele Köche) einen Maintainer haben, der die Verbesserungen (Patches) verwaltet.

Das Lösen von Versionskonflikten wurde von CVS ohne weitere Intelligenz auf rein textueller Basis gelöst, es kann die die semantische Logik nicht erkennen.

Die lokalen Einstellungen

Das Repository wurde am LS XII unter

```
/home/pg/rock/INTERNAL/CVSROOT
```

angelegt. Durch den Befehl

```
cvs -d :ext:<username>@ls12sp.cs.uni-dortmund.de:/home/pg/rock/INTERNAL/CVSROOT \
       co <modulname>
```

– wobei man den Backslash weglassen und alles in eine Zeile schreiben sollte – kann man sich eine aktuelle Kopie der Projekte besorgen.

Die wichtigsten Anwendungsfälle

cvs co *modul*	Erstellen einer lokalen Arbeitskopie
cvs status *datei*	Mögliche Probleme finden
cvs ci *datei*	Lokale Änderungen einpflegen
cvs up *datei*	Repository-Version in lokale Version patchen

TkCVS

TkCVS [Str] (Abbildung A.12) ist eine auf Tcl/Tk-basierende graphische Oberfläche für CVS. Es zeigt den Status der Dateien im ausgewählten Verzeichnis und stellt Schalter und Menüs zur Verfügung um CVS-Kommandos auf den gewählten Dateien auszuführen. Eine weitere Anwendung, TkDiff, ist integriert, um Browsen und Mergen von Änderungen zu unterstützen.

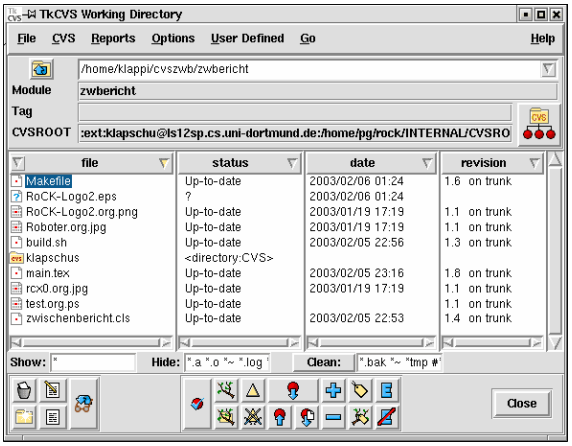


Abbildung A.12: Screenshot von TkCVS

A.3 Hilfsmittel zum Compilerbau

Im Entwicklungsprozess des Compilers standen uns einige Tools zur Verfügung, die den Prozess sehr vereinfacht und beschleunigt haben. Dennoch muss man sagen, dass es auch sehr viel Zeit kostet, sich mit den teilweise nur spärlich dokumentierten Tools anzufreunden. Letzteres ist auch einer der Gründe, in diesem Kapitel einen Überblick und einige nähere Erläuterungen zu den Hilfsmitteln zu geben.

A.3.1 LANCE

An dieser Stelle soll nun genauer die Funktionsweise von LANCE erklärt werden. Hierzu zählt die Erzeugung einer Zwischendarstellung zu gegebenem C-Code und die maschinenunabhängige Optimerung mittels LANCE. Nach Anpassung der config-Datei und dem Setzen der Pfade wird bei Aufruf des compile Befehls das C-Frontend von LANCE gestartet und aus der angegebenen Quelldatei `source.c` die Zieldatei `source.ir.c` erzeugt. Die Intermediate Representation wird von LANCE als 3-AC aus dem Quellprogramm generiert. Hierbei muss man zwischen der internen und der externen Darstellung unterscheiden. Intern benutzt LANCE ein eigenes Format für die IR, nach außen hin stellt es eine kompilierbare ANSI-C-Datei für jede Quellcodedatei zur Verfügung. Diese Dateien bestehen aus einer Teilmenge von C, die einen 3-AC für jedes Programm liefern. Beispiel einer Generierung:

C-Code:

```
void main()
{
   int i, A[10];
   i = A[2]++ > 1 ? 2 : 3;
}
```

Symboltabelle mit Hilfsvariablen:

```
int A[10];
char *t1,*t3;
int i,t2,t5,t6,t7,t8;
int *t4;
```

Laden von A[2]:

```
t3 = (char *)A; // cast auf char*
t2 = 2 * 4;             // Offsetberechnung
t1 = t3 + t2;   // effektive Adresse in t1
t4 = (int *)t1; // cast zurück auf int
t5 = *t4;               // laden aus Speicher nach t5
```

Inkrementieren von A[2]:

```
t6 = t5 + 1;    // Inkrement
*t4 = t6;               // Speicher in A[2]
```

Aufspalten des bedingten Ausdrucks:

```
t7 = t5 > 1;    // Vergleich
if (t7) goto L1;        // Sprung zum then-Zweig
t8 = 3;         // else-Zweig
goto L2;                // Sprung zum nächsten Statement
L1:
t8 = 2;         // then-Zweig
L2:
i = t8;         // Ergebnis nach i
```

Wie man an obigem Beispiel sieht, ist der generierte Code nicht optimal; er enthält Befehle, deren Durchführung zur Laufzeit unnötig sind. Optimierungen der IR sind maschinenunabhängig und sollen die Darstellung vereinfachen. Dabei sollen redundante Berechnungen und überflüssiger Code entfernt werden. Im

oben angegebenen Beispiel fällt die Berechnung `t2 = 2 * 4` auf. Sie liefert immer dasselbe Ergebnis, eine Berechnung zur Programmzeit ist also unnötig. Daher kann diese Zeile ersetzt werden durch `t2 = 8`. Eine derartige Optimierung nennt man constant folding. Ist diese Zuweisung geschehen, so erkennt man, dass der Wert von `t2` nicht mehr verändert wird, also konstant bleibt. Man könnte nun jede Benutzung von t2 durch den Wert 8 ersetzen; die folgende Zeile wird also zu `t1 = t3 + 8`. Der Wert von t2 wird also an spätere Stellen im Code weitergeschoben; daher auch der Name constant propagation. Nun fällt auf, dass die Zeile `t2 = 8` absolut überflüssig geworden ist, sie kann also gestrichen werden (dead code elimination), ebenso die Deklaration von t2 an sich (unused variable elimination). LANCE bietet Möglichkeiten zur IR Optimierung an, die jeweils eine Optimierung auf dem Code durchführen. Es existieren Tools u. a. für constant folding, constant propagation, copy propagation, CSE elimination, dead code elimination und jump optimization, sowie die Möglichkeit der iterativen Anwendung aller Optimierungen. Durch den Aufruf des Befehls `iropt source.irc.c` wird die optimierte `source.ir.c` erzeugt. Welche einzelnen Optimierungen LANCE vorgenommen hat. wird in der Datei `iropt.log` festgehalten. In dem Beispiel wird der Code von 15 IR-Statements auf 7 IR-Statements reduziert.

LANCE bietet weiterhin Bibliotheksfunktionen, mit denen man sowohl Kontroll- als auch Datenflussgraphen erzeugen kann. Für CFGs liefert der Konstruktor der Klasse `ControlFlow Graph` für jede Funktion – als Parameter des Konstruktors – eine CFG Darstellung. Weiterhin existiert die Möglichkeit, den Graphen durch die Funktion `showcfg` im VCG-Format zu exportieren. Analog liefert die LANCE-Klasse `DataFlowAnalysis` eine DFG-Darstellung für eine gegebene Funktion (ebenfalls als Parameter des Konstruktors); eine Exportfunktion hier ist `showdfg`. Aufgrund der Laufzeitkomplexität der Weiterverarbeitung von allgemeinen DFGs im Backend (optimale Überdeckung ist NP-hart) sowie der OLIVE++-Schnittstellen (akzeptiert nur DFTs), werden die Graphen an den CSEs aufgebrochen. Auch dies liefert LANCE bereits in Form der Klassen `LANCEDataFlowTree` bzw. `DFTManager`. Der Konstruktor dieser Klasse generiert dabei für eine gegebene Funktion eine DFT-Darstellung. Dabei besteht diese intern aus einer Liste von Basisblöcken, von denen jeder wiederum eine Liste von DFTs enthält. Die so erstellten Bäume sind direkt von OLIVE++ weiterbenutzbar.

A.3.2 OLIVE++

Olive++[Avi] ist ein so genannter *Codegenerator-Generator* der auf Erkenntnissen mit seinen beiden Vorgängern Twig und iburg basiert. Entwickelt wurde Olive++ vom ICD derart, dass man nur die strukturelle Beschreibung des Befehlssatzes in Form einer Baumgrammatik (vgl. 4.6.2 auf Seite 90) erstellen muss, um durch Olive++ einen Codegenerator zu erhalten, der dann automatisch in linearer Zeit eine Überdeckung für den übergebenen Datenflussbaum findet. Bei der Suche nach einer kostenminimalen Überdeckung eines Baumes (dem sog. Tree Parsing) geht Olive++ (genau wie iburg) nach dem Prinzip der dynamischen Programmierung vor (Quellen: [SR97] und [Leu97]): Für jeden Baum T kann eine kostenminimale Überdeckung gefunden werden, indem man für jeden Teilbaum von T die kostenminimale Überdeckung findet und diese mit den Kosten für die Wurzel von T verrechnet. Dabei wird der Baum T genau zweimal durchlaufen:

1. **Bottom-up labelling Phase**: In dieser Phase wird der Eingabebaum von den Blättern zur Wurzel durchlaufen, und jedem Knoten eine Menge von passenden Grammatikregeln zugeordnet, sowie deren jeweilige Kosten vermerkt (als auch die minimalen Kosten für die jeweiligen Teilbäume).

2. **Top-down rule emmision Phase**: Der Baum wird noch einmal von der Wurzel zu den Blättern durchlaufen und die optimale Überdeckung wird explizit erzeugt aus den Informationen der ersten Phase an den Knoten.

Da jeder Knoten in den beiden Phasen je genau einmal besucht wird, ist die Laufzeit linear zur Größe des Baumes und einem Faktor aus der Komplexizität der zugrunde liegenden Grammatik (denn für jeden

Knoten müssen alle passenden Regeln gefunden werden).
Die Regeln der Grammatik, die für Olive++ erstellt werden muss, haben folgendes Format:

- `rule -> nonterm : tree [cost] action;`
- `tree -> term( tree-list ) | term | nonterm`
- `tree-list -> ( tree-list, child ) | child`
- `child -> tree`
- `cost -> C-code | C-expression`
- `action -> C-code`

Beispielsweise sieht eine Regel in der Grammatik dann folgendermaßen aus:

```
stm: cs_RETURN(acc)
{
  $cost[0].cost = 1 + $cost[2].cost;
}
=
{
  $action[2]();
  EmitRETURN(0);
};
```

Zur Erläuterung: Diese Regel wäre passend für die Wurzel eines Baumes, der im originalen C-Code einem „Return“-Befehl entsprach. In diesem Beispiel wird erwartet, dass der Wert, der zurück gegeben werden soll, durch einen Teilbaum, der auf „acc“ abbildet, im Akkumulator der Maschine untergebracht ist. Die Berechnung der minimalen Kosten geschieht in diesem Fall, indem man die eigenen Kosten (hier 1, die dem Return entsprechen) mit den Kosten des in `$action[2]()` aufgerufenen Teilbaums addiert. `$cost[0].cost` sind die Kosten an diesem Knoten, `$action[2]()` sorgt für die Ausführung des Teilbaums, der in der ersten Zeile an Position 2 steht (`stm = 0, cs_RETURN = 1, acc = 2`), und `EmitRETURN(0)` generiert dann letztendlich den Assembler-Befehl `'RETURN'` in dem Ausgabeprogramm. Die Kosten spielen bei der Überdeckung natürlich eine zentrale Rolle, so dass man besondere Platzhalter braucht, um z. B. die Auswahl einer Regel unter bestimmten Bedingungen (die dann im Kosten-C-Codeteil abgefragt werden) zu verhindern. Dazu bedient man sich dann des Platzhalters `'INFINITY'`, der dann auf unendliche Kosten verweisen soll, so dass jede andere Regel dieser vorgezogen wird, bzw. diese Regel nie zur Überdeckung gewählt wird.
Schliesslich gilt es noch das Verfahren zu beschreiben, mit dem Olive++ in ein Backend eingebunden werden kann: Dazu ruft man Olive++ auf, z. B. mit dem Flag -I(`olive++ -I -o pic.c grammar.brg`) wenn man Debug-Prozeduren mit im Codegenerator haben will. Dann steht dem Backend die Funktion `'burm_label()'` zur Verfügung, der man den gerade zu überdeckenden Baum (bzw. dessen Wurzel) übergibt und die dann quasi die Schnittstelle zum Codegenerator darstellt. Ausserdem müssen sich Frontend (Lance2) und Codegenerator natürlich über die zu übersetzenden Operationen abstimmen, was in Form von Terminalsymbolen geschieht, die auch im Frontend definiert sind. Nachfolgend eine Liste der Operationen (bzw. Terminalsymbole), die in unserem Compiler von Olive++ verarbeitet werden:

```
cs_PLUS    cs_MINUS   cs_MULT    cs_DIV     cs_MOD     cs_SHL     cs_SHR
cs_CAST    cs_LEQ     cs_GEQ     cs_EQ      cs_NEQ     cs_AND     cs_OR
cs_NOT     cs_LOGNOT  cs_UPLUS   cs_UMINUS  cs_ADDR    cs_LOAD    cs_STORE
cs_WRITE   cs_LABEL   cs_CALL    cs_JUMP    cs_CJUMP   cs_RETURN  cs_NOARG
cs_XOR     cs_LESS    cs_READ    cs_PASSARG cs_READARG          cs_WRITEARG
cs_INTCONST           cs_FLOATCONST         cs_GLOBALSYM       cs_GREATER
cs_STRUCTCOPY         cs_VOIDRETURN
```

Da die einzelnen Operationen, die von diesen Terminalsymbolen dargestellt werden, fast alle selbsterklärend sind, wird an dieser Stelle nicht weiter darauf eingegangen. Es sollte jedoch noch erwähnt werden, dass die Anzahl der Argumente für diese Operationen natürlich auch festgelegt wurden. Unter anderem aus diesen Symbolen konstruiert man dann eine Grammatik für Olive++, die alle möglichen Eingabekombinationen dieser Operationen in Form von Bäumen überdecken kann.

A.3.3 LLIR

Die LLIR (Low Level IR)[Inf] ist eine Datenstruktur für eine Zwischendarstellung, die das gesamte in Assembler dargestellte Programm umfasst. Im Gegensatz zur LANCE-IR sind die einzelnen Instruktionen hier jedoch bereits in maschinenabhängigem Code vorhanden. Hierzu enthält die LLIR eine Reihe von Klassen, die eine solche Struktur ermöglichen.

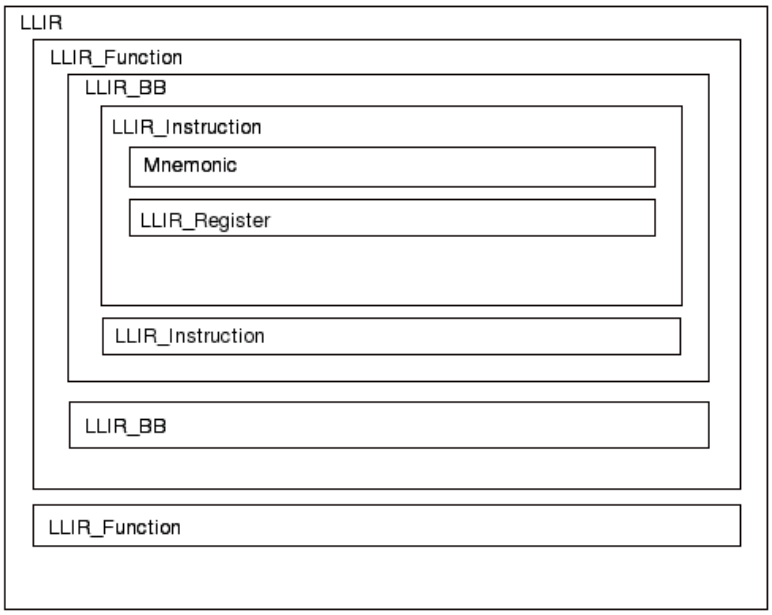


Abbildung A.13: Klassendiagramm der LLIR

Die LLIR ist selbst eine derartige Klasse (s. Abb. A.13), die auf diverse weitere Klassen und einige vom Benutzer zu liefernde Daten zurückgreift. Sie besteht aus einer Menge von LLIR-Funktionen, die namentlich in der Regel denen des C-Sourcefiles entsprechen. Diese Funktionen beinhalten die bereits aus dem Frontend bekannten Basisblöcke (hier: LLIR_BB), welche wiederum aus Listen von Befehlen (LLIR_Instruction) zusammengesetzt sind. Alle Instanzen dieser Klassen enthalten jeweils Verweise auf ihre unmittelbaren Vorgänger und Nachfolger, was einen Durchlauf durch das Programm, eine seiner Funktionen oder deren Basisblöcke vereinfacht. Desweiteren enthält die LLIR Klassen für physikalische und virtuelle Register sowie u. a. eine eigenständige Lifetime-Analyse.

A.3.4 Compiler HOW-TO

Um den Compiler nutzen zu können, gilt es, erst einige Voraussetzungen zu erfüllen:

1. Der Compiler liegt bisher nur in einer Version für Linux vor, welche auf Debian- und SUSE-Linux getestet wurde.

2. Alle Binaries müssen vorhanden sein. Dazu gehört das Lance2-Frontend (mit u. a. `compile` und `iropt`), sowie das Backend (insbesondere `picback` und `asmparser`).
3. Die Umgebungsvariablen müssen korrekt gesetzt sein:
   - `LANCE2_CPP="gcc -E"`
   - `LANCE2_CONFIG="/PFAD_ZUR_/config.pic"`
   - `PEEP_CONF="/PFAD_ZUR_picpeep.dat/"` (nur Pfadangabe!)
   - Die `PATH`-Variable muss ergänzt werden um die Pfade zu den Binaries.

   Der einfachste Weg diese Änderungen durchzuführen besteht im Anpassen des `env_conf.sh`-Bash-Skriptes und dessen Ausführung durch `source env_conf.sh` .
4. Man wechselt in das Verzeichnis, in dem sich die zu kompilierende C-Datei (hier als `myapp.c` bezeichnet) befindet.
5. Der Compiler kann jetzt mit dem Befehl `piccc myapp.c` (piccc steht für PIC-C-Compiler) verwendet werden.

Der Befehl `piccc myapp.c` bietet noch die Möglichkeit, als Option `'-b'` und `'-s'` anzugeben. Dabei bewirkt die Option `'-b'`, dass im Code, anstelle der Accessbank, das Bank-Select-Register verwendet wird, während `'-s'` in jede Funktion ein `DB0xFFFF` als Pseudo-NOP einfügt, um einem Hardware-Bug in älteren PIC18-Prozessoren entgegen zu wirken.

Sollte das `piccc`-Skript nicht wie erwartet laufen, kann die Befehlskette zum Kompilieren auch von Hand eingegeben werden:

1. `compile myapp.c` (Lance2-Aufruf um die Intermediate-Representation (IR, `myapp.ir.c`) zu erstellen)
2. `iropt myapp.ir.c` (Optimierungsskript für die IR)
3. `picback myapp.ir.c [-b] [-s]` (Backend mit Code-Generator, Registerallokation und Peephole-Optimierungen)
4. `asmparser -o myapp.asm [-b 1]` (Parser, um benötigte Header zu ergänzen)

Sobald der Compiler erfolgreich terminiert, sollte neben der ursprünglichen `myapp.c` eine Datei mit der Endung `.asm` (hier `myapp.asm`) existieren, in der – im Windows kompatiblen Text-Format (wird vom MPLAB benötigt) – der generierte Assembler-Code zu finden ist. Zum Debugging des Compilers werden außerdem noch der Peephole-Tree (`peephole.vcg`) und der Interferenzgraph der Registerallokation (`new.vcg`) erstellt, die man mit Hilfe von `xvcg` betrachten kann. Auch die IR (`myapp.ir.c`) und der nach Funktionen sortierte C-Code (`myapp.pp`) werden erzeugt, sind aber eigentlich – wie auch die Graphen – von keinerlei Interesse für den Endanwender.

Anhang B

Softwarelizenzen

B.1 Allgemeines zu Softwarelizenzen

Viele Softwarepakete gelangen nur in binärer Form zum Anwender, also in einem Format, welches die Zielarchitektur direkt ausführen kann. Dieser Zustand wird auch als closed source bezeichnet, weil der Quelltext dem Anwender verschlossenen bleibt.

Bei Opensource-Software sind die Quelltexte dem Anwender zugänglich (und dazu alle Werkzeuge, um sie ins Binärformat zu überführen). Solche Programme können damit ohne Probleme an eigene Bedürfnisse angepasst werden. Insbesondere ist es einfach, sie als Basis für eigene Projekte zu nutzen. Ein solches Projekt wird als derived (engl. abgeleitet) bezeichnet, dies kann sowohl durch Linken einer fremden Bibliothek als auch durch sonstiges Einschließen von Code geschehen. Da die PG 420 ROCK ein Robot Construction Kit erstellen will, sind solche abgeleiteten Arbeiten ausdrücklich erwünscht und willkommen.

Eine Softwarelizenz ist zunächst nichts anderes als ein Vertrag, in dem eine Partei (Lizenzgeber) einer anderen (Lizenznehmer) bestimmte Nutzungsrechte an einer urheberrechtlich geschützten Software überlässt oder einschränkt [Ber]. Diese Rechte können beinhalten:

- Allgemeines Nutzungsrecht an der Software, Zahl der Nutzer, Art der Nutzung usw.
- Recht auf Weiterverbreitung der Software
- Recht auf Veränderung der Binärdateien
- Recht auf Veränderung des Quellcodes, sofern vorhanden
- Recht auf Weiterverbreitung veränderter Versionen der Binärdateien oder des Quellcodes
- Recht auf Verbindung (Linking) der Binärdateien oder des Quellcodes mit anderer Software

B.2 Die General Public License

Die General Public License [FSF91] ist eine einheitliche Lizenz für Opensource- Projekte, die folgende Freiheiten schützen soll:

- Freiheit, das Programm für jeden Zweck auszuführen
- Freiheit, den Quellcode zu studieren und anzupassen

- Freiheit, das Programm zu kopieren
- Freiheit, das veränderte Programm zu kopieren
- Die Haftung für entstehende Schäden wird ausgeschlossen.
- Jedes Derivat muss ebenfalls vollständig unter der GPL lizenziert werden
- Bei Weitergabe des Programmes in Binärform muss der Quelltext des gesamten Programms mitgeliefert oder auf Anfrage ausgehändigt werden

Heutige Linux-Systeme beruhen zu großen Teilen auf Software unter der GPL. Schon aus dieser Tradition heraus verwenden viele neue Opensource-Projekte die GPL. Die GPL garantiert, dass freie Software stets freie Software bleibt

Abbildung B.1 zeigt die Verteilung der gebräuchlichsten Opensource-Lizenzen auf Sourceforge. [OSD] Sourceforge bietet Opensource-Softwareprojekten eine kostenfreie, web-orientierte Entwicklungsumgebung und nimmt auf diesem Gebiet die Stellung eines Marktführers ein, neben Savannah und Berlios.

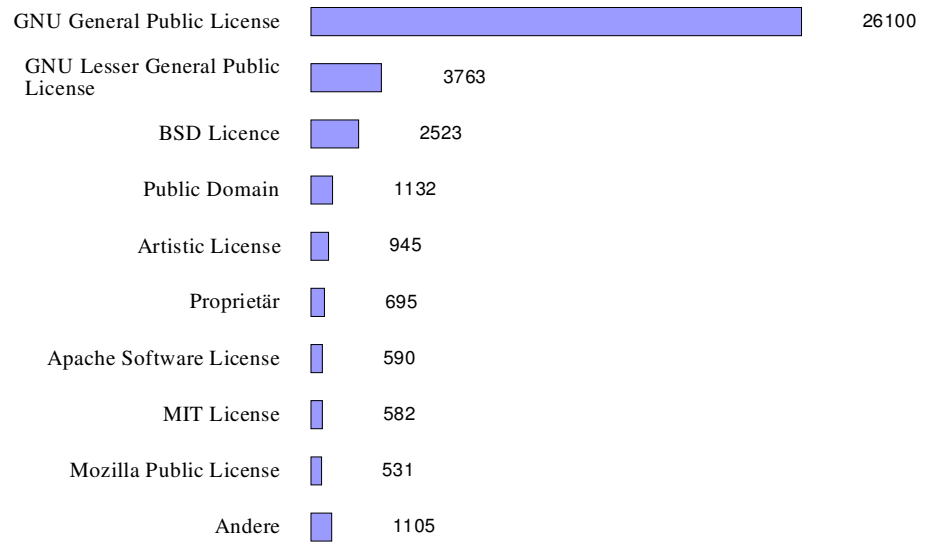


Abbildung B.1: Verteilung von Softwarelizenzen und Projekten auf Sourceforge [OSD]

Die Verwendung einer solchen, gut erprobten Lizenz bietet Sicherheit für den Entwickler und ist dazu anwenderfreundlich, weil der Benutzer in groben Zügen weiß, wie er die Software benutzen darf.

Die PG ROCK hätte gerne auf die GPL zurückgegriffen, konnte dies aber Aufgrund der Lizenzen einiger verwendeten Werkzeuge nicht tun. Die problematischen Klauseln werden im Folgenden erläutert.

Auch der Plan, die GPL zu modifizieren, musste verworfen werden, da sie genau dieses verbietet, was zur Schaffung einer eigenen ROCK-Lizenz führte, die sich philosophisch stark an der GPL und der BSD-Lizenz orientiert.

B.3 GPL-verhindernde Klauseln

Einschränkungen beim Bootloader

Der verwendete Bootloader basiert zum großen Teil auf der Application-Note AN851 [Mica] mit unseren Erweiterungen. Folgende zusätzliche Forderungen sind also zu beachten:

> The software supplied herewith by Microchip Technology Incorporated (the Company) is intended and supplied to you, the Company's customer, for use solely and exclusively on Microchip products.
> The software is owned by the Company and/or its supplier, and is protected under applicable copyright laws.
> All rights are reserved. Any use in violation of the foregoing restrictions may subject the user to criminal sanctions under applicable laws, as well as to civil liability for the breach of the terms and conditions of this license.
> THIS SOFTWARE IS PROVIDED IN AN AS IS CONDITION. NO WARRANTIES, WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE APPLY TO THIS SOFTWARE. THE COMPANY SHALL NOT, IN ANY CIRCUMSTANCES, BE LIABLE FOR SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES, FOR ANY REASON WHATSOEVER.

Einschränkungen bei Schaltplänen

CADSOFT macht folgende Aussagen über EAGLE:

> Educational License: Das Programm darf ausschließlich in Ausbildungsstätten wie Schulen, Universitäten oder Lehrwerkstätten zu Ausbildungszwecken verwendet werden.
>
> Student License: Das Programm darf ausschließlich für private Zwecke verwendet werden. Jede kommerzielle Anwendung ist untersagt. Es stellt eine Verletzung der Lizenzbedingungen dar, wenn Sie durch Gebrauch einer Studentenversion Geld verdienen.

LANCE, OLIVE++ sind Closed Source

Vom Lehrstuhl Informatik XII wurde uns mitgeteilt, dass es sich bei LANCE, OLIVE++ und LLIR um Closed-Source-Software handelt. Daher werden wir nur das Binärfile des Compilers weiterverbreiten.

B.4 Die RoCK-Lizenz

```
                        Die RoCK-Lizenz
                        ***************

          Detlev Bartsch, Torsten Denno, Pedram Hadjian,
        Nico Karnatz, Andre Kernchen, Andreas Klapschus,
          Marcus Ladwig, Michael Patzer, Matthias Reck,
    Christoph Schlagbaum, Daniel Smolinski, Thorsten Wilmer

                          Version 1.0
                Copyright (c) 2003, by PG RoCK (Die Autoren)
                          Alle Rechte vorbehalten


Vorwort

  Dieses Werk umfasst Hardware, Software und Dokumentation. Es wurde
von den oben genannten Autoren im Rahmen der Projektgruppe 420 Robot
```

Construction Kit erstellt.

Voraussetzungen und Bedingungen

Die Weiterverbreitung und Nutzung in Quelltext- oder Binärform mit oder ohne Modifizierungen wird hiermit unter der Bedingung gestattet, dass die folgenden Voraussetzungen eingehalten werden:

- Die Verbreitung in jeglicher Form, ob modifiziert oder unverändert darf nur unentgeldlich erfolgen. Porto und Verpackung, sowie Kosten für Speichermedien oder Übertragungskapazitaet sind hiervon ausgenommen.

- Jede verbreitete Version muss diese Lizenz enthalten.

- Dieses Werk ist nicht sublizenzierbar. Abgeleitete Arbeiten stehen auch unter der RoCK-Lizenz.

- Änderungen an den Quellen müssen mit verteilt werden.

- Der Bootloader des Betriebssystems SOUL darf nur auf von Microchip hergestellten Mikrokontrollern verwendet werden.

- Der Compiler darf nicht verändert oder reverse engineered werden und liegt nicht im Quelltext vor.

Haftungsausschluss

Da dieses Werk ohne jegliche Kosten lizenziert wird, besteht keinerlei Gewährleistung für das Werk, soweit dies gesetzlich zulässig ist. Sofern nicht anderweitig schriftlich bestätigt, stellen die Copyright-Inhaber und oder Dritte das Werk so zur Verfügung, wie es ist’’, ohne irgendeine Gewährleistung, weder ausdrücklich noch implizit, einschließlich - aber nicht begrenzt auf - Marktreife oder Verwendbarkeit für einen bestimmten Zweck. Das volle Risiko bezüglich Qualität und Leistungsfähigkeit des Programms liegt bei Ihnen. Sollte sich das Werk als fehlerhaft herausstellen, liegen die Kosten für notwendigen Service, Reparatur oder Korrektur bei Ihnen.

In keinem Fall, außer wenn durch geltendes Recht gefordert oder schriftlich zugesichert, ist irgendein Copyright-Inhaber oder irgendein Dritter, der das Werk wie oben erlaubt modifiziert oder verbreitet hat, Ihnen gegenüber für irgendwelche Schäden haftbar, einschließlich jeglicher allgemeiner oder spezieller Schäden, Schäden durch Seiteneffekte (Nebenwirkungen) oder Folgeschäden, die aus der Benutzung des Werkes oder der Unbenutzbarkeit des Programms folgen (einschließlich - aber nicht beschränkt auf - Datenverluste, fehlerhafte Verarbeitung von Daten, Verluste, die von Ihnen oder anderen getragen werden müssen, oder dem Unvermögen des Werkes, mit irgendeinem anderen Programm oder Hardware zusammenzuarbeiten), selbst wenn ein Copyright-Inhaber oder Dritter über die Möglichkeit solcher Schäden unterrichtet worden war.

Anhang C

Die Archiv-CD

Da wir mit ROCK ein leistungsfähiges Robot Construction Kit geschaffen haben, soll es natürlich nicht bei der Drop-Zone-Mission bleiben. Im Gegenteil – uns würde es freuen, wenn andere interessierte Menschen von unserer Arbeit profitieren würden!

Darum stellen wir auf unserer Homepage die gesamten Schaltpläne, Quellen (den Compiler leider nicht, vergleiche Anhang B.3), Seminararbeiten (Folien), Poster, Datenblätter, Howtos und Fotos als CD-Image (ISO 9660) zum Download bereit.

Quellen sind mit `tar` und `bzip2` gepackt als Archive vorhanden und können mit

```
tar xfvj beispiel.tar.bz2
```

entpackt werden.

Der Film, der unseren Roboter beim Lösen der Drop-Zone-Mission zeigt, kann gesondert bezogen werden.

Die Adresse unserer Homepage lautet: `http://ls12.cs.uni-dortmund.de/rock`.

Anhang D

Schaltpläne

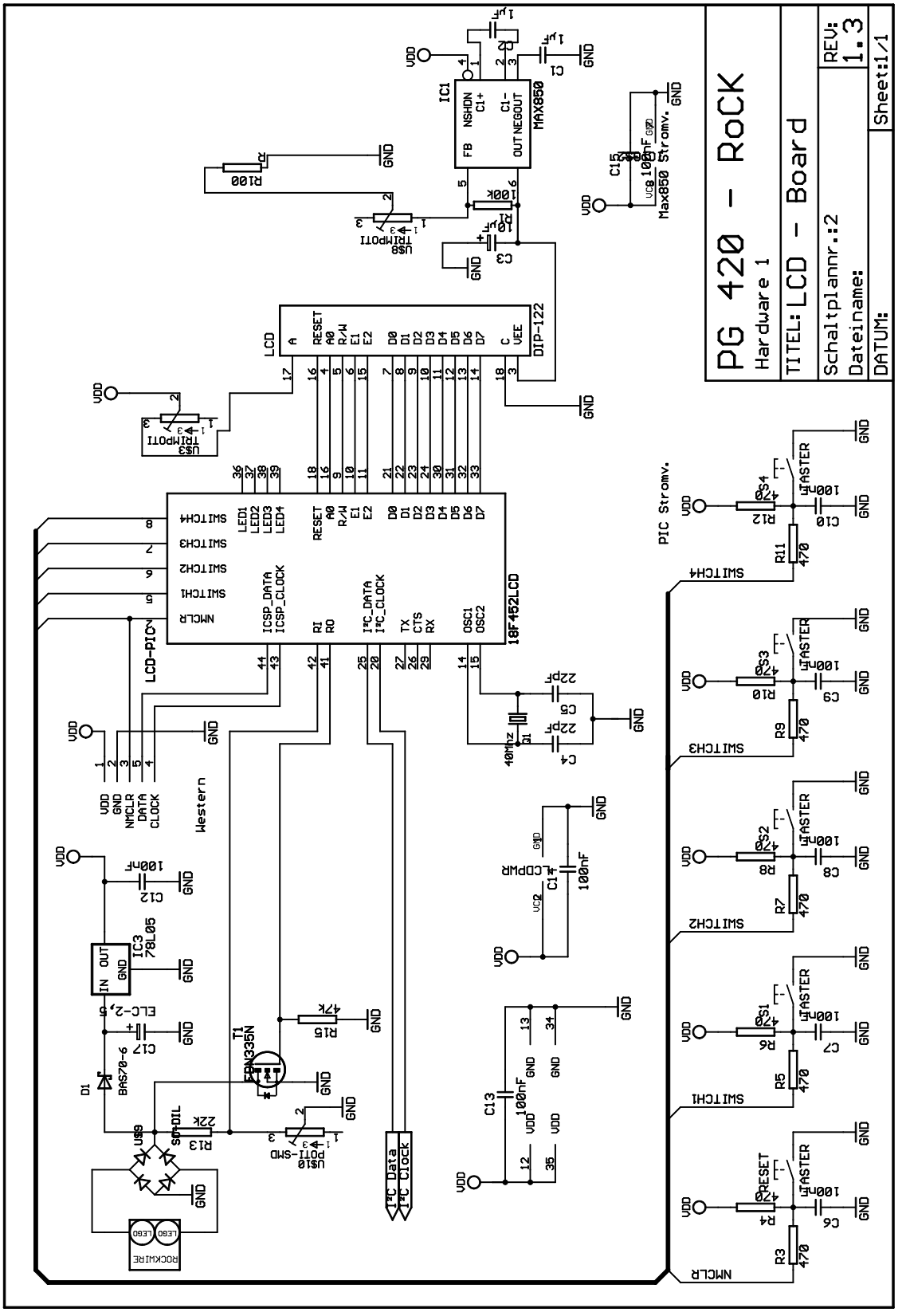


Abbildung D.1: Der Schaltplan des LCD-Boards

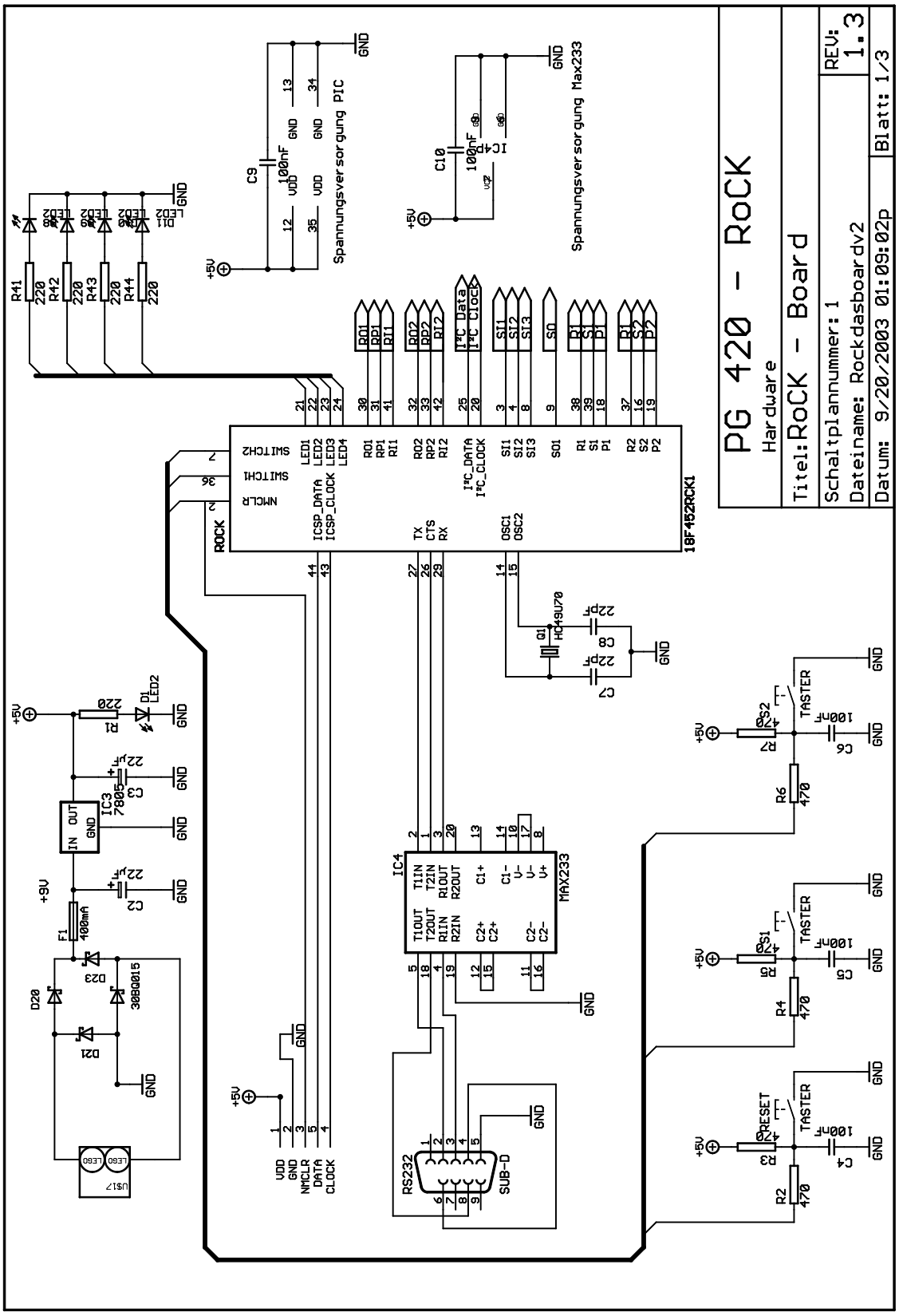


Abbildung D.2: Der Schaltplan des RoCK-Boards - Blatt 1

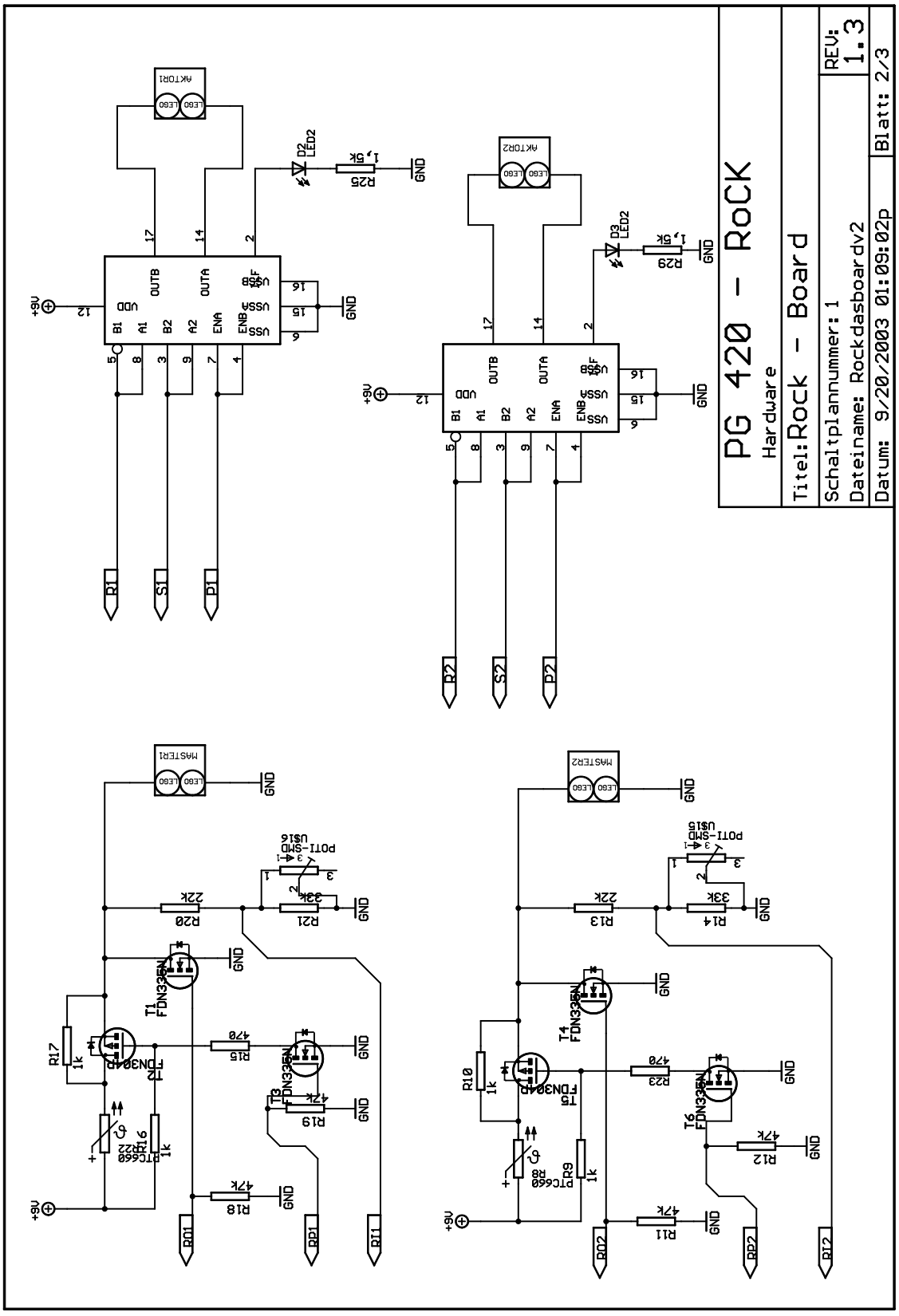


Abbildung D.3: Der Schaltplan des RoCK-Boards - Blatt 2

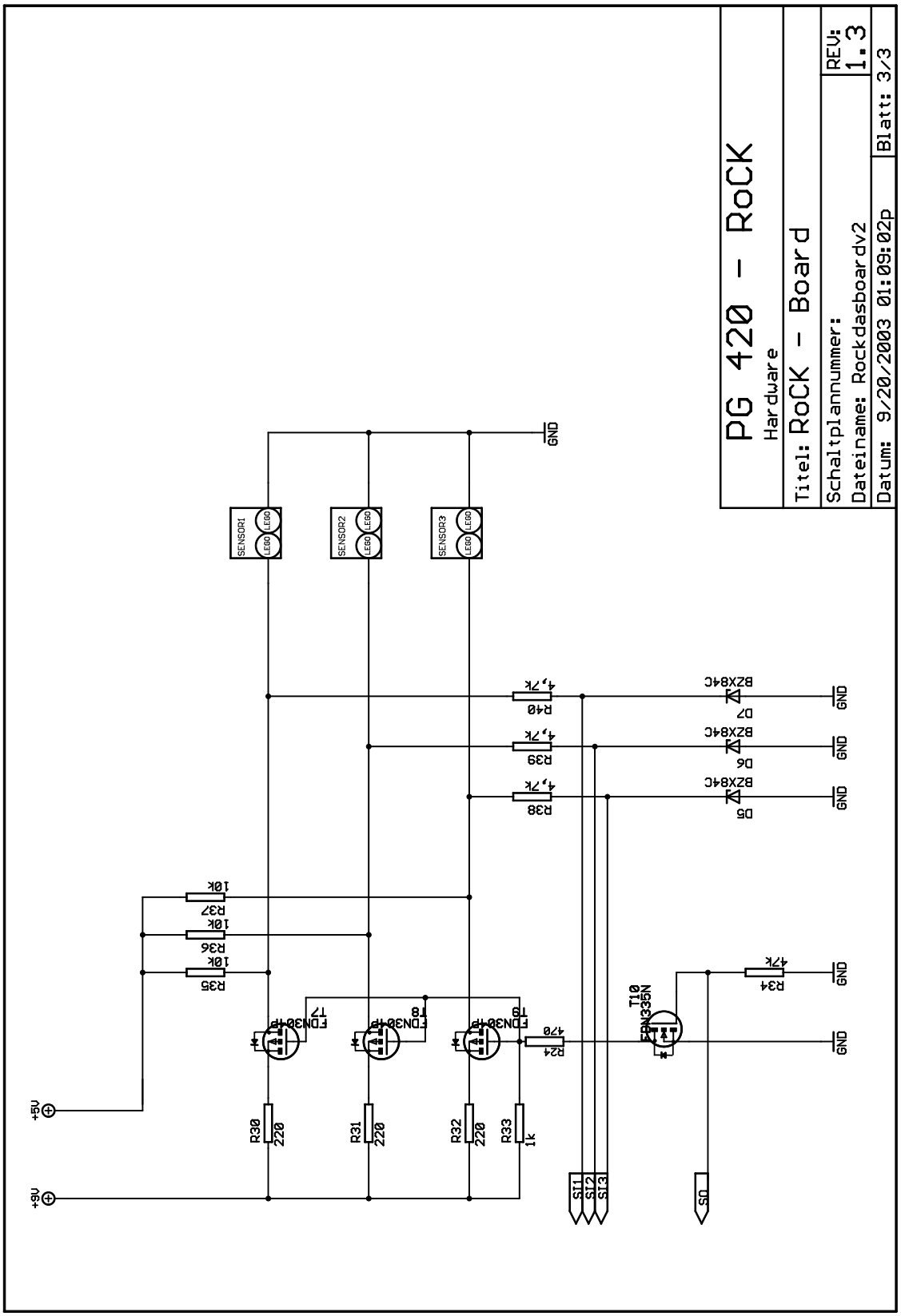


Abbildung D.4: Der Schaltplan des RoCK-Boards - Blatt 3

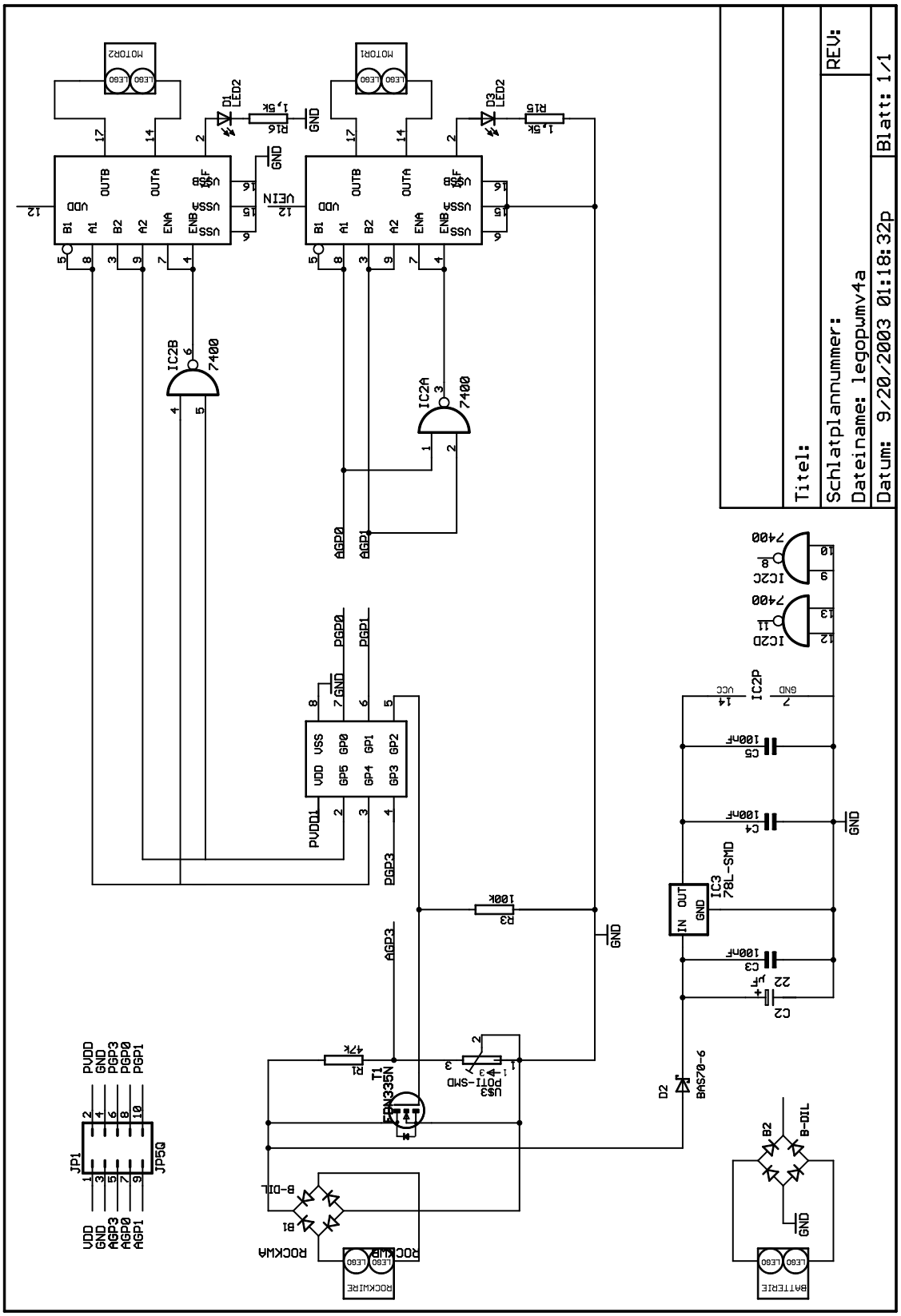


Abbildung D.5: Der Schaltplan des Motor Hub

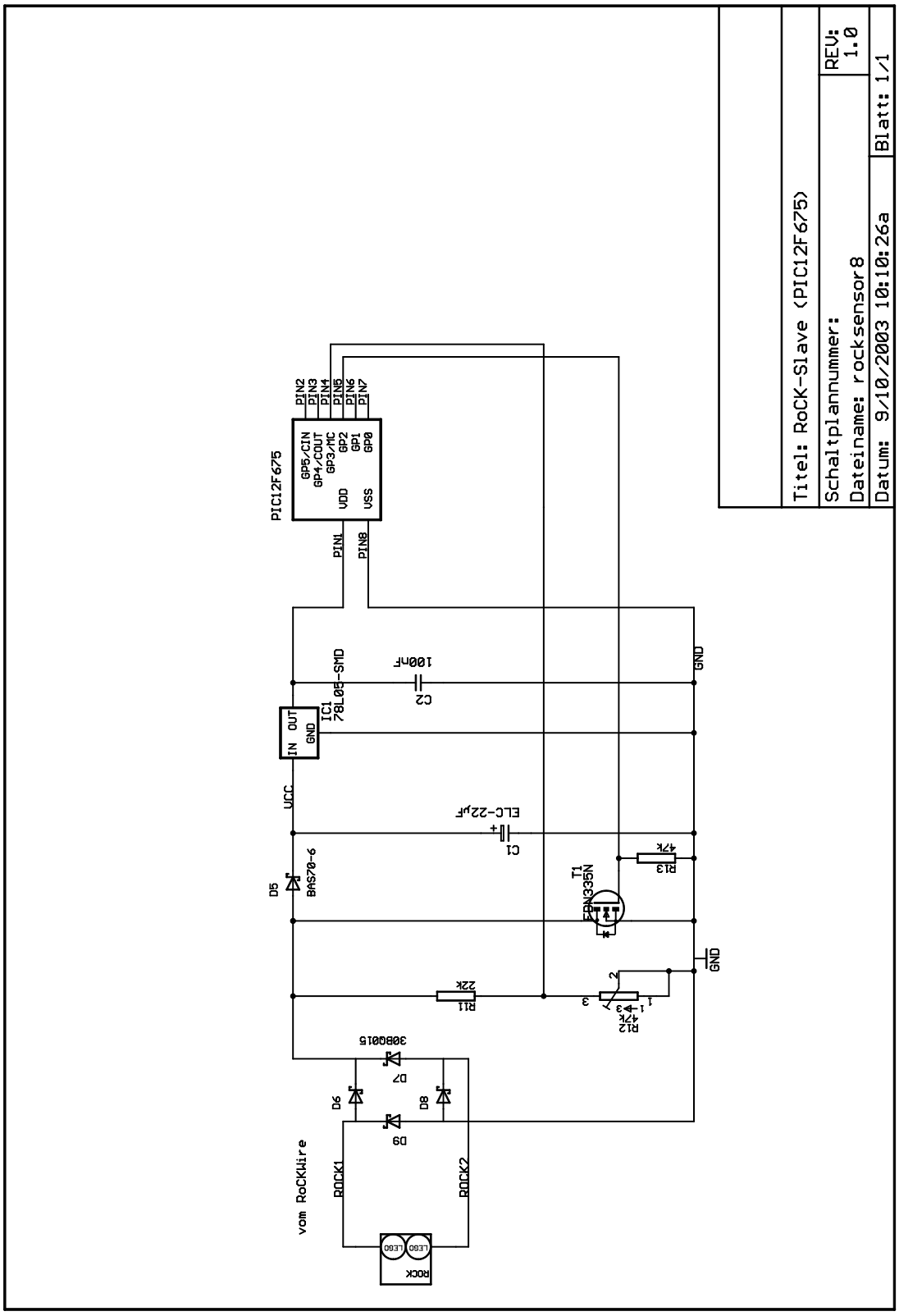


Abbildung D.6: ROCKWIRE-Slave

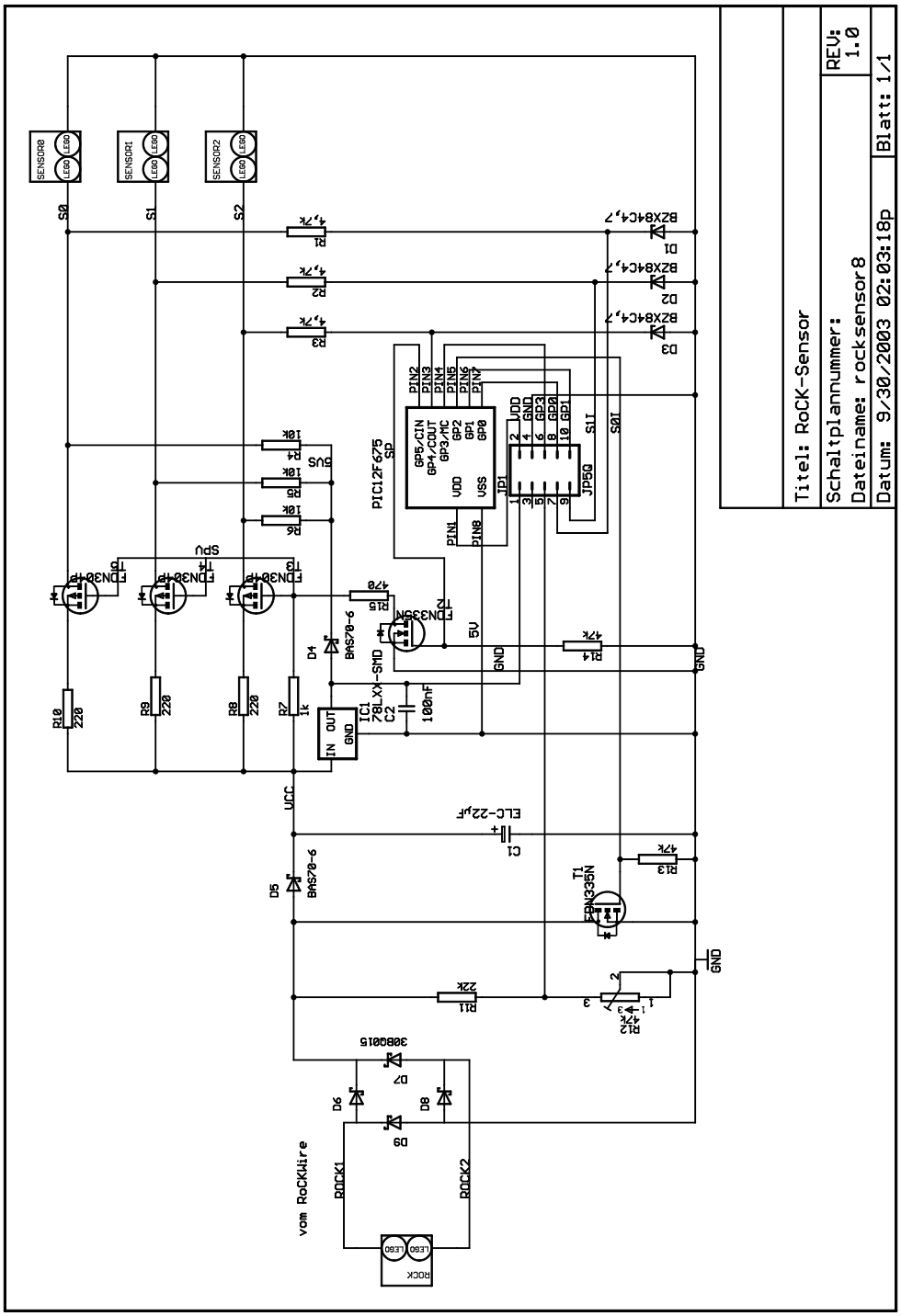


Abbildung D.7: Der Schaltplan des Sensor Hub mit D4. D4 wurde auf den Platinen entfernt.

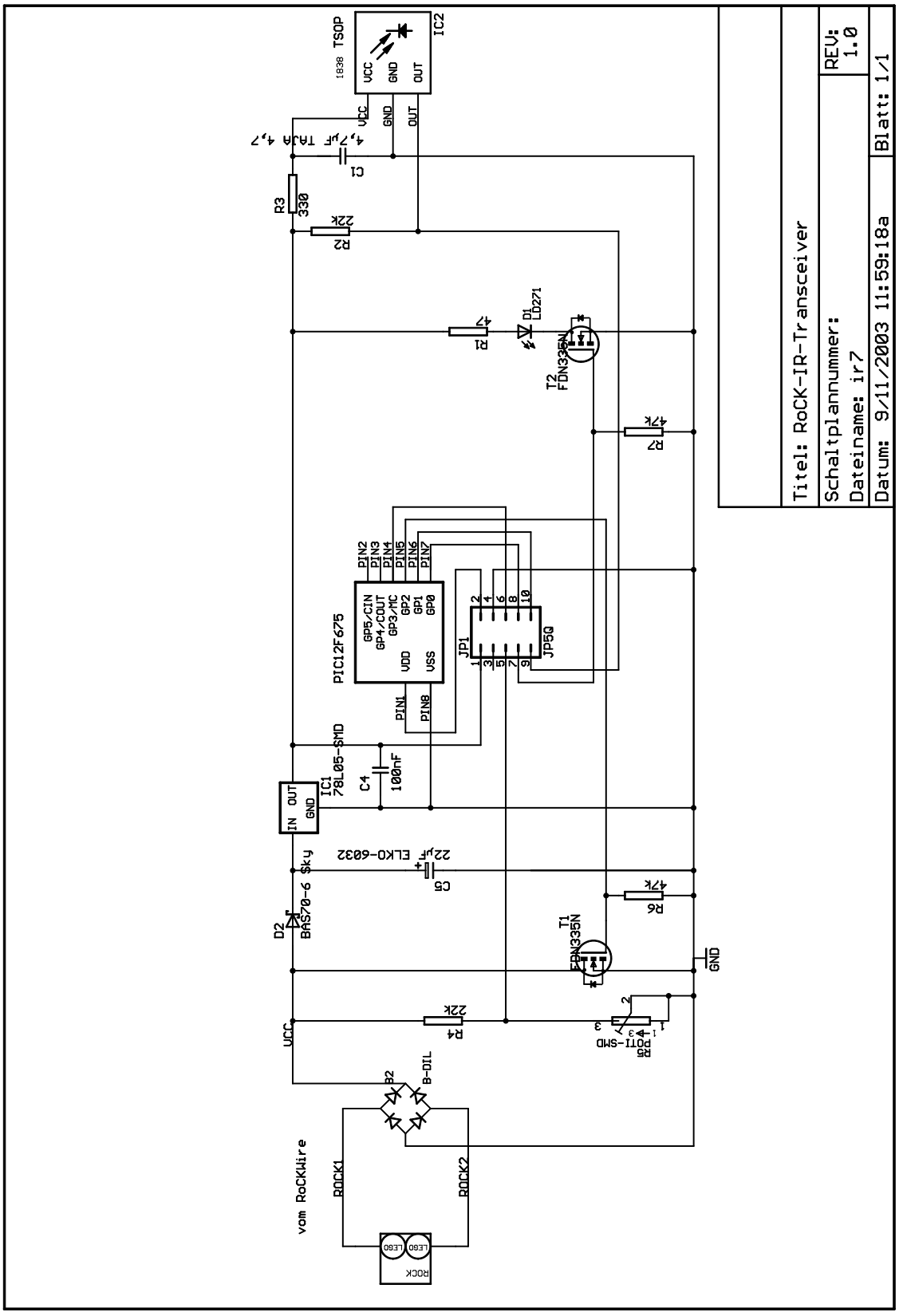


Abbildung D.8: Der Schaltplan des IR-Moduls

Anhang E

Platinenlayout

E.1 Das Routing der Platinen

Beim Routing handelt es sich um den Entwurf einer Leiterplatte anhand eines Schaltplanes. Zum Entwerfen der Schaltpläne und zum Routing der Platinen wurde der Layout-Editor EAGLE 3.55 [Cad] benutzt.

Es wurde dabei so vorgegangen, dass zunächst eine passende Gehäuseform gewählt wurde. Anschließend wurden die Bauteile möglichst Platz sparend auf beiden Seiten der Platine angeordnet und die anfangs nur als Airwires dargestellten Verbindungen durch Leiterbahnenzüge ersetzt. Die Anordnung der Bauteile und Leiterbahnen hängt voneinander ab, so dass baugruppenweise beides gleichzeitig festgelegt wurde. Bei diesem Prozess müssen die Designvorschriften des Platinenherstellers (Firma MVPCB [M&]) eingehalten werden. Des Weiteren wurden zusätzlich folgende strengere Designregeln eingehalten:

- Abstand zum Rand: 40 mil
- Abstand zwischen benachbarten Leiterzügen: 10 mil
- Leiterbahnbreite für Signale: 10 mil
- Leiterbahnbreite für größere Leistungen, z. B. ROCKWIRE: 20 mil
- Via Durchmesser: 40 mil
- Via Bohrung: 16 mil

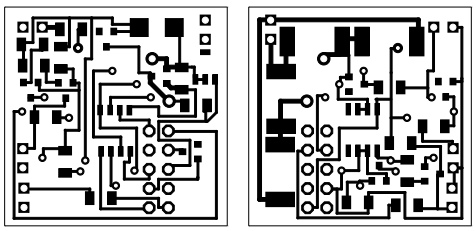


Abbildung E.1: Sensorhub-Platine. Links: top. Rechts: bottom.

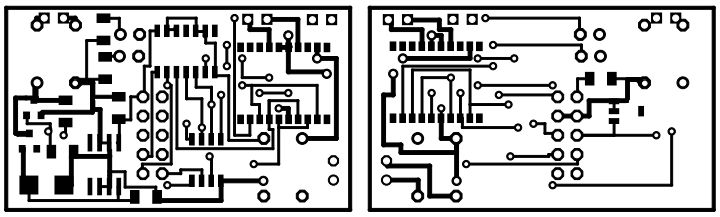


Abbildung E.2: Motorhub-Platine. Links: top. Rechts: bottom.

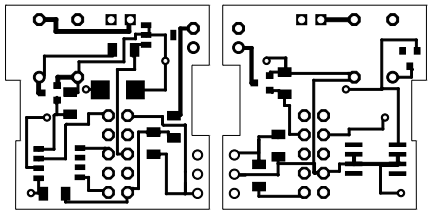


Abbildung E.3: IR-Modul-Platine. Links: top. Rechts: bottom.

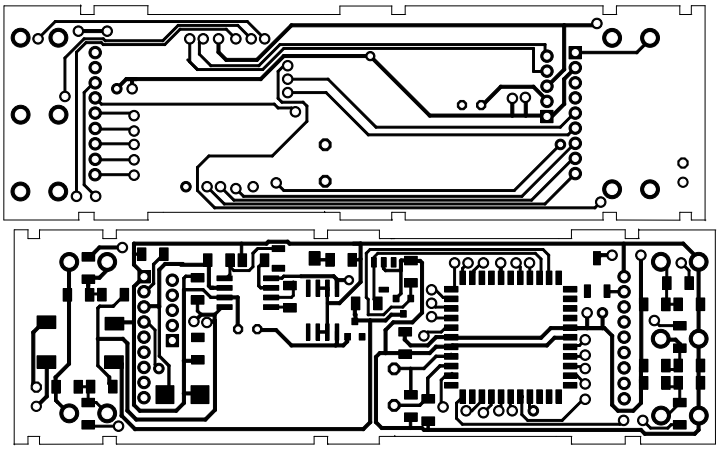


Abbildung E.4: LCD-Board-Platine. Oben: top. Unten: bottom.

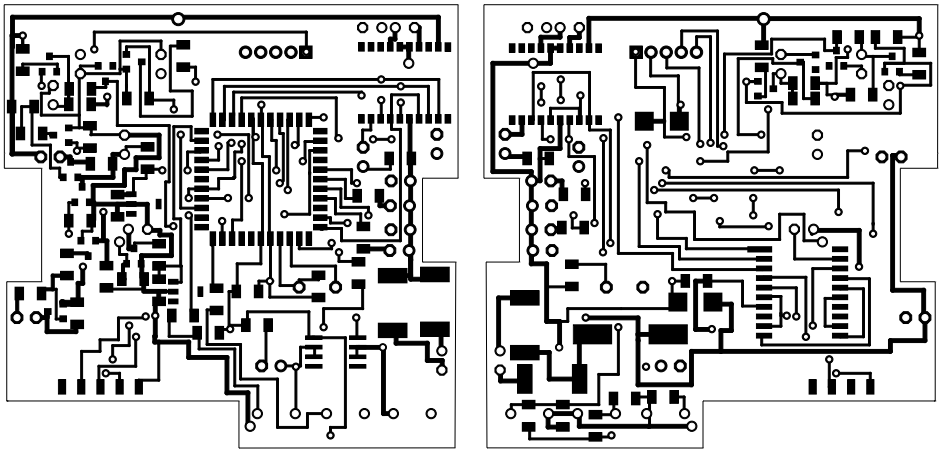


Abbildung E.5: ROCK-Board-Platine. Links: top. Rechts: bottom.

Literaturverzeichnis

[Agi] AGILENT TECHNOLOGIES. *Homepage des Herstellers Agilent Technologies*. http://www.agilent.com

[App98] APPEL, Andrew W.: *Modern Compiler Implementation: In C*. Cambridge University Press, 1998

[AVA86] ALFRED V. AHO, Jeffrey D. U.: *Compilers – Principles, Techniques and Tools*. Addison-Wesley Pub Co, 1986

[Avi] AVIJIT, Kumar. *OLIVE++ Manpage*. http://ls12.cs.uni-dortmund.de/info/olive++.ps

[Bau] BAUM, Dave. *NQC-Informationen*. http://www.baumfamily.org/nqc/

[Ben02] BENNETT, Matthew J. *PIC Based Imaging Sonar*. http://www.hazmat.com/~mjb/projects/picsonar/. 2002

[Ber] BERLIOS. *Openfacts Wissensdatenbank*. http://openfacts.berlios.de/index.phtml?title=Open-Source-Lizenzen

[Bra] BRANDL, Michael. *private Homepage eines Legobastlers*. http://insel.heim.at/mainau/330001/lego.htm

[Cad] CADSOFT COMPUTER GMBH. *Homepage von CadSoft*. http://www.cadsoft.de

[Ced] CEDERQVIST, Per. *CVS TexInfo Dokumentation*. http://www.cvshome.org/docs/manual/

[CMA] CMA. *LEGO DACTA Interface (09751)*. http://www.cma.science.uva.nl/english/products/09751.html

[DCP] DCP MICRODEVELOPMENTS LTD. *LEGO Sensor Adapter*. http://www.dcpmicro.com/lego/

[ecl] ECLIPSE.ORG CONSORTIUM. *Homepage von Eclipse*. http://www.eclipse.org

[Elea] ELECTRONIC ASSEMBLY. *EA DIP122-5NLED LCD- GRAFIK MODUL 122x32*. http://www.lcd-module.de/deu/pdf/grafik/dip122-5.pdf

[Eleb] ELECTRONIC ASSEMBLY. *Die Homepage der Firma Electronic Assembly*. http://www.lcd-module.de

[Faia] FAIRCHILD SEMICONDUCTOR CORPORATION. *FDN304P P-Channel 1.8V Specified PowerTrench MOSFET*. http://www.fairchildsemi.com/ds/FD/FDN304P.pdf

[Faib] FAIRCHILD SEMICONDUCTOR CORPORATION. *FDN335N N-Channel 2.5V Specified PowerTrench MOSFET*.
http://www.fairchildsemi.com/ds/FD/FDN335N.pdf

[FSF91] FSF (FREE SOFTWARE FOUNDATION). *General Public License*.
http://www.gnu.org/licenses/gpl.txt. 1991

[Gas] GASPERI, Michael. *MindStorms RCX Sensor Input Page*.
http://www.plazaearth.com/usr/gasperi/lego.htm

[Gig] GIGUERE, Eric. *ANSI-C-Standard-Überblick*.
http://www.et.brad.ac.uk/help/.packlangtool/.langs/.c/.ansi.html

[GL02] GOETZ, Thomas ; LANZ, Rolf. *Skript zu Betriebssystem-Technologie*.
http://www.hta-be.bfh.ch/~lanz/SKRIPTE/BsTech.pdf. Juni 2002

[Hee] VAN HEESCH, Dimitri. *Homepage von Doxygen*.
http://www.stack.nl/~dimitri/doxygen/

[Hur] HURBAN, Phillippe. *Seite zum Aufbau des Rotationssensors*.
http://philohome.free.fr/sensors/legorot.htm

[Inf] INFORMATIK CENTRUM DORTMUND. *Homepage Informatik Centrum Dortmund Abteilung ES*.
http://www.icd.de/es

[Kam92] KAMMERER: *Elektronik III Baugruppen der Mikroelektronik*. Pflaum Verlag München, 1992

[Knu99a] KNUDSEN, Jonathan B.: *The Unofficial Guide to LEGO MINDSTORMS*. O'Reilly, 1999

[Knu99b] KNUDSON, Jonathan B.: *The Unofficial Guide to Lego Mindstorms Robots*. O´REILLY, 1999

[Leu] LEUPERS, Rainer. *LANCE - Retargetable C compiler*.
http://ls12-www.cs.uni-dortmund.de/lance/

[Leu97] LEUPERS, Rainer: *Retargetable Code Generation for Digital Signal Processors*. Kluwer Academic Publishers, 1997

[Lin] LINEBERRY, Bob. *Hitachi HD44780 Datenblatt*.
http://ls12.cs.uni-dortmund.de/rock/info/hd44780.pdf

[Liv] LIVICK, Gary. *SRF08 ultra sonic range finder*.
http://www.robot-electronics.co.uk/htm/srf08tech.shtml

[M&] M&V LEITERPLATTEN VERTRIEBS GMBH. *M&V Leiterplatten*.
http://www.mvpcb.de/

[Mar] MARTIN, Fred. *Homepage von Fred Martin*.
http://web.media.mit.edu/~fredm/mindstorms/

[Max] MAXIM INTEGRATED PRODUCTS. *MAX850, MAX851, MAX852, MAX853 Low-Noise, Regulated, Negative Charge Pump Power Supplies for GaAsFET Bias*.
http://pdfserv.maxim-ic.com/en/ds/MAX850-MAX853.pdf

[Mica] MICROCHIP. *Application Note 851*.
http://www.microchip.com/1010/suppdoc/appnote/all/an851/

[Micb] MICROCHIP. *Microchip ICD 2 Webseiten*.
http://www.microchip.com/1000/pline/tools/picmicro/icds/icd2/detailed/index.htm

[Micc] MICROCHIP. *Microchip PICDEM 2 Plus Webseiten*.
http://www.microchip.com/1000/pline/tools/picmicro/icds/icd2/cupola/pcdmplus/index.htm

[Micd] MICROCHIP. *Microchip Webseiten*.
http://www.microchip.com

[Mic97] MICROCHIP: *PICmicro Mid-Range MCU Family Reference Manual*. 1. 1997

[Mic02a] MICROCHIP: *PIC12F629/675 Data Sheet*. 1. 2002

[Mic02b] MICROCHIP: *PIC18F4XX2 Data Sheet*. 1. 2002

[MIP] MAXIM INTEGRATED PRODUCTS, Inc. *Maxim 1-Wire and iButton: Communications components for identification, sensor, control, and memory functions.*
http://www.maxim-ic.com/1-Wire.cfm

[MITa] MIT. *Homepage des MIT Projektes: Cricket*.
http://llk.media.mit.edu/projects/cricket/

[MITb] MIT. *Homepage des MIT Projektes: Lifelong Kindergarten*.
http://llk.media.mit.edu/projects/

[MITc] MIT. *Homepage des MIT Projektes: Programmable Brick*.
http://lcs.www.media.mit.edu/groups/el/projects/programmable-brick/

[NAS] NASA. *Mars Pathfinder Home*.
http://mars.jpl.nasa.gov/MPF/index1.html

[Nel] NELSON, Russel. *LEGO MINDSTORMS Internals*.
http://www.crynwr.com/lego-robotics/

[Nog] NOGA, Markus L. *Offizielle Homepage von LegOS*.
http://www.noga.de/legOS/

[OSD] OSDN (OPEN SOURCE DEVELOPMENT NETWORK). *Sourceforge*.
http://www.sourceforge.net

[PG4] PG420. *Homepage der PG420-RoCK*.
http://ls12-www.cs.uni-dortmund.de/RoCK/info/

[Proa] PROJEKTGRUPPE MPLAY3. *Homepage MPlay3*.
http://ls12.cs.uni-dortmund.de/mplay3/

[Prob] PROUDFOOT, Kekoa. *RCX-Internals*.
http://graphics.stanford.edu/~kekoa/rcx/

[RE] RAUSCH, Martin ; EXLER, Roland. *PTC (Positive Temperature Coefficient, Kaltleiter)*.
http://www.emt.uni-linz.ac.at/roland/messtech_pr/node26.html

[ROB] ROBOBULL, Project. *Homebrew Sensors for the RCX*.
http://www.restena.lu/convict/Jeunes/RoboticsIntro.htm

[San] SANDER, Georg. *VCG Overview*.
http://www.cs.uni-sb.de/RW/users/sander/html/gsvcg1.html

[sem] INTERSIL SEMICONDUCTORS. *datashet HIP4020*.
http://www.intersil.com

[Sie] SIEMENS. *LD 271, LD 271 H LD 271 L, LD 271 HL GaAs-IR-Lumineszenzdiode*. http://www.villasophia.com/Rick/r4project/r4outline/documents/Docs/ld271.pdf

[SKW] SLOTHOUBER, Frans ; KETUNNEN, Petteri ; VAN WEERT, Jacco. *Homepage von Robodoc*. http://www.xs4all.nl/~rfsber/Robo/

[SLPRS] SPANIOL, Otto ; LINNHOFF-POPIEN, Claudia ; REICHL, Peter ; SCHUBA, Marco. *Skript zu Systemprogrammierung*. http://www2.s-inf.de/Skripte/SysPro.1997-WS-Spaniol.(i6).Skript.pdf

[Spr] SPRUT. *Die Homepage eines Hobby-Elektronikers*. http://www.Sprut.de

[SR97] SPAM RESEARCH, Group: SPAM Compiler User's Manual. 1997. – Forschungsbericht. http://www.ee.princeton.edu/spam

[Str] VON STRASSBURG, Delbert. *TkCVS Homepage*. http://www.twobarleycorns.net/tkcvs.html

[Vis] VISHAY. *TSOP 18xx family datasheet*. http://www.vishay.com/document/82047/82047.pdf

[Wiz02] WIZARD, Brat. *PIC 16C84 Ultrasonic Ranging System Software*. http://www.wizard.org/sonarcode.html. 2002

Abbildungsverzeichnis

Tabellenverzeichnis

Index