Objektorientierte Softwareentwicklung für eingebettete Systeme

Diplomarbeit

Reinhard Hanselmann

Fachhochschule Ulm

Fachbereich Technische Informatik

1996

Danksagung

Meinem Betreuer, Herrn Prof. Dr.-Ing. Bernardo Wagner, möchte ich für seine beratende Hilfe herzlich danken. Er hat es mir ermöglicht, die interessant kombinierte Aufgabenstellung aus Hard- und Software an einem so anschaulichen und konkreten Anwendungsbeispiel durchzuführen. Die selbständige Arbeitsweise unter seiner Betreuung hat mir dabei sehr entsprochen.

Ulm, im Juni 1996 Reinhard Hanselmann

Inhalt

Danksagung 2

Inhalt 3

1 Einführung 7

2 Objektorientierte Softwareentwicklung 10

2.1 Klassifizierung objektorientierter Software 10

2.2 Methodik des objektorientierten Softwareentwurfs 11

2.2.1 Objektorientierte Analyse 12

2.2.2 Objektorientiertes Design 12

2.2.3 Objektorientierte Programmierung 13

2.2.4 Objektorientierte Softwareentwicklung nach Booch 13

2.3 Objektorientierung und Nebenläufigkeit 14

2.3.1 Prozeßstrukturen 14

2.3.2 Entwurfsmethoden 15

2.3.3 Objektkommunikation und -synchronisation 16

3 Objektorientierte Softwareentwicklung anhand eines Mini-Roboters 17

3.1 Anwendungsbeispiel 17

3.1.1 Roboter „Rug Warrior“ 17

3.1.2 Mikrocontroller 19

3.1.3 Aufgabenstellung 20

3.2 Analyse 20

3.3 Design 25

3.3.1 Nebenläufigkeit 25

3.3.2 Modellierungsentscheidungen 30

3.4 Programmierung 33

3.4.1 Programmiersprachen 33

3.4.2 Verhalten 34

3.4.3 Ressourcenkontrolle 35

3.4.4 Motorensteuerung 36

3.4.5 Geradenkontrolle 36

3.4.6 Tonerkennung 38

3.4.7 Scheduler 48

3.4.8 Mutex 49

3.4.9 Objekterzeugung ... 49
3.5 Simulation ... 50
3.6 Gesamtmodell ... 52

4 IC-Programm ... 55

5 Ergebnisse ... 56
5.1 Entwicklungswerkzeuge ... 56
5.2 Durchführung ... 56
5.3 Verständlichkeit ... 57
5.4 Flexibilität ... 57
5.5 Wiederverwendbarkeit ... 58
5.6 Erweiterbarkeit / Änderbarkeit / Wartbarkeit ... 59
5.7 Portabilität ... 59
5.8 Sicherheit ... 60
5.9 Verifizierbarkeit ... 61
5.10 Produktivität und Entwicklungskosten ... 61
5.11 Einführungskosten ... 61
5.12 Speicherplatzverbrauch ... 62
5.13 Laufzeitverhalten ... 63
5.14 Anforderungen an Programmierung ... 64
5.15 Anforderungen an Programmiersprache ... 65
5.16 Anforderungen an Compiler ... 66

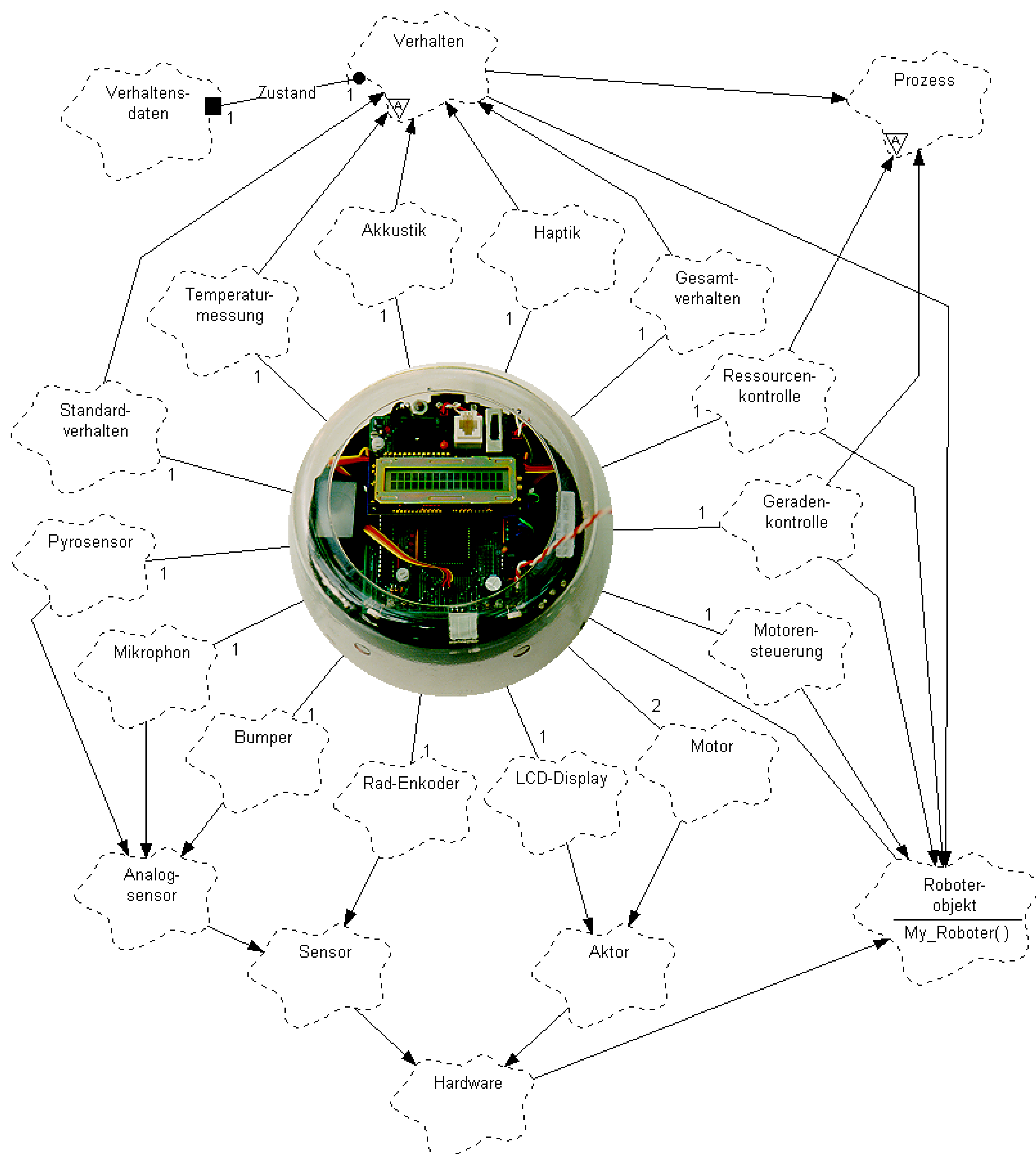
6 Schlußfolgerungen ... 68

Literatur ... 69

Anhang ... A-1
A) C++-Lösung ... A-1
QUEUE.HPP ... A-1
QUEUE.CPP ... A-1
SCHED.HPP ... A-2
SCHED.CPP ... A-3
ICSCHED.HPP ... A-4
ICSCHED.CPP ... A-4
MSCHED.HPP ... A-4
MSCHED.CPP ... A-5
MUTEX.HPP ... A-6

MUTEX.CPP A-6
PROZESS.HPP A-7
PROZESS.CPP A-7
HARDWARE.HPP A-7
HARDWARE.CPP A-8
SENSOREN.HPP A-10
SENSOREN.CPP A-11
AKTOREN.HPP A-14
AKTOREN.CPP A-15
VDATEN.HPP A-18
VDATEN.CPP A-18
VERHALT.HPP A-19
VERHALT.CPP A-20
RESCON.HPP A-23
RESCON.CPP A-23
GERADE.HPP A-23
GERADE.CPP A-24
TIMER.HPP A-25
TIMER.CPP A-26
FLIB.HPP A-26
FLIB.CPP A-26
MAIN.CPP A-26
BUMPWERT.HPP A-27
B) IC-Lösung A-28
ROBOT.LIS A-28
IC-Programm: ROBOT.C A-28
IC-Assembler-Modul: FREQUENZ.ASM A-33
IC-Assembler-Modul: SPEED.ASM A-34
IC-Programmbibliothek: SHAFT.C A-34
IC-Programmbibliothek: LIB_RW11.C A-34
C) Programmdiskette A-39

Objektorientierte Softwareentwicklung für eingebettete Systeme



1 Einführung

Eingebettete Systeme

Zwischen 1990 und 1995 hat sich der weltweite Absatz von Mikrocontrollern mehr als verdoppelt: Waren es 1990 13.9 Mrd. verkaufte Mikrocontroller, so wuchs die Zahl bis 1995 auf weltweit 30.7 Mrd. an. Das Einsatzgebiet für Mikrocontroller wird dabei immer umfangreicher und reicht von der Kaffeemaschine bis hin zur Raumfahrttechnik. Beispielhaft ist hierfür die Automobilindustrie, in der der Durchschnittspreis an Halbleitertechnik pro PKW zwischen 1990 und 1995 von 595 $ auf 1237 $ angestiegen ist.[1]

Solche Mikrocontroller sind jeweils physikalisch in ihr Umfeld eingebunden, aus dem sie nicht herausgelöst werden können. Sie stellen sogenannte „eingebettete Systeme" (ES) dar. Über Sensoren und Aktoren kommunizieren sie mit ihrer Umgebung und nehmen darin meist Überwachungs-, Steuerungs- und Regelungsaufgaben wahr. Dabei stellen ES häufig außergewöhnlich hohe Anforderungen an ihre Software: Gerade weil sie in Prozesse der realen Welt eingebunden sind, müssen sie besonders hohe Zuverlässigkeitsansprüche erfüllen und auch unter extremen Ausnahmebedingungen noch zuverlässig arbeiten. Ein Mikrocontroller im Navigationssystem eines Flugzeuges darf genauso wenig ausfallen wie ein Mikrocontroller im ABS-System eines Kraftfahrzeuges. ES zeigen reaktives Verhalten, müssen meist mehrere Aufgaben gleichzeitig erfüllen und haben dabei oft strenge Echtzeitanforderungen einzuhalten. Der Nachweis der fehlerfreien Funktion eines solchen Systems muß bereits vor dem erstmaligen Einbau in das Zielsystem möglich sein. Schließlich kann kein Flugzeug zu einem Testflug starten, ohne sich auf die Funktionsfähigkeit seiner Komponenten verlassen zu können. Softwarefehler, die erst nach Anlauf der Serienproduktion oder sogar erst nach Beginn der Auslieferung festgestellt werden, könnten ein ernsthaftes Sicherheitsrisiko für Mensch und Umwelt darstellen und Rückrufaktionen immense Kosten verursachen.

Zu all diesen hohen Softwarequalitätsanforderungen kommt bei ES hinzu, daß der Speicherplatz und die Rechenleistung allermeist sehr begrenzt sind. Hohe Stückzahlen bei der Produktion der Zielsysteme lassen die Kosten für mehr Speicherplatz oder höhere Rechenleistung schnell auf vielstellige Summen anwachsen, so daß sich selbst das Einsparen einzelner Bits auszahlen kann.

Wegen der knappen Ressourcen, der hohen Zuverlässigkeit und der hardwarenahen Programmierung wurde Software für ES bis vor kurzem überwiegend in Assembler realisiert. Allerdings erreichen auch solche Programme heute eine Größe und Komplexität, daß sie ohne

[1] Quelle der statistischen Angaben: Internet *http://www.hitex.com/automation/FAQ/primer/*, Kap. 3;

methodische Unterstützung aus dem Bereich der Softwaretechnologie, der Verwendung von Hochsprachen und der Nutzung leistungsfähiger CASE-Tools nicht mehr beherrscht werden können. Dieser Trend wird unterstützt durch neue Generationen von Mikrocontrollern, die über einen deutlich erweiterten Speicherplatz und verbesserte Rechenleistungen verfügen.

Objektorientierung für eingebettete Systeme?

Durch den Einsatz des objektorientierten Paradigmas konnten in den letzten Jahren besonders bei der Softwareentwicklung in großen, komplexen Problembereichen vielfältige Verbesserungen und Produktivitätssteigerungen erreicht werden. Erhöhung der Wiederverwendbarkeit, Steigerung der Kompatibilität und Dynamik, sowie die Handhabung von Komplexität sind einige wichtige Punkte. Doch vom Einsatz der OO bei der Programmierung im Kleinen ist weit weniger zu hören. Nur deshalb nicht, weil bisher einfach die Notwendigkeit dazu fehlte? Oder bringt die OO auch Nachteile mit sich, die sich ungünstig auf die Qualität, Sicherheit und Performance auswirken oder deren Einsatz dort gar verhindern? Gerade für ES ist diese Fragestellung besonders relevant, da die Komplexität der Software auch hier ständig zunimmt, andererseits jedoch höchste Anforderungen von der Software zu erfüllen sind.

Anwendungsbeispiel

In dieser Diplomarbeit wird für ein konkretes ES, den Mikrocontroller eines Mini-Roboters, eine objektorientierte Entwicklung der Software beispielhaft durchgeführt und dabei die oben genannten Fragen untersucht.

Bei dem Mini-Roboter handelt es sich um den am Messachusetts Institute of Technology (M.I.T.) entwickelten „Rug Warrior“. Er wird von dem Mikrocontroller Motorola 68HC11 (M68HC11) gesteuert und verfügt über vielfältige Sensoren und Aktoren. Beispiele hierfür wären neben anderen ein Pyrosensor, ein IR-System, ein Mikrophon, ein LCD-Display und zwei Fahrmotoren.

Die Steuerungssoftware, die in dem 32 KB kleinen RAM-Speicher Platz haben muß, soll den Roboter auf geradem Kurs fortbewegen. Gleichzeitig soll die Software verschiedene, vom Mikrofon wahrgenommene Töne unterscheiden und entsprechende Richtungsänderungen veranlassen, so daß der Roboter akustisch steuerbar ist. Bei Kollisionen mit Hindernissen soll die Steuerung diese Erkennen und den Roboter entsprechend ausweichen lassen.

Zur Lösung dieser Aufgabe wird ein Multi-Prozeß-System erstellt, das feine Nebenläufigkeit (Nebenläufigkeit innerhalb von Objekten) unterstützt und dabei flexibel den Einsatz von verschiedenen Schedulern gestattet. Ein Schutzmechanismus zum wechselseitigen Ausschluß von Zugriffen auf geschützte Bereiche wird eingeführt und ein Scheduler mit kooperativer

Prozeßverwaltung implementiert. Bei der Modellerstellung wird auf eine flexible Erweiterbarkeit geachtet. Der eigentliche Schwerpunkt der Arbeit liegt jedoch nicht auf der Realisierung der Steuerungssoftware, sonder auf dem Entwicklungsprozeß selbst und seinen Auswirkungen auf die Softwarequalitäts- und Produktivitätsmerkmale.

Die Softwareentwicklung wird nach der Methodik und Notation von *Booch* unter Zuhilfenahme des CASE-Tools *Rose* von *Rational* durchgeführt. Die Implementierung erfolgt mit *Borland C++* (Vers. 4.5). Leider kann diese objektorientierte Lösung mangels Verfügbarkeit eines Compilers für den M68HC11 derzeit jedoch nur am PC simuliert werden. Zum funktionalen Beweis wird deshalb zusätzlich eine strukturierte Lösung mit *Interactive C*[2] (IC) erstellt.

[2] *Interactive C* ist eine am M.I.T. Media Laboratory speziell für den M68HC11 entwickelte C-Programmiersprache. Sie enthält eine Untermenge von C-Sprachelementen und stellt Mechanismen zur dynamischen Prozeßerzeugung und -vernichtung zur Verfügung. Außerdem ist ein Kommandozeilen-Interpreter und -Debugger integriert.

2 Objektorientierte Softwareentwicklung

2.1 Klassifizierung objektorientierter Software

Wie [Fied 91, S. 16 ff] zeigt, lassen sich seit Beginn der Hochsprachenprogrammierung vier Generationen an Software unterscheiden:

1. Generation Unterprogramm-technik:	Das einzige Mittel zur Strukturierung stellt das Element des Unterprogramms dar, das im Wesentlichen zur Codierungseinsparung eingeführt wurde. Charakteristisch ist die ausschließliche Verfügbarkeit globaler Daten, auf welche jedes Unterprogramm uneingeschränkten zugreifen kann. Probleme: • Es ist eine hohe Anzahl an globalen Variablen erforderlich, auf die von allen Unterprogrammen ungeschützt zugegriffen werden kann. • Bei einer Typänderung einer Variablen müssen sämtliche Unterprogramme, die diese Variable verwenden, modifiziert werden. • Eine Anwendung, welche aus mehreren Unterprogrammen besteht, kann nur als Gesamtheit aufgefaßt werden. Eine getrennte Übersetzung, der isolierte Test oder aber die einfache Wiederverwendung der Unterprogramme ist nicht möglich.
2. Generation Datenstruktur-technik:	Möglichkeiten zur Beschreibung von Blockstrukturen, Datenstrukturen und Datentypen sind vorhanden, so daß erstmals die Möglichkeit parametrisierter Unterprogrammaufrufe besteht. Außerdem wird die getrennte Compilierung von Unterprogrammen unterstützt.
3. Generation Modulkonzept:	Module stellen Programmeinheiten dar, welche die von ihr manipulierten Daten von anderen Programmeinheiten abkapselt. Der Datenaustausch zwischen Modulen erfolgt über exakt definierte, parametrisierte Aufrufschnittstellen. Durch weitgehende Abkapselung der Daten können Module als eigenständige Einheiten aufgefaßt werden. Probleme: • Infolge der parametrisierten Schnittstelle ergeben sich Probleme bei der Softwarewartung und Modulintegration: Ändert sich die vereinbarte Schnittstelle, über die die Module miteinander kommunizieren, so müssen alle anderen Module, die diese Aufrufschnittstelle verwenden, geändert werden. • Gleiches gilt, wenn sich der Datentyp / die Datenstruktur der übergebenen Parameter ändert. Bei Systemen ohne Typprüfung kann dies zu Laufzeitfehlern führen, die besonders in Echtzeitumgebungen fatale Folgen haben können.
4. Generation Daten-orientierung:	Bei der Datenorientierung stehen nicht mehr Funktionen im Vordergrund, die Daten behandeln, sondern umgekehrt die Daten, auf welche ausgewählte Operationen ausgeführt werden können. Die Module werden also nicht mehr funktional orientiert, sondern daten- oder objektorientiert angelegt. Dieser Ansatz ist wesentlich beständiger, und Programmstrukturen werden dadurch weiter entkoppelt. Zur Unterstützung der Datenorientierung stehen abstrakte Datentypen, und zur Unterstützung der OO Klassen zur Verfügung.

Tabelle 1 - Softwaregenerationen

Der Begriff *„objektorientierte Programmierung“* wurde einige Zeit sehr weiträumig gefaßt. Inzwischen liegen Klassifizierungsschemen vor, mit denen datenorientierten Sprachen weiter abgegrenzt werden können. So unterschied Wegner 1989[3]:

Objekt-basierte Sprachen	Alle Sprachen, die datenorientierte Betrachtungsweisen mit Hilfe sprachlicher Elemente unterstützen, so daß Objekte eingerichtet werden können, auf die ein exakt definierter Satz von Operationen anwendbar ist.
Klassen-basierte Sprachen	**= Objekt-basierte Sprachen + Klassen** Klassen-basierte Sprachen erlauben es, Objekte mit gleichen Eigenschaften innerhalb einer Klassenbeschreibung zusammenzufassen.
Objekt-orientierte Sprachen	**= Klassen-basierte Sprachen + Vererbung** Objektorientierte Sprachen haben zusätzlich eine Klassenhierarchie mit Vererbungsmechanismen. So stellen Klassen auf einer höheren Hierarchieebene Klassen mit höherem Abstraktionsgrad dar und Klassen auf tieferer Hierarchieebene sind entsprechend mehr spezialisiert.

Tabelle 2 - Objektbasierte/-orientierte Sprachen

Im Gegensatz zu Wegner sind sich andere Autoren einig, daß Polymorphismus und dynamisches Binden weitere Grundbestandteile für eine objektorientierten Sprache sind (vgl. [Boo 94, S. 586], [Fied 91, S. 62]). [Mey 90, S. 65-68] stellt sogar folgende Eigenschaften als Anforderung an objektorientierte Sprachen:

- Objektbasierte, modulare Struktur;
- Datenabstraktion;
- Klassen;
- Mehrfache und wiederholte Vererbung;
- Polymorphismus und dynamisches Binden;
- Automatische Speicherplatzverwaltung;

2.2 Methodik des objektorientierten Softwareentwurfs

Objektorientierte Entwicklungsumgebungen und Programmiersprachen erzeugen nicht selbständig wiederverwendbare, wartbare und erweiterbare Software. Zum erreichen dieser Ziele ist ein sorgfältiger Entwurfsprozeß unter Berücksichtigung der Prinzipien und Mechanismen der OO notwendig. In den vergangenen Jahren sind zwar mehrere, teils konträre, Verfahren für den Entwurf objektorientierter Systeme entwickelt worden, die im Prinzip jedoch alle das folgende Schema zur Grundlage haben (vgl. [Schä 94, S. 6], [Boo 94, S. 295-311], [Stro 92, S. 406-410]):

1. Identifizierung von Klassen und Objekten;
2. Zuweisung von Attributen und Methoden;
3. Identifizierung der Beziehungen zwischen Klassen und Objekten;

[3] zitiert in [Fied 91, S. 61 f];

4. Implementierung des Entwurfs;

Ähnlich wie im strukturierten Ansatz gibt es auch bei der OO die drei Entwurfsphasen:

- *objektorientierte Analyse (OOA)*
- *objektorientiertes Design (OOD)* und
- *objektorientierte Programmierung (OOP).*

Inhaltlich unterscheiden sie sich jedoch grundlegend vom strukturierten Ansatz. So sind die einzelnen Phasen bei der objektorientierten Softwareentwicklung (OOSE) niemals in sich abgeschlossene autonome Entwurfsschritte, sondern stellen vielmehr einen Rahmen für ein iteratives, inkrementielles Vorgehen dar, deren Übergänge fließend sind. Jede Phase hat dabei dennoch ihre eigenen, klar definierten Inhalte und Ziele. (Vgl. zu 2.2.1 - 2.2.3 [Boo 94, S. 312-333] und [Schä 94, S. 21-33].)

2.2.1 Objektorientierte Analyse

Beim objektorientierten Paradigma versucht man, die Struktur des Problembereichs möglichst genau auf die Implementierung abzubilden. So ist der erste Schritt die Analyse, in der der Problembereich detailliert untersucht und beschrieben wird. Im Vordergrund steht dabei ganz eindeutig das „***was*** soll getan werden?“. Das „***wie*** soll es getan werden?“ wird in dieser Phase noch nicht berücksichtigt. Ziel ist es, alle Objekte des Problembereichs ausfindig zu machen und in Form von Klassen mit ihren Methoden und Attributen darzustellen. Dazu werden Verantwortlichkeiten zu Klassen zugeteilt, Beziehungen zwischen Klassen festgelegt und logische Abläufe definiert. Es soll ein Modell erstellt werden, das vollständig die funktionalen und operationalen Merkmale enthält und möglichst exakt der Realität entspricht. Strukturen der Implementierung bleiben in dieser Phase unbeachtet.

2.2.2 Objektorientiertes Design

Das Design ist der Zwischenschritt von der Analyse zur Implementierung. Hier wird erstmals der Lösungsraum (Programmiersprache, Betriebssystem, Hardware) betrachtet und überlegt, ***wie*** das Realitätsmodell aus der Analyse verwirklicht werden soll. Dabei müssen weitere implementierungsspezifische Klassen, Attribute und Methoden in das bestehende Modell eingefügt werden, die zur Lösung im Softwareraum notwendig sind. Typische Beispiele hierfür sind verkettete Listen oder Klassen zur Bildschirmausgabe, die in nahezu jeder Anwendung vorkommen. An dieser Stelle sollte auch die verwendete Klassenbibliothek untersucht werden, ob bereits existierende Klassen oder ganze Frameworks für die Lösung des Anwendungsproblems wiederverwendet werden können.

Es fällt auf, daß die Grenze zwischen Analyse und Design beim objektorientierten Ansatz

recht verschwommen ist. Die Ziele jedoch sind eindeutig: In der Analyse versucht man die reale Welt zu modellieren, indem man die darin vorkommenden Klassen und Objekte mit ihren Funktionalitäten ausfindig macht. Beim Design kümmert man sich um die Abstraktionen und Mechanismen, mit denen die Anforderungen der Anwendung verwirklicht werden können.

2.2.3 Objektorientierte Programmierung

Bei der objektorientierten Programmierung wird das erstellte Modell in die Implementierungssprache umgesetzt. Booch gibt für die objektorientierte Programmierung folgende Definition: *„Objektorientierte Programmierung ist eine Implementierungsmethode, bei der Programme als kooperierende Ansammlungen von Objekten angeordnet sind. Jedes dieser Objekte stellt eine Instanz einer Klasse dar, und alle Klassen sind Elemente einer Klassenhierarchie, die durch Vererbungsbeziehungen gekennzeichnet ist.“* [Boo 94, S. 57]
Da bereits während der Analyse und des Designs sämtliche Prinzipien und Mechanismen der OO eingeflossen sind, sollte die Programmierung im Gegensatz zu Analyse und Design einen weitaus geringeren zeitlichen Anteil einnehmen, als dies im strukturierten Ansatz der Fall wäre.

2.2.4 Objektorientierte Softwareentwicklung nach Booch

Gerade weil objektorientierte Softwareentwicklung ein iterativer, inkrementieller Prozeß ist, ist das klassische Wasserfallmodell nach Boehm für die Entwicklung bzw. den gesamten Lebenszyklus von objektorientierter Software unzureichend. Booch schlägt deshalb ein erweitertes Wasserfallmodell vor, das aus einem Macro- und einem Micro-Prozeß besteht. Der Macro-Prozeß, mit seinen Schritten:

- Konzeptualisierung (Festlegung der Kernanforderungen)
- Analyse (Einrichtung eines Modells des gewünschten Verhaltens)
- Design (Erzeugung einer Architektur)
- Evolution (Entwicklung der Implementierung)
- Wartung (Verwaltung der Entwicklung nach der Auslieferung)

bildet den Rahmen für den spiralförmigen Micro-Prozeß. Im Micro-Prozeß ist dann die Vorgehensweise innerhalb der einzelnen Macro-Phasen beschrieben:

- Festlegung der Klassen und Objekte auf einer bestimmten Abstraktionsebene
- Festlegung der Semantik dieser Klassen und Objekte
- Festlegung der Beziehungen zwischen diesen Klassen und Objekten
- Spezifikation der Schnittstellen und Implementation der Klassen und Objekte

(Vgl. [Boo 94, Kap. 6].)

Für diese Diplomarbeit wird die Methode und Notation von Booch unter Zuhilfenahme des CASE-Tools Rose von Rational verwendet. Zum Verständnis der nachfolgenden Objekt- und Klassendiagramme ist hier eine Übersicht der verwendeten Symbole gegeben:

Klassendiagramme:	
Klassenname Attribute Methoden( )	Klasse
	Assoziation
	Vererbung
	Verwendungsbeziehung
	Eigentumsbeziehung
	Enthaltensein per Wert
	Enthaltensein per Referenz
Exportsteuerung:	
	public
I	protected
II	private
Eigenschaften:	
A	abstrakte Klasse
Objektdiagramme:	
Name	Objekt

Tabelle 3 - Verwendete Symbole nach Booch-Notation

2.3 Objektorientierung und Nebenläufigkeit

2.3.1 Prozeßstrukturen

Zur Realisierung von Nebenläufigkeit mit objektorientierten Programmiersprachen stehen prinzipiell drei verschiedene Ansätze zur Auswahl[4]:

a) Ansatz mit übergeordneter Prozeßstruktur

Stellt das Betriebssystem bereits Mittel zur Prozeßkommunikation und -synchronisation zur Verfügung, so kann jede objektorientierte Programmiersprache um Nebenläufigkeit erweitert

[4] Eine detailiertere Beschreibung ist in [Hüs 95] zu finden.

werden. Die Prozeßstruktur ist hier also der Programmiersprache übergeordnet.

b) Objektbasierter Ansatz

Beim objektbasierten Ansatz wird das Prozeßkonzept mittels Klassen realisiert. Die Prozeßsynchronisation geschieht durch den Empfang von Botschaften. Es ist jedoch keine Spezialisierung durch Vererbung möglich.

c) Objektorientierter Ansatz

Beim objektorientierten Ansatz gibt es nochmals zwei Möglichkeiten zur Unterscheidung. Prozesse können entweder Single- oder Multiple-Thread-Charakter aufweisen:

c1) Prozesse mit Single-Thread-Charakter (grobe Nebenläufigkeit):

Hier stellen die Objekte selbst Prozesse dar. Innerhalb eines Objektes gilt der wechselseitige Ausschluß für verschiedene Methoden. Es kann also zu einem Zeitpunkt immer nur eine Methode eines Objektes aktiv sein. Die Kommunikationspunkte sind dabei die Methoden selbst.

c2) Prozesse mit Multiple-Thread-Charakter (feine Nebenläufigkeit):

Im Gegensatz zum Single-Thread-Charakter können beim Multiple-Thread-Charakter innerhalb eines Objektes mehrere Methoden gleichzeitig ausgeführt werden. Die Synchronisation muß dabei explizit implementiert werden, wobei alle bekannten Synchronisationskonzepte möglich sind.

2.3.2 Entwurfsmethoden

Zum Entwurf von objektorientierten Anwendungen mit Nebenläufigkeit gibt es vielfältige Ansätze und Methoden. Prinzipell können nach [Hüs 95] jedoch Methoden mit klassischem und Methoden mit integriertem Prozeßkonzept unterschieden werden.[5]

Beim *klassischen Prozeßkonzept* werden die Mittel (wie Prozesse, Semaphoren o.ä.) innerhalb des Entwurfs durch Klassen und Objekte repräsentiert. Dabei muß die Synchronisation explizit entworfen und implementiert werden, was Freiheit für grobe oder feine Nebenläufigkeit läßt.

Beim *integrierten Prozeßkonzept* repräsentiert jedes Objekt gleichzeitig einen Prozeß. Die Methodenaufrufe stellen hierbei die Synchronisationspunkte dar, was grobe Nebenläufigkeit impliziert. Allerdings existieren hierfür bisher nur wenige Entwurfsmethoden und Programmiersprachen als Prototypen. Somit haben sie noch kaum Verbreitungsgrad.

[5] [Boo 94, S. 101] unterscheidet neben immanenter und nicht-immanenter Nebenläufigkeit noch eine dritte Nebenläufigkeit, in der ein Interrupt periodisch einen Methodenaufruf veranlaßt und somit Nebenläufigkeit simuliert.

2.3.3 Objektkommunikation und -synchronisation

In objektorientierten Systemen mit Nebenläufigkeit gibt es generell zwei Arten von Objekten: *Aktive* und *Passive*. Dabei ist ein aktives Objekt nach [Boo 94, S. 607] *„Ein Objekt, das einen eigenen Steuerfluß (thread) besitzt“*. Ein passives Objekt ist entsprechend ein rein sequentielles Objekt ohne eigenen Steuerfluß. Zur Kommunikation von Objekten müssen diese entsprechend synchronisiert werden. Booch unterscheidet hierbei vier Synchronisationsarten:

- Synchron: Sender wartet, bis Empfänger bereit;
- Blockierend: Sender bricht ab, falls Empfänger nicht sofort reagiert;
- Timeout: Sender bricht ab, falls Empfänger nicht nach vorgegebener Zeit reagiert;
- Asynchron: kein Warten, Nachrichten werden in Warteschlange eingereiht;

3 Objektorientierte Softwareentwicklung anhand eines Mini-Roboters

3.1 Anwendungsbeispiel

3.1.1 Roboter „Rug Warrior“

Wie bereits aus dem Begriff „eingebettete Systeme“ hervorgeht, kann es keine Software für ES ohne umgebendes System geben. Die Untersuchungen dieser Diplomarbeit werden anhand des mobilen Mini-Roboters *„Rug Warrior“*[6] durchgeführt. Er enthält eine ganze Reihe von Sensoren und Aktoren, die in den zwei folgenden Tabellen aufgelistet und auf dem Board-Layout (Abbildung 1) zu sehen sind:

Sensor	Funktion
1 Mikrophon	Geräuschwahrnehmung
1 Pyrosensor	Wärmeerkennung
3 Mikroschalter (Bumper)	Kollisions- und Hinderniserkennung
2 Rad-Encoder	Drehwinkelbestimmung der Räder
2 Fotowiderstände	Helligkeitserkennung /-unterscheidung
2 Infrarot-LED’s	IR-Sender zur Hinderniserkennung
1 Infrarot-Detektor	IR-Empfänger zur Hinderniserkennung

Tabelle 4 - Sensoren des Rug Warrior

Aktor	Funktion
2 Fahrmotoren	Fortbewegung
32-stelliges LCD-Display	Textausgabe in 2 Zeilen mit je 16 Zeichen
Piezo-Piepser	Tonerzeugung
4 LED’s	Statusanzeige

Tabelle 5 - Aktoren des Rug Warrior

[6] Der mobile Mini-Roboter *„Rug Warrior“* wurde am Massachusetts *Institute of Technology* (*M.I.T.*) entwickelt und ist in Deutschland bei *JOKER Computertechnik und Robotik* (Internet: *http://www.joker-robotics.com*) erhältlich. Eine Kurzvorstellung ist in [Bräu 96] gegeben.

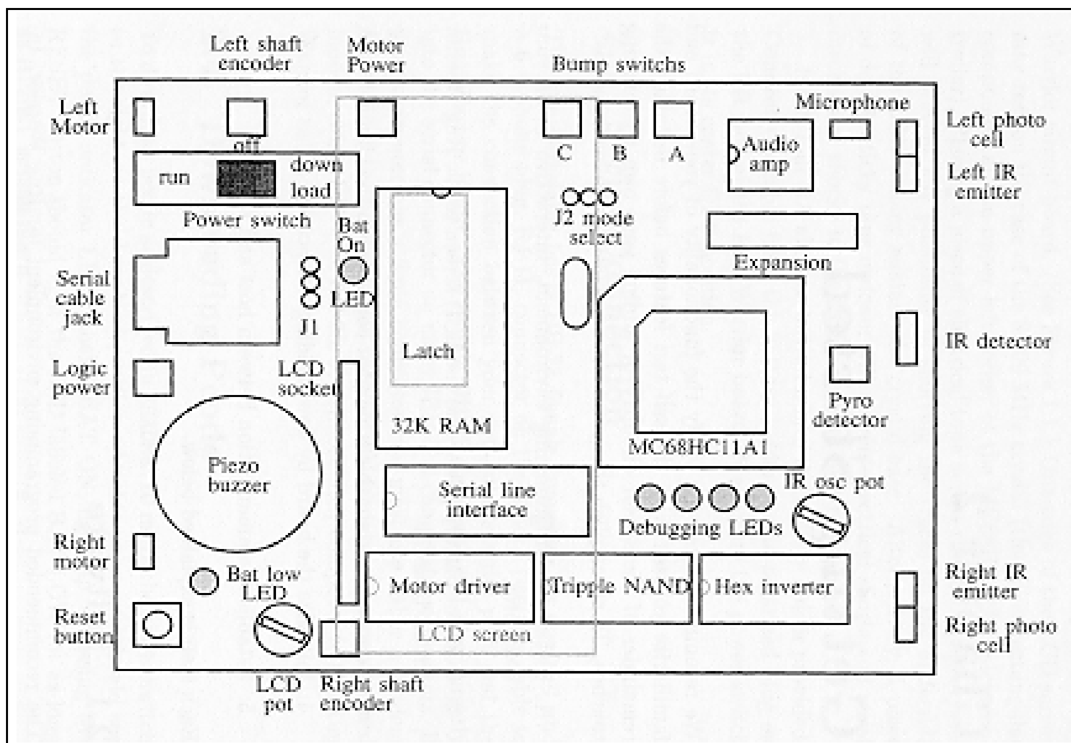


Abbildung 1 - Platinen-Layout des Rug Warrior [Jon 95, S. 3]

Auf dem Board des Roboters ist zusätzlich ein 32-KByte-RAM-Speicher vorhanden. Zur Programmierung des Mikrocontrollers steht eine serielle Schnittstelle zur Verfügung. Die Stromversorgung für die Motoren wird, getrennt von der Stromversorgung für den Mikrocontroller und der Sensorik, über NiCd-Akkublocks bereitgestellt. Die Stromkreise und Anschlußbeschaltungen der einzelnen Systemkomponenten ist in der folgenden Abbildung 2 zu sehen:

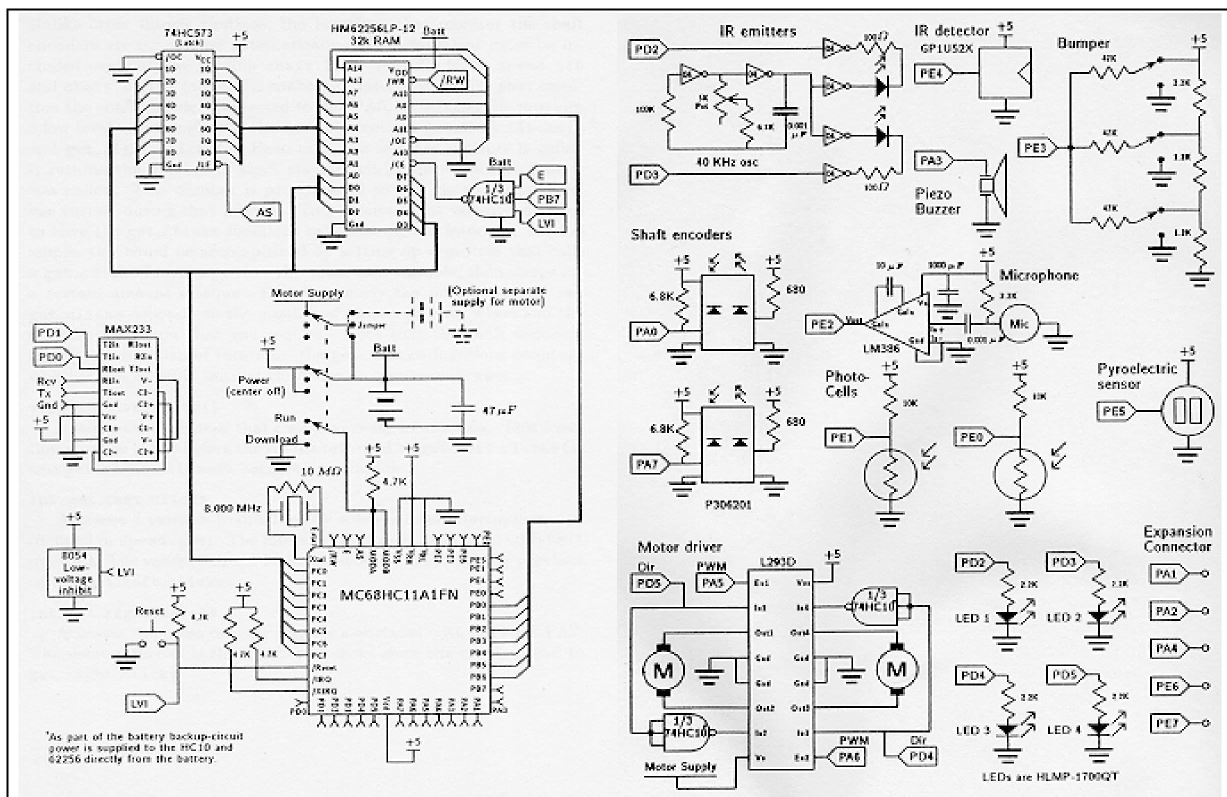


Abbildung 2 - Stromkreise und Anschlußbeschaltungen des Rug Warrior [Jon 95, S. 80 f]

3.1.2 Mikrocontroller

Der Roboter wird von einem Mikrocontroller Motorola 68HC11A1FN gesteuert. Wie in Abbildung 3 zu sehen ist, hat der Mikrocontroller folgende Komponenten in seinem Chip integriert:

- CPU;
- 256 Byte RAM;
- 1 Time-Counter-System;
- 16-Bit Adreßbus;
- 8-Bit Datenbus;
- 5 Ports (A-E):
  - Port A: Timersystem:
    3 digitale Eingänge;
    4 digitale Ausgänge;
    1 Steuerleitung (auch als Pulszähler verwendbar);
  - Port B: alternativ externer Bus oder 8-Bit Digitalausgang;
  - Port C: alternativ externer Bus oder 8-Bit Digitalein-/-ausgang;
  - Port D: Kommunikationsport (6 Bit frei konfigurierbar als Digitalein-/-ausgang);
  - Port E: alternativ 8-Bit Digitaleingang oder 8 Analogeingänge für ADU;

Der Systemtakt wird von einem 8.000 MHz-Quarz erzeugt, welcher durch einen Vorteiler durch vier geteilt wird, so daß der Bustakt (E-Takt) für alle Systemkomponenten 2.000 MHz beträgt. Um den externen 32 KB-RAM-Speicher nutzen zu können wird der Mikrocontroller im *Expanded Multiplexed Mode* betrieben. Der Zugriff auf diesen Speicher erfolgt über Port B und C, so daß diese beiden Ports also nicht mehr für die Anwendung zur Verfügung stehen. Zugriffe auf die Peripherie des Mikrocontrollers erfolgen mittels Memory-Mapped IO.

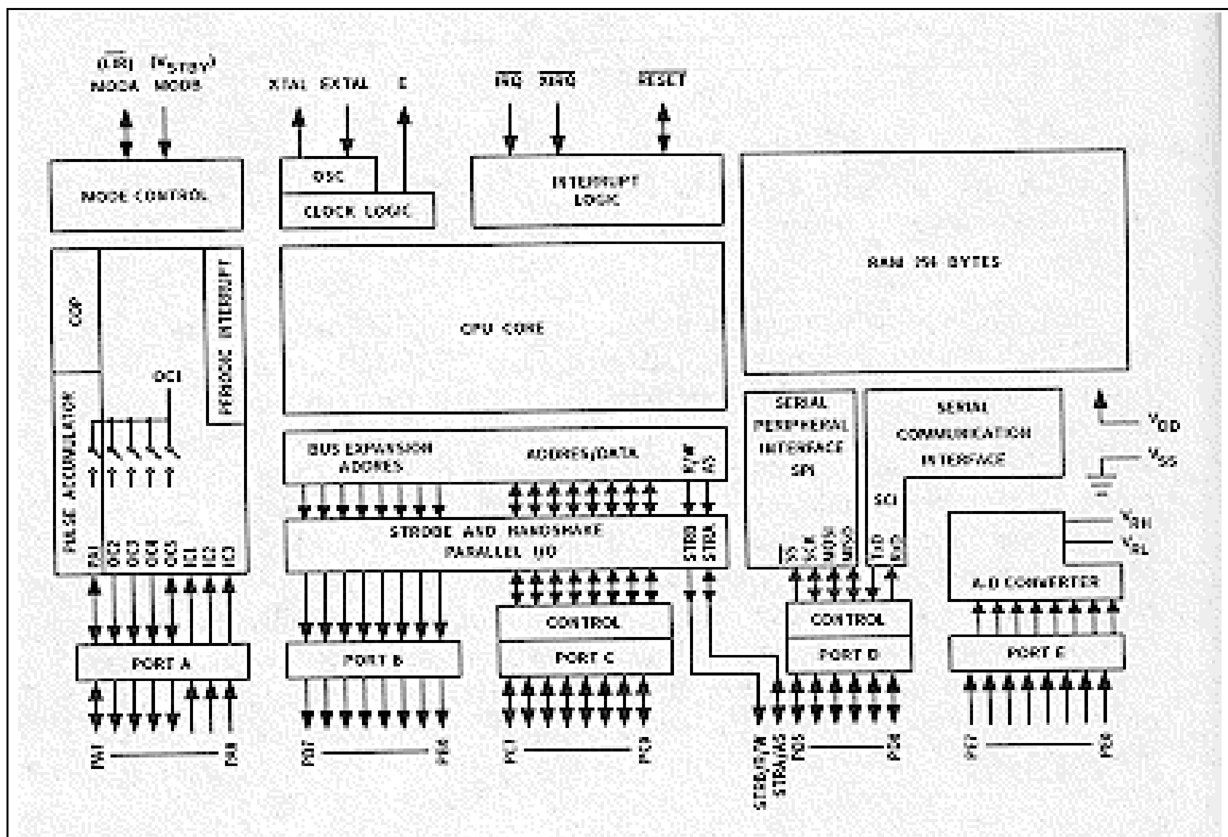


Abbildung 3 - Mikrocontroller Motorola 68HC11A0[7] [Jon 93, S. 44]

3.1.3 Aufgabenstellung

Zur Durchführung der Untersuchungen dieser Diplomarbeit ist folgende Aufgabenstellung definiert: Der Roboter soll sich im Normalfall geradeaus fortbewegen. Kollidiert er während der Fortbewegung mit einem Hindernis, so soll er darauf reagieren und dem Hindernis ausweichen. Gleichzeitig soll er mittels zweier verschiedener akustischer Signale steuerbar sein.

3.2 Analyse

Wie in Abschnitt 2.2.1 aufgezeigt, besteht das Ziel der OOA in der Erstellung eines möglichst orginalgetreuen Modells der Wirklichkeit. So ist es einfach, die ersten Objekte ausfindig zu machen, die zur Lösung der Aufgabenstellung beitragen. Eigentlich können alle Roboterkomponenten, wie sie in Tabelle 4 und Tabelle 5 aufgelistet sind 1:1 als Objekte in das Modell übernommen werden.

Einige Hardwarebestandteile bilden dabei bereits abgeschlossene Teilsysteme, die vom Prozessor auch nur als solche wahrgenommen werden. Ein Beispiel dafür sind die drei Mikroschalter (Bumper): Über Widerstände (siehe Abbildung 2) sind sie hardwareseitig bereits zu einem Widerstandsnetz zusammengeschaltet, so daß der Prozessor nur einen Analogwert an PE3 aus der Kombination der drei Schalterstellungen „sieht". Dieses Teilsystem geht einfach

[7] Die Abbildung des 68HC11A0 entspricht dem Bautyp 68HC11A1FN.

als Objekt „*Bumper*“ in das Modell ein. Es wäre unsinnig, solche Teilsysteme für das Modell wieder bis in ihre kleinsten Untereinheiten, also bis hin zum einzelnen Schalter und Widerstand aufzulösen, da diese anwendungstechnisch niemals einzeln angesprochen werden müssen. (Anders würde es sich verhalten, wenn die Schalter einzeln auf digitale Eingänge des Prozessors geführt wären und es Aufgabe des Prozessors wäre, die Kombination aus den Schalterstellungen zu ermitteln.) Es kann also schon hier festgehalten werden, daß die Sichtweise für die Modellbildung bei einem solchen ES vom Prozessor aus die Geeignetste ist. Schließlich befindet sich ja die entstehende Software später „in“ diesem Prozessor, und dann ist nur relevant, wie die Software ihre „reale Umgebung sieht“.

Im folgenden wird nicht der gesamte Roboter mit all seinen Komponenten modelliert, sondern nur die Aufgabenstellung betreffenden Komponenten. Später wird die Aufgabenstellung erweitert und daran gezeigt, welche vorteilhaften Auswirkungen durch die objektorientierte Modellierung ausgenutzt werden können. Unter Berücksichtigung der oben beschriebenen Sichtweise, sind die ersten Objekte für die Softwarelösung also folgende:

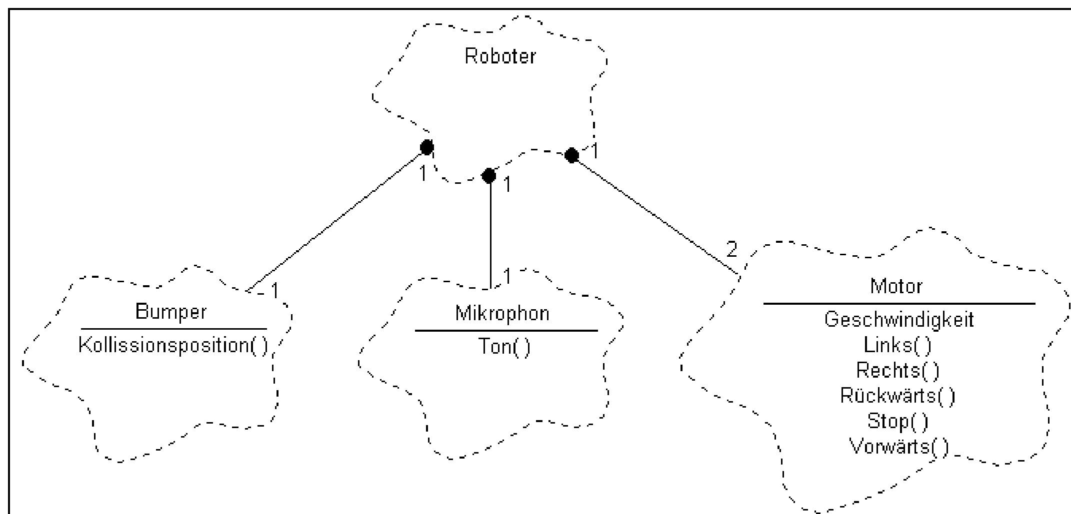


Abbildung 4 - Klassendiagramm OOA (1)

Wäre Abbildung 4 bereits das gesamte Modell, so hätte *Bumper* die Aufgabe, *Roboter* auftretende Kollisionen an verschiedenen Positionen zu melden. *Mikrophon* wäre dafür zuständig, auftretende Töne zu erkennen und ebenfalls an *Roboter* zu melden. *Motor* hätte die Fahrkommandos entgegenzunehmen und auszuführen. *Roboter* müßte hierbei alle sonstigen steuernden und koordinierenden Aufgaben erledigen, die notwendig sind. So müßte *Roboter* alle eintreffenden Signale der Sensoren entgegennehmen. Weiter müßte *Roboter* die Aufgabe übernehmen, die beiden Motoren so anzusteuern, daß eine sinnvolle Fahrtrichtung daraus entsteht. Daß dies nicht sinnvoll ist leuchtet ein, wenn man an eine möglichst flexible Erweiterbarkeit denkt. Eigentlich ist *Roboter* nur ein zusammenfassendes Objekt, das alle anderen Objekte enthält. Der Roboter hat allerdings implizit ein Verhalten, in dem seine Funktionalität

steckt. Es liegt also nahe eine Klasse *Verhalten* einzuführen, die ein solches Verhalten modelliert. In unserem Anwendungsbeispiel sind eigentlich gleich drei solcher Verhaltensweisen impliziert: Im Normalfall (d.h. wenn sonst nichts Wichtigeres zu tun ist) soll der Roboter einfach nur geradeaus fahren. Dieses Verhalten wird hier *Standardverhalten* genannt. Das zweite Verhalten besteht darin, akustische Signale zu erkennen und entsprechend darauf zu reagieren (*Akustik*). Und drittens soll der Roboter Kollisionen mit Gegenständen über seine Bumper erkennen und ausweichen (*Haptik*).

Die verschiedenen Verhalten des Roboters bekommen also die einzelnen Steuerungsaufgaben delegiert. Sie „wissen" dabei allerdings nichts von der Existenz zweier Motoren bzw. Räder. Deshalb kann es nicht Aufgabe eines Verhaltens sein, die beiden Räder so anzusteuern, daß eine Fortbewegung in einer sinnvollen Richtung stattfindet. Das Verhalten muß unabhängig von solchen Detailaufgaben bleiben und nur grundlegende Steuerkommandos wie z.B. „links" oder „rechts" aussenden. Die richtige Verarbeitung dieser Kommandos bekommt eine weitere Klasse *Motorensteuerung* zugeteilt, die sich um die richtige Ansteuerung der beiden Motoren kümmert. Schließlich wäre es auch denkbar, daß die beiden Räder einmal durch einen Kettenantrieb o.ä. ersetzt werden sollen. Dann wäre es äußerst aufwendig, alle Verhalten entsprechend der neuen Antriebstechnik anzupassen. Durch die Klasse *Motorensteuerung* bleibt die Anwendung jedoch unabhängig und flexibel änder- und erweiterbar.

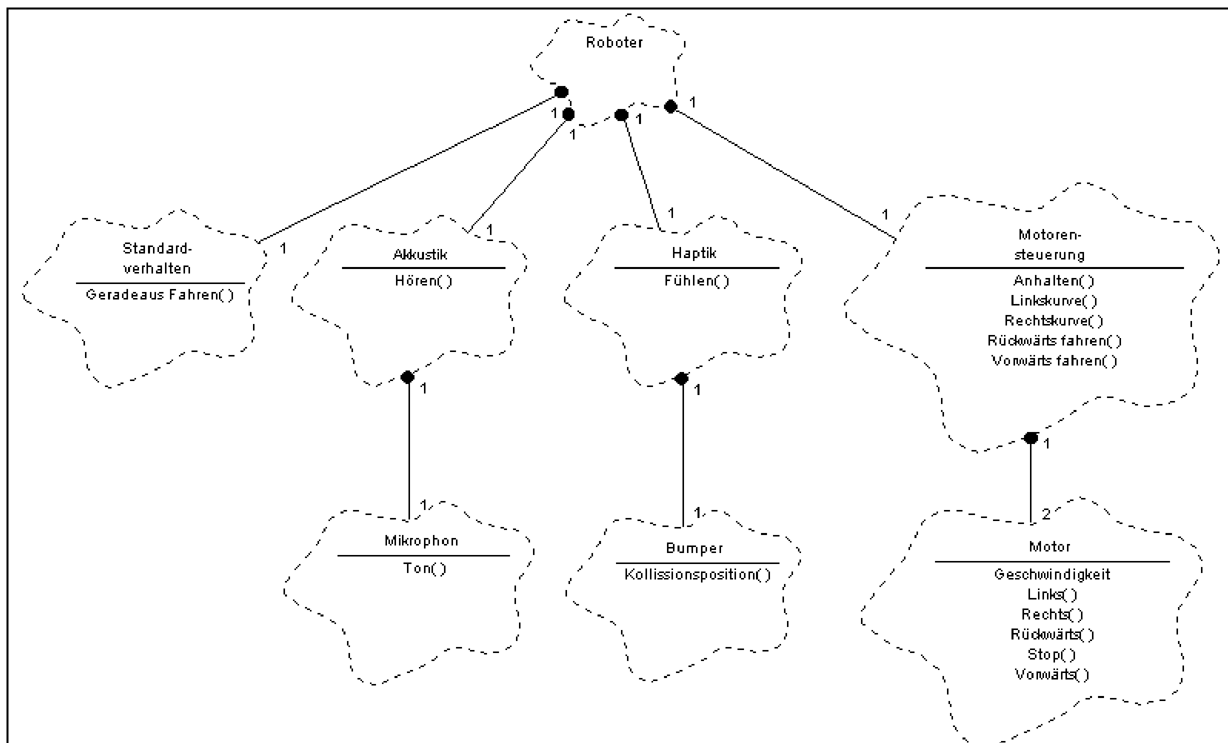


Abbildung 5 - Klassendiagramm OOA (2)

Ein großes Problem der Robotik ist, daß an der Steuerungseinheit normalerweise riesige Datenmengen von allen Sensoren anfallen. Diese Daten müssen dann in allen Kombinationen

ausgewertet und daraus eine sinnvolle Gesamtreaktion ermittelt werden. Das von *Brooks* unter dem Namen *„subsumption architekture"* (siehe [Jon 93, S. 247 ff]) bekannte Modell stellt hierzu eine echte Alternative dar. Dabei kann die Subsumption-Architektur mit dem objektorientierten Paradigma optimal realisiert werden, indem ein Verhalten in einer Klasse modelliert wird. Jedes Verhaltensobjekt ist dann selbst für die Auswertung seiner lokalen Sensordaten verantwortlich. Es schickt lediglich noch ein Kommando an die Ausführungseinheit weiter. Die große Datenflut und die daraus resultierende erschwerte Auswertung kommt erst gar nicht zustande.

Doch was passiert, wenn verschiedene Verhalten des Roboters sich gegenseitig widersprechen? Was passiert wenn das *Standardverhalten* „geradeaus" und die *Akustik* „links" anfordert? Es ist eine Kontrollinstanz notwendig, die bestimmt, welches Kommando Vorrang hat und ausgeführt werden soll und welches zurückstehen muß. Bezogen auf das bisherige Anwendungsbeispiel gibt es lediglich eine ausführende Einheit, nämlich die *Motorensteuerung* mit ihren Motoren. Der Roboter besitzt allerdings mehrere Aktoren, die jeweils nur einen einzigen (exklusiven) Zugriff erlauben. Um ein leicht zu erweiterndes Modell zu erstellen wird eine Klasse *Ressourcenkontrolle* eingeführt, die den Zugriff auf alle verfügbaren Ressourcen verwaltet. Damit die Problematik und Funktionalität der Ressourcenkontrolle besser gezeigt werden kann, wird die Aufgabenstellung diesbezüglich erweitert, daß jedes Verhalten zusätzlich Statusanzeigen auf dem LCD-Display ausgeben darf. Außerdem wird ein weiteres Verhalten namens *Temperaturmessung* eingeführt, das ständig die aktuelle Temperatur über den *Pyrosensor* ermitteln und auf dem *LCD-Display* anzeigen soll. Die einzelnen Verhalten bekommen Prioritäten zugeteilt, die der *Ressourcenkontrolle* die Wichtigkeit der übermittelten Steuerkommandos mitteilen. Jedes Verhalten kann dabei über jede Ressource des Roboters verfügen, ohne dies jedoch zu müssen. Die *Ressourcenkontrolle* vergibt dann je nach Prioritäten die Ressourcen. Daraus bildet sie ein Gesamtergebnis, welches sie an die Klasse *Gesamtverhalten* weiterleitet. *Gesamtverhalten* ist also die ausführende Einheit des Roboters, wohingegen die Verhalten *Haptik*, *Akustik*, *Temperaturmessung* und *Standardverhalten* die wahrnehmenden Einheiten darstellen.

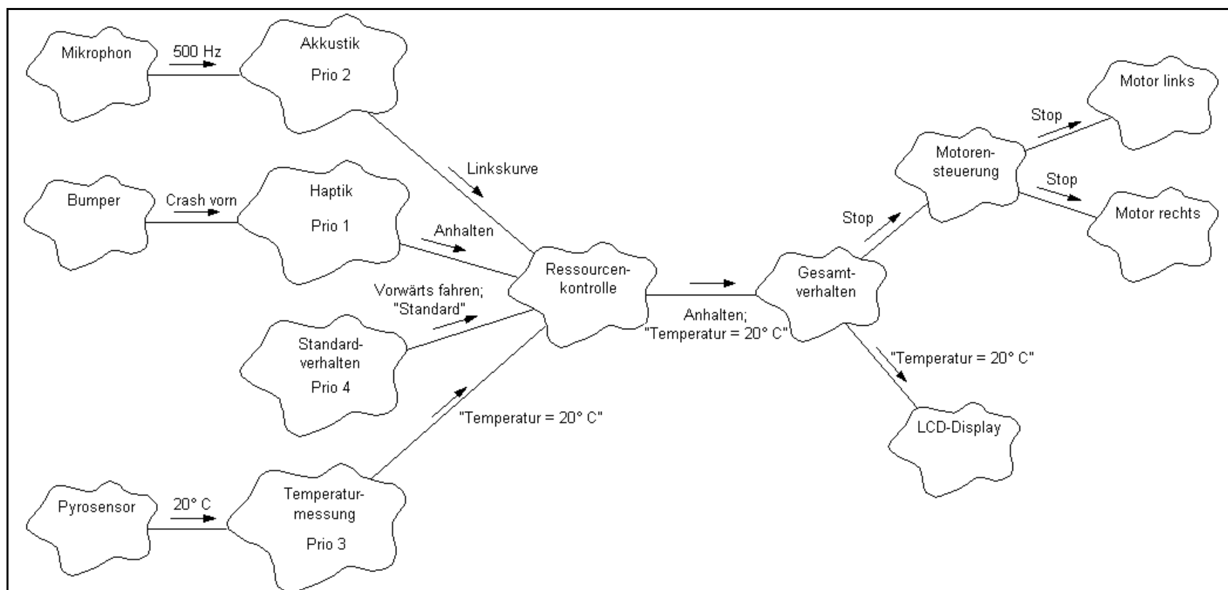


Abbildung 6 - Objektdiagramm Verhaltensanforderungen

Das Beispiel in Abbildung 6 zeigt, wie die verschiedenen Verhalten verschiedene Anforderungen an die *Ressourcenkontrolle* senden. *Haptik* hat dabei die höchste Priorität (1). Aus diesem Grund kommt ihr Motorsteuerungskommando „*Anhalten*“ zur Ausführung. Da *Haptik* jedoch keine Anforderung *LCD-Display* sendet, bleibt diese Ressource für Verhalten mit niedrigeren Prioritätsstufen frei, so daß hier *Temperaturmessung* mit Priorität 3 vor *Standardverhalten* mit Priorität 4 den Zuschlag für das *LCD-Display* bekommt. Der Text *„Temperatur = 20° C“* wird also angezeigt, der Text *„Standard“* wird wegen dem Ressourcenkonflikt unterdrückt.

Damit jedes Verhalten „weiß“ welche Ressourcen überhaupt zur Verfügung stehen, wird eine Klasse *Verhaltensdaten* eingeführt, in der alle möglichen Operationen des Roboters festgelegt sind. Jedes Verhalten hat seine eigenen Instanz dieser Klasse *Verhaltensdaten*. Über diese kommunizieren die einzelnen Verhalten mit der *Ressourcenkontrolle*, indem sie ihre *Verhaltensdaten* als Anforderung an die *Ressourcenkontrolle* schicken. Die *Ressourcenkontrolle* ermittelt nach den oben angegebenen Kriterien die resultierenden *Verhaltensdaten* und sendet sie an das Objekt *Gesamtverhalten*. Dieses versteht die *Verhaltensdaten* und veranlaßt deren Ausführung.

An der Stelle kann sehr effizient der Mechanismus der Vererbung ausgenutzt werden: Jedes einzelne Verhalten hat eine Methode für ihr spezifisches Verhalten, ein Attribut für die eigene Priorität und eigene Verhaltensdaten. Diese Eigenschaften können in einer allgemeinen, abstrakten Oberklasse *Verhalten* zusammengefaßt werden. Die Spezialisierung des individuellen Verhaltens kann dann in der jeweiligen Unterklasse implementiert werden.

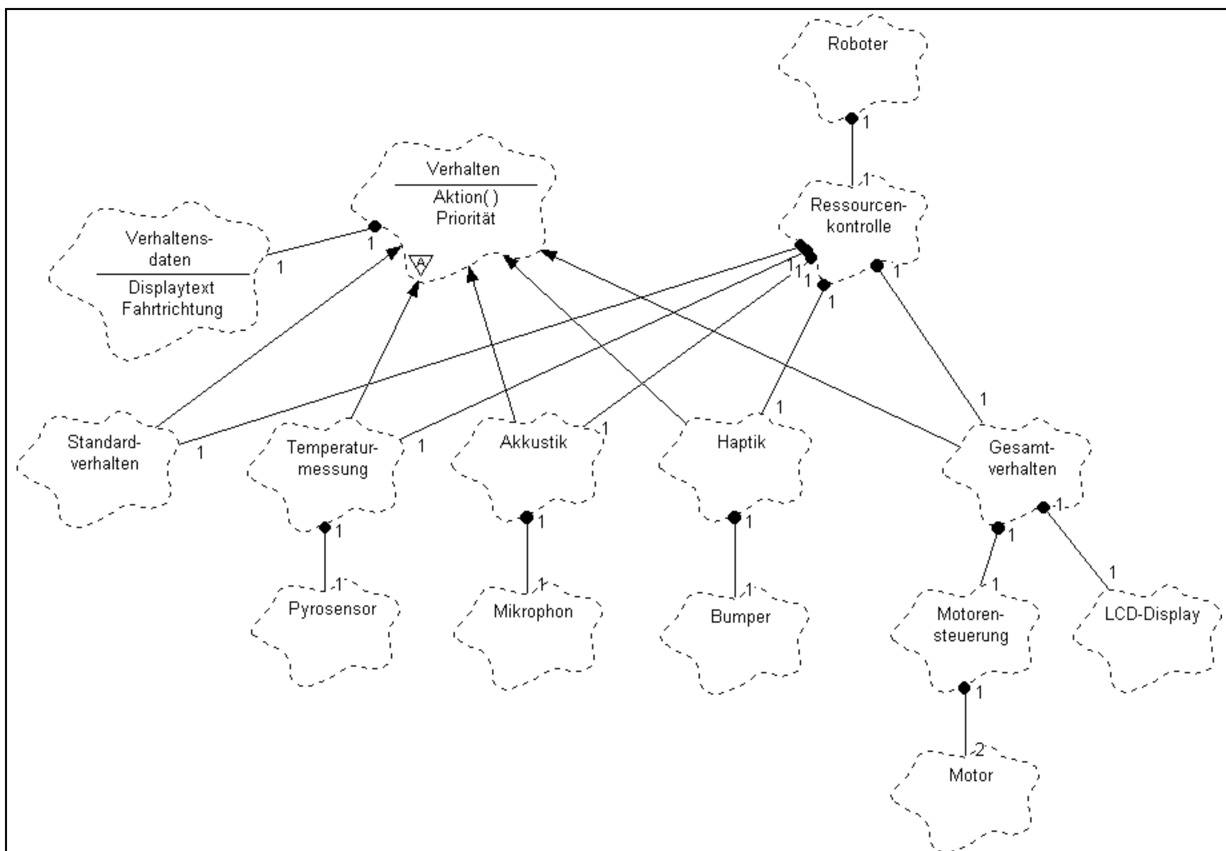


Abbildung 7 - Klassendiagramm OOA (3)

Damit stehen schon alle grundlegenden Klassen fest, um die Aufgaben des Anwendungsbeispiels zu erfüllen. Die weiteren Schritte beschäftigen sich mit der Vervollständigung des Modells um implementierungstechnische Eigenschaften.

3.3 Design

3.3.1 Nebenläufigkeit

Es ist selbstverständlich daß alle Aktivitäten wie Fühlen, Hören und Fortbewegen gleichzeitig geschehen müssen. Eine übergeordnete Prozeßstruktur nach 2.3.1.a steht bei dem Mikrocontroller nicht zur Verfügung. Die Nebenläufigkeit, die bisher implizit für das Modell vorausgesetzt wurde, muß nun also explizit modelliert und realisiert werden. Leider sind die Klassen aus dem Borland C++-Modul „*Thread*“ betriebssystemabhängig, so daß sie hier nicht zur Wiederverwendung für den M68HC11 zur Verfügung stehen. Darum wird für diese Diplomarbeit ein eigenes Multi-Prozeß-System entworfen. Es kann flexibel an verschiedene Scheduler angepaßt werden und unterstützt dabei die feine Nebenläufigkeit nach 2.3.1.c2. Zum Entwurf wird das klassische Prozeßkonzept verwendet, das mit der Booch-Notation unter Verwendung von Rose modelliert werden kann.

Die Möglichkeit, das Programm mittels verschiedener Scheduler zu betreiben, wird erst durch den Einsatz der OO-Technologie mit deren Polymorphismus eröffnet, was eine hohe Flexibi-

lität mit sich bringt: So können beispielsweise je nach Bedarf Scheduler für kooperative und nicht-kooperative Prozesse zum Einsatz kommen. Das Prozeßsystem ist dafür ausgelegt, daß sowohl preemptives, als auch non-preemptives Multitasking unterstützt wird. Hierfür müssen bei der Prozeßimplementierung selbstverständlich alle Regeln für die feine Nebenläufigkeit beachtet werden, so daß der gleichzeitige Zugriff auf Variablen durch Schutzmechanismen wie Mutex[8] oder Semaphoren verhindert wird.

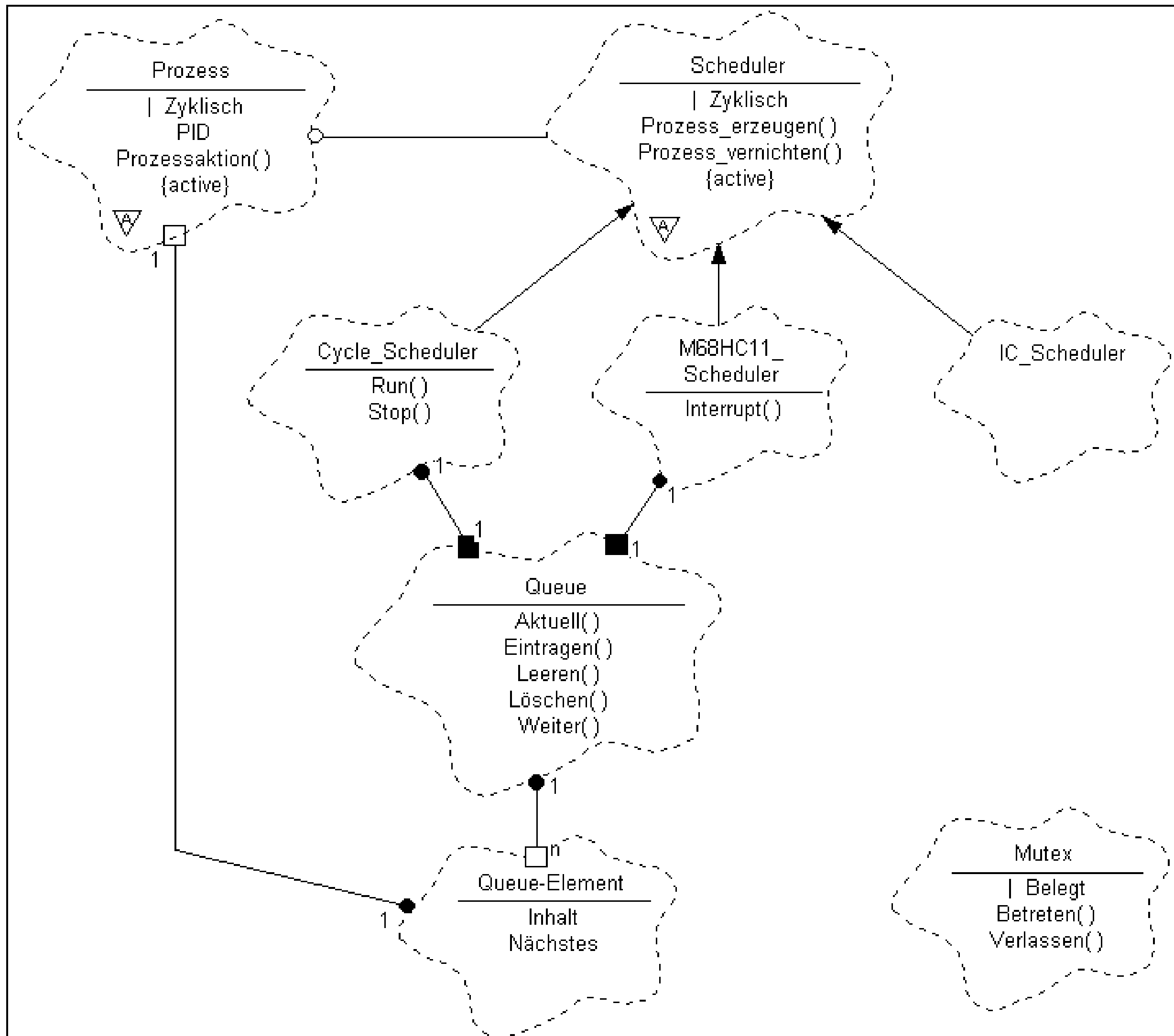


Abbildung 8 - Klassendiagramm Prozeßsystem

Prozesse

Die abstrakte Oberklasse *Prozess* faßt alle Eigenschaften zusammen, die ein Prozeß zu seiner Ausführung benötigt. Dies ist insbesondere die abstrakte Methode *Prozessaktion*, die jeder Prozeß enthalten muß. Darin wird in der jeweiligen Unterklasse die eigentliche Funktionalität des Prozesses implementiert. Durch die Möglichkeit der feinen Nebenläufigkeit darf es in *Prozess* aber durchaus weitere Methoden geben, die von anderen Objekten parallel dazu akti-

[8] *Mutex* = „Mutual Exclusion“ (wechselseitiger Ausschluß)

viert werden können.[9] Ferner sind in *Prozess* weitere Attribute zu Prozeßverwaltung wie etwa die Prozeß-ID (PID) enthalten.

<u>*Scheduler*</u>

Die abstrakte Oberklasse *Scheduler* schreibt alle Eigenschaften vor, die ein Scheduler unbedingt haben muß. Die abstrakten Methoden *Prozess_erzeugen* und *Prozess_vernichten* müssen dabei von den Unterklassen individuell redefiniert werden. *Zyklisch* ist ein Attribut, das der Scheduler an seine Prozesse weitergibt, um damit die Schleifen in der Methode *Prozessaktion* quasi von außen zu öffnen oder zu schließen. Dadurch kann *Prozessaktion* dynamisch an die Schedulerart für kooperatives oder nicht-kooperatives Multitasking angepaßt werden.

Cycle_Scheduler ist der im Rahmen dieser Arbeit vollständig realisierte Scheduler für kooperatives Multitasking. Nach seiner Aktivierung über die Methode *Run* führt er genau einen Durchlauf des aktuellen Prozesses aus, bevor er dann zum nächsten Prozeß wechselt und dort wiederum einen ganzen Durchlauf der Methode *Prozessaktion* ausführt. Der Prozeß muß sich also kooperativ verhalten und den Prozessor wieder freigeben, nachdem ein Schleifendurchlauf beendet ist. Dies wird durch das Attribut *Zyklisch* realisiert, das hier auf *false* steht.

```
Prozess::Prozessaktion() {
  do {
    ...
  } while(Zyklisch);
}
```

Programm 1 - Dynamik im kooperativen Verhalten der Prozesse

Anders verhält es sich bei einem Scheduler für preemptives Multitasking, wie dem *M68HC11_Scheduler*[10].

Da sich ein Prozeß hier nicht kooperativ verhalten muß, kann er endlos laufen. Der Scheduler selbst entzieht hier (über einen Interrupt, der die Methode *Interrupt* aktiviert) dem Prozeß die Rechenberechtigung und sorgt dafür, daß dieser später in unverändertem Status fortgesetzt wird. Das Attribut *Zyklisch* kann hier also auf *true* stehen, und die Schleife somit geschlossen sein.

IC_Scheduler ist ein weiteres Beispiel zur Demonstration der Dynamik dieses Konzeptes. Die Methoden *Prozess_erzeugen* und *Prozess_vernichten* kapseln hier lediglich die beiden Funktionen zur Prozeßerzeugung / -vernichtung der IC-Programmiersprache. Auf diese Art könnte

[9] Eine solche Zweiteilung von Objekten in objektorientierten Echtzeitanwendungen beschreiben auch Budde und Sylla in [Bud 95]. So unterscheiden sie zwischen einer Verhaltensbeschreibung, die die Reaktion auf Signale festlegt und den gewöhnten Attributen und Methoden.

[10] Der Scheduler *M68HC11_Scheduler* wurde im Rahmen dieser Diplomarbeit nur in groben Zügen entworfen. Ein späterer Ausbau ist möglich.

jeder bereits bestehende Scheduler in das System integriert werden[11].

```
// -------------------------------------------------------------------
void IC_Scheduler::Prozess_erzeugen(Prozess *p) {
  start_process(p.*Prozessaktion());   // Funktion von IC
}

// -------------------------------------------------------------------
void IC_Scheduler::Prozess_vernichten(int pid) {
  kill_process(pid);                   // Funktion von IC
}
```

Programm 2 - Kapselung von Prozeßfunktionen bereits vorhandener Scheduler

<u>*Queue*</u>

Jeder Scheduler braucht eine Warteschlange (Ready-Queue), in der er die rechenbereiten Prozesse speichert, die auf ihre Aktivierung warten. Die Klasse *Queue* realisiert eine solche Schlange. Für jeden neuen Prozeß wird ein neues Schlangenelement (*Queue-Element*) angelegt, das einen Zeiger auf seinen Prozeß enthält. Theoretisch könnten in diesem *Queue-Element* auch weitere Scheduler-spezifische Merkmale eines Prozesses wie z.B. die PID, die Stackgröße oder Angaben zur Rechenzeit abgelegt werden. Dies würde *Queue-Element* allerdings in seiner Wiederverwendbarkeit für andere Aufgaben einschränken, so daß diese Informationen besser im Objekt *Prozess* selbst gespeichert werden.

<u>*Mutex*</u>

Mit Instanzen der Klasse *Mutex* können kritische Abschnitte für den exklusiven Zugriff gesperrt werden. Theoretisch kann es unter Verwendung des realisierten *Cycle_Scheduler* gar nicht zu Reentrant-Problemen kommen, da jede Task einmal ganz durchlaufen wird, bevor die nächste zum Zuge kommt. Beim Einsatz preemptiver Scheduler ist der Reentrant-Schutz jedoch zwingend notwendig. Zur Aufrechterhaltung dieser Flexibilität und zur Demonstration der Einfachheit wird hier der Schutzmechanismus durch ein Mutex-Objekt eingeführt. Jeder Eintritt in einen kritischen Bereich muß dabei mit der Methode *Betreten* angemeldet werden. Befindet sich gerade eine andere Task im kritischen Bereich, so muß die eintrittswillige Task so lange warten, bis die andere Task den kritischen Bereich verlassen hat. Ein Verlassen wird durch die Methode *Verlassen* signalisiert. Das Warten ist hier aktiv realisiert. Ein passives Warten mittels Semaphoren ist nicht verwirklicht. Ein solches wäre jedoch wesentlich zeiteffizienter, wenngleich es einen erhöhten Verwaltungsaufwand erfordert. Bei Einsatz eines preemptiven Schedulers wird die Realisierung eines passiven Wartens empfohlen.

[11] Das angegebene Beispiel dient nur zur Veranschaulichung und ist so nicht einsatzfähig. Die Funktion *start_process(funktionsname(parameter))* von IC erwartet keine Adresse einer Funktion, sondern den Funktionsnamen selbst als Parameter. Dies entspricht nicht dem C-Standard. Außerdem gibt es derzeit keine Möglichkeit IC mit C++ zu koppeln.

Abstraktionsebene der Nebenläufigkeit

Es steht fest, daß das Fühlen parallel zum Hören parallel zum Handeln geschehen muß. Die implementierungstechnischen Mittel in Form von Prozessen sind eingeführt und stehen zur Verfügung. Doch auf welcher Abstraktionsebene soll die Nebenläufigkeit eingeführt werden? Hierfür gibt es mehrere Möglichkeiten: So könnte sie entweder auf niederer Ebene, bei den Sensoren, oder aber auf höherer Ebene, bei den Verhalten angesetzt werden. Im ersten Fall würde das Objekt *Mikrophon* ein *aktives Objekt* werden, daß ständig zyklisch seine ermittelten Töne an das Verhalten *Akustik* meldet. Akustik wäre dann nur ein *passives Objekt*, welches jeweils als Folge der Botschaften von *Mikrophon* aktiviert werden würde, und sodann ein entsprechendes Handeln veranlassen könnte. Im zweiten Fall wären die einzelnen Verhalten *aktive Objekte*, welche ihre untergeordneten Objekte (hier die Sensoren) jeweils nach Erfordernis in Anspruch nehmen würden.

Die letztere Möglichkeit erscheint als die sinnvollere, da einzelne Verhalten durchaus auf mehrere Sensoren und Aktoren angewiesen sind, und das im ersteren Fall zu erhöhtem Synchronisationsaufwand führt. Die Verhalten erscheinen außerdem als die beständigeren Objekte, wogegen Sensoren und Aktoren eher ausgetauscht oder durch andere ersetzt werden. Deshalb wird im Anwendungsbeispiel die zweite Möglichkeit angewandt. Die Realisierung ist durch den Vererbungsmechanismus sehr leicht: Die Klasse *Verhalten* wird einfach zum Erben der Klasse *Prozess.* In der Klasse *Prozess* wird dann das individuelle Verhalten in der Methode *Prozessaktion* spezifiziert. Zum Ausschluß eines gleichzeitigen Zugriffes auf die *Verhaltensdaten* eines Verhaltensprozesses sind die Datenzugriffe jeweils durch ein Objekt *Mutex* geschützt.

Außer den Verhalten wird die *Ressourcenkontrolle* zu einem eigenständigen Prozeß. Dadurch wird das ausführende Verhalten *Gesamtverhalten* von den wahrnehmenden Verhalten entkoppelt. D.h. das *Gesamtverhalten* kann ununterbrochen seine Aktivitäten ausführen und braucht nicht auf die einzelnen, evtl. langsameren wahrnehmenden Verhalten warten. Die Synchronisation wird von der unabhängigen *Ressourcenkontrolle* erledigt.

Zur Kontrolle der Geradeausfahrt wird ein weiterer eigenständiger Prozeß *Geradenkontrolle* notwendig. Er muß permanent überprüfen, ob sich der Roboter noch geradeaus bewegt, oder ob er von seinem Kurs abweicht. Dazu bedient er sich der Rad-Enkoder, die Impulse proportional zur Radumdrehung liefern. Gegebenenfalls muß die *Geradenkontrolle* dann eine Kurskorrektur veranlassen.

3.3.2 Modellierungsentscheidungen

Alle Sensoren und Aktoren des Roboters sind Hardwarekomponenten, die der Prozessor über bestimmte Adressen ansprechen kann. Theoretisch könnte somit jeder Sensor und Aktor seine eigene Adresse als Attribut enthalten. Da jedoch verschiedene Hardwarekomponenten oftmals mehr als eine, und somit unterschiedlich viele Adressen zur Adressierung haben (ein Beispiel hierfür wären Adressen für Statusregister und Ports), läßt sich dies nicht so ganz allgemein beschreiben. Außerdem müßten bei lokaler Haltung der Adressen bei einer Portierung auf einen anderen Prozessor sämtliche Adressen lokal aktualisiert werden. Viel sinnvoller ist es, sämtliche Adressen in einer Klasse *Hardware* zusammenzufassen. Jeder Sensor und Aktor (und ggf. weitere Hardwarekomponenten) sind dann Erben der Klasse *Hardware*, und haben dadurch Zugriff auf alle Adressen. Eine Portierung erfordert damit nur noch den Austausch einer einzigen Klasse, die gleichzeitig auch die Zugriffsfunktionen auf die Adressen enthält.
Eine weitere Klasse *Analogsensor* übernimmt die Zuteilung der Analogsensoren auf die physikalischen Ports des A/D-Wandlers im Mikrocontroller. Das Modell (ohne das Multi-Prozeß-Modul) ist somit zu folgender Klassenstruktur gewachsen:

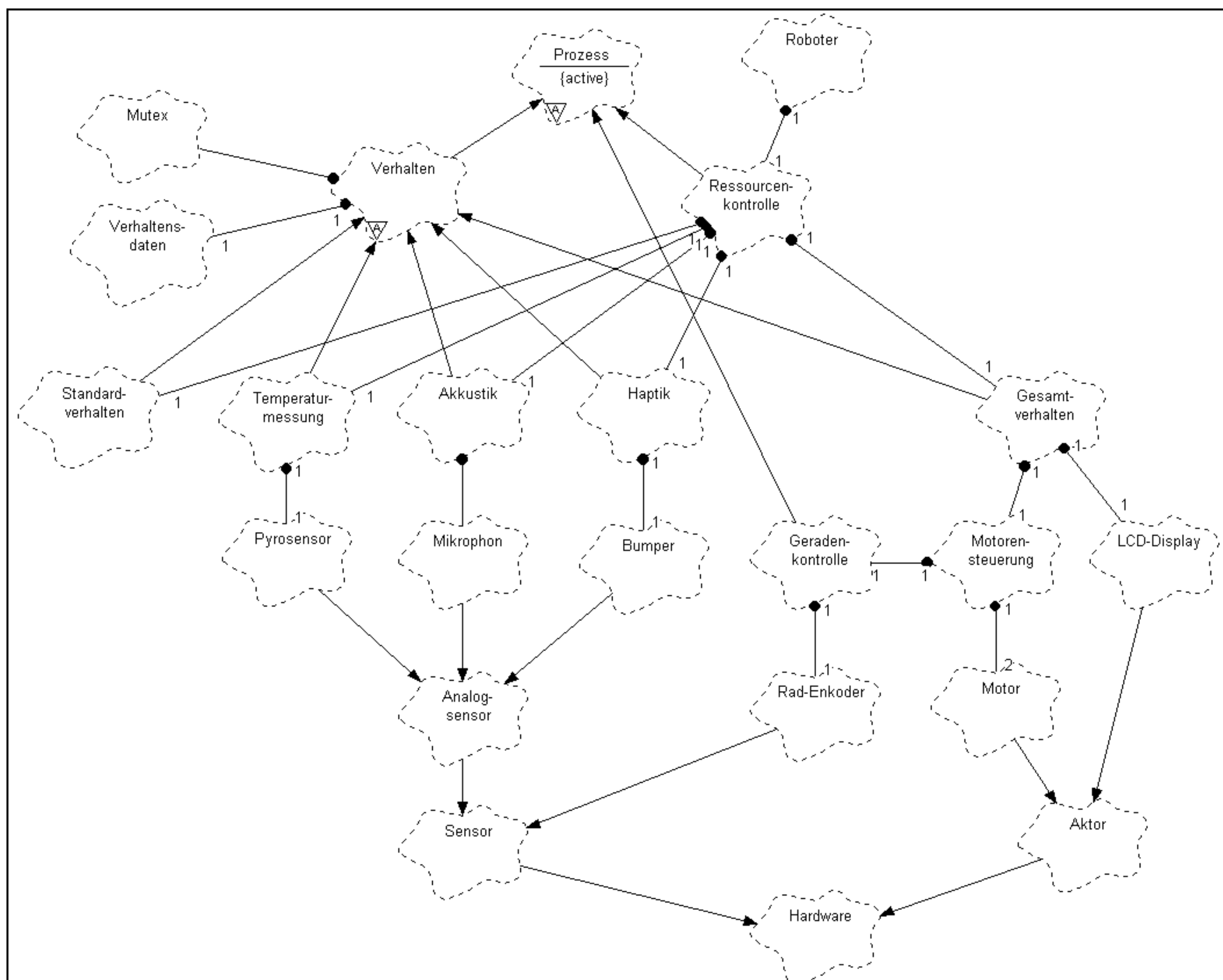


Abbildung 9 - Klassendiagramm Modell A

Betrachtet man das obige Klassendiagramm, so stellt sich die Frage: Wozu ist eigentlich die

Klasse *Roboter* noch notwendig ist. Die Antwort ist: Zu gar nichts! Das Modell (A) stellt in seiner Gesamtheit den Roboter dar. Die Aufgaben sind auf die einzelnen Komponenten verteilt, und jede Komponente hat ihre eigenen notwendigen Unterkomponenten. Rein funktionell hat die Klasse Roboter keine Aufgabe und ist somit überflüssig. Dennoch soll ein Modell möglichst exakt der Realität entsprechen, und in der Realität ist der Roboter als physikalisches Objekt vorhanden, das alle seine Einzelteile (wie Motoren, Sensoren, usw.) enthält. Es wäre also auch folgende Modellierung (Modell B) möglich:

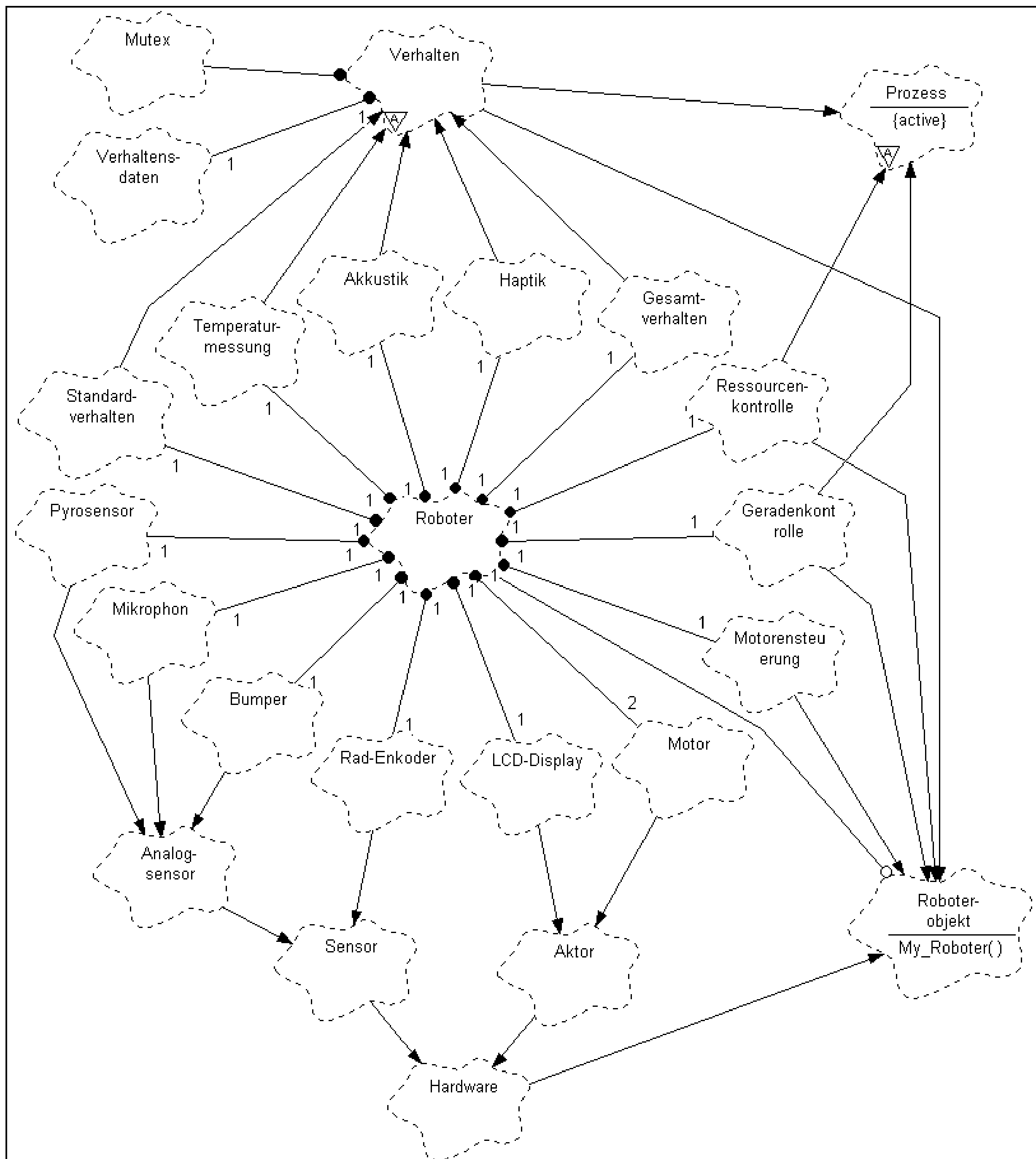


Abbildung 10 - Klassendiagramm Modell B

Wird die Software jedoch auf diese Art und Weise realisiert, so kennt *Roboter* zwar alle seine Komponenten, eine Komponente kennt jedoch nicht seinen *Roboter*. Dies ist jedoch notwen-

dig, damit eine Komponente indirekt über Roboter einer anderen Komponente Botschaften schicken kann. Zur Lösung ist (wie in Abbildung 10 gezeigt) die Klasse *Roboterobjekt* eingeführt, die einen Zugriff auf die Klasse *Roboter* ermöglicht. Jede einzelne Komponente ist Erbe der Klasse *Roboterobjekt,* und somit ist es jeder einzelnen Komponente ermöglicht, indirekt mit jeder anderen Komponente zu kommunizieren.

Die Vor- und Nachteile der beiden Designmöglichkeiten sind in Tabelle 6 erörtert:

	Modell A Modellierung ohne Klasse *Roboter* (Gesamtmodell stellt Roboter dar)	**Modell B** Modellierung mit Klasse *Roboter* als zusammenfassendes Objekt
Vorteile	• höhere Sicherheit dadurch, daß jedes Objekt nur Zugriff auf diejenigen Objekte hat, die es tatsächlich zur Ausführung seiner Aufgabe benötigt; • funktionales Zusammenwirken der Komponenten ist vollständiger beschrieben; die Funktionalität ist dadurch besser zu verstehen; • einfachere Implementierung durch direkte Referenzierung;	• Modell entspricht exakter der physikalischen Wirklichkeit;
Nachteile	• einzelne Komponenten sind von keiner Stelle aus direkt ansprechbar (das LCD-Display müßte beispielsweise über den Umweg *Ressourcenkontrolle - Gesamtverhalten - LCD-Display* angesprochen werden);	• Sicherheitseinbußen dadurch, daß jedes Objekt des Roboters Zugriff auf alle anderen Komponenten des Roboters hat, obwohl diese teilweise gar nicht notwendig sind; • funktionales Zusammenwirken der Komponenten ist unvollständig beschrieben und dadurch schwieriger zu verstehen; • aufwendigere Implementierung dadurch, daß alle Objekte indirekt über das Objekt *Roboter* referenziert werden müssen;

Tabelle 6 - Vor- und Nachteile der beiden Modelle im Vergleich

Der einzige Nachteil von Modell A tritt jedoch in einem ES so gar nicht auf, da bei richtigem Design jede Komponente schon im Modell den Zugriff auf ihre Komponenten ermöglicht bekommt. Es ist also nicht notwendig, daß zwei voneinander unabhängige Komponenten direkt miteinander kommunizieren können. Anders würde es sich in einem nicht-eingebetteten System verhalten, wo beispielsweise ein Benutzer von der Tastatur aus dem Roboter ein Kommando *„Text ‘HALLO’ anzeigen“* übermitteln möchte. Allein die verbleibende Tatsache, daß Modell B bei Betrachtung des Klassendiagramms eher nach Roboter und seinen Teilen aussieht, hebt nicht die Nachteile der Sicherheitseinbußen durch freie, globale, sozusagen ungeschützte Zugriffsmöglichkeiten auf, die zur Objektkommunikation notwendig sind. In Modell B ist in keiner Weise geklärt, wer welche Komponente ansprechen darf und wer nicht. Dies

würde im übertragenen Sinn globalen Variablen sehr nahe kommen und viele Vorteile der OO wären dadurch aufgehoben. Deshalb ist Modell A klar zu bevorzugen und im weiteren wird nurmehr Modell A betrachtet.

3.4 Programmierung

3.4.1 Programmiersprachen

Zur Implementierung der Roboter-Steuerung wurden mehrere Programmiersprachen erwogen. Leider war kein Compiler bekannt, der eine objektorientierte Programmiersprache direkt in ausführbaren Maschinencode des M68HC11 übersetzten könnte. Genauso war kein Precompiler im finanziellen Rahmen dieser Diplomarbeit zu erhalten, um z.B. C++-Code zunächst in C-Code zu übersetzen, der anschließend mit einem C-Compiler auf ausführbaren Maschinencode des M68HC11 weiter übersetzt werden könnte.[12] Auch der Einsatz von *Eiffel* wurde erwogen, das bereits einen integrierten Precompiler nach C enthält. Neben der ausschließlichen Verfügbarkeit auf einer UNIX-Plattform und des zusätzlichen Aufwandes zum Erlernen der Sprache, standen vor allem die Größe des erzeugten Codes im Widerspruch zu dessen Verwendung. So bemerken schon Fiedler / Rix / Zöllner in [Fied 91, S. 107] im Bezug auf den Einsatz von *Eiffel* in der Automatisierungstechnik: *„... kleinere Systeme bleiben aufgrund der umfangreichen Objektprogramme außen vor. Damit ist die Anwendung innerhalb der Automatisierungstechnik stark eingeschränkt, da hier zumeist auf Einplatinenrechner mit beschränkten Ressourcen zurückgegriffen wird.“* Diese Erkenntnis gilt um so mehr für die beschränkten Ressourcen von ES.

All diese Einschränkungen führten dazu, daß eine rein objektorientierte Lösung komplett mit Borland C++ 4.52 erstellt wurde. Diese kann wegen dem Fehlen eines Compilers derzeit nur auf einem PC simuliert werden. Die Zugriffe auf die Hardware des Roboters (Speicher / IO) werden dafür durch Funktionsaufrufe abgefangen und emuliert. Zum funktionalen Beweis der objektorientierten Lösung wurde die Steuerung zusätzlich prozedural mit der Programmiersprache IC implementiert und auf dem Roboter eingesetzt.

IC ist eine speziell für den M68HC11 entwickelte Programmiersprache, bestehend aus einer Untermenge von C-Sprachelementen. Sie stellt zusätzlich Mechanismen zur dynamischen Prozeßverwaltung zur Verfügung. Außerdem ist ein Kommandozeilen-Interpreter und -Debugger integriert. IC übersetzt den Quellcode zunächst in einen Stack-basierten Pseudo-Code, der anschließend von einem Interpreter ausgeführt wird. Die von den Entwicklern durch diesen ungewöhnlichen Ansatz erreichte bessere Portierbarkeit und einfachere Multi-

[12] Ein solcher Precompiler wäre z.B. *Cfront* von *AT&T*.

tasking-Realisierbarkeit geht leider stark auf Kosten der Ausführungsgeschwindigkeit. So wurde eine gegenüber Assembler-Code um Faktor 75 langsamere Ausführungsgeschwindigkeit gemessen[13]. Weiterführende Informationen über IC sind in [Jon 95, S. 39 ff] und [Jon 93, S. 39-41] zu finden.

3.4.2 Verhalten

Jedes *Verhalten* ist ein *Prozess* und wird durch den Konstruktor

```
Prozess::Prozess(Scheduler *scheduler) {
Zyklisch = scheduler->Zyklustyp();   // Zyklustyp von Sched. übernehmen
  scheduler->Prozess_erzeugen(this); // Prozess beim Scheduler anmelden
}
```

Programm 3 - Konstruktor von Prozess

automatisch bei seinem Scheduler angemeldet. Außerdem besitzt jedes Verhalten über *Prozess* automatisch eine Methode *Prozessaktion*, in der das reaktive Verhalten beschrieben wird. Dabei hat jedes Verhalten seine eigenen Verhaltensdaten

```
class Verhaltensdaten {
  public:
    Aktionen  Aktion;               // Fahrkommandos
    char      *Displaytext[32];     // Anzeigetext
    int       Displaywert;          // Anzeigewert
    float     Geschwindigkeit;      // Geschwindigkeit
};
```

Programm 4 - Definition von Verhaltensdaten

unter dem Namen *Zustand*. Zur Beschreibung des reaktiven Verhaltens werden nun in *Prozessaktion* individuell die Sensorwerte abgefragt und verarbeitet. Die Reaktion daraus wird über Zustand formuliert, der dann von der *Ressourcenkontrolle* weiterverarbeitet wird. Somit wird jedes Verhalten eine *Prozessaktion* mit folgender Struktur haben:

```
void Verhalten_X::Prozessaktion() {
  do {
    // Sensorwerte lesen
    // Sensorwerte verarbeiten
    // Reaktion vormulieren:
      Datenschutz.Betreten();
        *Zustand.Displaytext = ”AAA“;
        Zustand.Displaywert = BBB;
        Zustand.Aktion = CCC;
        Zustand.Geschwindigkeit = DDD;
        Status = true;
      Datenschutz.Verlassen();
  } while(Zyklisch);
}
```

Programm 5 - Struktur der Prozessaktion eines Verhaltens

Über die Methode *Daten* kann der angeforderte *Zustand* von der *Ressourcenkontrolle* abgefragt werden.

Das *Gesamtverhalten* als ausführende Einheit hat dagegen folgende *Prozessaktion*:

[13] Die Messung bezieht sich auf einen weiter unten beschriebenen Vergleich in der Ausführungsgeschwindigkeit an einer Routine zur Frequenzerkennung.

```
void Gesamtverhalten::Prozessaktion() {
  do {
    Datenschutz.Betreten();
      My_Motorensteuerung.Ausfuehren(Zustand.Aktion,
                                     Zustand.Geschwindigkeit);
      My_LCDDisplay.Anzeigen(*Zustand.Displaytext, Zustand.Displaywert);
    Datenschutz.Verlassen();
  } while(Zyklisch);
}
```

Programm 6 - Prozessaktion von Gesamtverhalten

3.4.3 Ressourcenkontrolle

Die *Ressourcenkontrolle* ist für die Vergabe der Ressourcen nach den Prioritäten der einzelnen Verhalten zuständig. Die Prioritätsvergabe ist implizit durch die Reihenfolge der Datenzuweisung realisiert (siehe Programm 7). Bei der Datenübernahme von Verhaltensdaten durch das Gesamtverhalten (siehe Programm 8) werden lediglich tatsächliche Ressourcenanforderungen überschrieben. Leere Anforderungen werden übergangen. Durch diesen Mechanismus entsteht ein hierarchisch kombiniertes Gesamtverhalten.

```
void Ressourcenkontrolle::Prozessaktion() {
  do {
    My_Gesamtverhalten.Datenuebernahme(My_Standardverhalten.Daten());
    if (My_Temperaturmessung.Aktiv())
       My_Gesamtverhalten.Datenuebernahme(My_Temperaturmessung.Daten());
    if (My_Akustik.Aktiv()) My_Gesamtverhalten.Datenuebernahme(My_Akustik.Daten());
    if (My_Haptik.Aktiv()) My_Gesamtverhalten.Datenuebernahme(My_Haptik.Daten());
  } while(Zyklisch);
}
```

Programm 7 - Implizite Prioritätsvergabe durch Zuweisungsreihenfolge

```
void Gesamtverhalten::Datenuebernahme(Verhaltensdaten *Quelle) {
  do {
    Datenschutz.Betreten();
      if (Quelle->Aktion != KLeer) Zustand.Aktion = Quelle->Aktion;
      if (*Quelle->Displaytext != "") {
        *Zustand.Displaytext = *Quelle->Displaytext;
        Zustand.Displaywert = Quelle->Displaywert;
      }
      if (Quelle->Geschwindigkeit != 0.0) Zustand.Geschwindigkeit =
        Quelle->Geschwindigkeit;
    Datenschutz.Verlassen();
  } while(Zyklisch);
}
```

Programm 8 - Methode Gesamtverhalten::Datenuebernahme

In der gezeigten Realisierung müssen der *Ressourcenkontrolle* ihre einzelnen Verhalten namentlich bekannt sein. Bei Erweiterungen um ein zusätzliches Verhalten müßten also auch Änderungen an der Klasse *Ressourcenkontrolle* vorgenommen werden. Um die Erweiterbarkeit dynamischer zu gestalten, wird hier folgende Verbesserung vorgeschlagen: Die Ressourcenkontrolle könnte um eine Verwaltungseinheit ihrer Verhalten ergänzt werden, bei der sich einzelne Verhalten dynamisch unter expliziter Angabe ihrer Priorität anmelden können. Die Ressourcenzuordnung könnte somit anonym, lediglich aufgrund der Prioritätsinformation erfolgen.

3.4.4 Motorensteuerung

Die Motorensteuerung ist dafür zuständig, die Fahrkommandos in Kommandos an die einzelnen Motoren umzuwandeln. Dabei nimmt die Methode *Motorensteuerung::Ausfuehren* die Fahrkommandos entgegen und ruft dann die entsprechenden Methoden innerhalb der Klasse auf. Zur besseren Austauschbarkeit der Motoren sind diese Methoden dabei nochmals in „öffentliche Motoraktionen“ und in „private Grundfunktionen“ aufgeteilt, wobei die öffentlichen Motoraktionen die privaten Grundfunktionen benutzen. So benutzt beispielsweise die Methode *Linksdrehung* die Methode *Links*, die letztendlich die beiden Motoren mit entsprechenden Geschwindigkeitswerten ansteuert (Aufruf von *LMotor.Go* bzw. *RMotor.Go*).

```
void Motorensteuerung::Linksdrehung(float v) {
  Links(v, Drehdifferenz);
  My_Timer.Sleep(Drehzeit);
}
```

Programm 9 - Öffentliche Motoraktion Motorensteuerung::Linksdrehung

```
void Motorensteuerung::Links(float v, float c) {
  float k;
  My_Geradenkontrolle.Setze_inaktiv();
  My_Geradenkontrolle.Setze_Geschwindigkeit(v);
  k = My_Geradenkontrolle.Korrekturfaktor();
  LMotor.Go(v + c - k);
  RMotor.Go(v - c + k);
}
```

Programm 10 - Private Grundfunktion Motorensteuerung::Links

```
void Motor::Go(float geschwindigkeit) {
  if (geschwindigkeit >= 0) BitSet(PORTD, DIR_Mask); // Vorwaerts
  else BitClear(PORTD, DIR_Mask);                    // Rueckwaerts
  if (Abs(geschwindigkeit) < 1) BitClear(OC1D, PWM_Mask); // Motor
                                 abschalten
  else BitSet(OC1D, PWM_Mask); // Motor von OC1 steuern
  if (Abs(geschwindigkeit) > 99) geschwindigkeit = 99.0;
  PokeWord(TimerIndex, (int) (655.56 * geschwindigkeit));
}
```

Programm 11 - Methode Motor::Go

3.4.5 Geradenkontrolle

Da die Motoren der beiden Räder unterschiedliche Übertragungskennlinien haben, haben gleiche Ansteuerungswerte nicht exakt die gleiche Radgeschwindigkeit zur Folge. Um einen geraden Fahrkurs zu erhalten ist es aber notwendig, daß beide Räder einen exakt gleichen Weg zurücklegen. Die Radgeschwindigkeit kann hierzu indirekt über die beiden Shaft-Encoder gemessen werden, die pro 1/32-Radumdrehung je einen Impuls liefern. Ist die Anzahl der pro Zeiteinheit gemessenen linken und rechten Impulse gleich, dann befindet sich der Roboter auf einem geraden Fahrkurs. Auf die gesamte zurückgelegte Strecke bezogen, müssen also die Summe der linken Impulse gleich der Summe der rechten Impulse sein. Zur Regelung dieser I-Regelstrecke wird ein digitaler PI-Regler mit folgender Übertragungsfunktion verwendet:

$$y(i) = y(i-1) + k\left(1 + \frac{T}{T_i}\right)e(i) - ke(i-1)$$

mit: i = momentaner Meßzeitpunkt;
i-1 = voriger Meßzeitpunkt;
e = Regelabweichung;
T = Abtastzeit;
T_i = Zeitkonstante der Strecke;
k = Verstärkungsfaktor;

Der Prozeß *Geradenkontrolle* (siehe Programm 12) stellt ständig den mit obiger Formel errechneten Korrekturfaktor aus der Abweichung der linken und rechten Summe von Impulsen für die *Motorensteuerung* bereit. Damit jedoch beabsichtigte Kurven von der *Geradenkontrolle* nicht als Kursabweichung fehlinterpretiert werden, muß die *Motorsteuerung* bei einer Kurvenfahrt die *Geradenkontrolle* deaktivieren und anschließend wieder aktivieren. Dies geschieht über die Methoden *Setze_inaktiv* bzw. *Setze_aktiv*. Wenn allerdings die Geradenkontrolle nach einer Kurvenfahrt wieder aktiviert wird, dann sind die letzten von den Enkodern gelieferten Werte falsch und dürfen nicht berücksichtigt werden. Zusätzlich muß die *Motorensteuerung* jede Geschwindigkeitsänderung an die *Geradenkontrolle* weitermelden, da die Geschwindigkeit mit in den Korrekturfaktor eingeht.
Im folgenden Programmausschnitt ist die Methode *Prozessaktion* der *Geradenkontrolle* gezeigt. Gleichzeitig wird darin der Schutz von Variablen durch die Mutex-Objekte ersichtlich:

```
void Geradenkontrolle::Prozessaktion() {
  long differenz = 0;               // aktuelle Differenz
  long differenz_alt = 0;           // vorige Differenz
  float ti = 0.0;                   // Zeitkonstante der Strecke
  float t = 1.0;                    // Meßintervall
  float k = 0.2;                    // Verstärkungsfaktor
  bool verwerfen = false;           // Kurvenimpulse unterdrücken
  do {
    L_Clicks = Enkoder.LImpulse();   // aktuelle Impulse lesen
    R_Clicks = Enkoder.RImpulse();   // "
    if (Aktiv) {                     // nur Geradeauslauf berücksicht.
      if (verwerfen) verwerfen = false;
      else {
        L_Sum = L_Sum + L_Clicks;    // Gesamtstrecke links
        R_Sum = R_Sum + R_Clicks;    // Gesamtstrecke rechts
        differenz = L_Sum - R_Sum;   // Differenz bez. Gesamtstrecke
        Geschwindigkeitsschutz.Betreten();
          if (Geschwindigkeit != 0) ti = L_Clicks / Geschwindigkeit;
        Geschwindigkeitsschutz.Verlassen();
        if (ti != 0) {
          Korrekturschutz.Betreten();
            Korrektur=Korrektur+k*(1+(t/ti))*differenz-k*differenz_alt;
          Korrekturschutz.Verlassen();
          differenz_alt = differenz; // Differenz merken
        }
      }
    }
    else verwerfen = true;
    My_Timer.Sleep(t);               // Meßintervall warten
} while(Zyklisch);
}
```

Programm 12 -Prozeßmethode zum Berechnen der Geradenkorrektur

Der Korrekturwert wird der Geradenkontrolle über die folgende Methode zur Verfügung gestellt:

```
float Geradenkontrolle::Korrekturfaktor() {
  float wert;
  Korrekturschutz.Betreten();
    wert = Korrektur;
  Korrekturschutz.Verlassen();
  return(wert);
}
```

Programm 13 - Methode zur Übermittlung des Korrekturfaktors

In der Praxis stellte sich allerdings heraus, daß der Roboter selbst bei guter Ausregelung auf eine Differenz von Null nicht exakt geradeaus fährt. Es wird angenommen, daß der Grund dafür die elastischen Räder sind. Bei unterschiedlich starkem Druck auf die einzelnen Räder ändert sich dadurch der Radumfang und somit die zurückgelegte Strecke pro Radumdrehung. Da diese Störgröße nicht mit den Rad-Encodern erfaßbar ist, kann diese Abweichung auch nicht ausgeregelt werden. Es blieb nur die Möglichkeit einen empirisch ermittelten Korrekturwert zu bestimmen. So sollte bei mittlerer Geschwindigkeit das linke Rad mit Faktor 1.1 gegenüber dem rechten Rad angesteuert werden.

3.4.6 Tonerkennung

Der Roboter soll auf akustische Signale reagieren und dadurch extern steuerbar sein. Hierzu ist es notwendig, daß er verschiedene akustische Kommandos voneinander unterscheiden kann. Prinzipiell stehen die Unterscheidung der Lautstärke (Amplitude), der Tonlänge (Zeit)

oder der Tonhöhe (Frequenz) zur Auswahl.

Als Akustiksensor enthält der Roboter ein Mikrophon, dessen Signale zunächst über einen Verstärker (LM386N-1) verstärkt werden. Diese verstärkten Analogsignale sind auf den Eingangskanal PE2 des Mikrocontrollers geführt. Über den internen A/D-Wandler werden die Analogsignale mit einer Auflösung von 256 Binärwerten digitalisiert. Zur Umsetzung werden 32 E-Takte benötigt, was bei einer Taktfrequenz von 2.000 MHz eine Umsetzzeit von 16 µs ergibt. Im Mehrkanalbetrieb arbeitet der A/D-Wandler jedoch vier Eingangsports sequentiell hintereinander ab, so daß ein Kanal erst nach 128 E-Takten erneut bearbeitet wird. Zur Tonerkennung steht also alle 64 µs ein Digitalwert zwischen 0 und 255 zur Verfügung steht.

<u>*Lautstärke*</u>

Zur Ermittlung der Lautstärke wird das anliegende Meßsignal über einen gewissen Zeitraum abgetastet und ausgewertet. Ist kein Ton vorhanden, so schwingt das Meßsignal mit undefinierter Frequenz und kleiner Amplitude um die Grundlinie (Abbildung 11). Diese ist auf den Mittelwert des A/D-Wertebereiches gelegt und ergibt somit einen digitalen Mittelwert von 128. Sobald das Mikrophon Töne aufnimmt, werden die Amplituden mit zunehmender Lautstärke größer (Abbildung 12). Eine Mittelwertbildung über den Beobachtungszeitraum würde aber Null (bzw. hier 128) ergeben, da die Schwingung achsensymmetrisch ist. Ein repräsentativer Wert für die Lautstärke entsteht erst, nachdem das Meßsignal auf Null normiert und daraus der Absolutwert gebildet wird (abs(analogwert - 128); siehe Abbildung 13). Die Methode *Mikrophon::DurchschnittsLautstärke* (Programm 14) erledigt diese Aufgabe.

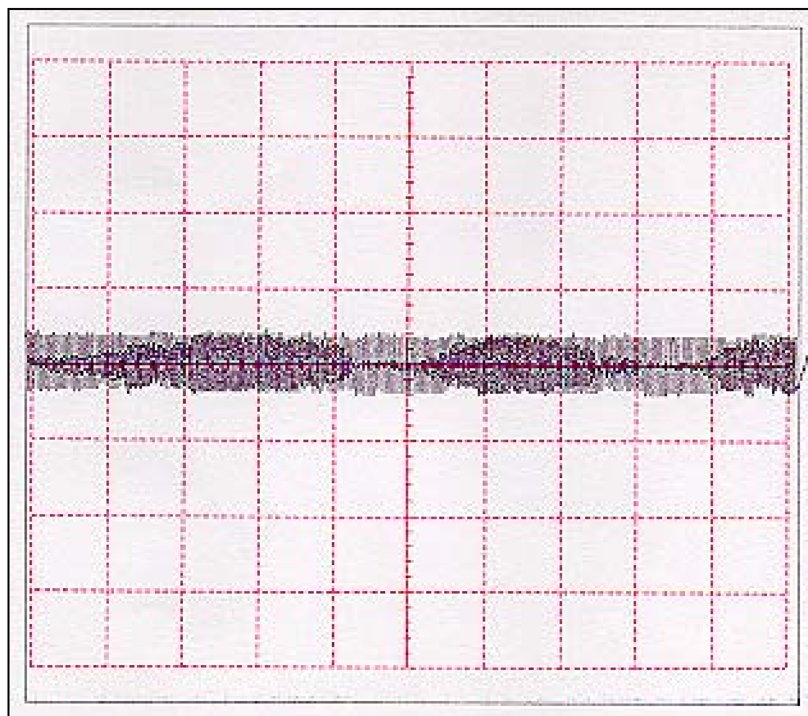


Abbildung 11 - Verrauschtes Signal ohne definierten Ton

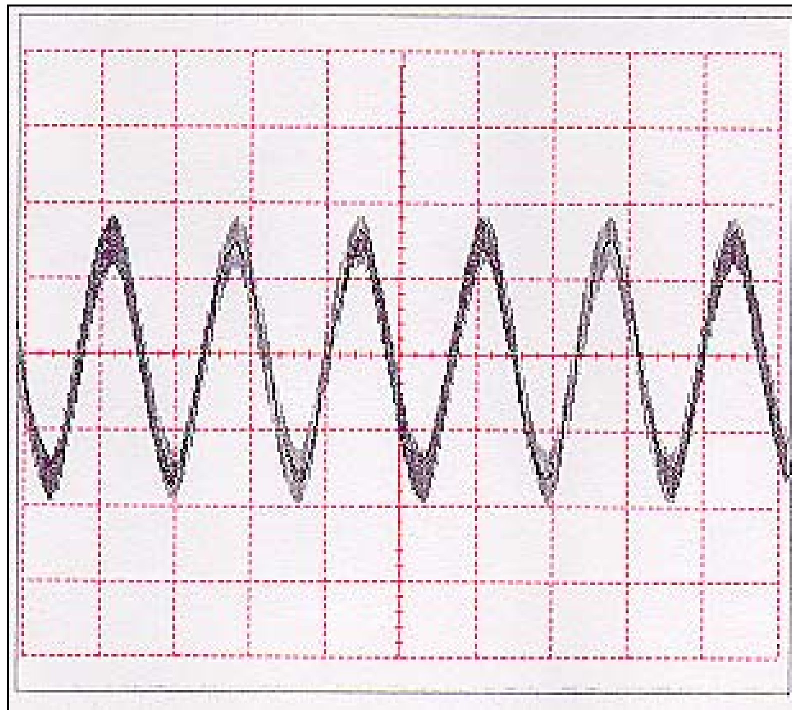


Abbildung 12 - Signal mit definiertem Ton

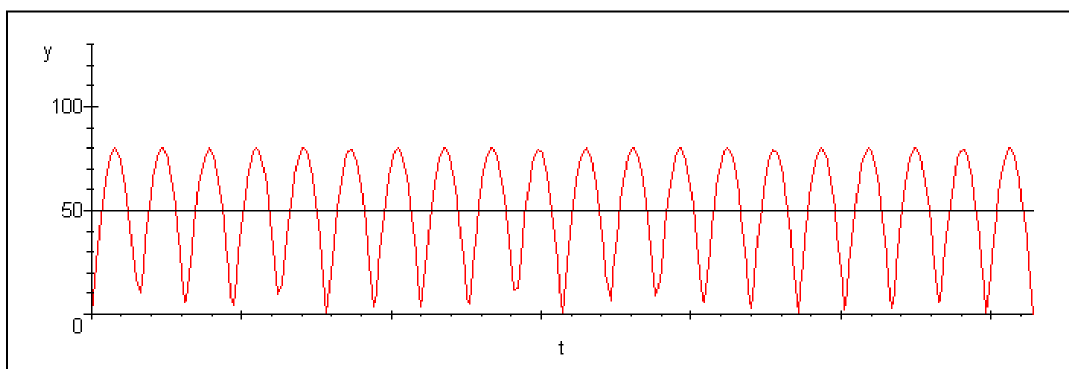


Abbildung 13 - Signal mit Mittelwert als Lautstärke

```
int Mikrophon::DurchschnittsLautstaerke() {
  int i;
  int summe = 0;
  for (i = 0; i < Abtastintervall; i++) {
    summe = summe + Abs(Analogwert() - 128);
  }
  return (summe / Abtastintervall);
}
```

Programm 14 - Methode Mikrophon::DurchschnittsLautstaerke

Tonlänge

Zur Erkennung von verschiedenen Tonlängen ist es zunächst notwendig zu definieren, wann ein Ton vorliegt und wann nicht. Hierzu dient ein Schwellwert als Grenze für die minimale Lautstärke, die noch als Ton gilt. Ein ständig vorhandenes Grundrauschen wird somit unterdrückt, und es werden nur tatsächlich vorhandene Töne ab einer gewissen Lautstärke herausgefiltert. Die Methode *Mikrophon::Ton* (Programm 15) löst diese Aufgabe unter Zuhilfenahme der oben besprochenen Methode *Mikrophon::DurchschnittsLautstaerke* und liefert nur noch *true* für einen vorhanden Ton bzw. *false* für einen nicht vorhandenen Ton zurück.

```
bool Mikrophon::Ton() {
  int geraeusch;
  geraeusch = DurchschnittsLautstaerke();
  if (geraeusch > Tonschwelle) return(true);
  else return(false);
}
```

Programm 15 - Methode Mikrophon::Ton

Soll in einer Multitasking-Umgebung eine Zeitintervall absolut bestimmt werden, so muß die Zeit-Bezugsgröße unabhängig von der Laufzeit des aufrufenden Tasks sein. Eine Möglichkeit hierfür ist ein Zähler, der interruptgesteuert und somit kontinuierlich weitergezählt wird. Die Auflösung der Tonlänge (also die minimal erkennbare Länge eines Tones) ist natürlich abhängig davon, wieviel Rechenzeit die Task bekommt. Bekommt die Task genügend Rechenzeit, so ist die maximale Auflösung gleich der Frequenz, mit der der Zähler erhöht wird. Bekommt die Task nicht genügend Rechenzeit vom Scheduler zugeteilt oder ist die Laufzeit für einen Task-Zyklus höher als die Zählfrequenz, so ist die maximale Auflösung gleich der Task-Wiederholzeit.

Die nachfolgende Methode *Mikrophon::Tondauer* (Programm 16) mißt die Zeit in Millisekunden, während der ein als Ton erkanntes Signal ansteht. *My_Timer.MSeconds* stellt dabei die Anzahl der seit dem letzten Reset verstrichenen Millisekunden zur Verfügung. Ist die Wiederholzeit der aufrufenden Task kleiner oder gleich einer Millisekunde, beträgt die maximale Auflösung der Tonlänge hierbei eine Millisekunde.

```
long Mikrophon::Tondauer() {
  bool ein = false;
  long start = 0;
  long stop = 0;
  while(stop == 0) {
    switch (ein) {
      case false: if (Ton()) {
                    start = My_Timer.MSeconds();  // Startwert setzen
                    ein = true;
                  }
                  break;
      case true:  if (!Ton()) {
                    stop = My_Timer.MSeconds();   // Stopwert setzen
                    ein = false;
                  }
                  break;
    }
  }
  return(stop-start);
}
```

Programm 16 - Methode Mikrophon::Tondauer

Tonhöhe

Zur Unterscheidung von verschiedenen Tonhöhen muß die Frequenz eines Tones gemessen werden, was sehr hohe Anforderungen sowohl an die Hardware als auch an die Software stellt. Anhand der folgenden Abbildung soll die programmtechnische Frequenzerkennung erläutert werden:

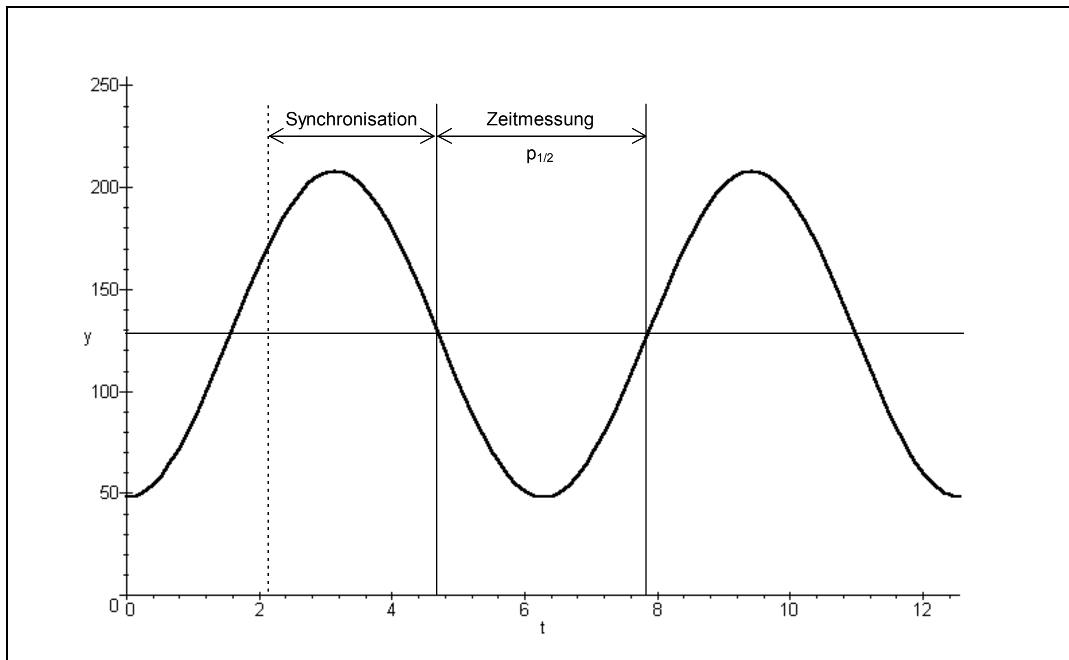


Abbildung 14 - Frequenzerkennung

Ein Ton hat einen sinusförmigen Schwingungsverlauf. Die Frequenz des Tones ergibt sich aus dem Kehrwert der Periodendauer. Während einer Periodendauer passiert das Signal zwei Mal den Wert Null (hier digital 128). Diese beiden Nulldurchgänge müssen programmtechnisch erkannt, und die Zeit dazwischen gemessen werden. Die Frequenz ergibt sich dann mit $f = \left(2 \cdot p_{1/2}\right)^{-1}$. Die Methode zur Frequenzerkennung in Programm 17 realisiert dies auf folgende Art und Weise: Zunächst wird auf den ersten Nulldurchgang gewartet (siehe Abbildung 14: *Synchronisation*). Passiert das Signal diesen ersten Nulldurchgang, so wird der aktuelle Zählerstand festgehalten. Dann wird auf das nächste Passieren des Nulldurchgangs gewartet. Tritt dieser ein, so wird erneut der Zählerstand ausgelesen. Die Differenz der beiden, multipliziert mit der Zeitdauer eines Zählschrittes ergibt die halbe Periodendauer. Diese Messung wird mehrfach durchgeführt und das Ergebnis aus dem Mittelwert gebildet. Neben der Überlaufkontrolle des Zählregisters ist es vor allem wichtig zu erkennen, ob die gemessene Frequenz tatsächlich von einem Ton her stammt, oder ob es nur eine zufällige Frequenz eines undefinierten Rauschens ist. Eine solche Kontrolle könnte prinzipiell in die Frequenzerkennung mit eingebaut werden. Da die Frequenzerkennung allerdings zeitkritisch ist (und jede verzichtbare Rechenoperation unnötig auf Kosten der Frequenzbandbreite geht) wird eine Lösung realisiert, in der die Frequenzerkennung nur gestartet wird, wenn ein definierter Ton vorliegt. Zu dieser Überprüfung wird die Methode *Mikrophon::Ton* verwendet.

```
float Mikrophon::Frequenz() {
  long start, stop, summe, differenz;
  int i, zaehler;
  float schnitt, frequenz;
  zaehler = 0;
  summe = 0;
  if (Ton()) {                        // Frequenzerkennung nur bei Ton
    for (i=0; i<20; i++) {
      while(Analogwert() >= 128) {;}    // Synchronisation
      while(Analogwert() < 128) {;}     // 1. Nulldurchgang
      start = My_Timer.Ticks();
      while(Analogwert() >= 128) {;}    // 2. Nulldurchgang
      stop = My_Timer.Ticks();
      differenz = stop - start;
      if (differenz>=0) {summe=summe+differenz; zaehler++;} //
                                      Überlaufschutz
    }
    if (zaehler > 0) {
      schnitt = summe / zaehler;
      frequenz = 1 / (2 * schnitt * 0.0000005); // 1 Imp. = 500 ns; 2
                                      Halbwellen
    }
    else frequenz = 0;
  }
  else frequenz = 0;
  return(frequenz);
}
```

Programm 17 - Methode Mikrophon::Frequenz

Zur exakten Frequenzerkennung muß nach dem Abtasttheorem von Shannon[14] mindestens mit der doppelten Signalfrequenz abgetastet werden, damit keine Alliazing-Effekte auftreten. Bei Echtzeitverarbeitung der Signale muß die Verarbeitungsgeschwindigkeit des Prozeßrechners die Abtastfrequenz nochmals übersteigen, damit Zeit für die Signalverarbeitung bleibt. Das langsamste Glied in der Kette bestimmt also die maximal erkennbare Frequenz des Eingangssignals. Die oben beschriebene Methode ist auch als Funktion mit IC realisiert. Dabei wurde mit der folgenden Meßroutine eine Zeit zum Auslesen und Vergleichen des Analogports sowie zur Zeitermittlung (grau unterlegt) von 337 µs ermittelt.

```
start = (long) peekword(0x100e);
  while(peek(0x1031) < 0) {;}
stop = (long) peekword(0x100e);
```

Programm 18 - IC-Funktion zur Ermittlung der Abtastzeit

Dieser Wert liegt dabei wesentlich über den 64 µs, die der A/D-Wandler zur Umsetzung benötigt[15]. Unter Berücksichtigung des Shannon-Abtasttheorems wäre dabei lediglich eine maximal erkennbare Frequenz von $(2 * 2 * 337\ \mu s)^{-1} = 742$ Hz möglich. Dieser niedrige Wert, der sich durch die interpretierende Ausführung mit IC ergibt, ist sehr unbefriedigend. Aus diesem Grund wurde die Frequenzerkennung auch in Assembler (siehe Programm 19) realisiert. Dabei werden für das Auslesen und Vergleichen des Analogwertes sowie zur Zeitermitt-

[14] Abtasttheorem von Shannon: $f_{Verarbeitung} > f_{Abtastung} > 2 \cdot f_{Signal}$

[15] Wie Messungen gezeigt haben, würde die Ausnutzung des Einkanalbetriebes, indem periodisch vier Register hintereinander mit einer Umsetzzeit von 16 µs beschrieben werden, im vorliegenden Fall der Echtzeitverarbeitung keine Vorteile bringen. Vielmehr würde der Aufwand zum Vergleich und zur Verarbeitung dieser vier Register den Zeitvorteil wieder vollständig verschlingen.

lung (wiederum grau unterlegt) lediglich noch 9.5 µs benötigt, was 35 Mal schneller ist und somit eine maximal meßbare Frequenz von $(2 * 2 * 9.5 \mu s)^{-1}$ = 26315 Hz zuläßt.

		Ausführungszeit in Zyklen[16]
BASE EQU $1000	Basisadresse zur indizierten Adressierung	
TCNT EQU $100E	Timer Count Register	
ADR1 EQU $1031	A/D Result Register 1	
ORG MAIN_START		
start FDB #0		
stop FDB #0		
subroutine_freq:		
PSHX	X-Register sichern	2
LDX #BASE	Offset der Registeradressen	5
* Synchronisation:		
x1: LDAA ADR1,x	Wert lesen	4
SUBA #128	Warten, solange >= 128	2
BPL x1	"	3
* Startwert ermitteln:		
x2: LDAA ADR1,x	Wert lesen	4
SUBA #128	Warten, solagen < 128	2
BMI x2	"	3
LDD TCNT,x	Counterstand laden	5
STD start	Counterstand speichern	5
* Stopwert ermitteln:	Wert lesen	
x3: LDAA ADR1,x		4
SUBA #128	Warten, solagen >= 128	2
BPL x3	"	3
LDD TCNT,x	Counterstand laden	5
STD stop	Counterstand speichern	5
* Überlauf in Counter überprüfen:		
LDD stop	Stopwert laden	5
SUBD start	Startwert subtrahieren	6
BMI x1	Wiederholung, falls negativ	3
* Routine verlassen:		
PULX	X-Register restaurieren	5
RTS	Subroutine verlassen mit D-Register als Returnwert	5

Programm 19 - Assembler-Routine zur Frequenzerkennung

Zum Assemblieren steht im Internet[17] ein Server zur Verfügung, der den Assembler-Code in Maschinensprache übersetzt und daraus ein ICB-File erstellt. Diese kann über ein LIS-File in ein IC-Programm eingebunden werden, so daß die Assembler-Routine dort mittels Funktionsaufruf verfügbar ist. Damit reduziert sich der IC-Code auf folgendes Maß, und bringt gleichzeitig eine 35-fache Bandbreitenvergrößerung mit sich:

[16] 1 Zyklus entspricht bei einem 2-MHz-Quarz einer Zeit von 500 ns.
[17] Adresse: *http://www.newtonlabs.com/ic/icb.html*

```
float Mikrophon_Frequenz() {
  float summe = 0.0;
  int i, anzahl;
  float frequenz;
  if (Mikrophon_Ton()) {
    poke(0x1039, 0b10000000);
    poke(0x1030, 0b00100010);
    anzahl = 50;
    for (i=0; i<anzahl; i++) summe = summe + (float) freq(0);
    frequenz = 1.0 / (2.0 * (summe /(float) anzahl) * 0.0000005);
    return(frequenz);
  }
  else return(0.0);
}
```

Programm 20 - IC-Lösung zur Frequenzerkennung unter Verwendung der Assembler-Routine

In der IC-Realisierung des Anwendungsbeispiels wird die Assembler-Routine verwendet. Gegenüber der Zeit von 9.5 µs zur Abfrage des Analogports ist hier jedoch die Umsetzzeit des A/D-Wandlers mit 64 µs höher. Das heißt, daß hier die Obergrenze für die maximal richtig erkennbare Frequenz durch den A/D-Wandler bestimmt wird. Sie liegt mit $(2 * 2 * 64\ \mu s)^{-1}$ bei 3906 Hz.

Der Timer, mit dem die Zeit zwischen den beiden Nulldurchgängen gemessen wird, ist ein 16-Bit-Register (TCNT), das vom Eingangstakt gespeist kontinuierlich hochgezählt wird. Bei einem 8-MHz-Quarz und einem 4-fachen Vorteiler ergibt sich also ein Meßbereich von 0 bis 32.768 ms, mit einer Auflösung von 500 ns.

In der folgenden Tabelle sind nochmals alle Zeiten der beteiligten Komponenten aufgeführt.

	Ausführungszeit	**Grenzen für die richtig erkennbaren Signalfrequenzen**
Leseschleife Assembler	4.5 µs	max. 55.555 kHz
Leseschleife C++	Simulationswert nicht repräsentativ	Simulationswert nicht repräsentativ
Leseschleife IC	337 µs	max. 741 Hz
A/D-Wandler	64 µs	max. 3906 Hz
Timer	0 ... 32.768 ms (Auflösung 500 ns)	7.6 Hz ...500 kHz

Tabelle 7 - Grenzen der richtig erkennbaren Signalfrequenzen

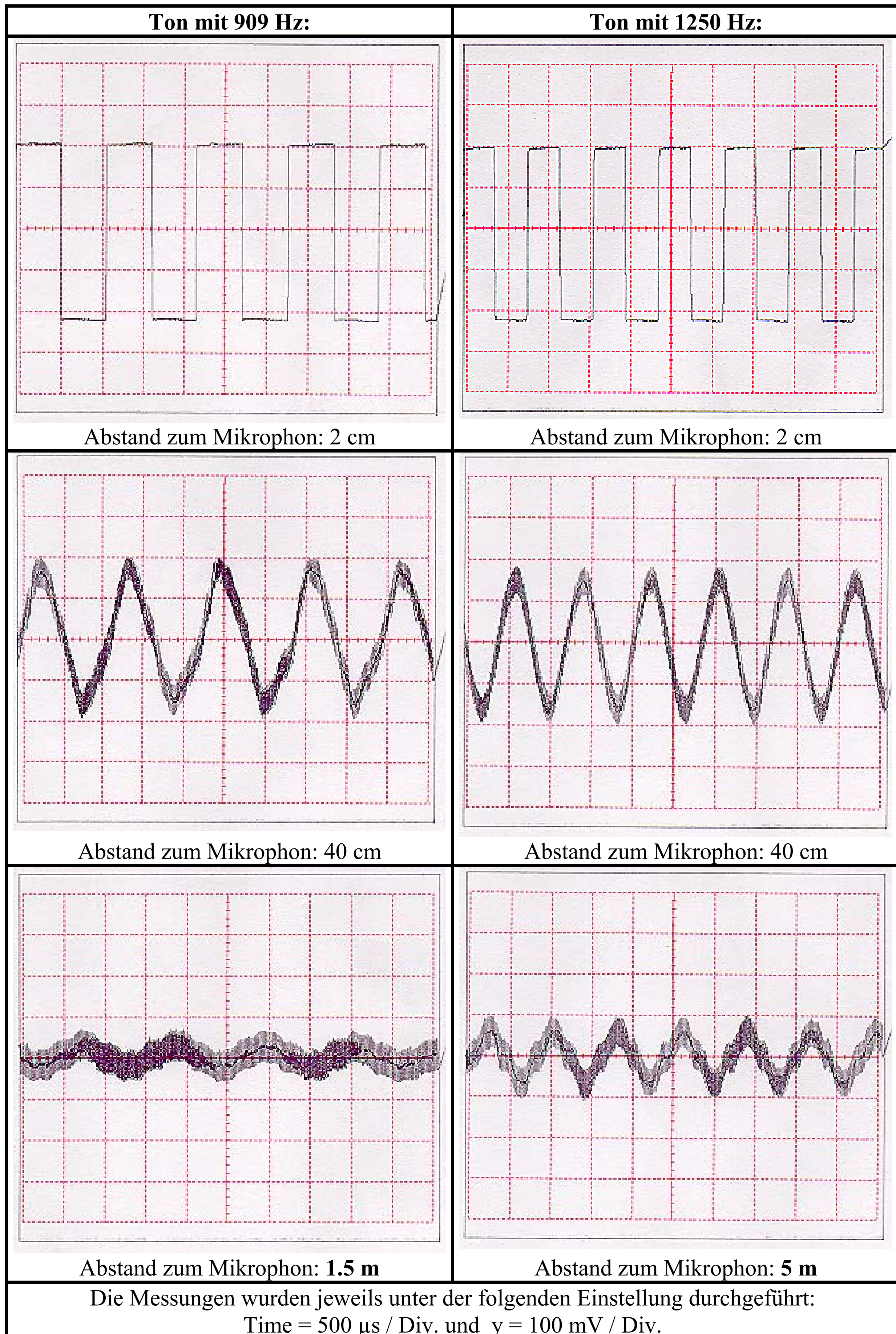
Hieraus kann je nach Einsatz der verschiedenen Software-Realisierungen die theoretische Bandbreite für die Tonerkennung herausgelesen werden. In der IC-Lösung mit Assembler-Routine beträgt die Bandbreite also 7.6 bis 3906 Hz und in der IC-Lösung ohne Assembler-Routine 7.6 bis 741 Hz. Zum Vergleich sind in der folgenden Tabelle verschiedene Größen von hörbaren Tönen aufgelistet:

Ton	Frequenz [Hz]	Periodendauer [ms]	Abstand zweier Nulldurchgänge [µs]
a’ *)	440	2.273	1136
c’’	523	1.923	956
a’’	880	1.136	568

*) Ton des Telefon-Freizeichens;

Tabelle 8 - Tonwerte

In der Praxis bereitet allerdings die mangelhafte Qualität des Mikrophons Probleme. Oszilloskopaufnahmen von verschiedenen Tonsignalen aus unterschiedlichen Entfernungen (jeweils mit gleicher Lautstärke) sollen dies verdeutlichen:



Ton mit 909 Hz:	**Ton mit 1250 Hz:**
Abstand zum Mikrophon: 2 cm	Abstand zum Mikrophon: 2 cm
Abstand zum Mikrophon: 40 cm	Abstand zum Mikrophon: 40 cm
Abstand zum Mikrophon: **1.5 m**	Abstand zum Mikrophon: **5 m**
Die Messungen wurden jeweils unter der folgenden Einstellung durchgeführt: Time = 500 µs / Div. und y = 100 mV / Div.	

Tabelle 9 - Frequenz- und Entfernungsabhängigkeit des Mikrophonsignals

Die Aufnahmen zeigen, daß der Abstand von der Tonquelle zum Mikrophon sehr entschei-

dend für die Qualität des Meßsignals ist, und nur in einem bestimmten Abstand brauchbare Ergebnisse liefert. Befindet sich die Schallquelle zu nah am Mikrophon, so ist das Mikrophonsignal am Verstärkerausgang völlig übersteuert (siehe Aufnahmen aus 2 cm Entfernung). Dies ist jedoch für die Frequenzerkennung irrelevant, da lediglich der Nulldurchgang entscheidend ist. Bei einem Abstand von 40 cm zwischen Mikrophon und Schallquelle ist das Ergebnis zufriedenstellend. Bei größeren Entfernungen verschwimmt das Signal zunehmend, so daß die Ergebnisse schnell unbrauchbar werden. Ein Ton mit ca. 900 Hz hat aus 1.5 m Entfernung in seiner Amplitude bereits schon so stark abgenommen, daß das Signal nur noch um die Nullinie schwimmt, und keine Frequenzerkennung mehr möglich ist. Hier könnte lediglich noch ein Tiefpaß-Filter Abhilfe schaffen. Bei steigender Frequenz der Töne nimmt diese Entfernungsempfindlichkeit allerdings ab. So kann bei einem Ton mit 1250 Hz die Frequenz selbst aus 5 m Entfernung noch mit einigermaßen ausreichender Genauigkeit ermittelt werden. Im Beispiel wäre hier ein Toleranzbereich von ±350 Hz hinzunehmen.

Nach dieser Erkenntnis sollten zur Steuerung des Roboters möglichst hochfrequente Töne benutzt werden, damit der Roboter auch bei Entfernung von der Tonquelle die Töne noch unterscheiden kann. Eine Mindestfrequenz von 1200 Hz wäre hierzu sinnvoll. Die Obergrenze wird durch den A/D-Wandler allerdings wie gezeigt auf 3906 Hz begrenzt. Zur Tonerzeugung stand jedoch nur eine Blockflöte mit einem Frequenzbereich von ca. 520 bis 1300 Hz zur Verfügung, was die Steuerung per Tonhöhen schwierig gestaltete. Zusätzlich erschwert wurde die Frequenzerkennung dadurch, daß eine konstante Lautstärke mit einer Blockflöte nur sehr eingeschränkt realisierbar war, und diese auch vom menschlichen Ohr nur subjektiv wahrgenommen werden kann.

3.4.7 Scheduler

Die prinzipielle Funktion des *Cycle_Schedulers* wurde bereits beim Design unter 3.3.1 erklärt. Sobald der Scheduler mit der Methode *Run* aktiviert ist, arbeitet er zyklisch die Proze-ßaktionen aller angemeldeten Prozesse ab, bis er durch die Methode *Stop* deaktiviert wird.

```
void Cycle_Scheduler::Run() {
  Aktiv = true;
  while(Aktiv) {
    if (Cyclingqueue.Aktuell() != 0)
      Cyclingqueue.Aktuell()->Prozessaktion(); // akt. Prozess ausführen
    Cyclingqueue.Weiter();           // Prozess weiterschalten
  }
}
```

Programm 21 - Methode Cycle_Scheduler::Run

Alle weiteren Implementierungsdetails bezüglich der Queue-Verwaltung haben für die Robotersteuerung keine direkte Bedeutung und können im Anhang abgedruckten Code eingesehen werden.

3.4.8 Mutex

Der wechselseitige Ausschluß zum Zugriff auf geschützte Bereiche wird durch ein Objekt Mutex mittels aktivem Warten realisiert. Leider ist auf dem M68HC11 kein atomares 'Bit Test and Set' als Maschinensprachebefehl verfügbar. Aus diesem Grund müssen in der Methode *Betreten* alle Interrupts global gesperrt werden, damit zwischen dem Prüfen des internen Status *Belegt* und dem Setzen des Status keine anderen Anforderungen den Status zerstören. Wie bereits erwähnt kann das Warten zeiteffizienter passiv realisiert werden. Dazu müßte dann allerdings eine Warteschlange und ein Aktivierungssignal eingeführt werden.

```
void Mutex::Betreten() {
  bool erfolg = false;
  do {
    DI(); // Interrupts sperren (da kein atomares 'Test and Set'
                                  verfügbar)
      if (!Belegt) {              // Falls frei
        Belegt = true;            // belegen
        erfolg = true;
      }
    EI();                         // Interrupts freigeben
  } while(!erfolg);               // aktives Warten
}
```

Programm 22 - Aktives Warten beim Betreten eines Mutex-geschützten Bereiches

Ein Beispiel, bei dem ein Bereich durch solch ein Mutex-Objekt geschützt wird, ist die Abfrage eines Ports des A/D-Wandlers. Diese Abfrage ist nicht reentrant-fähig. Deshalb darf sie nicht durch einen Zugriff von einem anderen Prozeß auf denselben, oder einen anderen Port unterbrochen werden. Der Mutexschutz kann lokal in der Abfragemethode untergebracht werden, so daß dieser nach außen völlig unsichtbar bleibt:

```
int Analogsensor::Analogwert() {
  int wert;
  Schutz.Betreten();              // Schutz für exklusiven Zugriff
    Poke(ADCTL, 0x20 | Port);     // Aktueller Port in SCAN-Mode
    wert = Peek(ADR1);            // Wert auslesen
  Schutz.Verlassen();
  return(wert);
}
```

Programm 23 - Schutz der Analogportabfrage durch Mutex

3.4.9 Objekterzeugung

Die Robotersteuerung ist so entworfen, daß keine Objekte zur Laufzeit erzeugt oder vernichtet werden müssen, was jeweils wertvolle Laufzeit kosten würde. Im Hauptprogramm wird zunächst ein Objekt der Ressourcenkontrolle *My_Ressourcenkontrolle* erzeugt. Da die Ressourcenkontrolle alle ihre Objekte statisch enthält, werden diese bereits im Konstruktoraufruf der Ressourcenkontrolle angelegt. (Das selbe gilt natürlich wiederum für die darin enthaltenen statischen Objekte.) Auch die dynamisch erzeugten Objekte *Queue-Element* in der Scheduler-Warteschlange werden bereits in der Initialisierungsphase angelegt, da ja alle Prozesse bereits in der Initialisierungsphase beim Scheduler angemeldet werden. Erst danach wird der

Scheduler mit der Methode *Run* gestartet. Es müssen also keine Objekte zur Laufzeit erzeugt werden.

```
int main() {
  Cycle_Scheduler My_Scheduler;
  Ressourcenkontrolle My_Ressourcenkontrolle(&My_Scheduler);
  My_Scheduler.Run();
  return 0;
}
```

Programm 24 - Hauptprogramm

3.5 Simulation

Zum Test der objektorientierten C++-Lösung auf dem PC sind in der Klasse *Hardware* die tatsächlichen Hardwareaufrufe durch Simulationswerte ersetzt. So sind beispielsweise in der Methode *Peek* unter der Adresse des Analogports (1031h) Simulationsdaten für die Analogsensoren wie Mikrophon (Port 2), Bumper (Port 3) und Pyrosensor (Port 5) erzeugt:

```
unsigned char Hardware::Peek(int adr) {
  int x;
  if (adr==0x1031) { // Zufallswerte für Analogports erzeugen:
    switch(RAM[0x1030-1]-32) {
      case 2: // Zufallswert Mikrophon:
              x = random(100);
              if      (x < 25) return(random(127)+1);   // 25% < 128
              else if (x < 50) return(random(127)+129); // 25% > 128
              else             return(128);             // 50% = 128
      case 3: // Zufallswert Bumperkette:
              x = random(7);
              if (x < 2) return(random(145)+16); // 2:7 x Crash
              else return(random(16));           // 5:7 x Frei
      case 5: // Zufallswerte Pyrosensor:
              return(random(256));
      default: printf("PORT %d undefiniert!\n", RAM[adr-1]);
    }
  }
  return(random(256));
}
```

Programm 25 - Modifizierte Methode Hardware::Peek zur Erzeugung von Simulationsdaten

Zusätzlich werden an entsprechenden Stellen des Programms Bildschirmausgaben eingebaut, so daß die Funktionalität am Bildschirm überprüft werden kann:

```
TEMPERATURMESSUNG(1): Motoraktion: Leer; Text: Temperatur: 58;
STANDARDVERHALTEN(1): Motoraktion: Vorwaerts; Text: ; Geschw.: 50.0;
RESSOURCENKONTROLLE
GERADENKONTROLLE: Differenz: 0; Korrektur: 18.6
GESAMTVERHALTEN(1): Motoraktion: Vorwaerts; Text: Temperatur: 58; Geschw.: 50.0;

TEMPERATURMESSUNG(1): Motoraktion: Leer; Text: Temperatur: 31;
AKUSTIK(1): Motoraktion: LinksDrehung; Text: Ton tief: 761; Geschw.: 40.0;
STANDARDVERHALTEN(1): Motoraktion: Vorwaerts; Text: ; Geschw.: 50.0;
RESSOURCENKONTROLLE
GERADENKONTROLLE: Differenz: -1; Korrektur: 13.4
GESAMTVERHALTEN(1): Motoraktion: LinksDrehung; Text: Ton tief: 761; Geschw.: 40.0;

TEMPERATURMESSUNG(1): Motoraktion: Leer; Text: Temperatur: 143;
HAPTIK(1): Motoraktion: LinksRueckDrehung; Text: ; Geschw.: 40.0;
AKUSTIK(1): Motoraktion: LinksDrehung; Text: Ton tief: 834; Geschw.: 40.0;
STANDARDVERHALTEN(1): Motoraktion: Vorwaerts; Text: ; Geschw.: 50.0;
RESSOURCENKONTROLLE
GERADENKONTROLLE: Differenz: -1; Korrektur: 13.4
GESAMTVERHALTEN(1): Motoraktion: LinksRueckDrehung; Text: Ton tief: 834; Geschw.:
 40.0;
```

Abbildung 15 - Bildschirmauszug aus dem Testprotokoll

Ein Verhalten gibt seine Daten nur auf dem Bildschirm aus, wenn es aktiv ist, das heißt wenn es eine Anforderung an die *Ressourcenkontrolle* stellt (gekennzeichnet mit „(1)" hinter dem Verhaltensamen). Die Darstellung umfaßt drei Durchläufe aller Prozesse. Im ersten Durchlauf melden lediglich die *Temperaturmessung* und das *Standardverhalten* Anforderungen an die *Ressourcenkontrolle* an. Da die *Temperaturmessung* eine höhere Priorität hat als das *Standardverhalten*, darf sie deren Anforderungen überschreiben. Im ersten Durchlauf kommt es allerdings zu keinem Ressourcenkonflikt, da lediglich die *Temperaturmessung* einen Text auf dem Display ausgeben möchte und genauso nur das *Standardverhalten* ein Motorkommando *„Vorwaerts"* mit einer Geschwindigkeit von *50.0* anfordert.

Im zweiten Durchlauf hat die *Akustik* einen tiefen Ton mit *761 Hz* erkannt und möchte dies auf dem *Display* anzeigen. Zusätzlich möchte sie dadurch eine Linksdrehung mit einer Geschwindigkeit von *40.0* veranlassen. Da die *Akustik* die höchste Priorität der drei Anforderungen hat, bekommt sie alle benötigten Ressourcen zugeteilt. Gleichzeitig wurde zwischen dem ersten und zweiten Durchgang festgestellt, daß das linke Rad gegenüber dem Rechten um einen Tick langsamer war. Der Korrekturfaktor wird um 5.2 erniedrigt, was das linke Rad etwas schneller macht. Weil in diesem Durchgang eine Kurvenfahrt gestartet wurde, hat die *Motorensteuerung* die *Geradenkontrolle* deaktiviert. Deshalb ist der Meßwert vom zweiten zum dritten Durchlauf unverändert.

Im dritten Durchlauf wurde von der *Haptik* ein Hindernis erkannt und entsprechend eine LinksRueckDrehung angefordert. Diese kommt wegen der höchsten Priorität der *Haptik* gegenüber der Anforderung der *Akustik* und des *Standardverhaltens* zum Zuge. Da die *Haptik* jedoch keine Anforderung an das *Display* stellt, bleibt dieses für die Anzeige des *Textes „Ton tief: 834"* der *Akustik* mit nächstniedrigerer Priorität frei.

3.6 Gesamtmodell

Erst bei der Programmierung haben sich letzte implementierungstechnische Details geklärt, die auch immer wieder kleinere und größere Veränderungen am Gesamtmodell zur Folge hatten. Letztendlich ist ein Programm entstanden, das aus 29 Klassen mit 1090 Codezeilen besteht und einen Gesamtumfang von 8397 Bytes an Code und Daten hat[18]. Die Klassen sind auf die folgenden 16 Module aufgeteilt:

Modul	Klasse	Funktion
Queue	Queue	Schlange
	QElement	Element der Klasse Queue
Sched	Scheduler	Abstrakte Oberklasse für Scheduler
	Cycle_Scheduler	Scheduler für kooperatives Multitasking
ICSched	IC_Scheduler	Scheduler mit Nutzung der IC-Multitaskingfunktionen
MSched	M68HC11_Scheduler	Demo eines Scheduler für nicht-kooperatives Multitaskings (nur andeutungsweise realisiert)
Prozess	Prozess	Abstrakte Oberklasse für Prozesse
Mutex	Mutex	Objekt zum wechselseitigen Ausschluß von Zugriffen
Hardware	Hardware	Adressen und Zugriffsmethoden auf die Hardware
Sensoren	Sensor	Oberklasse für Sensoren
	Analogsensor	Oberklasse für Analogsensoren
	Mikrophon	Mikrophon
	Bumper	Bumper-System
	Pyrosensor	Pyrosensor
	Radenkoder	Rad-Enkoder
Aktoren	Aktor	Oberklasse für Aktoren
	LCDDisplay	LCD-Display
	Motorensteuerung	Ansteuerung der einzelnen Motoren
	Motor	Motor
VDaten	Verhaltensdaten	Eigenschaften, die ein Verhalten haben kann
Verhalt	Verhalten	Abstrakte Oberklasse für Verhalten
	Standardverhalten	Standardverhalten (Geradeausfahrt)
	Akustik	Akustik (Reaktion auf Töne)
	Haptik	Haptik (Reaktion auf Kollisionen)
	Temperaturmessung	Temperaturmessung
	Gesamtverhalten	Resultirendes Gesamtverhalten
ResCon	Ressourcenkontrolle	Vergabe der Ressourcen nach Prioritäten der Verhalten
Gerade	Geradenkontrolle	Prozeß zur Überwachung der Geradeausfahrt
Timer	Timer	Zeitsystem
FLib	---	Diverse Grundfunktionen
BumpWert	---	Definition der Bumperwerte
Main	---	Hauptprogramm

Tabelle 10 - Module

Bei der Programmausführung werden insgesamt 40 Objekte erzeugt, von denen 8 aktives und 32 passives Verhalten zeigen. Ein Klassendiagramm des Gesamtmodells ist im Folgenden

dargestellt. Dabei sind aus Gründen der Übersichtlichkeit einige lokale Attribute und Methoden von der Darstellung ausgenommen und manche benutzt-Beziehungen lediglich textuell und nicht als Verbindungslinie dargestellt.

[18] Der tatsächliche Programmumfang ist geringer, da auf dem Mikrocontroller kein Code zur Simulation und zur Bildschirmausgabe notwendig ist. (Siehe hierzu auch 5.12)

Verhalten
| Datenschutz : Mutex
Aktiv( )

Verhaltens-daten
Ausgabe( )
Displaytext
Displaywert
Geschwindigkeit
Motoraktion

Zustand

Ressourcen-kontrolle

Prozess
| Zyklisch
PID
Prozessaktion( )
StackSize
Ticks
{active}

Scheduler
| Zyklisch
Prozess_erzeugen( )
Prozess_vernichten( )
Zyklustyp( )
{active}

IC_Scheduler

Cycle_Scheduler
| Aktiv
Run( )
Stop( )

M68HC11_Scheduler

Cyclingqueue
Readyqueue

Queue
| Root
| Zähler
Aktuell( )
Eintragen( )
Leeren( )
Löschen( )
Weiter( )

Queue-Element
Inhalt
Nächstes

My_Standardverhalten
My_Akkustik
My_Temperaturmessung
My_Haptik
My_Gesamtverhalten

Standard-verhalten

Akkustik

Temperatur-messung

Haptik

Gesamtverhalten
Datenübernahme( )

My_Mikrophon
My_Pyrosensor
My_Bumper
My_Motorensteuerung
My_LCDDisplay

Mikrophon
| My_Timer : Timer
DurchschnittsLautstärke( )
Frequenz( )
MaximalLautstärke( )
Ton( )
Tondauer( )

Pyrosensor
Temperatur( )

Bumper
Kollissionsposition( )

Geradenkontrolle
| Geschwindigkeitsschutz : Mutex
| Korrekturschutz : Mutex
Korrekturfaktor( )
Links( )
Rechts( )
Setze_aktiv( )
Setze_Geschwindigkeit( )
Setze_inaktiv( )

My_Geradenkontrolle

Motorensteuerung
| LMotor : Motor
| RMotor : Motor
Ausführen( )

Analogsensor
| Port
Analogwert( )

Enkoder
My_Timer
My_Timer
LMotor / RMotor

Schutz

Rad-Enkoder
Interrupt( )
LImpulse( )
RImpulse( )

Timer
MSeconds( )
Sleep( )
Ticks( )

Motor
Go( )

LCD-Display
Anzeigen( )

Mutex
| Belegt
Betreten( )
Verlassen( )

Sensor

Aktor

Hardware
Adressen
Adresszugriffe( )
DI( )
EI( )
Time( )

Abbildung 16 - Klassendiagramm mit Gesamtmodell der Robotersteuerung

4 IC-Programm

In dem prozeduralen IC-Programm ist die Funktionalität des objektorientierten C++-Programms mit Hilfe von Funktionen und Prozeduren nachgebildet. Hierzu wurde die mit IC gelieferte Programmbibliothek „LIB_RW11.C“ verwendet.

Zur Angleichung des kooperativen Verhaltens der C++-Lösung an die nicht-kooperativen Tasks der IC-Lösung werden alle Prozesse jeweils bei Schleifeneintritt durch die Prozedur *Hog_Processor* gesperrt und am Schleifenende durch die Prozedur *Defer* wieder freigegeben. Dadurch werden auch die nicht reentrant-fähigen Abfragen der Analogports vor gegenseitigem Überschreiben geschützt.

Die Prozeßkommunikation geschieht über globale Variablen. Zur Verdeutlichung der Zugehörigkeit sind sie jeweils im Abschnitt ihrer „Klasse“ deklariert. Natürlich hat die Deklarationsposition keine Einfluß auf die Sichtbarkeit. Den Schutz, den die objektorientierte Lösung bietet, kann hier nicht gegeben werden. Genauso ist bei Funktionen und Prozeduren jeweils der „Klassen“-Namen vorangestellt. Dies hat natürlich genauso wenig Einfluß auf deren Schutz vor globaler Zugänglichkeit.

Die Einbindung der Bibliotheken und ICB-Files (Maschinencode der Assembler-Routinen) geschieht über sogenannte LIS-Files, durch die alle zugehörigen Module eingebunden werden.

Der Code der IC-Realisierung, die Funktionsbibliothek „LIB_RW11.C“ sowie die LIS- und ASM-Files sind im Anhang abgedruckt und ergeben insgesamt einen Umfang des P-Codes von 4451 Bytes aus 792 Codezeilen.

5 Ergebnisse

5.1 Entwicklungswerkzeuge

Wer heute einen Mikrocontroller für ES objektorientiert programmieren möchte, wird Schwierigkeiten haben, einen geeigneten Compiler zu finden. So gab es auf dem Markt keinen Compiler für eine objektorientierte Programmiersprache, der direkt Maschinencode für den im Anwendungsbeispiel eingesetzten Mikrocontroller M68HC11 erzeugt. Gerade weil in ES aus Kostengründen meist sehr kleine, billige Mikrocontroller verwendet werden müssen, gestaltet sich die Suche um so schwieriger. Bis heute sind nur für größere Mikrocontroller (siehe [Mor 96]) C++-Compiler auf dem Markt verfügbar. Die Vermutung, ungenügende Performanceresultate seien die Ursache für die geringe Verfügbarkeit, ist nach Meinung des Autors nicht der ausschlaggebende Punkt. Vielmehr konzentrieren sich Compilerhersteller aus Rentabilitätsgründen zuerst auf Prozessoren, die für größere Applikationen eingesetzt werden. Die Notwendigkeit, einen Prozessor für wenig komplexe Anwendungen objektorientiert zu programmieren, ist einfach nicht so hoch wie für Anwendungen, deren Komplexität ohne OO nicht mehr bewältigt werden kann. Die Leistungsfähigkeit und vorteilhafte Nutzbarkeit der OO für kleine Prozessoren ist jedoch genauso gegeben und darf nicht an der Verfügbarkeit der Compiler gemessen werden.

Die Modellierung nach der Notation von Booch unter Verwendung des CASE-Systems Rose eignete sich ohne Einschränkung für die Entwicklung der Robotersteuerung. Die Durchgängigkeit von der Analyse über das Design bis hin zur Programmierung war ohne Bruch gegeben und das iterative, inkrementielle Vorgehen wurde unterstützt. Dadurch, daß die Prozeßmodellierung nach dem klassischen Prozeßkonzept erfolgte, war die Booch-Notation auch hierfür geeignet.

5.2 Durchführung

An den Ausführungen unter 3.2 bis 3.4 zeigt sich, daß kein reiner Top-Down-Entwurf bei der OOSE möglich war. Speziell für die Softwareentwicklung innerhalb dieser Diplomarbeit kam erschwerend hinzu, daß sämtliche Bereiche, von der Robotik im Allgemeinen bis hin zur Sensorik und spezifischen Mikrocontroller-Programmierung, für den Autor Neuland waren und somit alle Abstraktionsebenen gleichzeitig nebeneinander betrachtet werden mußten. Vielmehr bestätigte sich die Eignung der von Booch vorgeschlagenen Vorgehensweise, indem einzelne Klassen unabhängig fokusiert und ausgebaut wurden. Für den kommerziellen Einsatz bedeutet dies, daß relativ schnell erste Ergebnisse oder Prototypen erstellt werden können, die

später nach und nach ausgebaut und verfeinert werden können.

5.3 Verständlichkeit

Das objektorientierte Paradigma trägt in vielen Punkten zur besseren Verständlichkeit der Software bei. Klassen sind weitgehend unabhängige Softwarebausteine mit klar definierten Verantwortlichkeiten. Detailwissen bleibt in ihnen verborgen, so daß das Gesamtmodell klar verständlich bleibt. Je nach Notwendigkeit können sich Entwickler aufgabenspezifisch in die Implementierungsdetails vertiefen oder aber für sie Irrelevantes einfach unberücksichtigt lassen. Diese Eigenschaft zeigt sich auch an dieser Ausführung der Diplomarbeit. Es sind nur einzelne Ausschnitte der Implementierung herausgegriffen und genauer beschrieben. Trotzdem kann das Gesamtmodell in sich verstanden werden.

Die Softwareentwicklung gestaltet sich auch dadurch einfacher, daß Objekte des Problemraumes 1:1 in Objekte im Softwarraum übertragen werden können. Das Softwaremodell bildet also ein Abbild der realen Welt, das intuitiv verstanden und modelliert werden kann. Die Kommunikation über Botschaften entflechtet die Software nicht nur von Abhängigkeiten von komplizierten Datenstrukturen, sie trägt ebenfalls zur besseren Verständlichkeit bei. Eine Methode wie *Motor.stop* ist einfach viel klarer verständlich als ein Funktionsaufruf *Motor(0, 0)*, der das selbe bewirkt. Genauso verhält es sich mit der Möglichkeit der Operandenüberladung. Bekannte Operatoren wie z.B. + oder - können in ihrer Klasse überladen werden, so daß sie von späteren Benutzern dieser Klassen intuitiv gebraucht werden können, ohne zu wissen, welche komplexen Operationen eventuell dahinter verborgen sind. Die OO gestattet durch die Möglichkeit der Methodenüberladung erstmals, daß gleiche Operationen auf verschiedene Klassen auch gleich benannt werden können. Für den Programmierer werden dadurch Programme einfacher zu lesen und er braucht sich nicht mehr viele spezielle Funktionsaufrufe zu merken. So könnten im Anwendungsbeispiel sowohl die Motorensteuerung als auch die beiden Motoren jeweils eine Methode *Links* besitzen. Es müssen nicht mehr, wie so oft bei der strukturierten Programmierung Namenskonflikte künstlich vermieden werden.

5.4 Flexibilität

Die OO erhöht die Flexibilität von Software. Dadurch, daß Klassen klar definierte Verantwortlichkeiten haben und durch das Botschaftenkonzept schlanke, einfache Schnittstellen besitzen, ist der Austausch von Klassen meist relativ einfach möglich. Eines von mehreren Beispielen im Anwendungsmodell wäre die flexible Austauschmöglichkeit der Antriebstechnik. Wollte man diese etwa durch einen Vierradantrieb oder einen Kettenantrieb ersetzen, so müßte lediglich die Klasse *Motorensteuerung* (und eventuell die Klasse *Geradenkontrolle*) ausge-

tauscht werden. Alle anderen Klassen wären von einer solchen Änderung unberührt.
Weiter tragen dazu der Polymorphismus und das dynamische Binden bei. In Oberklassen kann ein Verhalten und seine Methoden festgelegt werden, welches in Unterklassen, die erst zur Laufzeit ausgewählt werden, verschiedenartig implementiert ist. Software bekommt dadurch eine neue Dimension an Flexibilität. Im Anwendungsbeispiel können für das Multitasking verschiedene Scheduler zum Einsatz kommen, ohne daß dafür an anderen Klassen Veränderungen vorgenommen werden müssen. Alle Eigenschaften, die ein Scheduler haben muß, sind in der abstrakten Oberklasse *Scheduler* festgeschrieben. Spezielle Realisierungen dafür können in deren Unterklassen beliebig variiert werden. Auf die Möglichkeit des dynamischen Bindens wurde bei diesem ES verzichtet, da es zu vermeidbaren Laufzeiteinbußen geführt hätte (siehe auch 5.12).

5.5 Wiederverwendbarkeit

Die größten Vorteile der OO liegen wohl in ihrer aktiven Unterstützung der Wiederverwendung, was die Produktivität steigern und die Entwicklungskosten senken kann. Dabei unterstützt die OO die Wiederverwendung gleich in dreifacher Weiße: Klassen können erstens innerhalb einer Applikation vielfach instanziert werden. Zweitens können Klassen genauso für andere Applikationen wiederverwendet werden. Und drittens wird die Wiederverwendung bei der Programmierung durch Vererbung ermöglicht. Bei Spezialisierungen einer Klasse reduziert sich also die Implementierungsarbeit lediglich auf die Veränderung gegenüber der allgemeineren Oberklasse. Alles andere kann durch die Vererbung übernommen werden.
Gerade im hardwarenahen Bereich sind viele Elektronikkomponenten als Module verfügbar. Durch die OO können nun zugehörige Softwaremodule in Form von Klassen erstellt werden, die sich nach einmaliger Entwicklung für alle Applikationen wiederverwenden lassen. Aus unserem Anwendungsbeispiel würde das für sämtliche Sensoren und Aktoren wie Mikrophon, Pyrosensor, Rad-Encoder, Motoren und LCD-Display zutreffen. Auch für die nicht verwendeten Komponenten wie LED's, IR-LED's, Photowiderstände und Piezo-Piepser könnten einmalig solche Komponenten erstellt und vielfältig eingesetzt werden. Von der Wiederverwendung innerhalb von Klassen durch Vererbung wurde ebenfalls aktiv Gebrauch gemacht. Allein für ein solches, relativ kleines Anwendungsbeispiel konnten folgende Vererbungsbeziehungen vorteilhaft ausgenutzt werden:

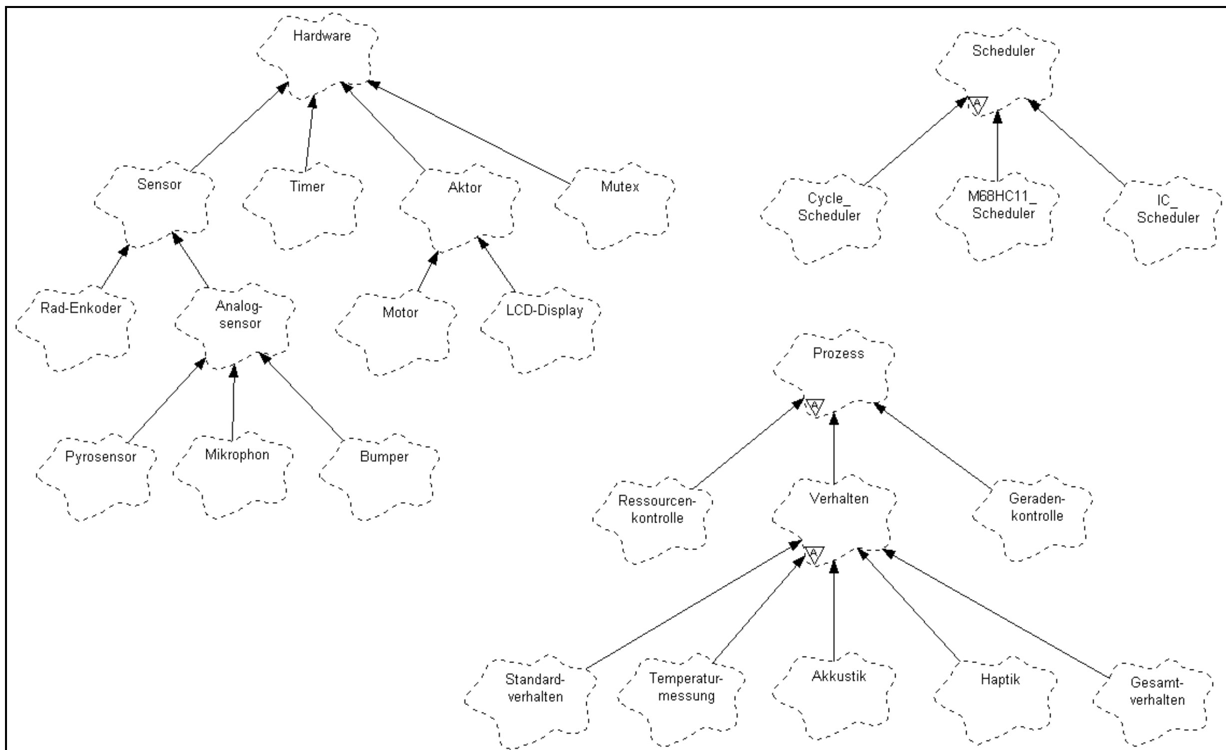


Abbildung 17 - Klassendiagramm der Vererbungsstrukturen

5.6 Erweiterbarkeit / Änderbarkeit / Wartbarkeit

Diese Vererbungsbeziehungen werden sich auch auf spätere Erweiterungen vorteilhaft auswirken. Soll beispielsweise ein neuer Analogsensor eingeführt werden, so stehen ihm durch das Beerben der Klasse *Analogsensor* bereits sämtliche Eigenschaften, von den Hardwareadressen bis hin zur Portzuteilung zur Verfügung. Die Einführung eines neuen Verhaltens hätte durch Vererbung automatisch zur Folge, daß es durch seine Instanziierung automatisch beim Prozeßsystem angemeldet wird. Die Erweiterungsarbeit besteht lediglich noch in der Implementierung des spezifischen Verhaltens selbst und seiner Verarbeitung in der Ressourcenkontrolle.

Auch Änderungsarbeiten reduzieren sich auf ein geringes Maß: Eine Änderung an einer Oberklasse wird automatisch an alle Ihre Unterklassen weitervererbt. In anderen Paradigmen müßten sämtliche betroffenen Stellen im Code mühsam aufgesucht und einzeln abgeändert werden. Auch hier wirken sich Zeiteinsparungen und Vermeidung von Fehlern positiv aus.

5.7 Portabilität

Die enge Hardwarekopplung und die daraus resultierende mangelhafte Portabilität von Softwarprodukten wurde Eingangs als großes Manko bemängelt. Dieses ist bereits bei nicht eingebetteten Systemen ein großes Problem. Um wie viel mehr ist diese Hardwarekopplung bei ES vorhanden, deren spezifische Eigenschaft es ja ist, in ein Hardwareumfeld eingebunden zu

sein? Im Anwendungsbeispiel wird gezeigt, daß allein der Austausch einer Klasse (*Hardware*) genügt, um die Applikation auf einen anderen Prozessor zu portieren! Es lassen sich viele Kosten einsparen, wenn Applikationen prozessorunabhängig entwickelt und prozessorübergreifend eingesetzt werden können.

5.8 Sicherheit

ES stellen in Sachen Zuverlässigkeit und Sicherheit meist viel höhere Anforderungen als andere Systeme, weil sie in Real-Life-Prozesse eingebunden sind und bei Fehlfunktionen Gefahren für Menschen und Umwelt verursachen können. Objektorientierung nimmt für sich in Anspruch, Software sicherer zu machen. Dies geschieht auf mehrfache Weise:

Zunächst einmal haben Klassen ihre eigenen Daten, auf die nur sie selbst Zugriffsberechtigung haben. Nach außen sind diese Daten vor unberechtigter Manipulation geschützt. Daten, die von anderen als der besitzenden Klasse gebraucht werden, können über Zugriffsmethoden gekapselt werden, so daß diese nur indirekt (nach voriger Berechtigungs- oder Plausibilitätskontrolle) verändert oder weitergegeben werden können. Beispielhaft sei hier das Attribut *Status* in der Klasse *Verhalten* genannt. Sie zeigt der *Ressourcenkontrolle* einen Anspruch auf Ressourcen an. Allerdings hat die *Ressourcenkontrolle* keine direkte Zugangsmöglichkeit zu diesem Attribut, so daß eine unberechtigte Manipulation ausgeschlossen bleibt. Die *Ressourcenkontrolle* kann das Attribut *Status* nur über die Methode *Aktiv* lesend abfragen. *Status* ist also durch *Aktiv* gekapselt. Die Integrität bleibt gewahrt.

Genauso steigt die Sicherheit damit, daß durch Wiederverwendung und Vererbung weniger Code neu geschrieben werden muß, was natürlich immer eine Quelle für mögliche Fehler ist. Dadurch, daß Klassen vielfach wiederverwendet und somit in verschiedensten Anwendungen zum Einsatz kommen, haben sie auch einen höheren Qualitätsnachweis vorzuweisen, weil Fehler mit einer höheren Wahrscheinlichkeit aufgedeckt werden. Verbesserungen an einer Klasse kommen allen verwendenden Applikationen zu gute und wirken sich unmittelbar darauf aus.

Im Gegensatz zu Assembler oder C können viele objektorientierte Programmiersprachen eine strenge Typprüfung vorweisen. Dadurch werden Laufzeitfehler vermieden.

Einen besonders hohen Qualitätsanspruch kann die Programmiersprache Eiffel bieten. Mit ihren Invariant-Klausel können für jede Klasse Vor- und Nachbedingungen angegeben werden, was die Korrektheit von Programmen beweisbar macht.

Zur Erhöhung der Robustheit von Programmen sollten Ausnahmebehandlungen möglich sein. Nicht zuletzt wird die Sicherheit auch durch die unter 5.3 angeführte klarere Verständlichkeit und Übersichtlichkeit erhöht.

5.9 Verifizierbarkeit

Klassen können Botschaften empfangen und Botschaften senden. Abhängigkeiten von komplizierten Datenstrukturen entfallen. Das erleichtert den Test von Klassen. Botschaften können in der Regel viel einfacher erzeugt werden, als komplexe Datenstrukturen. So werden Klassen gegen ihre Schnittstelle (also gegen ihre Methoden) getestet und das Ergebnis mit dem Soll verglichen. Dadurch daß Klassen eigenständige Module darstellen, können sie meist auch separat von anderen Klassen getestet werden.

Auch der Vererbungsmechanismus kann zum Klassentest günstig ausgenutzt werden. So wurde zum Test des Anwendungsbeispiels die Klasse *Verhaltensdaten* um eine Ausgabemethode ihrer Daten ergänzt. Diese Ausgabemethode stand durch Vererbung in jedem einzelnen Verhalten zur Verfügung, so daß jedes Verhalten somit seine spezifischen *Verhaltensdaten* anzeigen konnte.

5.10 Produktivität und Entwicklungskosten

All die bisher genannten Vorteile tragen direkt oder indirekt zur Kostenreduzierung und Produktivitätssteigerung bei der Softwareentwicklung bei: Die Implementierungszyklen werden durch die Wiederverwendung kürzer, weil sich neu zu schreibender Code nur noch auf Änderungen bezieht. Die Änderung, Wartung und Erweiterung von Software wird vereinfacht und beschleunigt; Klassen, Frameworks und Applikationen werden sicherer, flexibler und portabler. Die Entwicklung im Team wird aktiv unterstützt und Experten können zielgerechter eingesetzt werden.

5.11 Einführungskosten

Bei der Einführung des objektorientierten Paradigmas können allerdings neue Kosten entstehen. Ein Aspekt sind die Kosten die anfallen, um Mitarbeiter auf die OO-Technologie umzuschulen. Eventuell müssen dazu sogar neue Mitarbeiter eingestellt werden. Genauso sind Anschaffungskosten für neue Entwicklungstools und Programmiersprachen aufzubringen. Für spezielle Anwendungsbereiche müssen gegebenenfalls spezifische Klassenbibliotheken eingekauft oder neu erstellt werden, was ebenfalls Kosten verursacht.

Beispielhaft ist, daß im hier entwickelten Anwendungsbeispiel keine einzige Klasse der Borland-Entwicklungsumgebung wiederverwendet werden konnte. Alle Klassen mußten zunächst neu entworfen und implementiert werden. Gerade bei ES ist dieser Aspekt besonders relevant: Durch ihre hohen Anforderungen an das Laufzeitverhalten müssen (wie weiter unten noch gezeigt wird) manche Einschränkungen bei der Wiederverwendung in Kauf genommen wer-

den. Theoretisch wäre es möglich gewesen, für die Warteschlange des Schedulers die Queue-Klassen von der Borland-Entwicklungsumgebung zu beerben. Allerdings sind diese zur dynamischeren Wiederverwendbarkeit als Templates realisiert, was beim verwendeten Compiler zu unnötigem RAM-Verbrauch geführt hätte. So wurde aus Effizienzgründen eine viel speziellere, schlankere Queue neu implementiert. Dieses exemplarische Beispiel zeigt, daß die Einführung mit besonderen Kosten verbunden sein kann.

5.12 Speicherplatzverbrauch

Selbstverständlich kann unter Verwendung einer Hochsprache der Speicherplatz nicht mehr so gezielt bis in das letzte Bit ausgereizt werden, wie dies unter Verwendung eines Assemblers möglich ist. Letztendlich wird es wieder eine Rentabilitätsfrage bleiben, ob zur Bewältigung der Komplexität eine (objektorientierte) Hochsprache günstiger kommt, als die Einsparung von Speicher. Bei dieser Frage wird für ES sehr genau die Qualität des Compiler berücksichtigt werden müssen, von dem ganz entscheidend die generierte Codegröße abhängig ist. Deshalb kann hier nur auf Kapitel 5.16 verwiesen werden.

Fest steht jedoch, daß compilierenden Sprachen zur Ermöglichung des dynamischen Bindens Tabellen zur Adressverwaltung von virtuellen Funktionen anlegen müssen, die zu mehr Speicherplatzverbrauch führen. Da es sich hierbei aber lediglich um eine Möglichkeit und nicht um ein „Muß“ für den Programmierer handelt, ist der Speicherplatzverbrauch durch die OO nicht zwangsläufig höher.

Das Bestreben, Klassen möglichst allgemeingültig zu Beschreiben und somit vielfältig wiederverwenbar zu machen, bringt allerdings auch den Effekt mit sich, daß Klassen oftmals für einen Anwendungsbereich überspezifiziert sind. Wertvoller Speicherplatz für Code von nicht benötigten Methoden wird somit verschenkt. Wenn dies nicht akzeptabel ist, wird eine gezielte Einführung einer allgemeiner formulierten Oberklasse, oder gleich eine Neuimplementierung, notwendig. Bei der Robotersteuerung ist beispielsweise das Attribut *Aktiv* für das *Gesamtverhalten* überflüssig, da *Gesamtverhalten* als ausführende Einheit grundsätzlich immer aktiv ist. Die Einführung einer zusätzlichen Oberklasse würde für die restlichen Verhalten jedoch zusätzlichen Overhead verursachen.

Vergleicht man den Code- und Datenumfang der objektorientierten C++-Lösung von 8397 Bytes mit der strukturierten IC-Lösung von 4451 Bytes, so erscheint die Differenz als relativ hoch. Allerdings ist dabei folgendes zu beachten: In der C++-Lösung müssen zusätzliche Bibliotheken zur Erzeugung von Simulationsdaten (*STDLIB.H*, *TIME.H*, *DOS.H*) und sowie zur Bildschirmausgabe (*STDIO.H*) eingebunden werden. Auch die Berechnung der Simulationsdaten selbst, sowie der Text für die Bildschirmausgabe beanspruchen zusätzlich Speicher-

platz. Des weiteren wird bei der C++-Lösung Speicherplatz explizit zur RAM-Simulation alokiert, wogegen dieser auf dem Roboter physikalisch als Hardware vorhanden ist. Außerdem ist im C++-Code zusätzlich das gesamte Prozeß-Verwaltungssystem enthalten, was dagegen bei der IC-Lösung bereits in dem IC-„Betriebssystem“ (zusätzliche 1113 Bytes) enthalten ist. Ferner ist unbekannt, von welcher Art der Stack-basierte Pseudo-Code ist, den IC aus dem Quellcode generiert. Genauso wenig kann aus dem Vergleich der Codezeilen (C++: 1090 Zeilen; IC: 792 Zeilen) eine objektive Aussage gemacht werden. In Anbetracht dessen, daß in der C++-Lösung sämtliche Methodendefinitionen durch Verwendung von Header-Dateien doppelt vorkommen, erscheint der Unterschied als eher gering. Allerdings wurden Umgekehrt nicht alle Funktionen der IC-Bibliothek verwendet. All diese Faktoren führen dazu, das keine sinnvolle Schlußfolgerungen aus den beiden Codeumfangsgrößen zu ziehen sind.

5.13 Laufzeitverhalten

Durch die OO können Einbußen im Laufzeitverhalten entstehen: Objektorientierung nutzt als Prinzip zur Modularisierung die Abstraktion. Folglich zieht ein Methodenaufruf einer Klasse auf hoher Abstraktionsebene einen kaskadenförmigen Aufruf von Methoden aller niedrigerer Ebenen nach sich. Jeder Methodenaufruf benötigt dafür Zeit zum Berechnen und Ausführen des Sprungs sowie für die Parameterübergabe. Der Gewinn an Abstraktion und Modularität mit all seinen Vorteilen bringt eben diesen negativen Effekt mit sich, der so ausgeprägt bei anderen Paradigmen nicht auftritt. Am deutlichsten kann man die Tiefe von Abstraktionsebenen beim Debuggen von Smalltalk-Programmen beobachten. Smalltalk, als die reinste objektorientierte Sprache, besteht ja komplett aus Objekten. Selbst Ziffern, Buchstaben und Operatoren sind Objekte, die Botschaften verstehen. Eine Tiefe von 15 Abstraktionsebenen kann dabei durchaus vorkommen. Verstärkt wird dieser Effekt durch die Technik des Kapselns von Daten. Der Sicherheitsgewinn geht dabei auch auf Kosten der Zeit für einen zusätzlichen Methodenaufruf.

Die Möglichkeit des dynamischen Bindens erfordert neben zusätzlichem Speicherplatz auch zusätzliche Rechenzeit. Im Gegensatz zur statischen Bindung ergibt sich die tatsächlich verwendete Methode hier erst zur Laufzeit. Wenn dynamisches Binden bei ES nicht vermieden wird, muß also die Zeit zum Ermitteln der Adresse über die virtual-function-tables zusätzlich in Kauf genommen werden.

Genauso verhält es sich mit der dynamischen Objekterzeugung und -vernichtung: Da jeder Konstruktor- und Destruktoraufruf zusätzliche wertvolle Rechenzeit verbraucht, wird man bei ES wohl sehr bemüht sein, alle Objekte nach Möglichkeit statisch anzulegen, so wie es auch bei der Robotersteuerung realisiert wurde.

Eine automatische Speicherbereinigung (Garbage Collection) wird für ES wohl nicht zum Einsatz kommen können. Der Aufwand des hierfür benötigten zusätzlichen Objektcodes, der nicht explizit kontrollierbare Speicher und der hohe Rechenzeitbedarf stehen in keinem wirtschaftlichen Verhältnis zum Nutzen. Die Speicherverwaltung wird bei ES die Aufgabe des Programmierers bleiben.

Aufgrund der Sicherheitsanforderungen von ES wird eine strenge Typprüfung Grundanforderung an die verwendete Programmiersprache sein. Allerdings ist hierbei die Verwendung von dynamischer im Gegensatz zu statischer Typprüfung (erreichbar durch Templates) für harte Zeitanforderungen ebenfalls auszuschließen. Nach [Stro 92, S. 435] ist das Laufzeitverhalten bei dynamischer Typprüfung um Faktor Drei bis Zehn langsamer, als bei einer Statischen. So führten diese erhöhten Laufzeitanforderungen und der höhere Speicherplatzverbrauch dazu, daß die Klasse *Queue* im Anwendungsbeispiel neu implementiert, und nicht die Template-Realisierung der Entwicklungsumgebung wiederverwendet wurde.

Schutzmechanismen zum Erreichen der Reentrant-Fähigkeit sind bei Assembler-Realisierungen lediglich für Interrupt-Unterbrechungen zu beachten. Allermeist stehen hierfür bereits Maschinensprachebefehle zur Verfügung, die beim Eintritt in eine Interrupt-Routine die entsprechenden Register sichern und beim Verlassen restaurieren. Andere Unterbrechungen als Interrupts treten hier nicht auf. Bei multitaskingfähigen Hochsprache-Realisierungen sind die Schutzmechanismen für die Reentrant-Fähigkeit sowie zur Task-Kommunikation und -Synchronisation jedoch viel aufwendiger. Laufzeiteinbußen durch Task-Wechselzeiten, Task-Verwaltung, und eventuelle (aktive) Wartezeiten für Task-Kommunikation, Task-Synchronsiation und Reentrant-Schutz treten auf. Dies gilt jedoch generell für die multitaskingfähige Hochsprachenprogrammierung und nicht allein nur für OO.

Ein aussagekräftiger Vergleich des Laufzeitverhaltens zwischen objektorientierter und strukturierter Lösung ist leider genauso wenig möglich, wie der Vergleich des Speicherplatzverbrauches. Es ergäbe keine brauchbares Ergebnis, wenn man die Ausführungszeit der PC-Simulation mit der interpretierten Ausführungszeit auf dem Roboter vergleichen würde. Es bestünde lediglich die Möglichkeit, einzelne Teile des übersetzten C++-Codes (ohne Nutzung der Hardwarekomponenten) mit der Ausführungszeit einer reinen Assembler-Lösung zu vergleichen, was letztendlich genauso die Notwendigkeit eines hochwertigen Compilers herausstellen würde.

5.14 Anforderungen an Programmierung

Erhöhte Anforderungen an Programmierer für ES bleiben auch nach Einführung der OO bestehen, wenngleich sich die Schwerpunkte verlagern. Der wichtigste Punkt ist, daß der Pro-

grammierer die Schwächen seines Compilers kennen muß, um diese „von Hand“ beheben zu können.

Das Abwägen zwischen der Benutzung von Inline- gegenüber Member-Funktionen mit deren Vor- und Nachteilen bleibt das Selbe wie bei der prozeduralen Programmierung. So sind Inline-Funktionen nur bei sehr kleinem Funktionscode sinnvoll. Sobald der Funktionscode größer wird als der Code für den Einsprung und die Rückkehr, muß die Zeit gegen den Codeumfang abgewogen werden. Ein C++-Programmierer sollte jedoch beachten, daß eine Funktionsdefinition innerhalb einer Klassendefinition standardmäßig einer Inline-Funktion entspricht. Deshalb sollten, sofern dies nicht ausdrücklich erwünscht ist, Funktionsdefinitionen immer außerhalb von Klassendefinitionen erfolgen.[19]

In den vorhergehenden Abschnitten wurde bereits erläutert, daß das dynamische Binden mehr Speicherplatz und Rechenzeit erfordert als das Statische, und deshalb bei ES oft ungeeignet ist. Genauso muß der Einsatz von Templates mit dynamischer Typprüfung sowie die dynamische Objekterzeugung und -vernichtung individuell und compilerabhängig auf ihre Eignung bezüglich des Laufzeitverhaltens geprüft werden.

5.15 Anforderungen an Programmiersprache

Der Einsatz interpretierender Sprachen wie zum Beispiel Smalltalk ist für ES ungeeignet. Es ergibt sich (von Ausnahmen bei der Entwicklung abgesehen) gar keine Notwendigkeit für den Einsatz von Interpretern, weil sich am Code des Endproduktes ab der Fertigstellung nichts mehr ändern wird. Es wäre absolut absurd, fehleranfällig, speicherplatz- und zeitverschwendend Quellcode im ROM oder (E)EPROM abzulegen, und diesen bei jeder Ausführung neu zu interpretieren.

ES werden aufgrund ihrer Hardwaregebundenheit generell Programmiersprachen mit Möglichkeiten für direkte Zugriffe auf Speicher- und IO-Zugriffe benötigen. Zur Gewährleistung der hohen Sicherheitsanforderungen werden Sprachen mit strenger Typprüfung erforderlich sein, um Laufzeitfehler zu vermeiden und eventuell fatale Folgen zu unterbinden. Hierzu sollten die Sprachen zusätzlich Ausnahmebehandlungsmechanismen zur Verfügung stellen. Eine bessere Validation kann durch Sprachen mit Vor- und Nachbedingungen erreicht werden. Zur Erfüllung des reaktiven Verhaltens von ES muß die Programmiersprache Möglichkeiten bieten, Methoden durch Interrupts zu aktivieren.

Gerade für ES kann der Einsatz von hybriden Sprachkonzepten überaus interessant sein: Erstens könnte dadurch die Umstellung auf das neue Paradigma erleichtert werden, indem bisher

[19] Das Geheimnisprinzip ist zwar in diesem Zusammenhang irrelevant, wenngleich es ebenfalls für dieses Vorgehen spricht.

erzeugter Code in der OO-Umgebung weiterverwendet werden kann. Die Umstellung könnte somit fließend erfolgen. Zweitens könnten durch den Einsatz hybrider Konzepte viele Methodenaufrufe dadurch eingespart werden, daß niedere Abstraktionsebenen eben nicht mehr durch Objekte, sondern durch andere Sprachelemente repräsentiert sind.[20] Und drittens könnten besonders kritische Zeitabschnitte durch Inlines, vorzugsweise in Assembler, realisiert werden.

5.16 Anforderungen an Compiler

Die Anforderungen an den Compiler bei der Entwicklung von Software für ES sind unvergleichbar hoch. Nicht zuletzt werden Standards für Compiler-Techniken gerade an ES gesetzt. Schließlich ist der Compiler (in Verbindung mit dem Linker) letztendlich dafür verantwortlich, wie effektiv der abgesetzte Code in seinem Laufzeitverhalten und in seinem Speicherplatzverbrauch ist. Und wie bereits erwähnt: Bein ES in hohen Stückzahlen können die Einsparung von Bits vielstellige Einsparungen bei den Herstellungskosten des Endproduktes zur Folge haben. Da ja letztendlich immer Wirtschaftlichkeitsaspekte den Ausschlag für Entscheidungen geben, wird der Auswahl des Compilers eine Schlüsselfunktion in der Entscheidung für oder gegen das OO-Paradigma zukommen, selbst wenn die technischen Vorteile überwiegen.

Die Ausführungen von [Let 95] und [Pfei 95] über die die Effizienz von C++-Programmen enthalten wertvolle allgemeine Aspekte zur Compilerauswahl für ES: So gibt es große Unterschiede bei der Verwirklichung von Konstruktor- und Destrukturaufrufen der verschiedenen Compiler. Prinzipiell sollten bei Applikationen für ES alle Objekte statisch erzeugt werden. Wer dies jedoch dynamisch machen muß, sollte auf Effizienz des Compilers in diesem Punkt achten.

Ebenfalls große Unterschiede gibt es in der Art, wie Compiler Objekt-Code für Templates generieren. Hier sollte darauf geachtet werden, daß der Compiler keinesfalls für jedes File, in der die Template-Klasse verwendet wird, separat eine Kopie des Objekt-Codes erzeugt. Für jeden Typ ist der Objektcode lediglich einmalig notwendig. Alles andere wäre redundant und eine Speicherplatzverschwendung.

Genauso legen manche Compiler für jede Quelldatei, in der eine virtuelle Funktion aufgerufen wird, separat eine Function-Table an. Für ES sollte die viel effizientere Technik bevorzugt werden, bei der die Tabelle mit den Funktionsadressen lediglich einmalig in der Datei angelegt wird, in der die Funktion definiert ist.

[20] Ein Vergleich von C++ (die im Gegensatz zu rein-objektorientierten Sprachen als hybride Sprache angesehen werden kann) zu Smalltalk läßt diesen Unterschied besonders deutlich werden.

Eine abgeleitete Klasse, die von n Klassen mit der selben virtuellen Basisklasse abgeleitete ist, darf optimalerweise nur eine, und nicht n, Kopien der virtuellen Basisklasse anlegen.

ES benötigen Compiler, die das erstellen ROM-fähiger Applikationen unterstützen. Dazu gehören nach [MR 95] neben der Möglichkeit automatisch Start-Up-Code zu erzeugen, auch Möglichkeiten, wie das Positionieren von Daten und Code an explizite Adressen. Vielfach sollen neben Code, Heap und Stack sogar Konstanten, Strings, initialisierte Daten und nicht-initialisierte Daten getrennt auf verschiedene Segmente oder Speicherbereiche aufteilbar sein.

Neben allen angebotenen Optimierungstechniken, die ohne Zweifel Effizienzsteigerungen bringen, ist die beste Optimierung jedoch immer noch eine gute Programmiertechnik.

6 Schlußfolgerungen

Selbst wenn bisher der Codeumfang von eingebetteter Software noch Assembler-Realisierungen zuließ, so erfordert die Komplexitätszunahme inzwischen doch auch bei ES den Einsatz von methodischer Unterstützung und Hochsprachen. Das realisierte Anwendungsbeispiel zeigt, daß die Verwendung der OO selbst für den Einsatz bei ES ein vielversprechender Ansatz ist, und daß damit vielfältige Verbesserungen erzielt werden können: Senkung von Entwicklungskosten, Steigerungen in der Produktivität, Wiederverwendbarkeit, Wartbarkeit, Flexibilität, Portabilität, Verständlichkeit und teilweise in der Sicherheit sind erreichbar und die Komplexität ist leichter bewältigbar.

Zur Frage nach den Performanceauswirkungen wären weiterführende Untersuchungen zwischen dem Maschinencode aus der objektorientierten Lösungen und dem Maschinencode einer Assembler-Realisierung interessant. Sie könnten tiefere Aufschlüsse bezüglich des Laufzeitverhaltens und des Speicherplatzverbrauches geben und sollten deshalb bei Verfügbarkeit eines entsprechenden Compilers durchgeführt werden.

Doch selbst wenn diese Untersuchungen derzeit nicht möglich waren, können Performanceeinbußen durch die aufgezeigten Maßnahmen bei Compilern und bei der Programmierung vermieden werden. Außerdem kompensieren die Leistungssteigerungen in den neuen Mikrocontroller-Generationen eventuelle Performanceeinbußen bei Weitem.

Letztendlich werden Wirtschaftlichkeitsaspekte für oder gegen die Einführung des neuen Paradigmas ausschlaggebend sein. Deshalb muß individuell geklärt werden, ab wann sich Mehrausgaben zur Einführung und eventuelle Mehrausgaben für größere Controller gegenüber Einsparungen bei der Produktivitätssteigerung in der Entwicklung bezahlt machen. Mit Ausnahme von ganz kleinen Anwendungsgebieten wird sich unter Berücksichtigung der Komplexitätszunahme langfristig gesehen sicherlich die OO für viele Anwendungen rentieren. Hier ist jetzt die Industrie gefordert, auf die Entwicklung zu reagieren und entsprechende OO-Compiler auch für Mikrocontroller anzubieten.

Literatur

[Boo 94]...... BOOCH, Grady (1994): *Objektorientierte Analyse und Design: Mit praktischen Anwendungsbeispielen*. Bonn: Addison-Wesley

[Bräu 96] .... BRÄUNL, Thomas (1996): Der Teppich-Krieger. In: *c't 01/96*, S. 290-296

[Bud 95]...... BUDDE, Reinhard; SYLLA, Karl-Heinz (1995): Objektorientierte Echtzeit-Anwendungen auf Grundlage perfekter Synchronisation. In: *OBJEKTspektrum* 2/95, S 54-60

[Fied 91] ..... FIEDLER, Jörg; RIX, Karl F.; ZÖLLER, Horst (1991): *Objekt-orientierte Programmierung in der Automatisierung*. Düsseldorf: VDI-Verlag

[Hüs 95]...... HÜSENER, Thomas; BALZERT, Helmut (Hrsg.) (1995): *Objektorientierter Entwurf von nebenläufigen, verteilten und echtzeitfähigen Softwaresystemen*. Heidelberg: Spektrum, Akademischer Verlag

[Jon 93]....... JONES, Joseph L.; FLYNN, Anita M. (1993): *Mobile Robots: Inspiration to Implementation*. Wellesley: A K Peters, Ltd.

[Jon 95]....... JONES, Joseph L. (1995): *The Mobile Robot Assembly Guide*. Wellesley: A K Peters, Ltd.

[Let 95] .... LETHABY, Nick (1995): Looking at Code Size and Performance With C++. In: *NewBits*, vol. 12, no. 3, S. 4-7, Microtec Research Inc. - Firmenschrift

[Mey 90] ..... MEYER, Bertrand (1990): *Objektorientierte Softwareentwicklung*. München, Wien: Hanser Verlag; London: Prentice Hall Int.

[Mor 96] ..... MORGENROTH, Karlheiz (1996): µC-Tools: Entwicklungswerkzeuge für Mikrocontroller und Embedded Control. In: *ELRAD 1996, Heft 2*, S. 36-44

[MR 95] ...... MICROTEC RESEARCH (1995): C and C++ Compiler. In: *NewBits*, vol. 12, no. 3, Anhang S. 5-7, Microtec Research Inc. - Firmenschrift

[Pfei 95]...... PFEIFFER, Andreas (1995): *Speicherhunger und Leistung von C++ Applikationen*. München / Ottobrunn: Microtec Research GmbH. - Firmenschrift

[Schä 94] .... SCHÄFER, Steffen (1994): *Objektorientierte Entwurfsmethoden: Verfahren zum objektorientierten Softwareentwurf im Überblick*. Bonn: Addison-Wesley

[Stro 92]...... STROUSTRUP, Bjarne (1992): *Die C++ Programmiersprache*. 2. Aufl. Bonn: Addison-Wesley

Anhang

A) C++-Lösung

QUEUE.HPP

```
#if !defined QUEUE_HPP
#define QUEUE_HPP

class Prozess;

// ============================================================================
// ============================================================================
// Klasse QElement
class QElement {
  public:
    QElement();
    QElement *Naechstes;         // Zeiger auf nächstes Queue-Element
    Prozess *Inhalt;             // Zeiger auf Prozess
};

// ============================================================================
// ============================================================================
// Klasse Queue
class Queue {
  protected:
    QElement *Root;              // Root-Zeiger
    int Zaehler;                 // Zähler zur Vergabe der PID's
  public:
    Queue();
    ~Queue();                    // Destruktor zum Löschen der Queue
    int Eintragen(Prozess *);    // Aufnehmen eines neuen Prozesses
    void Loeschen(int);          // Löschen des Prozesses mit angegebener PID
    void Leeren();               // Gesamte Queue leeren
    void Weiter();               // Schaltet die Queue um ein Element weiter
    Prozess *Aktuell();          // Gibt einen Zeiger auf den aktuellen Prozess zurück
};

#include "prozess.hpp"

#endif // QUEUE_HPP
```

QUEUE.CPP

```
#include "queue.hpp"

// ============================================================================
// ============================================================================
// Klasse QElement

// ----------------------------------------------------------------------------
QElement::QElement() {
  Naechstes = 0;                 // Zeiger erden
  Inhalt = 0;                    // "
}

// ============================================================================
// ============================================================================
// Klasse Queue

// ----------------------------------------------------------------------------
Queue::Queue() {
  Root = 0;                      // Zeiger erden
  Zaehler = 0;                   // Queue mit 0 Elementen
}
```

```
// ---------------------------------------------------------------------------
Queue::~Queue() {
  Leeren();                     // vor Verlassen Queue leeeren
}

// ---------------------------------------------------------------------------
int Queue::Eintragen(Prozess *arg) {
  QElement *neu;
  QElement *temp;
  neu = new(QElement);
  Zaehler++;                    // neue PID ermitteln
  neu->Inhalt = arg;            // Zeiger auf Prozess zuweisen
  neu->Inhalt->PID = Zaehler;   // PID zuweisen
  if (Root == 0) {Root = neu; neu->Naechstes = neu;} // erstes Element anlegen
  else {temp = Root->Naechstes; Root->Naechstes = neu; neu->Naechstes = temp;} // wei-
                                tere Elemente anlegen
  return(Zaehler);              // Returnwert: zugewiesene PID
}

// ---------------------------------------------------------------------------
void Queue::Loeschen(int id) {
  QElement *temp, *nach;
  temp = Root;
  if (temp != 0) {              // Queue darf nicht leer sein
    do {
      nach = temp;              // Nachläufer
      temp = temp->Naechstes;   // Suchen
    } while(!(temp==Root || temp->Inhalt->PID==id)); //   bis Queueende oder Gefunden
    if (temp->Inhalt->PID==id) { // Prüfung, ob tatsächlich gefunden
      if (temp==Root) Root = 0; // letztes Element löschen
      else {nach->Naechstes = temp->Naechstes;}      // sonstiges Element löschen
      delete(temp);             // Speicherplatz freigeben
      Zaehler--;                // PIDnummer freigeben
    }
  }
}

// ---------------------------------------------------------------------------
Prozess * Queue::Aktuell() {
  return (Root->Inhalt);        // Zeiger auf aktuellen Prozess zurückgeben
}

// ---------------------------------------------------------------------------
void Queue::Weiter() {
  Root = Root->Naechstes;       // Queue um 1 Element weiterschalten
}

// ---------------------------------------------------------------------------
void Queue::Leeren() {
  while(Zaehler > 0) {
    Loeschen(Zaehler);
    Zaehler--;
  }
}
```

SCHED.HPP

```
#if !defined SCHED_HPP
#define SCHED_HPP

#include "queue.hpp"
#include "prozess.hpp"

// ===========================================================================
// ===========================================================================
// KLASSE SCHEDULER
class Scheduler {
  protected:
    bool Zyklisch;              // Attribut zum Öffnen / Schließen der Endlos-Schleifen
                                der Tasks
  public:
    Scheduler(bool);            // Scheduler (Initialisierung mit Attribut Zyklisch)
```

```
    virtual int Prozess_erzeugen(Prozess *) = 0;     // Methode zur Prozesserzeugung
                               mit Standardwerten; Rückgabewert: PID
    virtual int Prozess_erzeugen(Prozess *, int, int) = 0; // Methode zur Prozesser-
                               zeugung (Initialisierung mit Prozess, Rechenzeit,
                               Stackgröße) Rückgabewert: PID
    virtual void Prozess_vernichten(int) = 0; // Methode zur Prozessvernichtung
    bool Zyklustyp();          // Kapselung von Zyklisch
};


// ================================================================================
// ================================================================================
// KLASSE CYCLE_SCHEDULER
class Cycle_Scheduler : public Scheduler {
  protected:
    Queue Cyclingqueue;        // Ready-Queue
    bool Aktiv;                // Flag zum Aktivieren / Anhalten des Schedulers
  public:
    Cycle_Scheduler();
    int Prozess_erzeugen(Prozess *);    // konkrete Implementation der virtuellen Me-
                               thodendefinition aus Klasse Scheduler
    int Prozess_erzeugen(Prozess *, int , int);      // konkrete Implementation der
                               virtuellen Methodendefinition aus Klasse Scheduler
    void Prozess_vernichten(int);       // konkrete Implementation der virtuellen Me-
                               thodendefinition aus Klasse Scheduler
    void Run();                // Methode zum Aktivieren des Schedulers
    void Stop();               // Methode zum Anhalten des Schedulers
};

#endif // SCHED_HPP
```

SCHED.CPP

```
#include "sched.hpp"

// ================================================================================
// ================================================================================
// KLASSE SCHEDULER

// --------------------------------------------------------------------------------
Scheduler::Scheduler(bool zykl) {
  Zyklisch = zykl;             // Initialisierung mit Parameterwert
}

// --------------------------------------------------------------------------------
bool Scheduler::Zyklustyp() {
  return(Zyklisch);
}


// ================================================================================
// ================================================================================
// KLASSE CYCLE_SCHEDULER

// --------------------------------------------------------------------------------
Cycle_Scheduler::Cycle_Scheduler() : Scheduler(false) {
  Aktiv = false;               // Scheduler standardmäßig deaktiviert (Aktivierung mit
                               Methode RUN())
}

// --------------------------------------------------------------------------------
int Cycle_Scheduler::Prozess_erzeugen(Prozess *objekt, int, int) {
  return Prozess_erzeugen(objekt);     // Keine Möglichkeit zur Angabe von Ticks und
                               Stack
}

// --------------------------------------------------------------------------------
int Cycle_Scheduler::Prozess_erzeugen(Prozess *objekt) {
  return(Cyclingqueue.Eintragen(objekt));     // Prozess in Ready-Queue eintragen
}

// --------------------------------------------------------------------------------
void Cycle_Scheduler::Prozess_vernichten(int pid) {
```

```
  Cyclingqueue.Loeschen(pid);   // Prozess mit angegebener PID aus Queue löschen
}

// ------------------------------------------------------------------------------
void Cycle_Scheduler::Run() {
  Aktiv = true;
  while(Aktiv) {
    if (Cyclingqueue.Aktuell() != 0) Cyclingqueue.Aktuell()->Prozessaktion();  // Ak-
                                tuellen Prozess ausführen
    Cyclingqueue.Weiter();      // Prozess weiterschalten
  }
}

// ------------------------------------------------------------------------------
void Cycle_Scheduler::Stop() {
  Aktiv = false;                // Scheduler deaktivieren
}
```

ICSCHED.HPP

```
#if !defined ICSCHED_HPP
#define ICSCHED_HPP

#include "queue.hpp"
#include "prozess.hpp"

// ==============================================================================
// ==============================================================================
// KLASSE IC_SCHEDULER
class IC_Scheduler : public Scheduler {
  public:
    IC_Scheduler();
    int Prozess_erzeugen(Prozess *);
    int Prozess_erzeugen(Prozess *, int, int);
    void Prozess_vernichten(int);
};

#endif // ICSCHED_HPP
```

ICSCHED.CPP

```
#include "icsched.hpp"

// ==============================================================================
// ==============================================================================
// KLASSE IC_SCHEDULER

// ------------------------------------------------------------------------------
IC_Scheduler::IC_Scheduler() : Scheduler(true) {
}

// ------------------------------------------------------------------------------
int IC_Scheduler::Prozess_erzeugen(Prozess *objekt, int ticks, int ssize) {
  return start_process(objekt->Prozessaktion(), ticks, ssize);
}

// ------------------------------------------------------------------------------
int IC_Scheduler::Prozess_erzeugen(Prozess *objekt) {
  return Prozess_erzeugen(objekt, 5, 256);
}

// ------------------------------------------------------------------------------
void IC_Scheduler::Prozess_vernichten(int pid) {
  kill_process(pid);
}
```

MSCHED.HPP

```
#if !defined M_SCHED
```

```
#define M_SCHED

#include "queue.hpp"
#include "prozess.hpp"

// ================================================================================
// ================================================================================
// Klasse M68HC11_SCHEDULER
class M68HC11_Scheduler : public Scheduler {
  protected:
    int Tickkonto;
    Queue ReadyQueue;
    Prozess *AktProz;
    int Stackmerker;
  public:
    M68HC11_Scheduler();
    int Prozess_erzeugen(Prozess *);
    int Prozess_erzeugen(Prozess *, int, int);
    void Prozess_vernichten(int);
    void Interrupt();
};

#endif // M_SCHED
```

MSCHED.CPP

```
#include "msched.hpp"

// ================================================================================
// ================================================================================
// Klasse M68HC11_SCHEDULER

// --------------------------------------------------------------------------------
M68HC11_Scheduler::M68HC11_Scheduler() : Scheduler(true) {
}

// --------------------------------------------------------------------------------
int M68HC11_Scheduler::Prozess_erzeugen(Prozess *objekt, int ticks, int ssize) {
  objekt->Ticks = ticks;
  objekt->StackSize = ssize;
  // int temp, startadresseL, startadresseH;
  // startadresseL = &objekt::Prozessaktion;
  // startadresseH >> 8;
  // startadresseL << 8;
  // startadresseL >> 8;
  // Stack für Erstaufruf vorbelegen:
  // Interrupts sperren;
  // temp = aktueller Stackpointer;
  // aktueller Stackpointer = stackmerker;
  // push CCR;
  // push ACCB;
  // push ACCA;
  // push IXH;
  // push IXL;
  // push IYH;
  // push IYL;
  // push startadresseH;
  // push startadresseL;
  // p->sp = aktueller Stackpointer;
  // aktueller Stackpointer = temp;
  // stackmerker = stackmerker + ssize;
  // Interrupts freigeben;
  return(ReadyQueue.Eintragen(objekt));
}

// --------------------------------------------------------------------------------
int M68HC11_Scheduler::Prozess_erzeugen(Prozess *objekt) {
  return Prozess_erzeugen(objekt, 5, 256);
}

// --------------------------------------------------------------------------------
void M68HC11_Scheduler::Interrupt() {
  Tickkonto--;
```

```
  if (Tickkonto == 0) {         // Prozeßwechsel
  // - alle Register, Flags, Befehlszeiger auf aktuellem Stack sichern (automatisch
                                bei INT);
  // - Interrupts sperren;
  // - Stackpointer sichern;
  // - aktuellen Prozess in Ready-Queue hinten einreihen;
  // - neuen Prozess aus Ready-Queue herausholen;
  // - neuen Stackpointer setzen;
  // - Tickkonto setzen;
  // - Interrupts zulassen;
  // - Register regenerieren (automatisch bei RTI);
  }
};
```

MUTEX.HPP

```
#if !defined MUTEX_HPP
#define MUTEX_HPP

#include "hardware.hpp"
#include "flib.hpp"

// ================================================================================
// ================================================================================
// Klasse Mutex
class Mutex : Hardware {
  protected:
    bool Belegt;                // Status
  public:
    Mutex();
    void Betreten();            // Methode zum Betreten des geschützten Bereiches
    void Verlassen();           // Methode zum Verlassen des geschützten Bereiches
};

#endif // MUTEX_HPP
```

MUTEX.CPP

```
#include "mutex.hpp"

// ================================================================================
// ================================================================================
// Klasse Mutex

// --------------------------------------------------------------------------------
Mutex::Mutex() {
  Belegt = false;               // Status mit frei initialisieren
}

// --------------------------------------------------------------------------------
void Mutex::Betreten() {
  bool erfolg = false;
  do {
    DI();                       // Interrupts sperren (da kein atomares 'Test and Set'
                                verfügbar)
      if (!Belegt) {            // Falls frei
        Belegt = true;          //    belegen
        erfolg = true;
      }
    EI();                       // Interrupts freigeben
  } while(!erfolg);             // aktives Warten
}

// --------------------------------------------------------------------------------
void Mutex::Verlassen() {
  DI();                         // Interrupts sperren
    Belegt = false;             // freigeben
  EI();                         // Interrupts freigeben
}
```

PROZESS.HPP

```
#if !defined PROZESS_HPP
#define PROZESS_HPP

#include "flib.hpp"

class Scheduler;

// ===============================================================================
// ===============================================================================
// Klasse Prozess
class Prozess {
  protected:
    bool Zyklisch;
  public:
    Prozess(Scheduler *);
    virtual void Prozessaktion() = 0;   // virtuelle Definition von Prozessaktion
    int PID;                    // Prozess-ID (wird von Scheduler zugeteilt)
    int Ticks;                  // Laufzeit des Prozesses (für preemptiven Scheduler)
    int StackSize;
};

#include "sched.hpp"

#endif // PROZESS_HPP
```

PROZESS.CPP

```
#include "prozess.hpp"

// ===============================================================================
// ===============================================================================
// Klasse Prozess

// -------------------------------------------------------------------------------
Prozess::Prozess(Scheduler *scheduler) {
  Ticks = 0;
  StackSize = 0;
  Zyklisch = scheduler->Zyklustyp();    // Zyklustyp von Scheduler übernehmen
  scheduler->Prozess_erzeugen(this);    // Prozess beim Scheduler anmelden
}
```

HARDWARE.HPP

```
#if !defined HARDWARE_HPP
#define HARDWARE_HPP

// ===============================================================================
// ===============================================================================
// Klasse Hardware
class Hardware {
  public:
    Hardware();
    int PORTA;
    int PIOC;
    int PORTC;
    int PORTB;
    int PORTCL;
    int DDRC;
    int PORTD;
    int DDRD;
    int PORTE;
    int CFORCE;
    int OC1M;
    int OC1D;
    int TCNT;
    int TIC1;
    int TIC2;
    int TIC3;
```

```
    int TIC3INT;
    int TOC1;
    int TOC2;
    int TOC3;
    int TOC4;
    int TOC5;
    int TCTL1;
    int TCTL2;
    int TMSK1;
    int TFLG1;
    int TMSK2;
    int TFLG2;
    int PACTL;
    int PACNT;
    int SPCR;
    int SPSR;
    int SPDR;
    int BAUD;
    int SCCR1;
    int SCCR2;
    int SCSR;
    int SCDR;
    int ADCTL;
    int ADR1;
    int ADR2;
    int ADR3;
    int ADR4;
    int OPTION;
    int COPRST;
    int PPROG;
    int HPRIO;
    int INIT;
    int TEST1;
    int CONFIG;
    unsigned char Peek(int);    // Lesen des Speichers
    void Poke(int, unsigned char);      // Schreiben des Speichers (Byte)
    void PokeWord(int, int);    // Schreiben des Speichers (Word)
    void BitSet(int, int);      // Setzen einzelner Bits
    void BitClear(int, int);    // Löschen einzelner Bits
    long Time();                // Auslesen der Zeit in ms seit Systemstart
    long Tcnt();                // Simulation von TCNT-Register
    void DI();                  // Sperren aller Interrupts
    void EI();                  // Aktivieren aller Interrupts
};

#endif // HWADR_HPP
```

HARDWARE.CPP

```
#include "hardware.hpp"
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <dos.h>

int RAM[0x103f-0x1000];

// ============================================================================
// ============================================================================
// Klasse Hardware

// ----------------------------------------------------------------------------
Hardware::Hardware() {
  PORTA  = 0X1000;
  PIOC   = 0X1002;
  PORTC  = 0X1003;
  PORTB  = 0X1004;
  PORTCL = 0X1005;
  DDRC   = 0X1007;
  PORTD  = 0X1008;
  DDRD   = 0X1009;
  PORTE  = 0X100A;
  CFORCE = 0X100B;
```

```
  OC1M   = 0X100C;
  OC1D   = 0X100D;
  TCNT   = 0X100E;
  TIC1   = 0X1010;
  TIC2   = 0X1012;
  TIC3   = 0X1014;
  TOC1   = 0X1016;
  TOC2   = 0X1018;
  TOC3   = 0X101A;
  TOC4   = 0X101C;
  TOC5   = 0X101E;
  TIC3INT = 0XFFEA;
  TCTL2  = 0X1021;
  TMSK1  = 0X1022;
  TFLG1  = 0X1023;
  TMSK2  = 0X1024;
  TFLG2  = 0X1025;
  PACTL  = 0X1026;
  PACNT  = 0X1027;
  SPCR   = 0X1028;
  SPSR   = 0X1029;
  SPDR   = 0X102A;
  BAUD   = 0X102B;
  SCCR1  = 0X102C;
  SCCR2  = 0X102D;
  SCSR   = 0X102E;
  SCDR   = 0X102F;
  ADCTL  = 0X1030;
  ADR1   = 0X1031;
  ADR2   = 0X1032;
  ADR3   = 0X1033;
  ADR4   = 0X1034;
  OPTION = 0X1039;
  COPRST = 0X103A;
  PPROG  = 0X103B;
  HPRIO  = 0X103C;
  INIT   = 0X103D;
  TEST1  = 0X103E;
  CONFIG = 0X103F;
}

// --------------------------------------------------------------------------------
unsigned char Hardware::Peek(int adr) {
  int x;
  if (adr==0x1031) {           // Zufallswerte für Analogports erzeugen:
    switch(RAM[0x1030-0x1000-1]-32) {
      case 2:                  // Zufallswert Mikrophon:
              x = random(100);
              if      (x < 25) return(random(127)+1);      // 25% < 128
              else if (x < 50) return(random(127)+129);    // 25% > 128
              else             return(128);   // 50% = 128
      case 3:                  // Zufallswert Bumperkette:
              x = random(7);
              if (x < 2) return(random(145)+16);     // 2:7 x Crash
              else return(random(16));  // 5:7 x Frei
      case 5:                  // Zufallswerte Pyrosensor:
              return(random(256));
      default: printf("PORT %d undefiniert!\n", RAM[adr-0x1000-1]);
    }
  }
  return(random(256));
}

// --------------------------------------------------------------------------------
void Hardware::Poke(int adr, unsigned char wert) {
  RAM[adr-0x1000-1] = wert;
}

// --------------------------------------------------------------------------------
void Hardware::PokeWord(int adr, int wert) {
  RAM[adr-0x1000-1] = wert;
}

// --------------------------------------------------------------------------------
void Hardware::BitSet(int adr, int wert) {
  RAM[adr-0x1000-1] = RAM[adr-0x1000-1] | wert;
```

```
}

// ----------------------------------------------------------------------------
void Hardware::BitClear(int adr, int wert) {
  RAM[adr-0x1000-1] = (RAM[adr-0x1000-1] ^ (RAM[adr-0x1000-1] & wert));
}

// ----------------------------------------------------------------------------
long Hardware::Time() {
  return((long) clock());
}

// ----------------------------------------------------------------------------
long Hardware::Tcnt() {
  return(random(3000));
}

// ----------------------------------------------------------------------------
void Hardware::DI() {
  disable();
}

// ----------------------------------------------------------------------------
void Hardware::EI() {
  enable();
}
```

SENSOREN.HPP

```
#if !defined SENSOREN_HPP
#define SENSOREN_HPP

#include "hardware.hpp"
#include "bumpwert.hpp"
#include "flib.hpp"
#include "mutex.hpp"
#include "timer.hpp"

// ============================================================================
// ============================================================================
// Klasse Sensor
class Sensor : public Hardware {
  public:
    Sensor();
};


// ============================================================================
// ============================================================================
// Klasse Analogsensor
class Analogsensor : public Sensor {
  protected:
    int Port;                   // Port des Analog-Digital-Wandlers
    Mutex Schutz;               // Mutex zum exklusiven Zugriff
  public:
    Analogsensor(int);          // Konstruktor
    ~Analogsensor();            // Destruktor (zum Abschalten der Ladungspumpe)
    int Analogwert();           // Ermittelt aktuellen Analogwert
};


// ============================================================================
// ============================================================================
// Klasse Mikrophon
class Mikrophon : public Analogsensor {
  protected:
    Timer My_Timer;
    int Tonschwelle;            // Schwellwert für Tonmindestlautstärke
  public:
    Mikrophon(int);             // Mikrophon (Initialisierung mit Analogport)
    int   DurchschnittsLautstaerke();   // ermittelt die Durchschnittslautstärke inner-
                                halb einem vorgegebenen Intervall
```

```
    int   MaximalLautstaerke(); // ermittelt die Maximallautstärke innerhalb einem vor-
                                gegebenen Intervall
    bool  Ton();                // ermittelt, ob Ton vorliegt oder nicht
    long  Tondauer();           // ermittelt die Länge eines Tones
    float Frequenz();           // ermittelt die Frequenz eines Tones
};


// ================================================================================
// ================================================================================
// Klasse Bumper
class Bumper : public Analogsensor {
  public:
    Bumper(int);                // Bumper (Initialisierung mit Analogport)
    Bumperwert Kollissionsposition();   // ermittelt die Kollisionsposition
};

// ================================================================================
// ================================================================================
// Klasse Pyrosensor
class Pyrosensor : public Analogsensor {
  public:
    Pyrosensor(int);            // Pyrosensor (Initialisierung mit Analogport)
    int Temperatur();           // ermittelt einen Temperaturwert
};


// ================================================================================
// ================================================================================
// Klasse Radenkoder
class Radenkoder : public Sensor {
  protected:
    int LC;                     // Zähler der linken Impulse
    int RC;                     // Zähler der rechten Impulse
  public:
    Radenkoder();
    void Interrupt();           // Methode, die von Interrupt angestoßen wird
    int LImpulse();             // Kapselung für LC
    int RImpulse();             // Kapselung für RC
};

#endif // SENSOREN_HPP
```

SENSOREN.CPP

```
#include "sensoren.hpp"
#include "flib.hpp"
#include "stdlib.h"

// ================================================================================
// ================================================================================
// Klasse Sensor

// --------------------------------------------------------------------------------
Sensor::Sensor() : Hardware() {
}


// ================================================================================
// ================================================================================
// Klasse Analogsensor

// --------------------------------------------------------------------------------
Analogsensor::Analogsensor(int argument) : Sensor() {
  Port = argument;
  Poke(OPTION, 0x80);           // Ladungspumpe einschalten
}

// --------------------------------------------------------------------------------
Analogsensor::~Analogsensor() {
  Poke(OPTION, 0x00);           // Ladungspumpe ausschalten
}
```

```
// -------------------------------------------------------------------------------
int Analogsensor::Analogwert() {
  int wert;
  Schutz.Betreten();            // Schutz für exklusiven Zugriff
    Poke(ADCTL, 0x20 | Port);   // Aktueller Port in SCAN-Mode
    wert = Peek(ADR1);          // Wert auslesen
  Schutz.Verlassen();
  return(wert);
}


// ===============================================================================
// ===============================================================================
// Klasse Mikrophon

// -------------------------------------------------------------------------------
Mikrophon::Mikrophon(int port) : Analogsensor(port) {
  Tonschwelle = 25;
}

// -------------------------------------------------------------------------------
bool Mikrophon::Ton() {
  int geraeusch;
  geraeusch = DurchschnittsLautstaerke();
  if (geraeusch > Tonschwelle) return(true);
  else return(false);
}

// -------------------------------------------------------------------------------
int Mikrophon::DurchschnittsLautstaerke() {
  int i;
  int summe = 0;
  int intervall = 20;           // Intervall zur Bildung des Durchschnittswertes
  for (i = 0; i < intervall; i++) {
    summe = summe + Abs(Analogwert() - 128);
  }
  return (summe / intervall);
}

// -------------------------------------------------------------------------------
int Mikrophon::MaximalLautstaerke() {
  int i;
  int max = 0;
  int aktuell = 0;
  for (i = 0; i < 20; i++) {    // Intervall zur Bildung des Maximalwertes
    aktuell = Abs(Analogwert() - 128);
    if (aktuell > max) max = aktuell;
  }
  return (max);
}

// -------------------------------------------------------------------------------
long Mikrophon::Tondauer() {
  bool ein = false;
  long start = 0;
  long stop = 0;
  while(stop == 0) {
    switch (ein) {
      case false: if (Ton()) {
                    start = My_Timer.MSeconds();     // Startwert setzen
                    ein = true;
                  }
                  break;
      case true:  if (!Ton()) {
                    stop = My_Timer.MSeconds();      // Stopwert setzen
                    ein = false;
                  }
                  break;
    }
  }
  return(stop-start);
}

// -------------------------------------------------------------------------------
float Mikrophon::Frequenz() {
  long start, stop, summe;
```

```
  int i, zaehler;
  float schnitt, frequenz;
  zaehler = 0;
  summe = 0;
  if (Ton()) {                   // Frequenzerkennung nur bei anliegendem Ton
    for (i=0; i<20; i++) {
      while(Analogwert() >= 128) {;}    // Synchronisation
      while(Analogwert() < 128) {;}     // 1. Nulldurchgang
      start = My_Timer.Ticks();
      while(Analogwert() >= 128) {;}    // 2. Nulldurchgang
      stop = My_Timer.Ticks();
      if ((stop-start) >= 0) {summe = summe + (stop-start); zaehler++; }// Überlauf-
                                 schutz
    }
    if (zaehler > 0) {
      schnitt = summe / zaehler;
      frequenz = 1 /(2 * schnitt * 0.0000005);      // 1 Imp. = 500 ns; 2 Halbwellen
    }
    else frequenz = 0;
  }
  else frequenz = 0;
  return(frequenz);
}


// ===============================================================================
// ===============================================================================
// Klasse Bumper

// -------------------------------------------------------------------------------
Bumper::Bumper(int port) : Analogsensor(port) {
}

// -------------------------------------------------------------------------------
Bumperwert Bumper::Kollissionsposition() {
  int wert;
  wert = Analogwert();
  if (wert <  16) return(frei); // (  5) keiner
  if (wert <  38) return(r);    // ( 27) rechts
  if (wert <  60) return(l);    // ( 49) links
  if (wert <  81) return(lr);   // ( 71) links + rechts
  if (wert < 101) return(h);    // ( 90) hinten
  if (wert < 124) return(rh);   // (112) rechts + hinten
  if (wert < 147) return(lh);   // (135) links + hinten
  return(lrh);                  // (159) links + rechts + hinten
}


// ===============================================================================
// ===============================================================================
// Klasse Pyrosensor

// -------------------------------------------------------------------------------
Pyrosensor::Pyrosensor(int port) : Analogsensor(port) {
}

// -------------------------------------------------------------------------------
int Pyrosensor::Temperatur() {
  return(Analogwert());
}


// ===============================================================================
// ===============================================================================
// Klasse Radenkoder

// -------------------------------------------------------------------------------
Radenkoder::Radenkoder() : Sensor() {
                                // Interrupt-Handler aktivieren:
  DI();
    Poke(TIC3INT, Peek(TOC3));
  EI();
                                // Enkoder initialisieren:
  Poke(PACTL, 0x50);            // PA7 als Eingang bei steigender Flanke
  Poke(PACNT, 0);               // Initialisierung auf Null;
  LC = 0;                       // "
```

```
  RC = 0;                        // "
  BitClear(TCTL2, 2);            // TCTL2 lediglich auf letzte 2 Bits aktivieren
  BitSet(TCTL2, 1);              // "
  BitSet(TMSK1, 1);              // TMSK1 für IC3-Interrupts aktivieren
};

// ---------------------------------------------------------------------------
void Radenkoder::Interrupt() {
  RC++;
  Poke(TFLG1, 1);
}

// ---------------------------------------------------------------------------
int Radenkoder::LImpulse() {
  int wert;
  LC = random(3);                // nur zur Simulation
  wert = LC;
  LC = 0;
  return(wert);
}

// ---------------------------------------------------------------------------
int Radenkoder::RImpulse() {
  int wert;
  RC = random(3);                // nur zur Simulation
  wert = RC;
  RC = 0;
  return(wert);
}
```

AKTOREN.HPP

```
#if !defined AKTOREN_HPP
#define AKTOREN_HPP

#include "sched.hpp"
#include "hardware.hpp"
#include "gerade.hpp"
#include "VDaten.hpp"
#include "Timer.hpp"

// ===========================================================================
// ===========================================================================
// Klasse Aktor
class Aktor : public Hardware {
  public:
    Aktor();
};


// ===========================================================================
// ===========================================================================
// Klasse LCDDisplay
class LCDDisplay : public Aktor {
  public:
    LCDDisplay();
    void Anzeigen(char *, int); // Methode zum Ausgeben einer 32-Stelligen Zeichenkette
                                   und eines Wertes
};


// ===========================================================================
// ===========================================================================
// Klasse Motor
class Motor : public Aktor {
  protected:
    int Index;                   // Motorindex: 0 / 1
    int TimerIndex;              // Timer-Index
    int DIR_Mask;                // Maske für Richtungsbits
    int PWM_Mask;                // Maske für Motorports
  public:
    Motor(int);                  // Konstruktor (Initialisierungswert Motorindex)
    void Go(float);              // Methode zm Setzen der Motorgeschwindigkeit
```

```
};


// ================================================================================
// ================================================================================
// KLASSE MOTORENSTEUERUNG
// Nimmt definierte Kommandos entgegen und steuert entsprechend beide Motoren an
class Motorensteuerung {
  protected:
    Timer My_Timer;
    Geradenkontrolle My_Geradenkontrolle;     // Task zur Kontrolle der Geradeausfahrt
    Motor LMotor;                // linker Motor
    Motor RMotor;                // rechter Motor
    float LRKorrektur;           // Korrekturwert für links-/rechts-Differenz
    float Rueckzeit;             // Zeit für Rückwärtsfahrt
    float Drehdifferenz;         // indirekte Kurvengeschwindigkeit
    float Drehzeit;              // Zeit für Drehung
    void Vorwaerts(float);       // Fahrkommando(Geschwindigkeit)
    void Rueckwaerts(float);     // Fahrkommando(Geschwindigkeit)
    void Links(float, float);    // Fahrkommando(Geschwindigkeit, Drehdifferenz)
    void Rechts(float, float);   // Fahrkommando(Geschwindigkeit, Drehdifferenz)
    void Stop();                 // Fahrkommando
  public:
    Motorensteuerung(Scheduler *);
    void Ausfuehren(Aktionen, float);  // Methode zum aktivieren eines Aktionskomman-
                                 dos mit definierter Geschwindigkeit
    void Geradeaus(float);       // Aktionskommando(Geschwindigkeit)
    void Zurueck(float);         // Aktionskommando(Geschwindigkeit)
    void Anhalten();             // Aktionskommando
    void Linksdrehung(float);    // Aktionskommando(Geschwindigkeit)
    void Rechtsdrehung(float);   // Aktionskommando(Geschwindigkeit)
    void LinksRueckdrehung(float);      // Aktionskommando(Geschwindigkeit)
    void RechtsRueckdrehung(float);     // Aktionskommando(Geschwindigkeit)
    void Linkswende(float);      // Aktionskommando(Geschwindigkeit)
    void Rechtswende(float);     // Aktionskommando(Geschwindigkeit)
};

#endif // AKTOREN_HPP
```

AKTOREN.CPP

```
#include "aktoren.hpp"
#include <stdio.h>

// ================================================================================
// ================================================================================
// Klasse Aktor

// --------------------------------------------------------------------------------
Aktor::Aktor() : Hardware() {
}


// ================================================================================
// ================================================================================
// Klasse LCDDisplay

// --------------------------------------------------------------------------------
LCDDisplay::LCDDisplay() : Aktor() {
}

// --------------------------------------------------------------------------------
void LCDDisplay::Anzeigen(char text[32], int wert) {
  printf("%s", text);           // Text ausgeben
  if (wert != 0) printf("%d", wert);   // ggf. Wert ausgeben
  printf("\n");
}


// ================================================================================
// ================================================================================
// Klasse Motor
```

```
// ---------------------------------------------------------------------------
Motor::Motor(int argument) : Aktor() {
  Index = argument;
  switch (Index) {
  case 0: TimerIndex = 0x1018;  // Index für Timer-Register
          DIR_Mask = 0x10;      // Richtungs-Bit Port D
          PWM_Mask = 0x40;      // PWM-Bits Port A
          break;
  case 1: TimerIndex = 0x101A;  // Index für Timer-Register
          DIR_Mask = 0x20;      // Richtungs-Bit Port D
          PWM_Mask = 0x20;      // PWM-Bits Port A
          break;
  }
                                // Initialisierungen:
  BitSet(DDRD, 0x30);           // PD4 u. PD5 als Ausgang für Motorrichtung
  Poke(OC1M, 0x60);             // OC1M für PA5 und PA6 aktivieren
  BitSet(TCTL1, 0xa0);          // OC3 -> PA5, OC2 -> PA6
  BitClear(TCTL1, 0x50);        // "
  PokeWord(TOC2, 0);            // OC1 reagiert für TCNT = 0
};

// ---------------------------------------------------------------------------
void Motor::Go(float geschwindigkeit) { // Entspricht IC-Funktion: "motor(int index,
                                float vel)"
  float absolutgeschwindigkeit;
  if (geschwindigkeit > 0.0) {  // Vorwärts:
    BitSet(PORTD, DIR_Mask);
    absolutgeschwindigkeit = geschwindigkeit;
  }
  else {                        // Rückwärts:
    BitClear(PORTD, DIR_Mask);
    absolutgeschwindigkeit = (-geschwindigkeit);
  }
  if (absolutgeschwindigkeit < 1.0) BitClear(OC1D, PWM_Mask);     // Motor abschalten
  else BitSet(OC1D, PWM_Mask);  // Motor von OC1 steuern
  if (absolutgeschwindigkeit > 99.0) absolutgeschwindigkeit = 99.0;
  PokeWord(TimerIndex, (int) (655.56 * absolutgeschwindigkeit));
}


// ===========================================================================
// ===========================================================================
// KLASSE MOTORENSTEUERUNG

// ---------------------------------------------------------------------------
Motorensteuerung::Motorensteuerung(Scheduler *sched) : My_Geradenkontrolle(sched),
                                LMotor(0), RMotor(1) {
  Rueckzeit = 1.0;
  Drehdifferenz = 20.0;
  Drehzeit = 0.8;
}

// Private Grundfunktionen:
// ---------------------------------------------------------------------------
void Motorensteuerung::Vorwaerts(float v) {
  My_Geradenkontrolle.Setze_aktiv();
  My_Geradenkontrolle.Setze_Geschwindigkeit(v);
  LMotor.Go(v - My_Geradenkontrolle.Korrekturfaktor());
  RMotor.Go(v + My_Geradenkontrolle.Korrekturfaktor());
}

// ---------------------------------------------------------------------------
void Motorensteuerung::Rueckwaerts(float v) {
  My_Geradenkontrolle.Setze_inaktiv();
  My_Geradenkontrolle.Setze_Geschwindigkeit(v);
  LMotor.Go((-v) - My_Geradenkontrolle.Korrekturfaktor());
  RMotor.Go((-v) + My_Geradenkontrolle.Korrekturfaktor());
}

// ---------------------------------------------------------------------------
void Motorensteuerung::Links(float v, float c) {
  My_Geradenkontrolle.Setze_inaktiv();
  My_Geradenkontrolle.Setze_Geschwindigkeit(v);
  LMotor.Go(v + c - My_Geradenkontrolle.Korrekturfaktor());
  RMotor.Go(v - c + My_Geradenkontrolle.Korrekturfaktor());
}
```

```
// ---------------------------------------------------------------------------
void Motorensteuerung::Rechts(float v, float c) {
  My_Geradenkontrolle.Setze_inaktiv();
  My_Geradenkontrolle.Setze_Geschwindigkeit(v);
  LMotor.Go(v - c - My_Geradenkontrolle.Korrekturfaktor());
  RMotor.Go(v + c + My_Geradenkontrolle.Korrekturfaktor());
}

// ---------------------------------------------------------------------------
void Motorensteuerung::Stop() {
  My_Geradenkontrolle.Setze_inaktiv();
  My_Geradenkontrolle.Setze_Geschwindigkeit(0);
  LMotor.Go(0);
  RMotor.Go(0);
}

// Öffentliche Motoraktionen:
// ---------------------------------------------------------------------------
void Motorensteuerung::Geradeaus(float v) {
  Vorwaerts(v);
}

// ---------------------------------------------------------------------------
void Motorensteuerung::Zurueck(float v) {
  Rueckwaerts(v);
  My_Timer.Sleep(Rueckzeit);
}

// ---------------------------------------------------------------------------
void Motorensteuerung::Anhalten() {
  Stop();
}

// ---------------------------------------------------------------------------
void Motorensteuerung::Linksdrehung(float v) {
  Links(v, Drehdifferenz);
  My_Timer.Sleep(Drehzeit);
}

// ---------------------------------------------------------------------------
void Motorensteuerung::Rechtsdrehung(float v) {
  Rechts(v, Drehdifferenz);
  My_Timer.Sleep(Drehzeit);
}

// ---------------------------------------------------------------------------
void Motorensteuerung::LinksRueckdrehung(float v) {
  Links(-v, Drehdifferenz);
  My_Timer.Sleep(Drehzeit);
}

// ---------------------------------------------------------------------------
void Motorensteuerung::RechtsRueckdrehung(float v) {
  Rechts(-v, Drehdifferenz);
  My_Timer.Sleep(Drehzeit);
}

// ---------------------------------------------------------------------------
void Motorensteuerung::Linkswende(float v) {
  Links(v, Drehdifferenz);
  My_Timer.Sleep(2.0 * Drehzeit);
}

// ---------------------------------------------------------------------------
void Motorensteuerung::Rechtswende(float v) {
  Rechts(v, Drehdifferenz);
  My_Timer.Sleep(2.0 * Drehzeit);
}

// ---------------------------------------------------------------------------
void Motorensteuerung::Ausfuehren(Aktionen kommando, float geschwindigkeit) {
  switch (kommando) {
    case KGeradeaus:          Geradeaus(geschwindigkeit); break;
    case KZurueck:            Zurueck(geschwindigkeit); break;
    case KAnhalten:           Anhalten(); break;
```

```
    case KLinksdrehung:       Linksdrehung(geschwindigkeit); break;
    case KRechtsdrehung:      Rechtsdrehung(geschwindigkeit); break;
    case KLinksRueckdrehung:  LinksRueckdrehung(geschwindigkeit); break;
    case KRechtsRueckdrehung: RechtsRueckdrehung(geschwindigkeit); break;
    case KLinkswende:         Linkswende(geschwindigkeit); break;
    case KRechtswende:        Rechtswende(geschwindigkeit); break;
    default: ;
  }
}
```

VDATEN.HPP

```
#if !defined DATEN_HPP
#define DATEN_HPP

// ===============================================================================
// ===============================================================================
// Mögliche Fahrkommandos
enum Aktionen {KLeer,
               KGeradeaus,
               KZurueck,
               KAnhalten,
               KLinksdrehung,
               KRechtsdrehung,
               KLinksRueckdrehung,
               KRechtsRueckdrehung,
               KLinkswende,
               KRechtswende
};


// ===============================================================================
// ===============================================================================
// Klasse Verhaltensdaten

class Verhaltensdaten {
  public:
    Verhaltensdaten();
    Aktionen Aktion;            // Fahrkommandos
    char *Displaytext[32];      // Anzeigetext
    int Displaywert;            // Anzeigewert
    float Geschwindigkeit;      // Geschwindigkeit
    void Ausgabe();             // Methode zur Datenausgabe
};

#endif // DATEN_HPP
```

VDATEN.CPP

```
#include "VDaten.hpp"
#include <stdio.h>

// ===============================================================================
// ===============================================================================
// Klasse Verhaltensdaten

// -------------------------------------------------------------------------------
Verhaltensdaten::Verhaltensdaten() {
  Aktion = KLeer;
  *Displaytext = "";
  Displaywert = 0;
  Geschwindigkeit = 0.0;
}

// -------------------------------------------------------------------------------
void Verhaltensdaten::Ausgabe() {
  char *Kommando[20];
  switch(Aktion) {
    case KLeer:  *Kommando = "Leer"; break;
    case KGeradeaus:  *Kommando = "Vorwaerts"; break;
    case KZurueck:  *Kommando = "Rückwaerts"; break;
```

```
    case KAnhalten:  *Kommando = "Stop"; break;
    case KLinksdrehung:  *Kommando = "LinksDrehung"; break;
    case KRechtsdrehung:  *Kommando = "RechtsDrehung"; break;
    case KLinksRueckdrehung:  *Kommando = "LinksRueckDrehung"; break;
    case KRechtsRueckdrehung:  *Kommando = "RechtsRueckDrehung"; break;
    case KLinkswende:  *Kommando = "LinksWende"; break;
    case KRechtswende:  *Kommando = "RechtsWende"; break;
    default: *Kommando = "FEHLER!!!";
  }
  if (*Kommando != KLeer) printf("Motoraktion: %s", *Kommando);
  if (*Displaytext != "") printf("; Text: %s", *Displaytext);
  if (Displaywert != 0) printf("%d", Displaywert);
  if (Geschwindigkeit != 0) printf("; Geschw.: %.1f", Geschwindigkeit);
  printf(";\n");
}
```

VERHALT.HPP

```
#if !defined VERHALT_HPP
#define VERHALT_HPP

#include "prozess.hpp"
#include "sensoren.hpp"
#include "aktoren.hpp"
#include "sched.hpp"
#include "flib.hpp"
#include "vdaten.hpp"
#include "bumpwert.hpp"

// ============================================================================
// ============================================================================
// Klasse Verhalten
class Verhalten : public Prozess {
  protected:
    bool Status;                // Flag zur Meldung von Ressourcenanforderungen
    Verhaltensdaten Zustand;    // Eigene Verhaltensdaten als Zustand
    Mutex Datenschutz;          // Mutex zum Reentrant-Schutz
  public:
    Verhalten(Scheduler *);
    Verhaltensdaten *Daten();   // Kapselung von Zustand
    bool Aktiv();               // Kapselung von Status
};


// ============================================================================
// ============================================================================
// Klasse Standardverhalten
class Standardverhalten : public Verhalten {
  public:
    Standardverhalten(Scheduler *);
    void Prozessaktion();
};


// ============================================================================
// ============================================================================
// Klasse Akustik
class Akustik : public Verhalten {
  protected:
    Mikrophon My_Mikrophon;
  public:
    Akustik(Scheduler *);
    void Prozessaktion();
};


// ============================================================================
// ============================================================================
// Klasse Haptik
class Haptik : public Verhalten {
  protected:
    Bumper My_Bumper;
  public:
```

```
    Haptik(Scheduler *);
    void Prozessaktion();
};


// ================================================================================
// ================================================================================
// Klasse Temperaturmessung
class Temperaturmessung : public Verhalten {
  protected:
    Pyrosensor My_Pyrosensor;
  public:
    Temperaturmessung(Scheduler *);
    void Prozessaktion();
};


// ================================================================================
// ================================================================================
// Klasse Gesamtverhalten
class Gesamtverhalten : public Verhalten {
  protected:
    LCDDisplay My_LCDDisplay;
    Motorensteuerung My_Motorensteuerung;
  public:
    Gesamtverhalten(Scheduler *);
    void Prozessaktion();
    void Datenuebernahme(Verhaltensdaten *);  // Methode zur Übernahme der Daten eines
                                Verhaltens
};

#endif // VERHALT_HPP
```

VERHALT.CPP

```
#include "verhalt.hpp"
#include <stdio.h>

// ================================================================================
// ================================================================================
// Klasse Verhalten

// --------------------------------------------------------------------------------
Verhalten::Verhalten(Scheduler *sched) : Prozess(sched) {
    Status = false;             // Verhalten standardmäßig deaktiviert
}

// --------------------------------------------------------------------------------
bool Verhalten::Aktiv() {
  return Status;
}

// --------------------------------------------------------------------------------
Verhaltensdaten* Verhalten::Daten() {
  Verhaltensdaten *arg;
  Datenschutz.Betreten();
    arg = &Zustand;
  Datenschutz.Verlassen();
  return(arg);
}


// ================================================================================
// ================================================================================
// Klasse Akustik

// --------------------------------------------------------------------------------
Akustik::Akustik(Scheduler *sched) : Verhalten(sched), My_Mikrophon(2) {
}

// --------------------------------------------------------------------------------
void Akustik::Prozessaktion() {
  int frequenz;
```

```
  const int hoch = 2000;
  const int tief = 1000;
  do {
    frequenz = My_Mikrophon.Frequenz();
    Datenschutz.Betreten();
      if ((frequenz >= tief) && (frequenz < hoch)) { // Frequenz tief
        *Zustand.Displaytext = "Ton tief: ";
        Zustand.Displaywert = frequenz;
        Zustand.Aktion = KLinksdrehung;
        Zustand.Geschwindigkeit = 40;
        Status = true;
      }
      else if (frequenz >= hoch) {      // Frequenz hoch
        *Zustand.Displaytext = "Ton hoch: ";
        Zustand.Displaywert = frequenz;
        Zustand.Aktion = KRechtsdrehung;
        Zustand.Geschwindigkeit = 40;
        Status = true;
      }
      else Status = false;      // keine Aktion
      if (Status) {printf("AKUSTIK(%d): ", Status); Zustand.Ausgabe();}
    Datenschutz.Verlassen();
  } while (Zyklisch);
}


// ================================================================================
// ================================================================================
// Klasse Haptik

// --------------------------------------------------------------------------------
Haptik::Haptik(Scheduler *sched) : Verhalten(sched), My_Bumper(3) {
}

// --------------------------------------------------------------------------------
void Haptik::Prozessaktion() {
  Bumperwert crash;
  do {
    crash = My_Bumper.Kollissionsposition();
    Datenschutz.Betreten();
      if (crash == frei) Status = false;      // Normallfall -> keine Änderung
      else if (crash == lr) {   // Crash vorn (links + rechts)
        Zustand.Aktion = KLinksRueckdrehung;
        Zustand.Geschwindigkeit = 40;
        *Zustand.Displaytext = "Crash vorn";
        Status = true;
      }
      else if (crash == l) {    // Crash links
        Zustand.Aktion = KRechtsRueckdrehung;
        Zustand.Geschwindigkeit = 40;
        *Zustand.Displaytext = "Crash links";
        Status = true;
      }
      else if (crash == r) {    // Crash rechts
        Zustand.Aktion = KLinksRueckdrehung;
        Zustand.Geschwindigkeit = 40;
        *Zustand.Displaytext = "Crash rechts";
        Status = true;
      }
      else if (crash == h) {    // Crash hinten
        Zustand.Aktion = KGeradeaus;
        Zustand.Geschwindigkeit = 40;
        *Zustand.Displaytext = "Crash hinten";
        Status = true;
      }
      if (Status) {printf("HAPTIK(%d): ", Status); Zustand.Ausgabe();}
    Datenschutz.Verlassen();
  } while (Zyklisch);
}


// ================================================================================
// ================================================================================
// Klasse Temperaturmessung

// --------------------------------------------------------------------------------
```

```
Temperaturmessung::Temperaturmessung(Scheduler *sched) : Verhalten(sched),
                               My_Pyrosensor(5) {
}

// ------------------------------------------------------------------------------
void Temperaturmessung::Prozessaktion() {
  int temperatur;
  do {
    temperatur = My_Pyrosensor.Temperatur();
    Datenschutz.Betreten();
      Zustand.Aktion = KLeer;
      *Zustand.Displaytext = "Temperatur: ";
      Zustand.Displaywert = temperatur;
      Status = true;
      if (Status) {printf("TEMPERATURMESSUNG(%d): ", Status); Zustand.Ausgabe();}
    Datenschutz.Verlassen();
  } while (Zyklisch);
}


// ==============================================================================
// ==============================================================================
// Klasse Standardverhalten

// ------------------------------------------------------------------------------
Standardverhalten::Standardverhalten(Scheduler *sched) : Verhalten(sched) {
}

// ------------------------------------------------------------------------------
void Standardverhalten::Prozessaktion() {
  do {
    Datenschutz.Betreten();
      Zustand.Aktion = KGeradeaus;
      Zustand.Geschwindigkeit = 50;
      *Zustand.Displaytext = "";
      Status = true;
      if (Status) {printf("STANDARDVERHALTEN(%d): ", Status); Zustand.Ausgabe();}
    Datenschutz.Verlassen();
  } while (Zyklisch);
}


// ==============================================================================
// ==============================================================================
// Klasse Gesamtverhalten

// ------------------------------------------------------------------------------
Gesamtverhalten::Gesamtverhalten(Scheduler *sched) : Verhalten(sched),
                               My_Motorensteuerung(sched) {
  Status = true;                 // Gesamtverhalten standardmäßig aktiviert
}

// ------------------------------------------------------------------------------
void Gesamtverhalten::Prozessaktion() {
  do {
    Datenschutz.Betreten();
      My_Motorensteuerung.Ausfuehren(Zustand.Aktion, Zustand.Geschwindigkeit);
      My_LCDDisplay.Anzeigen(*Zustand.Displaytext, Zustand.Displaywert);
    Datenschutz.Verlassen();
  } while(Zyklisch);
}

// ------------------------------------------------------------------------------
void Gesamtverhalten::Datenuebernahme(Verhaltensdaten *Quelle) {
  do {
    Datenschutz.Betreten();
      if (Quelle->Aktion != KLeer) Zustand.Aktion = Quelle->Aktion;
      if (*Quelle->Displaytext != "") {
        *Zustand.Displaytext = *Quelle->Displaytext;
        Zustand.Displaywert = Quelle->Displaywert;
      }
      if (Quelle->Geschwindigkeit != 0.0) Zustand.Geschwindigkeit = Quelle-
                               >Geschwindigkeit;
    Datenschutz.Verlassen();
  } while(Zyklisch);
}
```

RESCON.HPP

```
#if !defined RESSCON_HPP
#define RESSCON_HPP

#include "prozess.hpp"
#include "sched.hpp"
#include "verhalt.hpp"

// ================================================================================
// ================================================================================
// Klasse Ressourcenkontrolle
// (verteilt je nach Anforderung der einzelnen Verhalten die Ressourcen)

class Ressourcenkontrolle : public Prozess {
  protected:
    Standardverhalten My_Standardverhalten;
    Akustik           My_Akustik;
    Haptik             My_Haptik;
    Temperaturmessung My_Temperaturmessung;
    Gesamtverhalten   My_Gesamtverhalten;
  public:
    Ressourcenkontrolle(Scheduler *);
    void Prozessaktion();
};

#endif // RESSCON_HPP
```

RESCON.CPP

```
#include "rescon.hpp"
#include <stdio.h>

// ================================================================================
// ================================================================================
// Klasse Ressourcenkontrolle

// --------------------------------------------------------------------------------
Ressourcenkontrolle::Ressourcenkontrolle(Scheduler *sched) : Prozess(sched),
                               My_Standardverhalten(sched), My_Haptik(sched),
                               My_Akustik(sched), My_Temperaturmessung(sched),
                               My_Gesamtverhalten(sched) {
}

// --------------------------------------------------------------------------------
void Ressourcenkontrolle::Prozessaktion() {
  do {
    My_Gesamtverhalten.Datenuebernahme(My_Standardverhalten.Daten());
    if (My_Temperaturmessung.Aktiv())
                               My_Gesamtverhalten.Datenuebernahme(My_Temperaturmessung
                               .Daten());
    if (My_Akustik.Aktiv()) My_Gesamtverhalten.Datenuebernahme(My_Akustik.Daten());
    if (My_Haptik.Aktiv()) My_Gesamtverhalten.Datenuebernahme(My_Haptik.Daten());
    printf("RESSOURCENKONTROLLE\n");
  } while(Zyklisch);
}
```

GERADE.HPP

```
#if !defined GERADENKONTROLLE
#define GERADENKONTROLLE

#include "prozess.hpp"
#include "sched.hpp"
#include "mutex.hpp"
#include "sensoren.hpp"
#include "flib.hpp"
#include "timer.hpp"

// ================================================================================
// ================================================================================
```

```
// KLASSE Geradenkontrolle
class Geradenkontrolle : public Prozess  {
  protected:
    Timer My_Timer;             // Timer
    Radenkoder Enkoder;         // Radenkoder
    int  L_Clicks;              // Impulse links
    int  R_Clicks;              // Impulse rechts
    long  L_Sum;                // Summe der linken Impulse
    long  R_Sum;                // Summe der rechten Impulse
    float Korrektur;            // Korrekturfaktor für die L/R-Ansteuerung
    float Geschwindigkeit;      // aktuelle Geschwindigkeit
    bool Aktiv;                 // Attribut zum Aktivieren / Deaktivieren der Geraden-
                                kontrolle
    Mutex Korrekturschutz;      // Mutex zum Schutz exklusiven Zugriffes auf Korrektur
    Mutex Geschwindigkeitsschutz;       // Mutex zum Schutz exklusiven Zugriffes auf
                                Geschwindigkeit
  public:
    Geradenkontrolle(Scheduler *);
    void Prozessaktion();
    long Links();               // Kapselung von L_Clicks
    long Rechts();              // Kapselung von R_Clicks
    float Korrekturfaktor();    // Kapselung von Korrektur
    void Setze_aktiv();         // Aktivierung der Geradenkontrolle
    void Setze_inaktiv();       // Deaktivierung der Geradenkontrolle
    void Setze_Geschwindigkeit(float);  // Setzen der Geschwindigkeit
};

#endif // GERADENKONTROLLE
```

GERADE.CPP

```
#include "gerade.hpp"
#include "stdio.h"

// ================================================================================
// ================================================================================
// KLASSE Geradenkontrolle

// --------------------------------------------------------------------------------
Geradenkontrolle::Geradenkontrolle(Scheduler *sched) : Prozess(sched), Geschwindig-
                                keitsschutz(), Korrekturschutz() {
  L_Clicks = 0;
  R_Clicks = 0;
  L_Sum = 0;
  R_Sum = 0;
  Korrektur = 0;
  Geschwindigkeit = 0;
  Aktiv = false;                // Geradenkontrolle standardmäßig deaktiviert
}

// --------------------------------------------------------------------------------
long Geradenkontrolle::Links() {
  return(L_Sum);
}

// --------------------------------------------------------------------------------
long Geradenkontrolle::Rechts() {
  return(R_Sum);
}

// --------------------------------------------------------------------------------
void Geradenkontrolle::Prozessaktion() {
  long differenz = 0;           // aktuelle Differenz
  long differenz_alt = 0;       // vorige Differenz
  float ti = 0.0;               // Zeitkonstante der Strecke
  float t = 1.0;                // Meßintervall
  float k = 0.2;                // Verstärkungsfaktor
  bool verwerfen = false;       // Attribut zum Unterdrücken von Impulsen nach Kurven-
                                fahrt
  do {
    L_Clicks = Enkoder.LImpulse();      // aktuelle Impulse lesen
    R_Clicks = Enkoder.RImpulse();      // "
    if (Aktiv) {                // nur Radbewegungen von Geradeauslauf berücksichtigen
```

```
      if (verwerfen) verwerfen = false; // erste Werte nach Kurvenfahrt unterdrücken
      else {
        L_Sum = L_Sum + L_Clicks;       // Gesamtstrecke links
        R_Sum = R_Sum + R_Clicks;       // Gesamtstrecke rechts
        differenz = L_Sum - R_Sum;      // Differenz bezogen auf Gesamtstrecke
        Geschwindigkeitsschutz.Betreten();
          if (Geschwindigkeit != 0) ti = L_Clicks / Geschwindigkeit;
        Geschwindigkeitsschutz.Verlassen();
        if (ti != 0) {
          Korrekturschutz.Betreten();
            Korrektur = Korrektur + k * (1 + (t/ti)) * differenz - k * differenz_alt;
                                 // PI-Regler für IT1-Regelstrecke
          Korrekturschutz.Verlassen();
          differenz_alt = differenz;    // Differenz merken
        }
      }
    }
    else verwerfen = true;
    My_Timer.Sleep(t);          // Meßintervall warten
    printf("GERADENKONTROLLE: Differenz: %ld; Korrektur: %.1f\n", differenz, Korrek-
                                tur);
  } while(Zyklisch);
}


// --------------------------------------------------------------------------------
void Geradenkontrolle::Setze_aktiv() {
  Aktiv = true;
}

// --------------------------------------------------------------------------------
void Geradenkontrolle::Setze_inaktiv() {
  Aktiv = false;
}

// --------------------------------------------------------------------------------
void Geradenkontrolle::Setze_Geschwindigkeit(float v) {
  Geschwindigkeitsschutz.Betreten();
    Geschwindigkeit = v;
  Geschwindigkeitsschutz.Verlassen();
}

// --------------------------------------------------------------------------------
float Geradenkontrolle::Korrekturfaktor() {
  float wert;
  Korrekturschutz.Betreten();
    wert = Korrektur;
  Korrekturschutz.Verlassen();
  return(wert);
}
```

TIMER.HPP

```
#if !defined TIMER_HPP
#define TIMER_HPP

#include "hardware.hpp"

// ================================================================================
// ================================================================================
// Klasse Timer
class Timer : public Hardware {
  public:
    Timer();
    long MSeconds();            // Ermittelt die seit Programmstart vergangene Zeit in
                                ms
    long Ticks();               // Simulation des TCNT-Registers
    void Sleep(long);           // Watetet die gegebene Anzahl an ms
};

#endif // TIMER_HPP
```

TIMER.CPP

```
#include "timer.hpp"

// ============================================================================
// ============================================================================
// Klasse Timer

// ----------------------------------------------------------------------------
Timer::Timer() : Hardware() {
}

// ----------------------------------------------------------------------------
long Timer::MSeconds() {
  return(Time());
}

// ----------------------------------------------------------------------------
void Timer::Sleep(long wzeit) {
  long start;
  start = MSeconds();
  while (start + wzeit > MSeconds());
}

// ----------------------------------------------------------------------------
long Timer::Ticks() {
  return(Tcnt());              // Simulationswert
}
```

FLIB.HPP

```
#if !defined FLIB_HPP
#define FLIB_HPP

// ============================================================================
// ============================================================================
// STANDARDFUNKTIONEN

// ----------------------------------------------------------------------------
enum bool {false=0, true=1};   // Definiert boolsche Werte

// ----------------------------------------------------------------------------
int Abs(int arg);              // Funktion zum ermitteln des Absolutbetrages

#endif // FLIB_HPP
```

FLIB.CPP

```
#include "flib.hpp"

// ============================================================================
// ============================================================================
// STANDARDFUNKTIONEN

// ----------------------------------------------------------------------------
int Abs(int arg) {
  if (arg<0) return -arg;
  else return arg;
}
```

MAIN.CPP

```
#include "sched.hpp"
#include "rescon.hpp"

// ============================================================================
// ============================================================================
// HAUPTPOROGRAMM ROBOBTERSTEUERUNG
```

```
int main() {
  Cycle_Scheduler My_Scheduler;
  Ressourcenkontrolle My_Ressourcenkontrolle(&My_Scheduler);
  My_Scheduler.Run();
  return 0;
}
```

BUMPWERT.HPP

```
enum Bumperwert {frei=0, h=1, r=2, rh=3, l=4, lh=5, lr=6, lrh=7};
```

B) IC-Lösung

ROBOT.LIS

```
speed.icb          /* Interruptroutine (speed.asm) für Shaft-Encoder */
shaft.c            /* IC-Funktionen zum Lesen der Shaft-Encoder-Impulse */
frequenz.icb       /* Assemblerroutine (frequenz.asm) zur Frequenzerkennung */
robot.c            /* IC-Programm Robotersteuerung */
```

IC-Programm: ROBOT.C

```
/****************************************************************************/
/****************************************************************************/
/*                   ROBOTERANSTEUERUNG MIT INTERACTIVE C (IC)              */
/*                                                                          */
/*                Prozedurales Referenzprogramm zur Diplomarbeit:           */
/*       "Objektorientierte Softwareentwicklung für eingebettete Systeme"   */
/*                            Reinhard Hanselmann                           */
/****************************************************************************/
/****************************************************************************/

float Globalgeschwindigkeit = 40.0;
int   Ton = 1;

/* Konstanten: ------------------------------------------------------------ */
int   False     = 0;
int   True      = 1;
int   Leer      = 0;
float LeerF     = 0.0;
char  LeerS[32] = "";

/* Funktion ABS: ---------------------------------------------------------- */
int Abs(int arg) {
  if (arg < 0) return(-arg);
  else return(arg);
}

/* Funktion LCD_Anzeige: -------------------------------------------------- */
void LCDAnzeige(char text[32], int wert) {
  printf("%s", text);
  if (wert != Leer) printf(" %d", wert);
  printf("\n");
}

/* Funktion Bumper_Kollissionsposition: ----------------------------------- */
int Bumper_Kollissionsposition() {
  int wert;
  wert = analog(3);
  if (wert <  16) return(0);    /* (  5) keiner */
  if (wert <  38) return(2);    /* ( 27) Rechts */
  if (wert <  60) return(4);    /* ( 49) Links */
  if (wert <  81) return(6);    /* ( 71) Links + Rechts */
  if (wert < 101) return(1);    /* ( 90) hinten */
  if (wert < 124) return(3);    /* (112) Rechts + hinten */
  if (wert < 147) return(5);    /* (135) Links + hinten */
  return(7);                    /* (159) Links + Rechts + hinten */
}

/* Funktion Mikrophon_Durchschnittslautstaerke: ---------------------------- */
int Mikrophon_DurchschnittsLautstaerke() {
  int i;
  int summe = 0;
  int anzahl = 20;
  for (i = 0; i < anzahl; i++) {
    summe = summe + Abs(analog(2) - 128);
  }
  return (summe / anzahl);
}

/* Funktion Mikrophon_Ton: ------------------------------------------------ */
```

```
int Mikrophon_Ton() {
  int geraeusch;
  int tonschwelle = 20;
  geraeusch = Mikrophon_DurchschnittsLautstaerke();
  if (geraeusch > tonschwelle) return(True);
  else return(False);
}

/* Funktion Mikrophon_Frequenz: --------------------------------------------- */
float Mikrophon_Frequenz() {
  float summe = 0.0;
  int i, anzahl;
  float frequenz;
  if (Mikrophon_Ton()) {
    poke(0x1039, 0b10000000);
    poke(0x1030, 0b00100010);
    anzahl = 50;
    for (i=0; i<anzahl; i++) summe = summe + (float) freq(0);
    frequenz = 1.0 / (2.0 * (summe /(float) anzahl) * 0.0000005);
    return(frequenz);
  }
  else return(0.0);
}

/* Task Geradenkontrolle: --------------------------------------------------- */
float Korrektur = 0.0;
int   Gerade = False;

void Geradenkontrolle() {
  int links, rechts;
  int lsum, rsum;
  int verwerfen = False;
  float diff, diff_alt, k, t, ti;
  init_velocity();             /* Rad-Encoder initialisieren */
  t = 1.0;
  diff_alt = Korrektur;
  k = 0.3;
  while(1) {
    links = get_left_clicks();
    rechts = get_right_clicks();
    if (Gerade) {
      if (verwerfen) verwerfen = False;
      else {
        lsum = lsum + links;
        rsum = rsum + rechts;
        diff = (float) (lsum - rsum);
        if (links > 0) {
          ti = (float) links / Gesamt_Geschwindigkeit;
          Korrektur = Korrektur + k * (1.0 + (t/ti)) * diff - k * diff_alt;    /* PI-
                                Regler (IT1-Strecke) */
          diff_alt = diff;
        }
      }
    }
    else verwerfen = True;
    sleep(t);
    /* printf("D:%f K:%f\n", diff, Korrektur); */
  }
}

/* Funktion Fahre: ---------------------------------------------------------- */
void Fahre(float v, float d) {
  motor(0, (v - d));
  motor(1, (v + d));
}

/* Funktion Motorensteuerung: ----------------------------------------------- */
int M_Halt        = 1;
int M_Links       = 2;
int M_Rechts      = 3;
int M_Vorwaerts   = 4;
int M_Rueckwaerts = 5;
int M_LinksRueck  = 6;
int M_RechtsRueck = 7;

void Motorensteuerung(int richtung, float geschwindigkeit) {
```

```
  if      (richtung == M_Vorwaerts)     { Gerade = True;  Fahre(geschwindigkeit, Kor-
                               rektur); }
  else if (richtung == M_Links)          { Gerade = False; Fahre(geschwindigkeit, 20.0
                               + Korrektur); }
  else if (richtung == M_Rechts)        { Gerade = False; Fahre(geschwindigkeit, -20.0
                               + Korrektur); }
  else if (richtung == M_Rueckwaerts)   { Gerade = False; Fahre(-geschwindigkeit, -
                               Korrektur); }
  else if (richtung == M_Halt)          { Gerade = False; Fahre(0.0, 0.0); }
  else if (richtung == M_LinksRueck)    { Gerade = False; Fahre(-geschwindigkeit, -
                               20.0 - Korrektur); }
  else if (richtung == M_RechtsRueck)   { Gerade = False; Fahre(-geschwindigkeit, 20.0
                               - Korrektur); }
}

/* Funktion Motoraktion: -------------------------------------------------- */
int Links              = 1;
int Rechts             = 2;
int LinksDrehung       = 3;
int RechtsDrehung      = 4;
int LinksRueckDrehung  = 5;
int RechtsRueckDrehung = 6;
int Geradeaus          = 8;
int Zurueck            = 9;
int Halt               = 10;

void Motoraktion(int aktion, float geschwindigkeit) {
  float rueckdauer = 1.0;
  float drehzeit = 0.8;
  float pausezeit = 0.2;
  if (aktion == Geradeaus) {
    Motorensteuerung(M_Vorwaerts, geschwindigkeit);
  }
  else if (aktion == LinksDrehung) {
    Motorensteuerung(M_Links, geschwindigkeit);
    sleep(drehzeit);
  }
  else if (aktion == RechtsDrehung) {
    Motorensteuerung(M_Rechts, geschwindigkeit);
    sleep(drehzeit);
  }
  else if (aktion == Links) {
    Motorensteuerung(M_Links, geschwindigkeit);
  }
  else if (aktion == Rechts) {
    Motorensteuerung(M_Rechts, geschwindigkeit);
  }
  else if (aktion == LinksRueckDrehung) {
    Motorensteuerung(M_Halt, 0.0);
    sleep(pausezeit);
    Motorensteuerung(M_LinksRueck, geschwindigkeit);
    sleep(drehzeit);
    Motorensteuerung(M_Halt, 0.0);
    sleep(pausezeit);
  }
  else if (aktion == RechtsRueckDrehung) {
    Motorensteuerung(M_Halt, 0.0);
    sleep(pausezeit);
    Motorensteuerung(M_RechtsRueck, geschwindigkeit);
    sleep(drehzeit);
    Motorensteuerung(M_Halt, 0.0);
    sleep(pausezeit);
  }
  else if (aktion == Zurueck) {
    Motorensteuerung(M_Rueckwaerts, geschwindigkeit);
  }
  else if (aktion == Halt) {
    Motorensteuerung(M_Halt, 0.0);
  }
}

/* Task HAPTIK: ----------------------------------------------------------- */
int   Haptik_Aktiv;
int   Haptik_Motoraktion;
float Haptik_Geschwindigkeit;
char  Haptik_Text[32];
```

```
int    Haptik_Anzeigewert;

void Haptik() {
  int crash;
  while(1) {
    hog_processor();
    crash = Bumper_Kollissionsposition();
    if (crash == 0) Haptik_Aktiv = False;
    else if (crash == 6) {      /* Crash vorn (links + rechts) */
      Haptik_Text = LeerS;
      Haptik_Anzeigewert = Leer;
      Haptik_Motoraktion = LinksRueckDrehung;
      Haptik_Geschwindigkeit = Globalgeschwindigkeit;
      Haptik_Aktiv = True;
    }
    else if (crash == 4) {      /* Crash links */
      Haptik_Text = LeerS;
      Haptik_Anzeigewert = Leer;
      Haptik_Motoraktion = RechtsRueckDrehung;
      Haptik_Geschwindigkeit = Globalgeschwindigkeit;
      Haptik_Aktiv = True;
    }
    else if (crash == 2) {      /* Crash rechts */
      Haptik_Text = LeerS;
      Haptik_Anzeigewert = Leer;
      Haptik_Motoraktion = LinksRueckDrehung;
      Haptik_Geschwindigkeit = Globalgeschwindigkeit;
      Haptik_Aktiv = True;
    }
    else if (crash == 1) {      /* Crash hinten */
      Haptik_Text = LeerS;
      Haptik_Anzeigewert = Leer;
      Haptik_Geschwindigkeit = Globalgeschwindigkeit;
      Haptik_Motoraktion = Geradeaus;
      Haptik_Aktiv = True;
    }
    else if (crash == 5) {      /* Crash links hinten */
      Haptik_Text = LeerS;
      Haptik_Anzeigewert = Leer;
      Haptik_Motoraktion = RechtsDrehung;
      Haptik_Geschwindigkeit = Globalgeschwindigkeit;
      Haptik_Aktiv = True;
    }
    else if (crash == 3) {      /* Crash rechts hinten */
      Haptik_Text = LeerS;
      Haptik_Anzeigewert = Leer;
      Haptik_Motoraktion = LinksDrehung;
      Haptik_Geschwindigkeit = Globalgeschwindigkeit;
      Haptik_Aktiv = True;
    }
    defer();
  }
}

/* Task AKUSTIK: ---------------------------------------------------------- */
int   Akustik_Aktiv;
int   Akustik_Motoraktion;
float Akustik_Geschwindigkeit;
char  Akustik_Text[32];
int   Akustik_Anzeigewert;

void Akustik() {
  float ton;
  float hoch = 2000.0;
  float tief = 1000.0;
  while(1) {
    hog_processor();
    ton = Mikrophon_Frequenz();
    if ((ton >= tief) && (ton < hoch)) {
      Akustik_Text = "Ton tief:";
      Akustik_Anzeigewert = (int) ton;
      Akustik_Motoraktion = LinksDrehung;
      Akustik_Geschwindigkeit = Globalgeschwindigkeit;
      Akustik_Aktiv = True;
    }
    else if (ton >= hoch) {
```

```
      Akustik_Text = "Ton hoch:";
      Akustik_Anzeigewert = (int) ton;
      Akustik_Motoraktion = RechtsDrehung;
      Akustik_Geschwindigkeit = Globalgeschwindigkeit;
      Akustik_Aktiv = True;
    }
    else Akustik_Aktiv = False;
    defer();
  }
}

/* Task TEMPERATURMESSUNG: -------------------------------------------------- */
int   Temperatur_Aktiv;
int   Temperatur_Motoraktion;
float Temperatur_Geschwindigkeit;
char  Temperatur_Text[32];
int   Temperatur_Anzeigewert;

void Temperaturmessung() {
int temperatur;
  while(1) {
    hog_processor();
    temperatur = analog(5);
    Temperatur_Text = "Temperatur:";
    Temperatur_Anzeigewert = temperatur;
    Temperatur_Aktiv = True;
    defer();
  }
}

/* Task STANDARD: ----------------------------------------------------------- */
int   Standard_Aktiv;
int   Standard_Motoraktion;
float Standard_Geschwindigkeit;
char  Standard_Text[32];
int   Standard_Anzeigewert;

void Standard() {
  while(1) {
    hog_processor();
    Standard_Text = "Standardmodus";
    Standard_Anzeigewert = Leer;
    Standard_Motoraktion = Geradeaus;
    Standard_Geschwindigkeit = Globalgeschwindigkeit;
    Standard_Aktiv = True;
    defer();
  }
}

/* Task RESSOURCENKONTROLLE: ------------------------------------------------ */
void Ressourcenkontrolle() {
  while(1) {
    hog_processor();
    if (Standard_Aktiv) {       /* Prio 4 */
      Gesamt_Motoraktion = Standard_Motoraktion;
      Gesamt_Geschwindigkeit = Standard_Geschwindigkeit;
      Gesamt_Text = Standard_Text;
      Gesamt_Anzeigewert = Standard_Anzeigewert;
    }
    if (Temperatur_Aktiv) {     /* Prio 3 */
      if (Temperatur_Motoraktion != Leer) Gesamt_Motoraktion = Temperatur_Motoraktion;
      if (Temperatur_Geschwindigkeit != LeerF) Gesamt_Geschwindigkeit = Tempera-
                                tur_Geschwindigkeit;
      if (Temperatur_Text[0] != 0) Gesamt_Text = Temperatur_Text;
      if (Temperatur_Anzeigewert != Leer) Gesamt_Anzeigewert = Temperatur_Anzeigewert;
    }
    if (Akustik_Aktiv) {        /* Prio 2 */
      if (Akustik_Motoraktion != Leer) Gesamt_Motoraktion = Akustik_Motoraktion;
      if (Akustik_Geschwindigkeit != LeerF) Gesamt_Geschwindigkeit = Akus-
                                tik_Geschwindigkeit;
      if (Akustik_Text[0] != 0) Gesamt_Text = Akustik_Text;
      if (Akustik_Anzeigewert != Leer) Gesamt_Anzeigewert = Akustik_Anzeigewert;
    }
    if (Haptik_Aktiv) {         /* Prio 1 */
      if (Haptik_Motoraktion != Leer) Gesamt_Motoraktion = Haptik_Motoraktion;
```

```
      if (Haptik_Geschwindigkeit != LeerF) Gesamt_Geschwindigkeit = Hap-
                                tik_Geschwindigkeit;
      if (Haptik_Text[0] != 0) Gesamt_Text = Haptik_Text;
      if (Haptik_Anzeigewert != Leer) Gesamt_Anzeigewert = Haptik_Anzeigewert;
    }
    defer();
  }
}

/* Task GESAMTVERHALTEN:  --------------------------------------------------- */
int   Gesamt_Motoraktion;
float Gesamt_Geschwindigkeit;
char  Gesamt_Text[32];
int    Gesamt_Anzeigewert;
int    Gesamt_Beep;

void Gesamtverhalten() {
  while(1) {
    hog_processor();
    Motoraktion(Gesamt_Motoraktion, Gesamt_Geschwindigkeit);
    LCDAnzeige(Gesamt_Text, Gesamt_Anzeigewert);
    defer();
  }
}


/* HAUPTPROGRAMM: ----------------------------------------------------------- */
void main() {
  test_number++;
  if ((test_number > 1) || (test_number < 0)) test_number = 0;
  if (test_number == 0) {
    start_process(Geradenkontrolle());
    start_process(Haptik());
    start_process(Akustik());
    start_process(Temperaturmessung());
    start_process(Standard());
    start_process(Ressourcenkontrolle());
    start_process(Gesamtverhalten());
  }
}
```

IC-Assembler-Modul: FREQUENZ.ASM

```
BASE  EQU    $1000
TCNT  EQU    $100E              Timer Count Register
ADR1  EQU    $1031              A/D Result Register 1

      ORG   MAIN_START
start FDB   #0
stop  FDB   #0

subroutine_freq:
      PSHX                      X-Register sichern
      LDX   #BASE               Offset der Registeradressen

* Synchronisation:
x1:   LDAA  ADR1,x              Wert lesen
      SUBA  #128                Warten, solange >= 128
      BPL   x1                  "

* Startwert ermitteln:
x2:   LDAA  ADR1,x              Wert lesen
      SUBA  #128                Warten, solagen < 128
      BMI   x2                  "
      LDD   TCNT,x              Counterstand laden
      STD   start               Counterstand speichern

* Stopwert ermitteln:
x3:   LDAA  ADR1,x              Wert lesen
      SUBA  #128                Warten, solagen >= 128
      BPL   x3                  "
      LDD   TCNT,x              Counterstand laden
      STD   stop                Counterstand speichern
```

```
* Überlauf in Counter überprüfen:
      LDD   stop               Stopwert laden
      SUBD  start              Startwert subtrahieren
      BMI   x1                 Wiederholung, falls negativ

* Routine verlassen:
      PULX                     X-Register restaurieren
      RTS                      Subroutine verlassen mit D-Register als Returnwert
```

IC-Assembler-Modul: SPEED.ASM

```
/* This section starts the 6811 assembler routines  */

TFLG1    EQU     $1023

  ORG MAIN_START

subroutine_initialize_module:   /* This module runs whenever a reset occurs */

     ldd #IC3_interrupt_handler /* Get 16-bit address of interrupt handler */
     std $FFEA                  /* Store in IC3 interrupt vector location */
     cli                        /* Enable interrupts globally */
     rts                        /* Return from initialzie_module subroutine */


variable_left_clicks:           /* Create right_clicks, a variable accessible by IC */
     fdb 0                      /* Fill double byte (16 bits).  right_clicks = 0 */

IC3_interrupt_handler:
     ldd variable_left_clicks   /* Get the value in right_clicks */
     addd #1                    /* Add one more encoder count */
     std variable_right_clicks  /* Store back to right_clicks */
     ldaa #%00000001            /* Clear the IC3 flag by writing a one */
     staa TFLG1                 /* Store in TFLG1 to clear IC3 flag */
     rti                        /* Return from interrupt */
```

IC-Programmbibliothek: SHAFT.C

```
/* Shaft.c */

/* Utility functions for monitoring the shaft encoders */

/* Functions for reading left and right "velocities" using PA7s pulse counter */
/* and an interrupt routine (in speed.icb) to catch pulses on PA0 */

void init_velocity()            /* Enable PA7 pulse counting and PA0 interrupt */
{ poke(0x1026,0b01010000);      /* PACTL  PA7 input, trigger on rising edge */
  poke(0x1027,0);               /* PACNT  init to 0 */
  left_clicks = 0;              /* Global referenced by interrupt routine */
  bit_clear(0x1021,0b10);       /* TCTL2 Assign values to the two lowest order bits */
  bit_set(  0x1021,0b01);       /* of tctl2 without changing any other bits */
  bit_set(0x1022,0b01);         /* TMSK1  enable IC3 interrupts */
}

int get_left_clicks()           /* Get the number of clicks and reset to 0 */
{ int clks = left_clicks;       /* If called at regular intervals this funciton returns
                                */
  left_clicks = 0;              /* the the left velocity in units of clicks per inter-
                                val */
  return clks; }

int get_right_clicks()          /* Same as above for right Shaft encoder */
{ int clks = peek(0x1027);      /* PACNT is only an 8-bit register -- be sure to */
  poke(0x1027,0);               /* call it often enough so that it dosen't overflow */
  return clks; }
```

IC-Programmbibliothek: LIB_RW11.C

```
/* lib_rw11.c */

/*
    VERSION HISTORY FOR lib_rw11.c

    V2.81  Jun 11 1994
    Created library for Rug Warrior from an earlier
    library by jsr and fgm

*/

persistent int test_number;

/*******************/
/* TIME PRIMITIVES */
/*******************/

/**********************************/
/* location of various time stuff: */
/* 0x14: time in milliseconds      */
/**********************************/

void reset_system_time()
{
  pokeword(0x14, 0);
  pokeword(0x12, 0);
}

/*  returns valid time from 0 to 32,767 minutes (approx)  */
float seconds()
{
  return ((float) mseconds()) / 1000.;
}

void sleep(float seconds)
{
    msleep((long)(seconds * 1000.));
}

void msleep(long msec)
{
    long end_time= mseconds() + msec;

    while (1) {
  /* if the following test doesn't execute at least once a second,
     msleep may not halt */
  long done= mseconds()-end_time;
  if (done >= 0L && done <= 1000L) break;
     }
}

void beep()
{   tone(500., .3);
  }

/*  1/2 cycle delay in .5us goes in 0x26 and 0x27  */
void tone(float frequency, float length)
{
    pokeword(0x26, (int)(1E6 / frequency));
    bit_set(0x1020, 0b00000001);
    sleep(length);
    bit_clear(0x1020, 0b00000001);

    /*  following is important to reduce # of interrupts
  when tone is off  */
    pokeword(0x26, 0);

    bit_clear(0x1000, 8);
}

void beeper_on()
{
    bit_set(0x1020, 0b00000001);
    bit_set(0x1022, 0b00001000);
}
```

```
void beeper_off()
{
    bit_clear(0x1022, 0b00001000);
    bit_clear(0x1020, 0b00000001);
    pokeword(0x26, 0);
    bit_clear(0x1000, 0b00001000);      /* turn power to spkr off */
}

void set_beeper_pitch(float frequency)
{
    pokeword(0x26, (int)(1E6 / frequency));
}

long timer_create_mseconds(long timeout)
{
    return mseconds() + timeout;
}

long timer_create_seconds(float timeout)
{
    return mseconds() + (long) (timeout * 1000.);
}

int timer_done(long timer)
{
    return timer < mseconds();
}

/* Rug Warrior's analog ports 6 and 7 are unassigned */

int analog(int port)
{   poke(0x1039, 0b10000000);
    poke(0x1030, port);
    return peek(0x1031);
}

/****************************/
/*** Multi-Tasking Support ***/
/****************************/

/*  gives process that calls it 256 ticks (over 1/4 sec)
    more to run before being swapped out
call repeatedly to hog processor indefinitely  */
void hog_processor()
{
    poke(0x0a, 0);
}

/* Debugging utilities */

/* Dump 8 bytes to the LCD screen, starting at addr */
void dump(int addr)
{ printf("%x: %x %x ", addr, peekword(addr), peekword(addr + 2));
  printf("      %x %x\n",  peekword(addr + 4), peekword(addr + 6));
}

/*************************************************************************/
/* Rug Warrior specific functions and constants                          */
/*************************************************************************/

/* Digital ports 1 and 2 are unassigned */

int digital(int port)           /* Return 1 bit from PA1 or PA2 (or PA0) */
{ return 1 & (peek(0x1000) >> (port & 3));
}

/* Indices for accessing sensors connected to the A/D converter.
   e.g. to read value of right photo cells use analog(photo_right) */

int photo_right = 0;
int photo_left = 1;
int microphone = 2;
int pyro = 5;

/**************************************************************************/
/*
```

```
/* Motor Control Primitives
/*
/*    init_motors()    - Must be called to enable motors
/*    motor(index, speed)  - Control velocity of motor 0 (left) or 1 (right)
/*    drive(trans_vel, rot_vel)  - Control robot translation and rotation
/*    stop()       - Stop both motors
 */

/* Setup two PWM channels for motor control */

/*                 Left        Right            */
int TOCx[2]     = {0x1018,     0x101A}; /* Index for timer register */
int dir_mask[2] = {0b00010000, 0b00100000};   /* Port D direction bits */
int pwm_mask[2] = {0b01000000, 0b00100000};   /* Port A PWM bits */

int init_motors()
{ bit_set(0x1009,0b110000);     /* Set PD4,5 as outputs for motor direction */
  poke(0x100C,0b01100000);      /* OC1M Output compare 1 affects PA5,6 */
  bit_set(  0x1020,0b10100000); /* TCTL1 OC3 turns off PA5, OC2 PA6 */
  bit_clear(0x1020,0b01010000); /* Use set and clear to avoid other bits */
  pokeword(0x1018,0);           /* When TCNT = 0, OC1 fires */
}

/* Make sure init_motors is called after a reset */

int init_motors_dummy = init_motors();

void stop()                     /* Stop both drive motors */
{ bit_clear(0x100D,pwm_mask[0]); /* Let OC1 turn off motors rather */
  bit_clear(0x100D,pwm_mask[1]); /* than turn them on */
}

/* Vel is in the range [-100, +100], index = 0 => Left, = 1 => Right */

void motor(int index, float vel)
{ float avel  ;                 /* Absolute value of velocity */
  if (vel > 0.0)
    { bit_set(0x1008, dir_mask[index]); /* Forward rotation */
      avel = vel; }
  else
    { bit_clear(0x1008, dir_mask[index]); /* Backward rotation */
      avel = (- vel); }
  if (avel < 1.0)               /* If we are going real slow */
    bit_clear(0x100D,pwm_mask[index]);  /* then just have OC1 turn off the motor */
  else
    bit_set(0x100D,pwm_mask[index]);    /* Otherwise let OC1 control the motors */
  if (avel > 99.0)              /* If we are gong real fast */
    avel = 99.0;                /* don't let the following multiply overflow */
  pokeword(TOCx[index], (int) (655.56 * avel));      /* Compute TOCx value */
}

/* Use drive to control motion of the robot.  A positive rot_vel makes the robot
   turn left. */

void drive(float trans_vel, float rot_vel)
{ motor(0,trans_vel - rot_vel);
  motor(1,trans_vel + rot_vel);
}

/* Enable and activate debugging LEDs. */
void leds(int val)
{ poke(0x1009,0b111100);        /* Set port D for output */
  poke(0x1008,val << 2);        /* Shift number over */
}

/* Return a 3-bit value representing which of the bumper switches are closed */
int bumper()
{ int bmpr;
  bmpr = analog(3);             /* Switch closure: */
  if      (bmpr <  11) return 0b000;   /* none */
  else if (bmpr <  32) return 0b001;   /* A   */
  else if (bmpr <  53) return 0b010;   /*  B  */
  else if (bmpr <  74) return 0b011;   /* AB  */
  else if (bmpr <  96) return 0b100;   /*  C  */
  else if (bmpr < 117) return 0b101;   /* A C */
  else if (bmpr < 132) return 0b110;   /*  BC */
```

```
  else                 return 0b111;   /* ABC - (mechanically impossible) */
}

/* ir_detect returns:
     0b00 => no reflection, 0b01 => reflection on right,
     0b10 => reflection on left, 0b11 => reflection on both sides */

int ir_detect()
{ int val1, val2, val3;
  val1 = peek(0x100A) & 0b10000; /* IR Detector connected to PE4 */
  bit_set(0x1008,0b1000);       /* Turn on Left emitter, PD3 */
  msleep(1L);                   /* Wait 1 millisecond */
  val2 = peek(0x100A) & 0b10000; /* Should be Low if signal detected */
  bit_clear(0x1008,0b1000);     /* Turn off Left emitter */
  bit_set(0x1008,0b0100);       /* Turn on Right emitter, PD2 */
  msleep(1L);                   /* Wait 1 millisecond */
  val3 = peek(0x100A) & 0b10000;
  bit_clear(0x1008,0b1100);     /* Turn emitters off */
  /* For detection, detector must be high when emitter is off, low when on */
  return ((val1 & ~val2) >> 3)  | ((val1 & ~val3) >> 4); /* HI -> LOW */
}
```

C) Programmdiskette