Design und Implementierung eines Simulators für das LEGO Mindstorms Robotics Invention System

Studienarbeit im Fach Informatik

vorgelegt von

Meik Felser

geboren am 15. Juni 1976 in Nürnberg

Angefertigt am

Institut für Mathematische Maschinen und Datenverarbeitung (IV)
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:	**Prof. Dr. rer. nat. Fridolin Hofmann** **Dr. Ing. Frank Bellosa**
Beginn der Arbeit:	03. März 2000
Abgabe der Arbeit:	18. Juli 2000

Ich versichere, dass ich meine Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 18. Juli 2000, ______________________________

Meik Felser

Inhaltsverzeichnis

1 Einleitung und Motivation 1

2 Allgemeiner Überblick über Simulatoren 3

2.1 Einsatzmöglichkeiten von Simulatoren 3

2.2 Simulationsansätze 4

2.2.1 Vollständige Simulation der Hardware auf Befehlssatzebene 4

2.2.2 Simulation auf Betriebssystemebene 5

2.2.3 Simulation einzelner Anwendungen 6

2.3 Beispiele 7

2.4 Simulation des LEGO-Kontrollbausteins 7

3 Simuliertes System 9

3.1 LEGO Mindstorms Robotics Invention System 9

3.2 RCX Kontrollbaustein 9

3.2.1 Aufbau und Möglichkeiten des RCX 9

3.2.2 Hitachi H8 Mikrocontroller 10

3.3 Software des RCX 13

3.3.1 Startvorgang 13

3.3.2 Firmware 13

4 Realisierung des RCX Simulators 16

4.1 Startvorbereitungen 17

4.1.1 Motorola S-Record Format 17

4.1.2 SRecordLoader-Klasse 18

4.2 Adressraumverwaltung 19

4.2.1 Überblick 19

4.2.2 Memory-Klasse 20
4.2.3 MemoryRegion-Klasse 21
4.3 CPU 24
4.3.1 Überblick 24
4.3.2 Implementierung 27
4.3.3 Ausnahme-Behandlung 32
4.3.4 Energiesparmodi 38
4.4 Ports 39
4.4.1 Überblick 39
4.4.2 Implementierung 41
4.5 Motor-Treiber 44
4.5.1 Überblick 44
4.5.2 Implementierung 44
4.6 Zähler 46
4.6.1 16-bit Zähler 46
4.6.2 8-bit Zähler 48
4.6.3 Watchdog Timer 49
4.7 Analog/Digital Wandler 49
4.7.1 Überblick 49
4.7.2 Implementierung 50
4.8 serielle Schnittstelle 51
4.8.1 Überblick 51
4.8.2 Implementierung 52

5 Verbesserungsmöglichkeiten **56**
5.1 Verbesserung der Geschwindigkeit 56
5.1.1 Beschleunigung des LC-Displays 56
5.1.2 Beschleunigung der Speicherverwaltung 57
5.2 Erhöhung der Simulationsgenauigkeit 57
5.3 Erweiterung des Simulationsumfangs 58

6 Zusammenfassung **59**

A Bedienungsanleitung **60**

A.1 Kompilierung des RCX Simulators . . . 60

A.2 Starten des RCX Simulators . . . 60

A.3 Bedienung des Simulators . . . 61

A.4 Laden der Firmware . . . 63

A.4.1 Veränderungen am ROM . . . 63

A.4.2 Firmware Downloader . . . 63

B Software-Architektur des RCX Simulators **65**

C Erstellen eines ROM Abbildes **69**

D Erstellen eines Crosscompilers für den H8/300 Prozessor **71**

Kapitel 1

Einleitung und Motivation

Simulatoren gibt es heute für alle Bereiche. Einer dieser Bereiche ist die Simulation von Computersystemen auf Softwarebasis. Die Einsatzgebiete solcher Simulationsprogramme sind vielfältig. Sie können zum Beispiel für die Entwicklung neuer Software und Betriebssysteme wertvolle Werkzeuge sein.

Insbesondere bei der Entwicklung von Betriebssystemen für eingebettete Systeme bieten sich Simulatoren an, da das Testen einer Software auf diesen Systemen oftmals umständlich ist. Zum einen muss die Software zuerst vom Entwicklungssystem zum eingebetteten System übertragen werden, um einen Testlauf zu starten. Zum anderen liefern eingebettete Systeme im Fehlerfall meist keine Hinweise auf die Ursache.

Allgemein erfreuen sich eingebettete Systeme jedoch immer größerer Beliebtheit und sind heute in jedem Lebensbereich anzutreffen. Gesteuert werden sie meist durch einen Mikrocontroller, der das Betriebssystem ausführt.

Das Ziel dieser Studienarbeit ist es einen Simulator für einen Mikrocontroller zu entwickeln. Dieser Mikrocontroller ist in einem Kontrollbaustein eingebettet, den der Spielwarenhersteller Lego[1] in einem seiner Roboter-Baukästen der Reihe "Mindstorms" verkauft. Der Steuerbaustein bietet sich als Beispiel an, da er leicht erhältlich und ein typischer Vertreter eines eingebetteten Systems ist

Die Simulation soll nicht nur den Mikrocontroller, sondern auch alle anderen Bestandteile des Lego-Kontrollbausteins umfassen. Es soll auch möglich sein Programme über die Infrarot-Schnittstelle zu übertragen, so wie es das Original vorsieht. Oberster Grundsatz bei der Implementierung war die Binärkompatibilität zum Original-System. Das heißt, dass der Simulator alle Programme die für den Mikrocontroller geschrieben wurden, unverändert ausführen kann.

In der folgenden Arbeit wird der im praktischen Teil angefertigte Simulator für den Lego-Steuerbaustein im Detail beschrieben.

Im ersten Kapitel wird zunächst ein Überblick über Simulatoren im Allgemeinen gegeben. Dabei wird besonders auf deren Einsatzmöglichkeiten und Nutzen, sowie auf spezielle Simulationsansätze eingegangen.

[1] Alle in dieser Arbeit genannten Produkt- und Markennamen sind urheberrechtlich geschützt und eingetragene Warenzeichen bzw. Markennamen der jeweiligen Inhaber.

Das zweite Kapitel stellt den Lego-Baustein vor. Dort wird zuerst die Hardware betrachtet; anschließend werden zwei verschiedene Betriebssysteme vorgestellt.

Das darauffolgende Kapitel beschäftigt sich mit dem Simulator selbst. Es werden dort die einzelnen Komponenten des Kontrollbausteins und speziell die Module des Mikrocontrollers vorgestellt und anschließend die Realisierung im Rahmen der Simulation erläutert.

Im letzten Teil der Arbeit werden schließlich einige Möglichkeiten aufgezeigt, die Simulation zu verbessern.

Der Anhang enthält eine kurze Bedienungsanleitung des Simulators, sowie ein Klassendiagramm welches die Software-Architektur des Simulators darstellt. Des Weiteren finden sich dort Anleitungen um ein ROM-Abbild und einen Crosscompiler für den Hitachi-Mikrocontroller zu erstellen.

Kapitel 2

Allgemeiner Überblick über Simulatoren

In diesem Kapitel wird ein kurzer Überblick gegeben, in welchen Bereichen Simulatoren eingesetzt werden können. Anschließend werden verschiedene Arten von Simulatoren vorgestellt und bewertet. Zum Abschluss wird der RCX Simulator eingeordnet.

2.1 Einsatzmöglichkeiten von Simulatoren

Neben wenigen Nachteilen, wie zum Beispiel einer geringeren Geschwindigkeit, bieten Simulatoren drei wichtige Vorteile gegenüber dem realen System: Zum ersten kann man sie als Systementwickler leichter verändern als ein reales System. Zum zweiten ist man mit einem Simulator unabhängig von der Existenz und der Verfügbarkeit des realen Systems und zum dritten kann ein Simulator hilfreich bei der Softwareentwicklung sein.

Den ersten Vorteil kann man zum Beispiel bei der Entwicklung neuer Prozessorarchitekturen ausnutzen. Erstellt man zuerst eine Simulation des neuen Prozessors, so bringt dies den Vorteil, dass man alternative Realisierungen einzelner Komponenten ausprobieren kann, bevor man sich für eine konkrete entscheiden muss. So kann man schon im Vorfeld sehen, ob eine Konfiguration den gestellten Anforderungen und Erwartungen entspricht. Um dieses Einsatzgebiet zu unterstützen, können Simulatoren eine Vielzahl von Analyse- bzw. Statistikdaten liefern, die man von einem realen System nicht oder nur sehr schwer erhalten kann. Somit kann ein Simulator unter Umständen die Schwachstellen einer Architektur besser aufzeigen als Analysen an einem realen System. Ein Simulator kann daher dazu beitragen, die Fehler bzw. die Fehlentscheidungen zu minimieren, die am endgültigen Produkt nur mit hohem finanziellem Aufwand zu korrigieren sind.

Der zweite Vorteil, die Unabhängigkeit vom realen System, zeigt sich zum Beispiel bei der Softwareentwicklung. Will man Software für ein System entwickeln, so benötigt man meist ein Exemplar dieses Systems um Testsläufe durchzuführen. Ist das System jedoch neu, so existiert eventuell erst ein Prototyp und die Serienproduktion ist noch gar nicht angelaufen, sodass man noch keine Exemplare erwerben kann. Hier kann ein Simulator nützlich sein, denn man kann mit seiner Hilfe die Entwicklung der Software schon frühzeitig beginnen. Eine

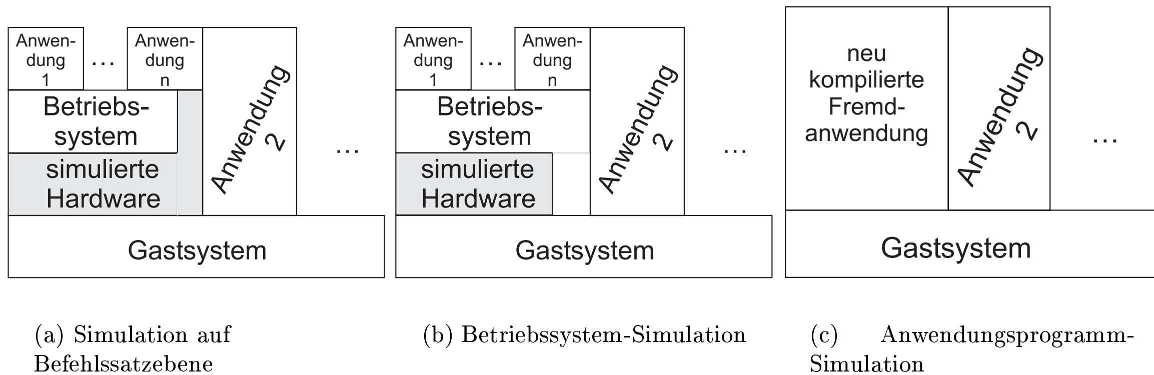
ähnliche Situation liegt vor, wenn das reale System relativ teuer ist und sich ein Software-Entwicklungsteam wenige Exemplare teilen muss. Auch hier kann es sinnvoll sein auf einen Simulator auszuweichen.

Bei der Programmierung eingebetteter Systeme kann die Unabhängigkeit von der realen Hardware ebenfalls vorteilhaft sein. Die Programme für solche Systeme werden oft auf einer Entwicklungs-Plattform erstellt und anschließend zum eigentlichen System übertragen. Dieser Vorgang nimmt meist einige Zeit in Anspruch und kann durchaus umständlich sein. Mithilfe einer Simulation des eingebetteten Systems könnte die gesamte Entwicklung auf einem System stattfinden, ohne ständig wechseln zu müssen.

Außerdem können Simulatoren die Softwareentwicklung unterstützen. Vor allem bei der Suche nach Programmfehlern können sie hilfreich sein. Ein eingebettetes System zum Beispiel bietet meist keine Möglichkeiten die Ursache eines Fehlers zu finden. Mithilfe eines Simulators kann man hingegen den Programmablauf verfolgen und dadurch die Fehlerquelle schneller finden.

2.2 Simulationsansätze

Nachdem die Vorteile von Simulatoren angesprochen wurden, soll nun auf verschiedene Ansätze, Simulatoren in Software zu realisieren, eingegangen werden. Dabei werden drei Klassen, wie in Abbildung 2.1 gezeigt, vorgestellt, die sich hauptsächlich in der Genauigkeit der Simulation unterscheiden.



(a) Simulation auf Befehlssatzebene

(b) Betriebssystem-Simulation

(c) Anwendungsprogramm-Simulation

Abbildung 2.1: Vergleich der Simulationsansätze

2.2.1 Vollständige Simulation der Hardware auf Befehlssatzebene

Beim ersten Ansatz soll die gesamte Hardware auf niedrigstem Niveau dargestellt werden, sodass der Unterschied zum realen System in der Software-Schicht nicht zu bemerken ist. Deshalb werden der Prozessor sowie alle Peripherie-Geräte auf Befehlssatzebene simuliert. Da für die Software üblicherweise keine Möglichkeit besteht tiefer in die Funktionsweise des Prozessors bzw. des Systems Einblick zu nehmen, ist ein Simulationsansatz auf einer noch tieferen Ebene unnötig und für die meisten Anwendungsfälle zu aufwendig.

Abbildung 2.1(a) soll verdeutlichen, dass durch einen Simulator auf Befehlssatzebene sowohl für das Betriebssystem als auch für die Anwendungsprogramme die gesamte Hardware zu existieren scheint. Der Simulator dient als Plattform und arbeitet die gestartete Software wie das Original-System ab. Jeder Befehl wird dabei vom Simulator bearbeitet und bewirkt eine entsprechende Änderung des simulierten Systems.

Der größte Vorteil ergibt sich aus der Eigenschaft, dass die Software, nicht zwischen Original und Simulation unterscheiden kann. Es sollte somit möglich sein jede Anwendung und jedes Betriebssystem unmodifiziert auf dem Simulator einzusetzen. Somit kann man Untersuchungen der Software-Schicht vornehmen die nicht durch vorher nötige Veränderungen verfälscht werden. Außerdem kann man den Simulator als vollständigen Ersatz des realen Systems einsetzen.

Man sollte jedoch nicht die Nachteile vernachlässigen, die von einer Simulation auf Befehlssatzebene ausgehen. Die Simulation jedes einzelnen Befehls bringt zwar Genauigkeit, kostet dafür aber viel Zeit. So wird jeder einzelne Befehl des simulierten Systems in mehrere Befehle des Gastsystems übersetzt. Dies führt zu einem enormen Geschwindigkeitsverlust, gegenüber dem Original-System. Dieser Nachteil verringert die Chancen einen Simulator als gleichwertigen Ersatz eines realen Systems einzusetzen.

Ein weiterer Punkt, den man beachten sollte, sind die Analyse- und Statistikdaten, die ein Simulator auf dieser Ebene erzeugen kann. Zum einen benötigt es zusätzlich Zeit, diese unter Umständen sehr großen Datenmengen zu sammeln und effizient zu verwalten, zum anderen ist die Verwertbarkeit dieser Daten in den meisten Fällen fraglich, da sie sich auf die Ebene der Maschinensprache beziehen. Eine einzelne Zeile Code in einer Hochsprache, wie zum Beispiel C, kann in Maschinencode viele Instruktionen umfassen. Da man im Normalfall aber nicht genau weiß, was der C-Compiler aus dem Code erzeugt, ist es sehr schwierig eine bestimmte Stelle eines Programms auf dieser Ebene zu überwachen. Das ist aber nötig, wenn man Daten über höher liegende Konzepte, wie zum Beispiel die Prozessverwaltung, sammeln möchte.

2.2.2 Simulation auf Betriebssystemebene

Eine Lösung des letzten Problems kann erreicht werden, indem die Simulation auf eine höhere Abstraktionsebene angehoben wird. Ziel soll es nun sein den Anwendungsprogrammen ein Betriebssystem zur Verfügung zu stellen, welches alle Aufgaben des Originals erfüllt. Dieser Ansatz will nicht das gesamte System auf niedrigster Ebene simulieren, sondern nur soweit nach unten absteigen, wie es die Simulation des Betriebssystems erfordert.

Wird die Funktionsweise eines Gerätes vom Betriebssystem verborgen, so muss es auch nicht exakt simuliert werden, es reicht, wenn das Betriebssystem Interaktionen mit diesem Gerät auf Geräte des Gastsystems umsetzt. Abbildung 2.1(b) zeigt, dass die Anwendungsprogramme den Zugriff auf die Hardware dem Betriebssystem überlassen müssen und dass dieses Funktionen des Gastsystems nutzen muss. Teile der Hardware, zum Beispiel der Prozessor, müssen unter Umständen trotzdem noch simuliert werden.

Als Vorteil gegenüber der vollständigen Simulation der Hardware ist die geringere Komplexität und die höhere Geschwindigkeit zu nennen. Wie schon angedeutet, muss man nicht den Aufwand auf sich nehmen, die externen Geräte auf Befehlsebene zu simulieren, sondern es reicht normalerweise aus so genannte Pseudogeräte zur Verfügung zu stellen. Das Betriebssystem muss dann alle Zugriffe auf diese Geräte an echte Geräte des Gastsystems umleiten.

Außerdem kann man Teilbereiche, die vom Betriebssystem transparent für die Anwendungen durchgeführt werden, bei der Simulation unter Umständen vernachlässigen. Als Beispiel hierfür ist die Speicherverwaltung zu nennen. Normalerweise fordern die Anwendungen Speicher an und bekommen diesen vom Betriebssystem zur Verfügung gestellt. Die Verwaltung des freien und virtuellen Speichers übernimmt dabei das Betriebssystem und ist daher für die Programme unsichtbar. Deshalb könnte der Simulator die Verwaltung des Speichers zum größten Teil dem Gastsystem überlassen und die Anforderungen der Anwendungsprogramme nur weiterleiten.

Durch die angehobene Simulationsebene ändern sich auch die Art der Statistik- und Analysedaten gegenüber denen eines Simulators auf Maschinencode-Ebene. Ist man zum Beispiel an der Häufigkeit der Prozesswechsel interessiert, so kann das simulierte Betriebssystem dies mitprotokollieren. Würde man auf Befehlssatzebene simulieren, so ließen sich diese Analysen nur schwer durchführen, denn das Konzept verschiedener Prozesse wird häufig erst durch das Betriebsystem eingeführt.

Die Verlagerung der Simulation auf eine höhere Ebene bringt aber auch einige Nachteile mit sich. So macht es dieser Ansatz nötig, den hardwareabhängigen Teil des Betriebssystems zu verändern, um Zugriffe auf das Gastsystem umzuleiten. Dazu benötigt man aber normalerweise Zugriff auf die Quellen des Betriebssystems, was zu Lizenzproblemen führen kann. Außerdem sind die Modifikationen für jedes Betriebssystem und jede neue Version eines Betriebssystems erneut durchzuführen.

Des Weiteren kann nicht gewährleistet werden, dass alle Programme, die auf dem Original-System laufen auch auf der Simulationsumgebung einwandfrei arbeiten. Greift ein Programm zum Beispiel direkt auf ein Gerät zu, so wird das in der Simulation nicht möglich sein, da nur der Zugriff über die dafür vorgesehenen Betriebssystem-Funktionen ausreichend unterstützt wird.

Außerdem kann die Vereinfachung der Simulation die Ergebnisse einer Analyse beeinflussen oder sogar verfälschen. Dies wird zum Beispiel dann deutlich, wenn man das Laufzeitverhalten einer Anwendung messen möchte. Dazu ist ein Simulator nötig, der das zeitliche Verhalten aller Geräte zueinander korrekt darstellt. Da das Laufzeitverhalten unter anderem auch stark von den Ein-/Ausgabegeräten, wie beispielsweise den Festplatten abhängt, müssen auch diese in einer korrekten zeitlichen Proportion zueinander stehen. Da der besprochene Simulationsansatz aber speziell die peripheren Geräte nur umleitet, ist keine korrekte Simulation des Laufzeitverhaltens mehr möglich, da die Hardware des Gastsystems einen nicht unwesentlichen Einfluss nimmt.

2.2.3 Simulation einzelner Anwendungen

Als konsequente Fortführung des zweiten Simulationsansatzes gibt es noch die Möglichkeit die Simulation auf die Anwendungssoftware zu beschränken. Dazu wird eine Anwendung erstellt, die als solche auf dem Gastsystem läuft und nicht durch einen Simulator interpretiert werden muss. Hierfür ist es nötig die Original-Applikation neu auf dem verfügbaren System zu kompilieren, wobei alle benötigten Betriebssystem-Funktionen als Bibliothek zur Verfügung gestellt werden. Das resultierende Programm lässt sich dann ganz normal auf dem Gastsystem starten, so als ob es für dieses geschrieben wurde.

Dieser Simulationsansatz ähnelt stark einer Portierung des Programms. Der entscheidende Unterschied ist jedoch, dass der Quellcode der Anwendung nicht verändert werden soll. Die

Anpassung an das Gastsystem sollen nur durch Verwendung geeignete Bibliotheken erledigt werden.

Ein Vorteil diese Simulationstechnik ist die Geschwindigkeit, die erzielt werden kann. Da das Programm in Maschinencode des Gastsystems vorliegt, kann es direkt von der CPU ausgeführt werden und muss nicht erst durch einen Simulator umgesetzt werden.

Allerdings läuft das Programm unter dem Betriebssystem des Gastrechners, welches die Verwaltung der Betriebsmittel unter Umständen völlig anders handhabt als das Original-System. Dadurch wird das Einsatzgebiet dieses Ansatzes stark eingeschränkt. So wird man diese Art der Simulation nur verwenden können um die Anwendung für fremde Systeme verfügbar zu machen und so ihre Funktionalität, unabhängig von einer bestimmten Hardware, überprüfen oder nutzen zu können. Eine genauere Analyse des Ablaufgeschehens ist aufgrund der unterschiedlichen Arbeitsumgebung meist nicht möglich oder sinnvoll.

Zudem ist noch zu beachten, dass man Zugriff auf den Quellcode des Anwendungsprogramms haben muss. Deshalb wird man diese Simulation wohl nur wählen, wenn man neue Software für ein System entwickelt, welches kaum verfügbar ist oder keine hinreichende Unterstützung zur Softwareentwicklung bietet. Wenn man beispielsweise Software für ein eingebettetes System entwickelt, so muss man das meist auf einem Entwicklungssystem machen. Dabei kann es sinnvoll sein das Ergebnis dort auch testen zu können, anstatt das Programm zum Original-System transferieren zu müssen.

2.3 Beispiele

Betrachtet man existierende Simulatoren, so stellt man fest, das die Grenze zwischen den oben gezeigten Simulationsansätzen oft nicht genau definiert werden kann. Trotzdem lassen sich die meisten Simulatoren einer Gruppe zuordnen.

Ein Beispiel für die Gruppe der vollständigen Hardware-Simulatoren ist der Simulator VMWare [VMW]. Dieser simuliert einen Intel PC inklusive aller Peripheriegeräte. Weitere Beispiel der ersten Gruppe sind die Simulatoren SimOS [Simb] und SimICS [Sima]. SimOS kann eine SGI-Workstation mit MIPS R4000 oder MIPS R10000 Prozessor auf einem beliebigen UNIX-System simulieren. SimICS ist ein Simulator einer SPARC-Architektur, wie sie in einer SUN-Workstation vorkommt.

Ein Beispiel für Betriebssystemsimulatoren ist das WINE-Projekt [WIN]. Es handelt sich dabei um ein Unix-Programm, welches in der Lage ist, Programme die für Microsoft Windows geschrieben wurden auszuführen.

Als letzte Beispiele sind die Simulatoren EmulegOS [FBF] und legOSim [Mül] zu nennen. Sie stellen Bibliotheken und Ausführungsumgebungen zur Verfügung, um Programme, die für das Betriebsystem legOS (siehe Abschnitt 3.3.2.2) geschrieben wurden, auf einem Unix-System zu testen.

2.4 Simulation des Lego-Kontrollbausteins

Der Simulator des Lego-Kontrollbausteins RCX, der im praktischen Teil dieser Studienarbeit entwickelt wurde, ist dem ersten Simulationsansatz zuzuordnen. Er simuliert den im Bau-

stein enthaltenen Mikrocontroller sowie alle seiner integrierten Module auf Befehlssatzebene. Das Hauptaugenmerk wurde dabei auf die Funktionalität gelegt, sodass einige Aspekte des zeitlichen Verhaltens vernachlässigt wurden. Zudem wurden auch bei manchen Komponenten des Mikrocontrollers Abstriche gemacht, wenn sie von der im Lego-Baustein enthaltenen Hardware nicht genutzt werden.

Der Sinn des Simulators ist hauptsächlich darin zu sehen, Programme testen zu können, ohne sie in den Baustein laden zu müssen. Die Simulation erleichtert somit die Entwicklung neuer Software und Betriebssysteme.

Kapitel 3

Simuliertes System

Der folgende Abschnitt gibt einen Überblick über das "LEGO Mindstorms Robotics Invention System". Zuerst wird das vorhandene System vorgestellt, anschließend wird die verwendete Hardware genauer betrachtet und auf den Startvorgang eingegangen. Zum Abschluss werden zwei Betriebssysteme vorgestellt, die man zusammen mit dem Robotics Invention System einsetzen kann.

3.1 LEGO Mindstorms Robotics Invention System

Unter dem Begriff "Mindstorms" verkauft LEGO drei Baukästen, denen alle der Bau von Robotern als Thema zugrunde liegt. Einer dieser Kästen ist das "Robotics Invention System", kurz RIS genannt. Er bietet von den drei Kästen die meisten Möglichkeiten und Freiheitsgrade Roboter zu entwickeln.

Neben vielen Standard-LEGO-Steinen liegt dem Kasten ein Kontrollbaustein Namens "RCX" bei. Dieser dient zur Steuerung der Roboter und verbirgt einen autonomen Mikrocontroller in einem LEGO-kompatiblen Gehäuse mit Display und Funktionstasten.

3.2 RCX Kontrollbaustein

3.2.1 Aufbau und Möglichkeiten des RCX

Der Steuerbaustein RCX ist 8x12 LEGO-Raster groß. Abbildung 3.1 zeigt den Baustein mit einigen angeschlossenen Sensoren und Motoren. Um die Energieversorgung sicherzustellen ist auf der Unterseite ein Fach für sechs Batterien vorgesehen.

Auf der Oberseite befindet sich ein Display, welches Statusinformationen anzeigen kann, sowie vier Knöpfe um Grundfunktionen auszuführen. Des Weiteren sind LEGO-gerechte und verpolungssichere Anschlüsse für drei Motoren und drei Sensoren vorhanden. LEGO bietet vier Sensortypen zur Verwendung mit dem RCX an: Berührungssensoren, Lichtsensoren, Rotationssensoren und Temperatursensoren. Im Robotics Invention System enthalten sind zwei

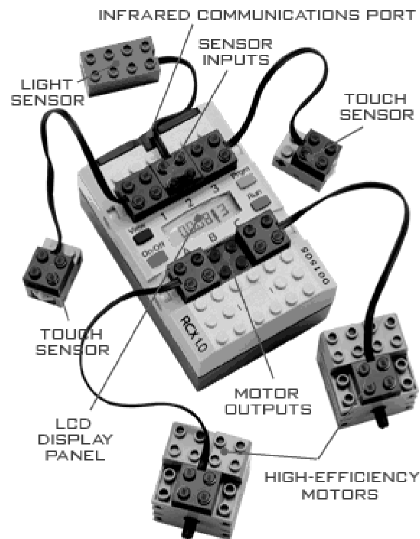


Abbildung 3.1: Der Kontrollbaustein „RCX" (Quelle:[LEG])

Berührungssensoren sowie ein Lichtsensor. Zusätzliche Sensoren können zwar separat erworben, aufgrund der geringen Anzahl anschließbarer Sensoren aber nicht gleichzeitig genutzt werden.

Für akustische Rückmeldungen befindet sich ein kleiner Piezo-Piepser im Inneren des Gehäuses, mit dem verschiedene Töne erzeugt werden können. Zur Übertragung von Programmen zum Kontrollbaustein dient die eingebaute Infrarotsende- und -empfangseinheit. Über diese kann man nicht nur das Betriebsystem, sondern auch Steuerkommandos übertragen. So lässt die IR-Schnittstelle zum Beispiel die Steuerung über eine Fernbedienung zu, indem empfangene Signale sofort in Aktionen umgesetzt werden. Auch die Kommunikation mit anderen RCX-Bausteinen ist eine Anwendungsmöglichkeit dieser Schnittstelle.

3.2.2 Hitachi H8 Mikrocontroller

Das RCX hat Lego in Zusammenarbeit mit dem Massachusetts Institute of Technology entwickelt. Als Steuereinheit wird ein Hitachi H8/3292 Mikrocontroller eingesetzt [Hitb].

3.2.2.1 Mikrocontroller im Allgemeinen

Einen Mikrocontroller könnte man als Ein-Chip-Computer bezeichnen. Er hat einen Prozessor, der sequentiell die Befehle abarbeitet, Speicher für Daten und Programme, Schnittstellen um mit der Außenwelt zu interagieren und einen Bus, der alles miteinander verbindet. Er entspricht somit der klassischen Von-Neuman Architektur wobei alles auf einem einzigen Chip vereint ist. Deshalb sind auch alle Komponenten etwas kleiner dimensioniert als man es von einem normalen Computer gewöhnt ist. So hat man nicht mehrere MByte Speicher zur Verfügung, sondern muss mit wenigen KByte zurechtkommen. Auf einen Hintergrundspeicher, wie zum Beispiel eine Festplatte, muss man völlig verzichten, und auch der Systemtakt ist oft auf wenige MHz begrenzt. Jedoch können Mikrocontroller häufig zusätzlich analoge und

digitale Signale verarbeiten. Dazu sind zum Beispiel Analog/Digital-Wandler, Zeitgeber und Impulswandler gleich auf dem Chip integriert.

Alle Komponenten auf einem Chip unterzubringen hat, gegenüber der Alternative, die Funktionalität aus einzelnen Chips aufzubauen, viele Vorteile. Da ein Mikrocontroller dadurch sehr klein ist kann er an Orten eingesetzt werden, für die andere Realisierungen zu groß sind. Nicht zu vernachlässigen ist auch die Unempfindlichkeit gegenüber äußeren Einflüssen, wie zum Beispiel Erschütterungen und Feuchtigkeit. Bei separat aufgebauten Schaltungen könnten dadurch Kontakte und Lötstellen beschädigt werden. Mikrocontroller sind in dieser Hinsicht robuster, da alle Verbindungen in einem Gehäuse eingegossen und keine Lötstellen vorhanden sind.

Ein weiterer Vorteil ist die bessere elektromagnetische Verträglichkeit. So werden durch die Integration der einzelnen Bestandteile die Verbindungsleitungen zwischen diesen viel kürzer. Eine Störung durch äußere elektromagnetische Einflüsse hat dadurch weniger Einfluss. Hinzu kommt, dass man für kürzere Leitungen geringere Spannungen einsetzen kann. Dadurch wird zum einen die Leistungsaufnahme reduziert, zum anderen verringert sich die elektromagnetische Abstrahlung.

Auch die geringen Kosten eines Mikrocontrollers verschaffen ihm einen großen Vorteil gegenüber anderen Lösungen. Zum einen sind die Herstellungskosten eines Mikrocontrollers geringer als die Produktion von mehreren einzelnen Chips, zum anderen ist auch die Montage eines Chips günstiger als die Bestückung einer Platine mit mehreren Chips.

3.2.2.2 Hitachi H8/3292

Der H8/3292, der im RCX eingesetzt wird, ist ein Mitglied der H8/3297 Mikrocontroller-Familie von Hitachi. Die Mitglieder einer Familie sind alle untereinander kompatibel und unterscheiden sich nur in der Speicherausstattung voneinander. So ist der Hitachi 3292, mit einer Ausstattung von nur 16 KByte ROM und 512 Bytes integriertem RAM, das kleinste Mitglied dieser Familie. Um das auszugleichen stehen ihm 32 KByte externes RAM im Kontrollbaustein zur Verfügung. Die Möglichkeit externes RAM zu nutzen bieten alle Mitglieder der H8/3297 Serie. Dazu haben sie einen speziellen Betriebsmodus, der einige ihrer Ein-/Ausgabe-Pins für die externe Erweiterung der Adress-, Daten- und Steuerleitungen reserviert.

Der Zugriff auf den Speicher erfolgt grundsätzlich über einen 16-bit-breiten Adressbus, egal ob externes oder internes RAM angesprochen werden soll. Jedoch lässt sich der Speicher nicht unbegrenzt erweitern, da man mit einer Adressbusbreite von 16 Bit maximal 65536 Speicherstellen ansprechen kann und außerdem Teile des Adressbereiches für das ROM und andere Geräte reserviert sind. Mit der 32 KByte RAM-Erweiterung im Lego-Kontrollbaustein ist deshalb auch schon die maximale Ausbaustufe des Mikrocontrollers erreicht.

Der Datentransport zwischen RAM bzw. ROM und CPU kann sowohl in Bytes zu 8 Bit als auch in Worten zu 16 Bit durchgeführt werden. Zwischen CPU und den anderen auf dem Chip integrierten Geräten ist hingegen nur ein 8-bit-breiter Datenbus verfügbar. Dieser Umstand führt manchmal zu Problemen beim Arbeiten mit 16-bit Registern, auf die später noch eingegangen wird.

Die Peripherie lässt sich auf dem Blockdiagramm in Abbildung 3.2 erkennen. Neben einem 10-bit, 8-Kanal Analog/Digital-Wandler, der eine angelegte Spannung in digitale Werte von 0 bis 1023 umwandeln kann, ist eine serielle Schnittstelle vorhanden, die zum Datenaustausch

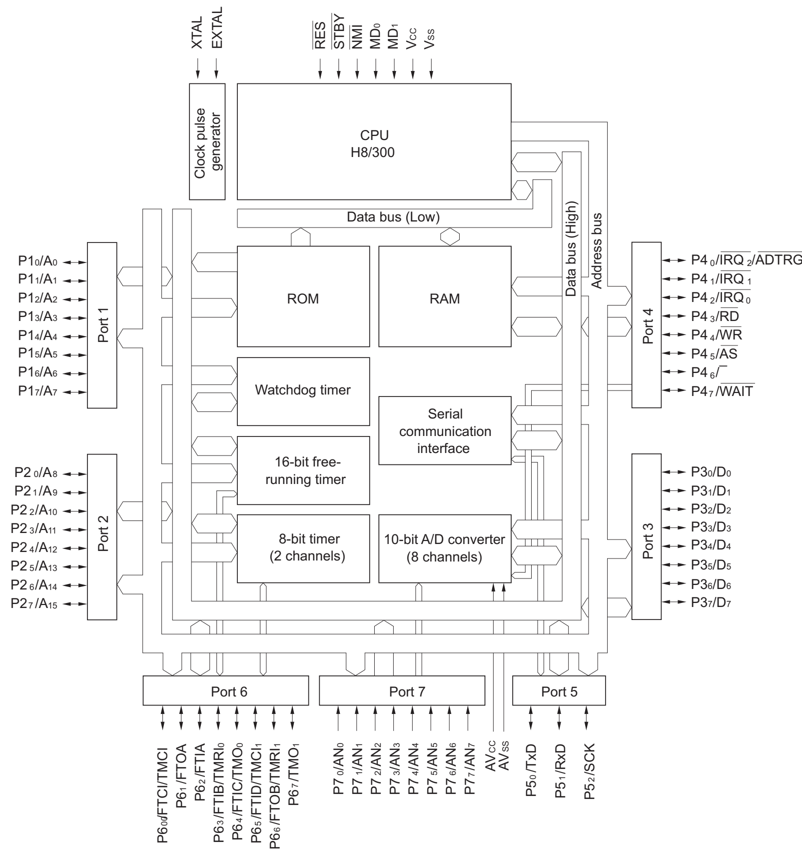


Abbildung 3.2: Blockdiagramm der H8/3297 Mikrocontroller-Familie (Quelle: [Hitb])

mit anderen Geräten genutzt werden kann. Des Weiteren sind ein 16-bit Zähler und zwei 8-bit Zähler verfügbar. Sie können in regelmäßigen Zeitintervallen Interrupts auslösen sowie externe Signale zu einer vordefinierten Zeit entgegennehmen. Als besonderer Zähler kommt noch der Watchdog-Timer hinzu, der sogar einen Reset des Mikrocontrollers auslösen kann.

Für Interaktionen mit seiner Umgebung stehen dem Hitachi Mikrocontroller 7 Schnittstellen, so genannte Ports, zur Verfügung. Einige davon können nur für die Eingabe, andere für Ein- und Ausgabe genutzt werden. Die meisten Ports werden durch interne Geräte beansprucht. Sie können jedoch auch von der Software frei verwendet und direkt abgefragt werden, wenn die Zusatzmodule deaktiviert sind.

Der Prozessor des Mikrocontrollers wird an späterer Stelle noch beschrieben, hier soll nur auf die Taktfrequenz hingewiesen werden. Der Hitachi 3292 wird in drei Versionen angeboten: eine 16 MHz Version, die 5 Volt Betriebsspannung benötigt, eine Version mit 12 MHz bei 4 Volt sowie eine 10 MHz Variante für 3 Volt Betriebsspannung. In Legos RCX wird die 16 MHz-Variante verwendet.

Bevor auf die Vorgänge nach dem Aktivieren des Mikrocontrollers bzw. des Lego-Kontrollbausteins eingegangen wird, sollen noch die drei Betriebsmodi des Hitachi Minicomputers erläutert werden. In den Modi 1 und 2, den so genannten „expanded" Modi, sind der Adress- und Datenbus nach außen durchgeführt, sodass man externe Geräte oder RAM direkt an den Bus anschließen kann. Der Unterschied der zwei Modi ist die Verwendbarkeit des

ROMs, welches im Modus 1 deaktiviert ist, im Modus 2 aber genutzt werden kann. Schließlich gibt es noch einen dritten Modus, den „single-chip” Modus, der die Erweiterung des Bussystems nicht zulässt. Die Wahl des Moduses wird über zwei Pins durchgeführt, welche im Rahmen der Resetsequenz des Mikrocontrollers überprüft werden. Im RCX wird der H8/3292 Mikrocontroller im Modus 2 betrieben, somit kann er das externe RAM und die Funktionen des Lego-ROMs nutzen.

3.3 Software des RCX

Nachdem die Hardware des Lego-Kontrollbausteins betrachtet wurde, soll nun ein Blick auf die Software geworfen werden, die den Mikrocontroller steuert. Dazu werden zuerst die Vorgänge nach dem Aktivieren des Steuerbausteins erläutert.

3.3.1 Startvorgang

Beim Einschalten ist zwischen dem Einlegen der Batterien, welches den Beginn der Spannungsversorgung darstellt, und einem Druck auf den “On-Off” Taster des Kontrollbausteins zu unterscheiden. Letzteres weckt den Mikrocontroller aus einem energiesparenden Schlafmodus, dem so genannten *Software-Standby* Modus. Nachdem der Mikrocontroller wieder aktiv ist, wird das aktuelle Programm wieder dort fortgeführt, wo es unterbrochen wurde. Eine genauere Erklärung der Energiesparmodi wird in Abschnitt 4.3.4 gegeben.

War der Kontrollbaustein jedoch von der Spannungsversorgung getrennt und wird wieder mit Spannung versorgt, so beginnt die Ausführung eines im ROM des Mikrocontrollers gespeicherten Programms. Dieses bietet mehrere von Lego vordefinierte Verhaltensarten für den zukünftigen Roboter an, welche man über die “Prgm” Taste am RCX auswählen kann und mit der “Run” Taste aktiviert.

Außerdem erlaubt es das Programm im ROM eine Art Betriebssystem, hier spricht man jedoch von Firmware, über die Infrarot-Schnittstelle zu empfangen und im RAM abzulegen. Nachdem der Transfer abgeschlossen ist, wird das neue Programm aufgerufen und ihm somit die Kontrolle übertragen. Vom ROM werden nun üblicherweise nur noch einzelne Funktionen, vor allem zur Ansteuerung der externen Peripherie, wie zum Beispiel der Motoren, des Displays oder der Sensoren, genutzt. Die weiteren Möglichkeiten hängen von der Firmware ab. Normalerweise kann man dann aber über die IR-Schnittstelle kleine Programme zum Kontrollbaustein übertragen, die anschließend von der Firmware verwaltet und zur Ausführung gebracht werden.

3.3.2 Firmware

In den folgenden Abschnitten werden zwei Betriebssysteme für den Lego-Kontrollbaustein vorgestellt, um einen Überblick über deren Möglichkeiten zu erhalten. Als erstes wird das Original-Lego-System, welches dem Robotics Invention System auf der CD beiliegt[1], betrachtet. Als zweites wird eine alternative Firmware Namens legOS (siehe [Nog]) vorgestellt.

[1] Die Original-Firmware liegt auf der Mindstorms-CD im Verzeichnis: \firm\firm309.lgo

3.3.2.1 Original-Lego-Firmware

Die von Lego für das RCX vorgesehene Firmware wird im Rahmen eines Lernprogramms, das von der mitgelieferten CD ausgeführt wird, zum Kontrollbaustein übertragen. Nach erfolgreicher Übernahme der Kontrolle meldet die Firmware ihre Bereitschaft durch ein akustisches Signal und wartet anschließend auf Befehle. Der Roboterentwickler kann nun mithilfe des im Robotics Invention System enthaltenen Windows-Programms selbst eine Verhaltensvorgabe für seinen Roboter zusammenstellen und anschließend zum RCX übertragen.

Die Erstellung dieser Verhaltensprogramme erfolgt über eine grafische Oberfläche, die es erlaubt einen Programmfluss mittels verschiedener "Puzzleteile" graphisch zu entwickeln. Es wurde dabei besonderer Wert auf die einfache und kindgerechte Benutzerführung gelegt, sodass auch Laien damit zurechtkommen.

Hat man ein Programm erstellt, so wird es in so genannten RCX-Byte-Code übersetzt und in den Kontrollbaustein geladen. Die Firmware spielt dabei die Rolle des Empfängers und verwaltet jedes Programm in einem von bis zu fünf getrennten Bereichen. Über die Tasten am RCX kann man dann das gewünschte Programm auswählen, starten und auch wieder stoppen.

Befindet sich der Roboter mit dem RCX in Reichweite des IR-Transmitters, so bietet die Lego-Firmware eine weitere Möglichkeit. Mit einer zu Diagnosezwecken mitgelieferten Software kann man Befehle übertragen, die sofort interpretiert und in die gewünschte Aktion umgesetzt werden. So kann man zum Beispiel Lichtsensoren kalibrieren, indem man die Messwerte direkt am Computerbildschirm abließt. Über dieses Prinzip kann man seinen Roboter auch per Fernbedienung steuern oder mehrere RCX zusammenkoppeln, um so zum Beispiel die Anzahl der Sensoren pro Roboter zu erhöhen.

Da die Möglichkeiten durch die grafische Programmieroberfläche stark begrenzt werden, bietet Lego auch ein ActiveX-Modul an. Mit dessen Hilfe kann man Programme in Visual Basic oder anderen Windows-Programmiersprachen, die ActiveX-Module unterstützen, schreiben und anschließend zum Roboter übertragen.

3.3.2.2 legOS

Als Alternative zum Lego-eigenen Betriebssystem ist die Firmware legOS zu sehen. legOS liegt als C-Quellcode vor und muss erst mit Hilfe eines Crosscompilers in Hitachi H8/300 Maschinencode übersetzt werden. Zur Übertragung in den Kontrollbaustein liegt ebenfalls ein Programm in C-Quellcode vor.

legOS übernimmt im RCX die Aufgabe eines Treibers und stellt den Benutzerprogrammen einfache Funktionen zur Verfügung, um beispielsweise die serielle Schnittstelle ansprechen zu können. Außerdem verwaltet legOS die Threads der einzelnen Benutzerprogramme durch einen zeitscheibengesteuerten Scheduler.

So wie legOS selbst müssen auch die "Verhaltensprogramme" durch einen Crosscompiler in H8/300 Maschinencode übersetzt werden. Ab legOS Version 0.2.0 können diese Programme dann zum RCX, auf dem legOS läuft, übertragen werden. Dort werden sie dann dynamisch gebunden und können dadurch die Firmware-Funktionen nutzen. Vor der Version 0.2.0 wurde die Firmware zusammen mit einem Benutzerprogramm auf dem Entwicklungssystem kompiliert und gebunden. Danach konnte man dieses Paket als Firmware zum RCX übertragen. Benutzerprogramme konnten also nicht nachgeladen werden.

Die Firmware legOS bietet einige Vorteile gegenüber der Original-Firmware. Man hat zum Beispiel umfangreichere Möglichkeiten eigene Programme zu schreiben. Man kann diese beispielsweise in C schreiben und hat somit alle in C möglichen Konstrukte zur Verfügung. Prinzipiell kann der Roboterentwickler jedoch auch jede andere Programmiersprache verwenden, die sich in Hitachi-Maschinencode übersetzen lässt und mit der man in der Lage ist die C-Funktionen von legOS zu verwenden. Dadurch ist die Programmentwicklung nicht nur auf Microsoft Windows beschränkt, sondern von jedem Computer und Betriebssystem aus möglich, auf dem man Programme für einen H8/300 Mikrocontroller erstellen kann.

Des Weiteren erzielt legOS einen erheblichen Geschwindigkeitsvorteil gegenüber der Original-Lego-Firmware. Die in Maschinencode übersetzten Programme können direkt von der CPU des Mikrocontrollers abgearbeitet werden. Die Lego-Firmware hingegen muss die in RCX-Byte-Code vorliegenden Programme interpretieren und benötigt deshalb mehr Zeit. Auch die Beschränkung der RCX-Code Programme auf 32 Variablen fällt mit diesem Prinzip weg. Die Größe des Benutzerprogramms ist nur durch den vorhandenen Speicher begrenzt.

Weiterhin bietet die Programmierung auf dieser Systemebene den Vorteil, uneingeschränkten Zugriff auf den Mikrocontroller und seine integrierten Module nehmen zu können. So muss man zum Beispiel nicht die ROM Funktionen zum Zugriff auf die Infrarot-Sendeeinheit benutzen, sondern könnte die serielle Schnittstelle des Hitachi-Mikrocontrollers direkt ansprechen und damit beispielsweise eine Fernbedienung für eine Stereoanlage oder ein Fernsehgerät verwirklichen.

Die Nachteile, die legOS mit sich bringt, sind zum Teil schon an den obigen Ausführungen zu erkennen. So werden vom Benutzer mehr Kenntnisse über das gesamte System verlangt. Man muss Programme in C schreiben und kompilieren können, außerdem benötigt man einen Crosscompiler, den man unter Umständen auch erst erstellen muss (siehe dazu Anhang D).

Kapitel 4

Realisierung des RCX Simulators

Nachdem das reale System vorgestellt wurde, soll nun die Simulation betrachtet werden. Das Ziel war es einen Simulator zu erstellen, der in der Lage ist, jedes Betriebssystem für den LEGO-Kontrollbaustein auszuführen, ohne es vorher anpassen zu müssen.

Als Vorbild diente ein Simulator für den MIPS R3000 Prozessor [Fan][Int94]. Einige Konzepte, vor allem die Adressraumverwaltung und wenige Teile der Prozessorsimulation, konnten nach einigen Anpassungen übernommen werden. Entwickelt wurde der Simulator in JAVA, wodurch der Vorteil entsteht, dass er auf vielen Plattformen einsetzbar und somit unabhängig von der Hardware des ausführenden Systems ist.

Eine grobe Übersicht der Software-Architektur findet sich in Abbildung 4.1. In Anhang B sind einige Klassen nochmals genauer abgebildet.

Im Zentrum der Simulation steht der Mikrocontroller und dessen integrierte Zusatzmodule. Eine zentrale Komponente ist deshalb auch die Klasse Microcontroller. Sie initialisiert den simulierten Prozessor und alle Zusatzmodule. Nachdem die Simulation gestartet wurde, steht die Klasse CPU im Mittelpunkt. Sie koordiniert den Ablauf der Simulation durch die Methode run, die als Thread ausgeführt wird. Der Aktivitätsträger bearbeitet dabei eine Endlosschleife, die zum einen für die Ausführung der Instruktionen zuständig ist, zum anderen die Zusatzmodule steuert. Ein Flussdiagramm der Hauptschleife findet man in Abschnitt 4.3.2.

Darüberhinaus wird noch ein zweiter Aktivitätsträger verwendet, der die run-Methode der Klasse IRI ausführt. Dieser dient aber nur der Simulation der seriellen Schnittstelle und ist im Gesamtkonzept des Simulators eher nebensächlich.

Der Zugriff auf alle Zusatzmodule ist durch einen simulierten Speicherbus vereinheitlicht. Realisiert wird er durch eine zentrale Adressverwaltung, die alle Zugriffe an die entsprechenden Objekte weiterleitet. Für diese Aufgabe ist eine Instanz der Klasse Memory vorgesehen. Die Zusatzmodule sind alle durch Unterklassen von MemoryRegion implementiert. Dadurch wird die einheitliche Schnittstelle des Speicherbusses gewährleistet.

In den folgenden Abschnitten werden einzelne Bereiche des Simulators genauer betrachtet. Dabei wird meist erst die Funktion der realen Hardware beschrieben, um anschließend auf die Umsetzung in Software einzugehen.

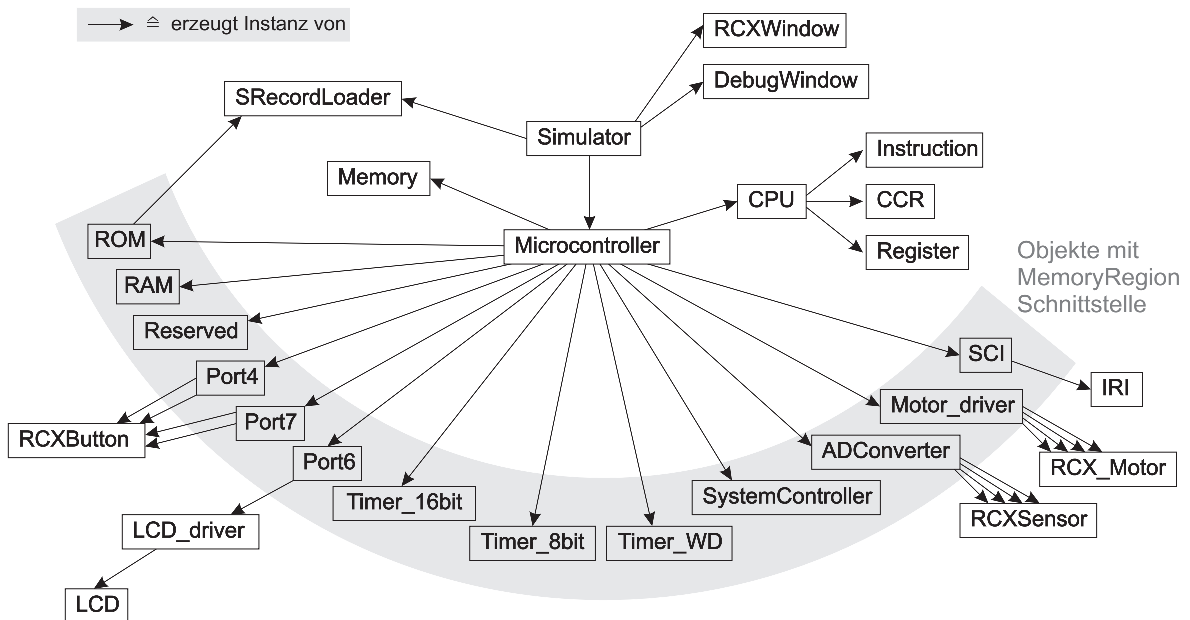


Abbildung 4.1: Übersicht der Klassen des RCX Simulators

4.1 Startvorbereitungen

Gestartet wird der Simulator über die Klasse Simulator. Diese verarbeitet die Kommandozeilenparameter und speichert die gewonnenen Informationen in statischen Variablen (siehe Klassendiagramm B.1 im Anhang). Außerdem erzeugt sie die Fenster für die Bildschirmdarstellung. Abbildung A.1 im Anhang zeigt das Erscheinungsbild des Simulators.

Die Hauptfenster-Klasse RCXWindow ist von der Java-Swing Komponente JFrame abgeleitet. Somit kann sie als Container dienen, zu dem alle Objekte, die mit dem Benutzer interagieren wollen hinzugefügt werden. Damit man das Aussehen des Fensters besser kontrollieren kann, werden alle Positionsangaben der einzelnen Komponenten in dieser Klasse zentral verwaltet.

Neben den Fenster-Objekten wird auch ein Microcontroller-Objekt erzeugt, welches alle Bestandteile der Simulation initialisiert (siehe Abbildung 4.1 und Klassendiagramm B.2 im Anhang). Bevor die Simulation gestartet werden kann, müssen aber noch die Funktionen des ROMs bereitgestellt werden. Da auf den im ROM des RCX enthaltenen Code jedoch Copyright-Rechte seitens Lego bestehen, ist er nicht fest in den Simulator eingebaut. Stattdessen muss er in Form eines ROM-Abbildes[1] beim Starten nachgeladen werden.

4.1.1 Motorola S-Record Format

Das Abbild des ROMs muss dabei im Motorola S-Record Format [Tra] vorliegen. Dieses Format legt die Art und Weise fest, wie die Rohdaten, die später im Speicher bzw. im ROM des Simulators liegen sollen, in einer ASCII-Datei gespeichert werden.

[1]zur Erstellung eines ROM-Abbildes siehe Anhang C

Dazu werden die Daten in kleine Blöcke von bis zu 32 Bytes zerschnitten. Jeder dieser Blöcke wird durch eine Zeile, S-Record genannt, in der Datei repräsentiert. Das Format dieser Zeilen wird in Abbildung 4.2 dargestellt.

Typ	Anzahl	Adresse	Daten	Prüfsumme
2 Zeichen	2 Zeichen	2 bis 6 Zeichen	0 bis 64 Zeichen	2 Zeichen

Abbildung 4.2: Aufbau eines S-Records

Die ersten beiden Zeichen beschreiben den Typ des S-Records. Hiervon gibt es 8 verschiedene: S0 bis S3, S5 und S7 bis S9. Typ S0 gibt an, dass es sich um einen Header-Record handelt, der Informationen allgemeiner Art, wie zum Beispiel eine kurze Beschreibung oder einer Versionsnummer des Codes, enthält. Records der Typen S1 bis S3 enthalten Daten, die im Speicher abgelegt werden sollen. Der Unterschied zwischen den Typen ist, dass die Adresse, an die diese Daten geladen werden sollen, durch 2, 3 bzw. durch 4 Bytes angegeben wird. Je nachdem, ob es sich bei dem Zielsystem um ein 16-bit, 24-bit oder 32-bit System handelt, werden entweder nur S1, nur S2 oder nur S3 Records verwendet. S5 gibt die Summe der zuvor übertragenen S-Records an, um zum Beispiel eine fehlende Übertragung feststellen zu können. Schließlich gibt es noch Einträge von Typ S7 bis S9. Diese geben die Startadresse an, an der nach der Übertragung die Programmausführung beginnen soll. Dabei kann die Adresse wieder durch 2, 3 oder 4 Bytes angegeben werden, analog zu den Daten-Records. In einer normalen S-Record Datei findet man von diesen Records höchstens einen.

Nach der Kennung folgen Gruppen zu je 2 Zeichen, die als hexadezimale Zahl interpretiert je ein Byte darstellen. Das erste so codierte Byte gibt die Anzahl der noch folgenden Bytes in dieser Zeile an. Direkt anschließend folgt eine Adresse. Bei einem S0 Record ist diese nicht genutzt und mit Null angegeben, bei Daten-Records (S1 bis S3) wird mit dieser Adresse der Startpunkt angegeben, an den die Daten im Speicher geladen werden sollen. Ein S5-Record speichert im Adressfeld die Anzahl bisher übertragener Daten-Zeilen, bei S7 bis S9 steht hier die Adresse, an der die Ausführung beginnen soll.

Alle folgenden Bytes bis auf das letzte sind die eigentlichen Daten. Records der Typen S5 und S7 bis S9 enthalten kein Daten-Feld, da bereits alle Informationen im Adressfeld angegeben wurden. Die letzten beiden Zeichen eines S-Records repräsentieren schließlich eine Prüfsumme über die gesamte Zeile.

Für den Hitachi-Simulator ist nur der Record-Typ S1 wichtig. Die Adressangabe besteht bei diesem Record aus 2 Bytes, also 4 Zeichen, und reicht aus, um den gesamten Speicher des Mikrocontrollers anzusprechen.

4.1.2 SRecordLoader-Klasse

Das Laden und Dekodieren einer S-Record Datei übernimmt im Simulator die Klasse SRecordLoader. Der Konstruktor verlangt nach einem Datei- und einem Memory-Objekt (siehe Abbildung B.1 im Anhang). Über das Datei-Objekt kann auf die S-Record Daten zugegriffen werden, das Memory-Objekt steht für den simulierten Speicher, in den die Daten geschrieben werden sollen.

Das SRecordLoader-Objekt liest Zeile für Zeile aus der Datei und analysiert den Typ des S-Records. Wird ein S1-Record gefunden, so werden die darin enthaltenen Daten über das Memory-Objektes an die spezifizierte Adresse im simulierten Speicher geschrieben. Zuvor werden jedoch die Daten dekodiert, das heißt, dass aus zwei ASCII-Zeichen immer ein Byte erzeugt wird. Records anderer Typen als S1 werden ignoriert, da sie keine relevanten Informationen enthalten.

4.2 Adressraumverwaltung

4.2.1 Überblick

Der Adressbus der H8/3297 Mikrocontroller-Familie ist 16-bit-breit. Somit können 2^{16} verschiedene Speicherstellen angesprochen werden. Abbildung 4.3 zeigt die Unterteilung des Adressraums des Hitachi H8/3292 Mikrocontrollers[2].

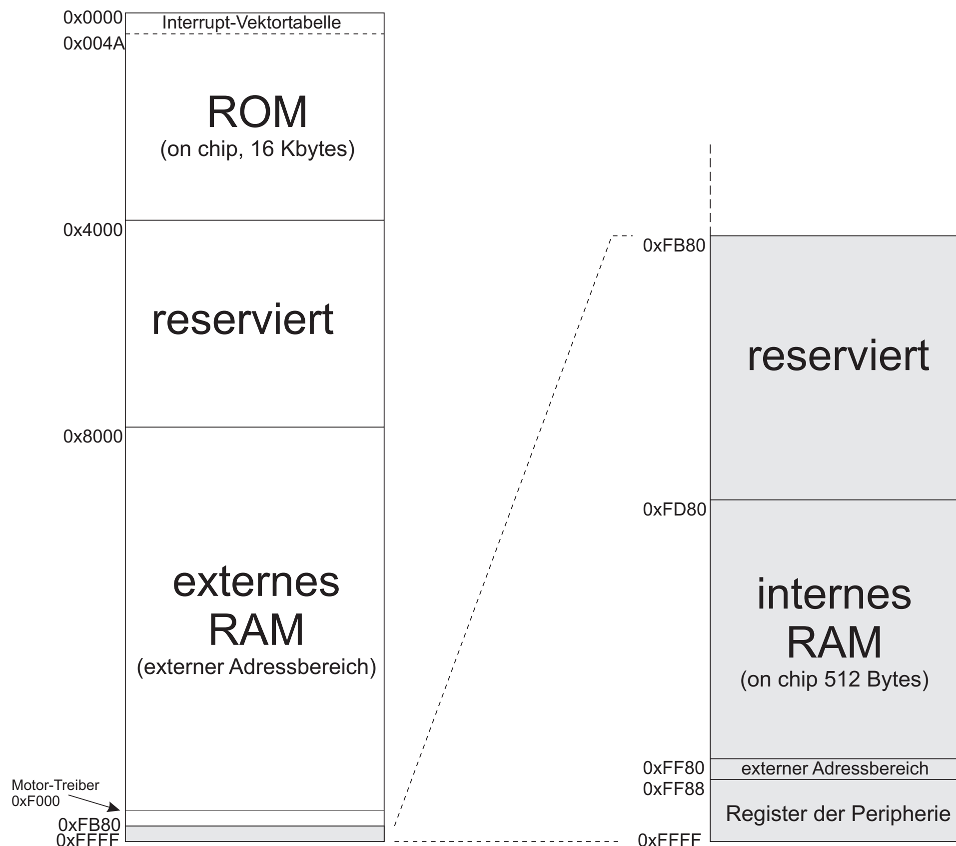


Abbildung 4.3: Adressraum des H8/3292 Mikrocontrollers

In einem realen System ist die Zuteilung der Adressen durch die Hardware festgelegt. Wird eine Adresse am Bus angelegt, so reagiert genau das Gerät, dem diese Adresse zugeordnet ist,

[2] Zur Darstellung von Zahlen im Hexadezimalsystem wird die in C oder Java übliche Schreibweise mit vorangestelltem 0x verwendet.

alle anderen Geräte müssen sich ruhig verhalten. Ein Gerät kann zum Beispiel die Adressumsetzung des Speichers oder ein peripheres Gerät sein, dessen Register sich über den Adressbus ansprechen lassen.

In der Simulation wird die Adressbereich-Zuordnung durch ein zentrales Verwaltungs-Objekt erledigt. Dieses nimmt alle Anfragen entgegen und verteilt sie auf die zuständigen Objekte.

4.2.2 Memory-Klasse

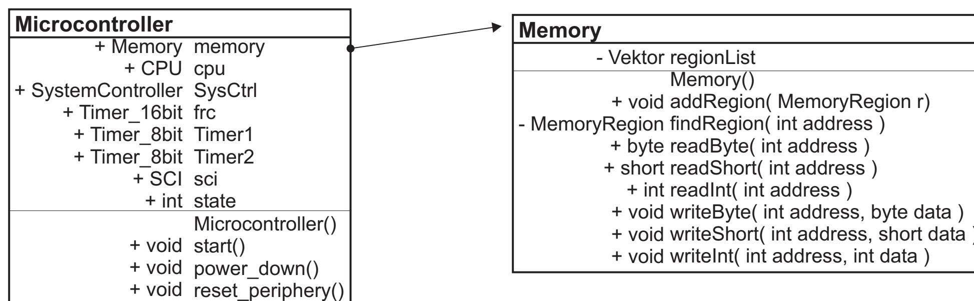


Abbildung 4.4: Die Klasse Memory

Die Adressraumverwaltung wird durch die Klasse Memory durchgeführt und wurde fast unverändert vom zugrundeliegenden MIPS R3000 Simulator übernommen. Der Simulator besitzt nur ein Objekt dieser Klasse, welches von dem Hauptobjekt Microcontroller den anderen Objekten zur Verfügung gestellt wird (siehe Abbildung 4.4[3] und Anhang B). Jede Anfrage an eine Speicherstelle wird an dieses Objekt gerichtet. Dort wird eine Vorauswahl getroffen, um sie an die richtige Klasse weiterzuleiten. Für den Zugriff auf Speicherstellen stehen die Methoden readByte, readShort, readInt, writeByte, writeShort und wirteInt zur Verfügung. Die Klasse Memory ist somit der Ersatz für den Adress- und den Datenbus des realen Mikrocontrollers.

Klassen, die einen Speicherbereich für sich beanspruchen möchten, müssen bei der Adressraumverwaltung registriert werden. Zu diesem Zweck ist die Methode addRegion der Klasse Memory vorgesehen. Diese Methode erwartet ein MemoryRegion-Objekt, welches den gewünschten Adressbereich angibt und an das die Anfragen für diesen Bereich weitergeleitet werden können. Alle Geräte, die über den Adressbus ansprechbar sein sollen, müssen deshalb durch Objekte repräsentiert werden, welche von MemoryRegion abgeleitet wurden.

Die Klasse Memory speichert intern eine Liste mit Referenzen auf MemoryRegion-Objekte. Jeder Aufruf von addRegion fügt einen neuen Verweis zu dieser Liste hinzu. Die Gesamtheit der von diesen Objekten unterstützten Adressen ergibt den gültigen Adressraum. Ein Aufruf einer read bzw. write Methode wird an eines dieser Objekte weitergeleitet. Dazu wird die private Methode findRegion verwendet. Sie wird mit der gewünschten Adresse versorgt und liefert daraufhin das entsprechende MemoryRegion-Objekt zurück.

Der Adressbereich, für den ein solches Objekt zuständig sein soll, ist durch die öffentlichen Variablen from und to definiert (siehe Abbildung 4.6). Die Methode findRegion geht die Liste der Speicherbereiche durch und prüft, von welchem Objekt die Adresse unterstützt wird. Sollte es vorkommen, dass sich die Adressbereiche zweier MemoryRegion-Objekte überschneiden,

[3] eine Legende der Symbole findet sich in Abbildung B.1 im Anhang

so findet die findRegion-Methode das zuerst registrierte Objekt, da addRegion neue Bereiche immer am Ende der Liste hinzufügt.

Die Technik, die bei der Adressraumverwaltung eingesetzt wird, macht es somit möglich, auf Adressen zuzugreifen, ohne sich darum zu kümmern, ob sie dem Speicher oder einem Schnittstellenregister zugeordnet sind.

4.2.3 MemoryRegion-Klasse

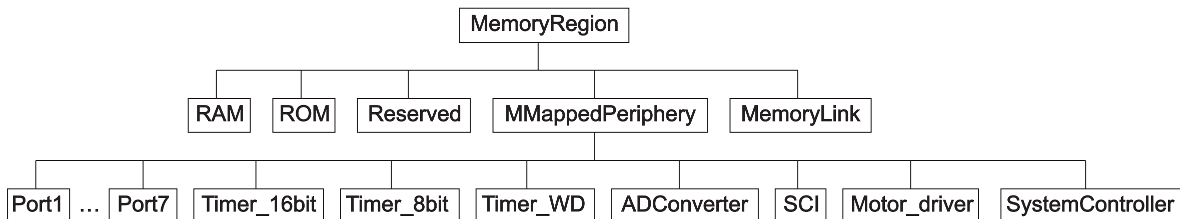


Abbildung 4.5: von MemoryRegion abgeleiteten Klassen

Wie bereits erwähnt müssen alle Klassen, die über den virtuellen Speicherbus erreichbar sein sollen, von der Klasse Memory als MemoryRegion-Objekt ansprechbar sein. Deshalb ist die Familie der abstrakten Klasse MemoryRegion relativ groß, wie Abbildung 4.5 zeigt (vgl. auch Klassendiagramm B.3 und B.4 im Anhang).

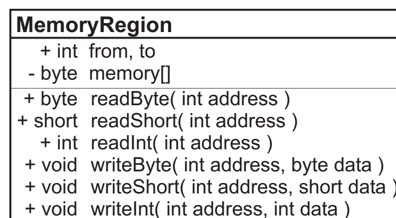


Abbildung 4.6: Die Klasse MemoryRegion

Die Implementierungen der read- und write-Methoden, die von der Klasse bereitgestellt werden, sind für den Zugriff auf große Speicherbereiche wie beispielsweise RAM oder ROM gedacht. So enthält ein MemoryRegion-Objekt neben Angaben zum Adressbereich, für das es zuständig ist, auch noch eine Referenz auf ein Byte-Array (siehe Abbildung 4.6). Dieses Array dient als Speicherbereich um die Daten, welche im RAM bzw. ROM gespeichert sind, aufzunehmen. Die Standard-Vorgehensweise eines MemoryRegion-Objektes ist es jede Anfrage mit dem Inhalt des Byte-Arrays zu beantworten und jeden schreibenden Befehl auf dem Array auszuführen. Dazu wird die Stelle im Array berechnet und die Daten dort heraus bzw. hinein kopiert. Der Speicher für das referenzierte Array wird jedoch erst von abgeleiteten Klassen reserviert, da nicht jedes auf MemoryRegion basierende Objekt ein Array benötigt.

4.2.3.1 RAM

In der vom MemoryRegion abgeleiteten Klasse RAM muss lediglich das Byte-Array zur Aufnahme des Speicherinhaltes angelegt werden. Alle anderen Aufgaben werden durch die von

der Vaterklasse geerbten Methoden erledigt.

4.2.3.2 ROM

Das ROM wird durch die gleichnamige Klasse simuliert, welche auch eine Unterklasse von MemoryRegion ist. Im Konstruktor wird, wie bei der Klasse RAM, ein Speicherbereich angelegt. Er dient hier zu Aufnahme des ROM Inhaltes. Der Zugriff auf diesen Speicherbereich wird durch die geerbten Methoden möglich. Da man allerdings nicht schreibend auf den ROM-Inhalt zugreifen darf, werden alle write-Methoden überschrieben. Die neuen, von der Klasse ROM bereitgestellten Methoden, werten jeden Aufruf als Fehler und werfen eine Java-Ausnahme.

Zum Befüllen des ROMs, vor dem Start der Simulation, wird die Klasse SRecordLoader verwendet, die das ROM-Abbild aus einer Datei lädt und installiert. Dazu benötigt diese Klasse jedoch Schreibzugriff auf den Bereich des ROMs. Da ein verändernder Zugriff auf den ROM-Inhalt aber nur noch innerhalb der Klasse möglich ist, steht die Methode load zur Verfügung (siehe Klassendiagramm B.3 im Anhang). Diese wird mit den Parametern für ein SRecordLoader-Objekt aufgerufen und schaltet die Schreib-Zugriffe auf das ROM vorübergehend frei.

Um das zu realisieren wird eine private Variable Namens writing_allowed verwendet, die sich auf die Funktion der Zugriffsmethoden auswirkt. Ist die Variable auf true gesetzt, so wird ein Aufruf einer write-Methoden an die Vaterklasse weitergeleitet. Dort ist ein Standard-Schreibzugriff implementiert, welcher wie bei der Klasse RAM den Inhalt des Byte-Arrays verändert. Ist die Variable auf false gesetzt, so führt jeder Aufruf dieser Methoden, wie oben erwähnt, zu einem Fehler. Die load-Methode muss daher vor Erzeugung des SRecordLoader-Objekts die Variable auf true setzen und nach Abschluss der Arbeit wieder zurück auf false.

4.2.3.3 MemoryLink-Klasse

Die Klasse MemoryLink ist ein Verbindungsglied zwischen der Adressraumverwaltung und den Klassen der Peripherie. Nötig wird diese spezielle Klasse, da die Adressraumverwaltung immer nur einen Adressbereich zu einem Objekt zuordnen kann. Speziell der Adressbereich eines Zusatzmoduls ist aber meist nicht zusammenhängend, sondern besteht aus mehreren einzelnen Abschnitten. Werden dazwischenliegende Adressen von anderen Geräten beansprucht, so müssten jedem Geräte-Objekt mehrere Adressbereiche zugeordnet werden.

Abbildung 4.7 zeigt die Situation an einem Beispiel. Jeder Port soll durch ein eigenes Objekt repräsentiert werden. Das Objekt des Port 2 beispielsweise stellt drei Register bereit, die über den Speicherbus erreichbar sein sollen. Die Adressen dieser Register sind aber nicht fortlaufend. Hinzu kommt, dass dazwischenliegende Adressen anderen Port-Objekten zugeordnet werden sollen.

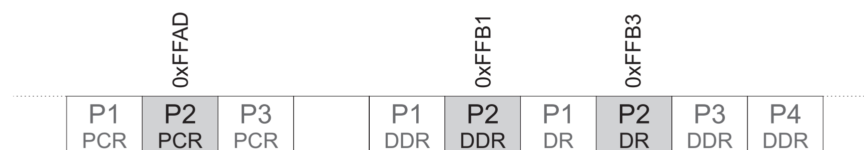


Abbildung 4.7: Adressbelegung des Port 2

Würde man dem Port 2 den gesamten Adressbereich, von der ersten verwendeten Adresse (0xFFAD) bis zur letzten (0xFFB3), zuordnen, so würde sich der Adressbereich mit dem der anderen Port-Objekte überschneiden. Wie schon erwähnt, erhält dann das Objekt die Zugriffe, welches zuerst bei der Adressraumverwaltung registriert wurde.

Da die Liste im Memory-Objekt nur Referenzen auf die MemoryRegion-Objekte enthält, kann man ein Objekt auch nicht mehrmals mit verschiedenen Adressen registrieren. Jeder zusätzliche Aufruf von addRegion würde nur eine weitere Referenz zu der Liste hinzufügen. Die Zuständigkeit eines Objekts wird jedoch erst bei einem Zugriff geprüft. Alle Verweise auf das Port-Objekt würden daher die gleiche Adresse repräsentieren.

Zur Lösung dieses Problems dient die Klasse MemoryLink. Durch sie kann das bestehende Konzept des MIPS R3000 Simulators erweitert werden, ohne die Basisklassen Memory und MemoryRegion zu verändern.

Für jeden Adressbereich, für den ein Objekt zuständig sein soll, wird ein MemoryLink-Objekt angelegt. Dabei wird dem Konstruktor eine Referenz auf ein MemoryRegion-Objekt übergeben (siehe Klassendiagramm B.3 im Anhang), an das die Aufrufe weitergeleitet werden sollen. Außerdem wird noch der Adressbereich festgelegt, der mit diesem Objekt verknüpft werden soll. Da die Klasse MemoryLink eine Unterklasse von MemoryRegion ist, wird der Bereich in die Variablen from und to eingetragen. In einem MemoryLink Objekt für die von Port 2 verwendet Adresse 0xFFB1, würde deshalb sowohl from als auch to auf 0xFFB1 gesetzt werden.

Die MemoryLink-Objekte werden dann anstatt des ursprünglichen Objekts an die addRegion-Methode der Klasse Memory übergeben. Im Fall des Port 2 sind das drei MemoryLink-Objekte, eines für jede Adresse.

Erfolgt nun ein Zugriff auf eine dieser Adressen, so wird die findRegion-Methode ein MemoryLink-Objekte finden, an das die Zugriffe weitergeleitet werden. Dort sind alle Zugriffsmethoden so definiert, dass ein Aufruf an das Objekt weitergeleitet wird, welches bei der Konstruktion spezifiziert wurde. Im Beispiel erhält somit das Objekt des Port 2 die Zugriffe.

Zur Verdeutlichung ist das Konzept noch einmal in der Abbildung 4.8 dargestellt.

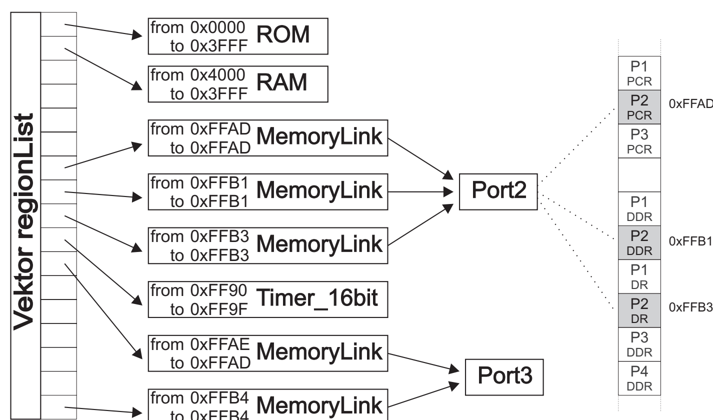


Abbildung 4.8: Funktionsweise von MemoryLink-Objekten

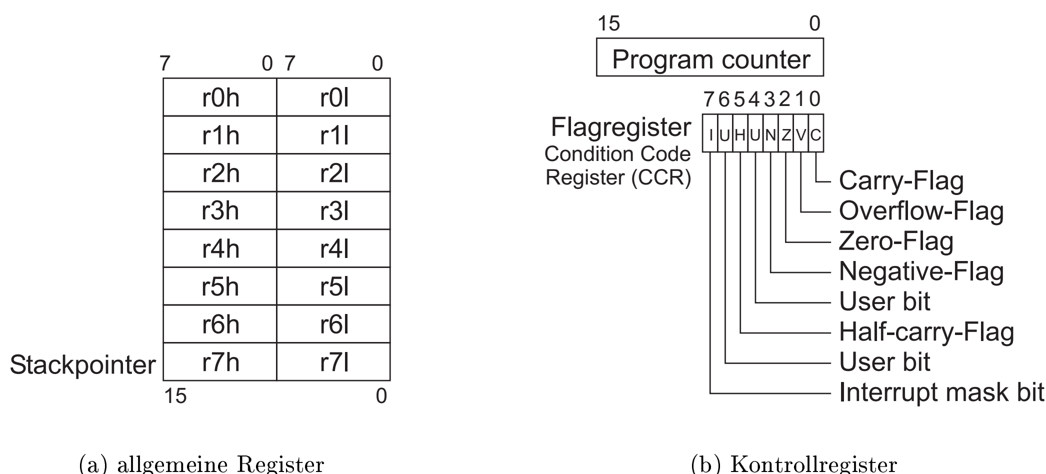
4.3 CPU

Im Zentrum eines Mikrocontrollers steht der Prozessor. Er führt alle Befehle aus und steuert alle anderen Komponenten. Auch im Simulator nimmt er einen zentralen Platz ein. Doch bevor man auf die Einzelheiten der Simulation eingehen kann, muss man sich zuerst einen genaueren Überblick über die Aufgaben und die Funktionsweise der CPU verschaffen.

4.3.1 Überblick

Wie schon erwähnt ist im Lego-Kontrollbaustein RCX ein Mikrocontroller von Hitachi im Einsatz. In diesem Mikrocontroller ist ein 16-bit Prozessor mit der Bezeichnung H8/300 integriert. Eine genaue Beschreibung des Prozessors und aller unterstützten Befehle findet sich in [Hita].

4.3.1.1 allgemeine Register



(a) allgemeine Register

(b) Kontrollregister

Abbildung 4.9: Die Register der Hitachi H8/3297 Mikrocontroller-Familie

Um Operanden aufnehmen zu können hat der Prozessor acht 16-bit Register (siehe Abbildung 4.9(a)). Eines dieser Register, `r7`, ist als Zeiger auf den Stack vorgesehen. Es lässt sich zwar prinzipiell auch für andere Operationen verwenden, jedoch sind die Befehle `push` und `pop`, die für die Stack-Benutzung vorgesehen sind, auf dieses Register festgelegt. Außerdem basiert jede implizite Benutzung des Stacks, beispielsweise die Sicherung der Rücksprungadresse bei einem Subroutinen-Aufruf, auf dem Wert in diesem Register.

Alle 16-bit Register lassen sich alternativ auch als zwei getrennte 8-bit Register ansteuern. So ist zum Beispiel der Inhalt des Registers `r5` auch über die beiden 8-bit Register mit der Bezeichnung `r5h` für die höherwertigen und `r5l` für die niederwertigen 8 Bit zu erreichen. Ob ein Befehl ein 8-bit oder ein 16-bit Register benötigt, kann nicht allgemein festgelegt werden. Manche Befehle stehen in einer 8-bit und einer 16-bit Variante zur Verfügung, wie zum Beispiel `mov`. Andere wiederum können nur mit 8-bit Werten arbeiten.

Neben den 8 bzw. 16 Arbeitsregistern gibt es noch einen Befehlszähler und ein 8-bit-breites Flagregister. Der Befehlszähler, im Englischen *programmcounter* (PC) genannt, zeigt auf den nächsten auszuführenden Befehl im Speicher. Das Flagregister wird im folgenden genauer betrachtet.

4.3.1.2 Condition Code Register

Das Flagregister, Condition Code Register (CCR) genannt, darf nicht als Ganzes gesehen, sondern muss bitweise betrachtet werden (siehe Abbildung 4.9(b)). Jedes Bit in diesem Register steht für eine bestimmte Information, die sich meist auf das Ergebnis des letzten Befehls bezieht. So wird das *Zero*-Flag gesetzt, wenn das Ergebnis einer arithmetischen oder einer Schiebeoperation Null ist. Das *Negative*-Flag zeigt an, ob das Vorzeichenbit des Ergebnisses gesetzt ist, ob es also negativ oder positiv zu interpretieren ist. Darüber hinaus gibt es noch das *Carry*-, das *Half-carry*- und das *Overflow*-Flag, welche hauptsächlich durch arithmetische Operationen verändert werden.

Die Carry-Flags werden zum Beispiel gesetzt, wenn eine Addition einen Übertrag auf eine höhere Stelle erzeugt oder wenn eine Subtraktion sich von dort etwas borgen muss. Je nach Größe der Arbeitsregister beziehen sich die Flags auf unterschiedliche Stellen. Arbeitet man auf einem 8-bit Register, so bezeichnet das *Carry*-Flag einen Übertrag des 8. Bits, das *Half-carry*-Flag des 4., bei einer 16-bit Operation wird ein Übertrag des 16. und des 8. Bits in den Flags vermerkt.

Das *Overflow*-Flag zeigt einen Überlauf bei einer Operation an. Das bedeutet, dass das Ergebnis auf Grund mangelnden Platzes nicht mit der theoretisch richtigen Lösung übereinstimmt. Dieser Fall tritt immer dann auf, wenn der gültige Zahlenbereich einer 8- bzw. 16-bit Zahl überschritten wird. Wenn zum Beispiel von der kleinsten darstellbaren 8-bit Zahl $(1000\ 0000)_2 = (-127)_{10}$ die Konstante 1 abgezogen wird, so entspricht der Wert im Ergebnisregister der positiven Zahl $(0111\ 1111)_2 = (+127)_{10}$. Das *Overflow*-Flag weist jedoch auf den Fehler hin.

Eine spezielle Stellung nimmt das *Interrupt-Mask* Flag ein, da es nicht durch arithmetische Operationen verändert wird, sondern nur durch explizite Befehle gesetzt oder gelöscht werden kann. Es beeinflusst die Interrupt-Behandlung und wird im Abschnitt 4.3.3 behandelt.

Da das Flagregister durch die bisher beschriebenen Bits noch nicht 8-bit breit ist, sind noch zwei *User*-Flags vorhanden, die wie das *Interrupt-Mask* Bit nur durch explizite Befehle verändert werden. Allerdings stehen sie einem Programm zur freien Verfügung und haben keinen Einfluss auf andere Teile des Mikrocontrollers.

4.3.1.3 fetch-decode-execute Zyklus

Die grundsätzliche Funktionsweise der H8/300 CPU gleicht der eines jeden anderen Prozessors [PH96][Kla89]. Zuerst wird der zu bearbeitende Befehl aus dem Speicher geholt. Woher der neue Befehl stammt, wird über den Befehlszähler gesteuert. Er enthält die Adresse der Speicherstelle, in der der nächste Befehl beginnt. Wurde der Befehl aus dem Speicher in das Rechenwerk geladen, so kann der Befehlszähler auf den nächsten Befehl gesetzt werden. Bei der H8/3297-Familie ist ein Befehl entweder zwei oder vier Byte lang.

Ist das Holen des Befehls abgeschlossen, so erfolgt die Dekodierung des neuen Kommandos. In dieser Phase wird festgestellt, um welchen Befehl es sich handelt und die Operanden werden dem Rechenwerk zur Verfügung gestellt.

Nach der Dekodierung erfolgt die Ausführung des Befehls. In dieser letzten Phase wird der Befehl bearbeitet und der Speicher sowie die Register und Flags entsprechend der Operation verändert. Bei Sprungbefehlen wird auch der Befehlszähler verändert. Wenn diese Phase abgeschlossen ist, kann mit einem erneuten Zyklus fortgefahren werden.

Die Dauer eines Zykluses ist befehlsabhängig. Die schnellsten Befehle der H8/300 CPU, z.B. `addx` oder `xor`, sind nach nur 2 Takten abgeschlossen, wohingegen die beiden langsamsten, `mulxu` und `divxu`, 12 Takte benötigen. Die Dauer hängt aber nicht nur von der Komplexität des Befehls an sich ab, sondern auch von der Adressierungsart der Operanden.

4.3.1.4 Adressierungsarten

Die meisten Kommandos des H8/300 sind 2-Adress-Befehle, das heißt, dass zwei Parameter mitgegeben werden, die die Orte der Operanden angeben. Oft spezifiziert dabei der zweite Parameter ein allgemeines Register, in das nach Ausführung auch das Ergebnis abgelegt wird. Manche Befehle benötigen auch nur einen Operanden, zum Beispiel `push` oder gar keinen wie beispielsweise `sleep`.

Der Prozessor des Mikrocontrollers kennt 9 verschiedene Arten, ein Objekt anzusprechen. In Tabelle 4.1 findet sich eine Zusammenfassung dieser Adressierungsarten mit je einem Beispiel. Bei der Erklärung der Beispiele entspricht der Pfeil (←) einer Zuweisung. Der Speicher wird durch ein Array Namens Mem und die Register durch ein Array Namens Regs dargestellt. Der Zugriff auf die Arrays ist in C ähnlicher Notation dargestellt. Mem[Regs[r2]] entspricht somit dem Wert an einer Speicherstelle, deren Adresse durch den Inhalt des Registers `r2` gegeben ist.

Adressierungsart	**Beispiel**	**Bedeutung**
immediate	`mov #3,r3`	Regs[r3]←3
register direct	`mov r2,r3`	Regs[r3]←Regs[r2]
register indirect	`mov @r2,r3`	Regs[r3]←Mem[Regs[r2]]
register indirect with displacement	`mov @(100, r2),r3`	Regs[r3]←Mem[100+Regs[r2]]
register indirect with post-increment	`mov @r7+,r5`	Regs[r5]←Mem[Regs[r7]] Regs[r7]←Regs[r7] + 2
register indirect with pre-decrement	`mov r5,@-r7`	Regs[r7]←Regs[r7] - 2 Mem[Regs[r7]]←Regs[r5]
absolut	`mov @1001,r3`	Regs[r3]←Mem[1001]
memory indirect	`jmp @@1001`	Regs[PC]←Mem[Mem[1001]]
PC-relativ	`bra 10`	Regs[PC]←Regs[PC]+10

Tabelle 4.1: Adressierungsarten der H8/300 CPU

Eine genaue Beschreibung der einzelnen Arten ist zum Beispiel in [PH96] oder in [Hita] nachzulesen. Zu Beachten ist noch, dass nicht jeder Befehl in jeder Adressierungsform existiert.

4.3.1.5 Befehlscodierung

Bevor auf die Simulation eingegangen werden kann, soll zuerst noch betrachtet werden, wie ein Befehl im Speicher abgelegt ist. Jede Instruktion des H8/300 Prozessors belegt mindestens zwei Byte Speicherplatz, manche sogar vier. In diesen Daten sind die gewünschte Operation und deren Parameter codiert. Im ersten Byte steht meist eine eindeutige Nummer, welche die Operation spezifiziert. Bietet ein Befehl die Möglichkeit an, die Operanden in verschiedenen Adressierungsarten anzugeben, so hat jede dieser Varianten eine eigene Nummer. Die folgenden Bytes enthalten die Parameter der Operation.

Der Befehl `sub r3h, r4l` wird beispielsweise durch den Wert 0x182C dargestellt. Die Operation wird hierbei durch den Wert 0x18 im ersten Byte codiert. Würde `sub` mehrere Adressierungsarten anbieten, so hätte jede Art einen anderen Wert. Als nächstes folgen die Parameter. `sub` erwartet zwei 8-bit Register.

Zur Beschreibung der acht allgemeinen 16-bit Register, dient ein 3-bit Wert. Um einen 8-bit Anteil dieser Register zu spezifizieren, wird ein weiteres Bit verwendet. 1 bedeutet dabei, dass das Low-Byte verwendet werden soll, 0 steht für das High-Byte. Der Wert 0x2C muss daher aufgetrennt werden in $(2)_{16} = (0\ 010)_2$ für den ersten Operanden, `r2h`, und $(C)_{16} = (1\ 100)_2$ für den zweiten, `r4l`.

4.3.2 Implementierung

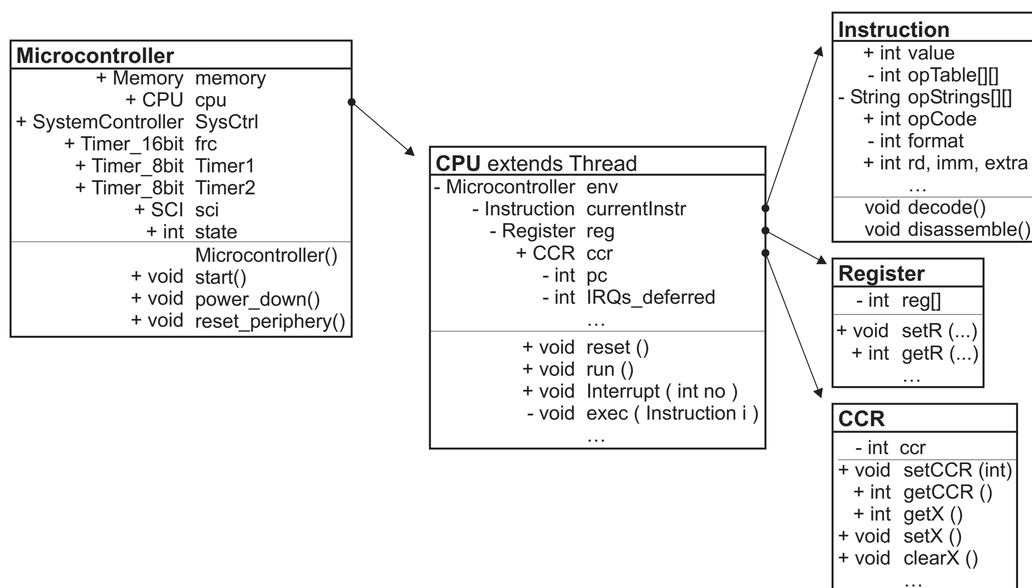


Abbildung 4.10: Klassendiagramm der Klasse CPU sowie einiger Hilfsklassen

Die Simulation des Prozessors wird durch die Klasse CPU durchgeführt. Zur Dekodierung eines Befehls dient eine Instanz der Klasse Instruction. Zur Abstraktion der Register werden Objekte der Klassen Register und CCR verwendet (siehe Abbildung 4.10).

Die Klasse CPU wird als Thread ausgeführt und enthält in ihrer Methode run die Hauptschleife des Simulators. Diese Schleife führt nicht nur den fetch-decode-execute Zyklus durch, sondern

enthält auch Aufrufe zu allen Modulen des Simulators, die periodisch Aufmerksamkeit benötigen, wie zum Beispiel die Zähler. Auf Vor- und Nachteile, die aus diesem Design entstehen, wird später, zum Beispiel im Rahmen der Ausnahme-Behandlung noch eingegangen. Abbildung 4.11 zeigt ein Flussdiagramm der Hauptschleife. Für die Simulation der CPU ist nur der fetch-decode-execute Zyklus relevant, die anderen Elemente der Abbildung werden in späteren Abschnitten noch erklärt.

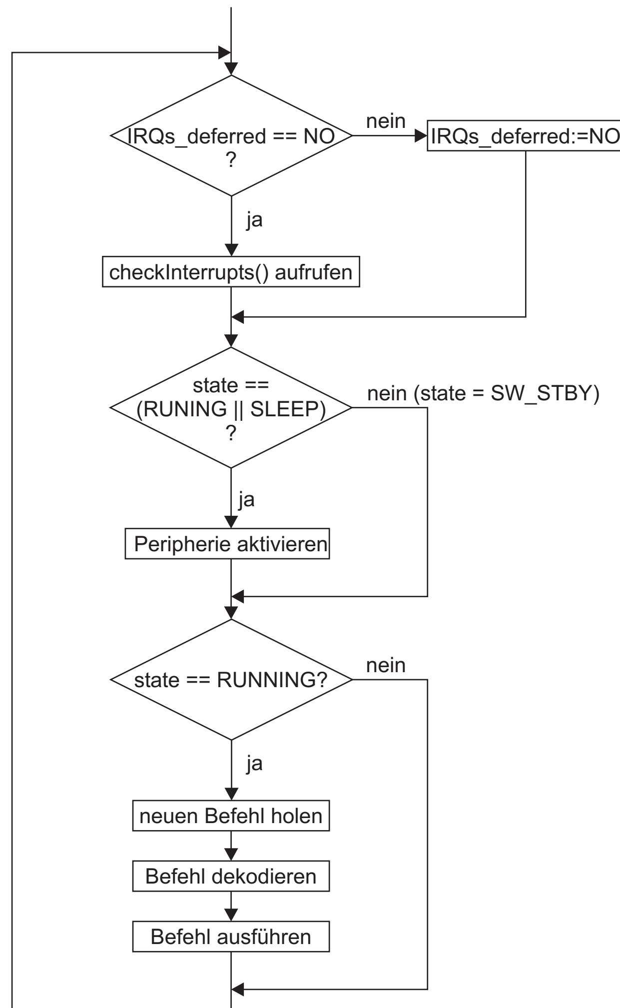


Abbildung 4.11: Die Hauptschleife des RCX Simulators.

4.3.2.1 Befehlshol-Phase

Jeder Zyklus beginnt mit dem Holen des nächsten Befehls. Dies wird durch einen Zugriff auf den simulierten Speicher erledigt. Die Adresse ist dabei durch den Befehlszähler festgelegt. Für den Speicherzugriff stehen die in Abschnitt 4.2.2 beschriebenen Funktionen zur Verfügung. Da die Länge der Befehle des H8 Mikrocontrollers nicht konstant ist, werden immer 4 Byte aus dem Speicher gelesen und zum Dekodieren an das Instruction-Objekt übergeben. Somit ist sichergestellt, dass alle Befehle vollständig erfasst werden, da sie maximal 4 Byte belegen. Als

Konsequenz werden jedoch bei einem 2-Byte-Befehl die zu viel gelesenen Byte im nächsten Zyklus erneut vom Speicher angefordert.

4.3.2.2 Dekodierungs-Phase

Zur Dekodierung des 4-Byte großen Wertes ist die Methode decode des Instruction-Objekts vorgesehen. Die Dekodierungs-Phase wird dort in zwei Schritten durchgeführt. Im ersten Schritt wird der Befehl inklusive dessen Adressierungsart ermittelt, im zweiten Schritt werden die Parameter extrahiert. Das Ergebnis ist eine Operationsnummer, die für jede Befehlsvariante eindeutig ist, außerdem werden die Argumente in separaten Variablen abgelegt.

Zur Durchführung der ersten Aufgabe wird meist nur das erste Byte der Instruktion benötigt, da es, wie schon beschrieben, den Befehl charakterisiert. Zur Unterscheidung der Befehle steht eine Tabelle in Form eines Arrays zur Verfügung. Jeder Eintrag in diesem Array enthält zwei Informationen, zum einen eine interne Befehlsnummer, die alle Befehle eindeutig identifizierbar machen soll, zum anderen die Adressierungsart dieses Befehls, die zur Extraktion der Parameter benötigt wird.

Eine Indizierung des Arrays mit dem ersten Byte der Instruktion führt im Idealfall zu einem Eintrag, der die eindeutige Nummer und die Adressierungsart des Befehls enthält. Abbildung 4.12 stellt das Vorgehen anhand eines Beispiels dar.

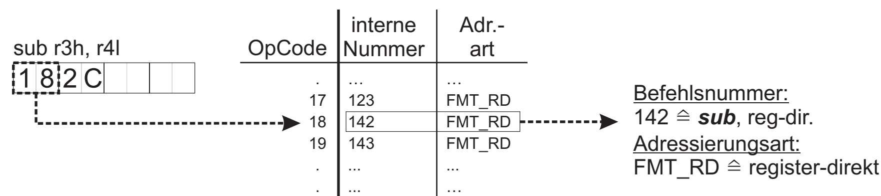


Abbildung 4.12: Dekodierung eines Befehls

Bei einigen Instruktionen ist das erste Byte allerdings nicht aussagekräftig genug, um den Befehl eindeutig von allen anderen zu trennen. Diese Sonderfälle sind im Array durch spezielle Einträge markiert und werden später genauer beschrieben.

Die zweite Phase der Dekodierung soll hauptsächlich die Parameter extrahieren. Dies geschieht unter Zuhilfenahme der Adressierungsart, die im ersten Schritt für den Befehl ermittelt wurde. Die Parameter sind bei allen Befehlen einer Adressierungsart in derselben Weise codiert, das heißt sie befinden sich an der gleichen Position innerhalb des Instruktionscodes. Zur Extraktion wird eine switch-Anweisung durchlaufen, die nach der zuvor ermittelten Adressart trennt. In den einzelnen Abschnitten werden die Parameter entsprechend der Adressierungsart aus dem Instruktionscode in Variablen der Klasse kopiert, um sie zur Ausführung bereitzustellen.

Sonderfälle

Mache Befehle wurden im ersten Schritt noch nicht eindeutig voneinander getrennt, da das erste Byte zweier verschiedenen Befehle gleich ist. Dies ist zum Beispiel bei Bitoperationen, die mit register-direkter Adressierung arbeiten der Fall. Bei diesen Befehlen muss zum ersten Byte noch das 9. Bit der Instruktion, also das höchstwertige Bit des zweiten Bytes, betrachtet werden, um eine eindeutige Trennung zu erreichen. Der erste Schritt, die Indizierung in die

Array-Tabelle, liefert somit für zwei dieser Befehle immer denselben Eintrag zurück. Die Informationen in diesem Eintrag können natürlich nicht für beide Befehle korrekt sein, deshalb müssen sie entsprechend des Befehls korrigiert werden.

Um das bewerkstelligen zu können muss zuerst erkannt werden, dass ein solches Problem vorliegt. Dazu ist im Adressierungsart-Feld aller betroffenen Einträge eine spezielle Nummer eingetragen. Die switch-Anweisung, welche die Einträge nach Adressierungsart getrennt bearbeitet, kann somit diesem Sonderfall einen eigenen Abschnitt zuordnen. In diesem Abschnitt wird dann das 9. Bit untersucht und der ermittelte Eintrag unter Umständen korrigiert.

Ist das zusätzliche Bit gelöscht, so ist keine Korrektur erforderlich. Die Befehlsnummer des ermittelten Eintrags stimmt für diesen Befehl. Die Instruktion kann wie ein normaler Befehl, mit register-direkter Adressierungsart, dekodiert werden.

Ist das zusätzlich betrachtete Bit jedoch gesetzt, so stimmt die Befehlsnummer, die der gefundene Eintrag liefert, nicht. Um eine effiziente Korrektur zu ermöglichen sind die internen Befehlsnummern so organisiert, dass man durch Addition von 3 zu der ermittelten Befehlsnummer die für diesen Befehl korrekte Nummer erhält. Die Extraktion der Parameter erfolgt dann analog wie vorher. Zur Verdeutlichung ist dieser Sonderfall an einem Beispiel in Abbildung 4.13 dargestellt.

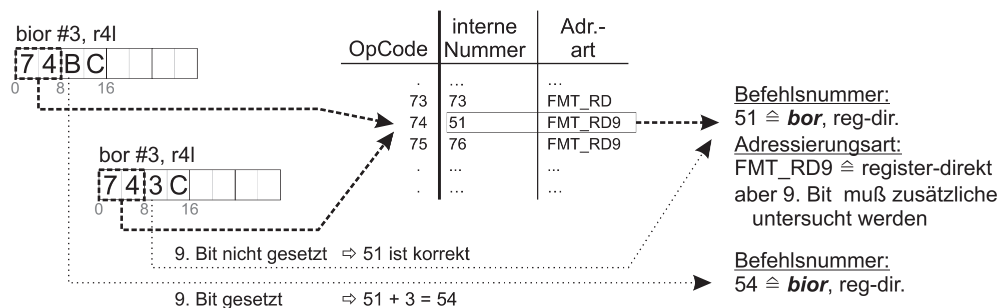


Abbildung 4.13: Trennung von Bitoperationen bei register-direkter Adressierung

Neben dem gerade erwähnten Sonderfall existiert noch ein weiterer, in dem die eindeutige Identifizierung mithilfe der Array-Tabelle versagt. Es handelt sich dabei um die absolute und register-indirekte Adressierungsform einiger Bitoperationen. Bei diesen vier-Byte-langen Befehlen trennt das erste Byte nur zwischen den Adressierungsarten. Das heißt, dass alle diese Befehle mit der gleichen Adressierungsart durch die Array-Tabelle nur auf wenige Einträge abgebildet werden, jedoch werden nie zwei Befehle verschiedener Adressierungsart den gleichen Eintrag liefern.

Die Indizierung der Array-Tabelle, zu Beginn der `decode`-Methode, liefert für diesen Spezialfall demzufolge einen Eintrag, der zwar die korrekte Adressierungsform enthält, die Befehlsnummer kann jedoch durch die Tabelle nicht ermittelt werden. Um diesen Missstand zu markieren, ist die Befehlsnummer in den betroffenen Einträgen auf einen speziellen Wert[4] gesetzt.

Trifft die switch-Anweisung zur Extraktion der Parameter auf eine der beteiligten Adressierungsarten, so wird die Befehlsnummer überprüft. Falls es sich um den beschriebenen Wert handelt, so muss die korrekte Nummer noch ermittelt werden. Dazu wird das dritte Byte der

[4]Für die Implementierung wurde dieser Wert willkürlich auf 26 festgelegt (vgl. auch Abbildung 4.14)

Instruktion untersucht, da dort bei diesen Befehlen der eigentliche Befehlscode steht. Der Code ist dabei identisch mit dem des jeweils gleichen Befehls bei register-direkter Adressierung.

So ist beispielsweise der Befehl `bset #3, @r5` in register-indirekter Adressierung durch den Code 0x7D 50 **70** 30 im Speicher abgelegt. Der gleiche Befehl bei register-direkter Adressierung (`bset #3, r5l`) wird durch 0x**70** 3D dargestellt. Die Bedeutung der beiden Instruktionen unterscheidet sich zwar, der Befehl selbst wird jedoch immer durch die $(70)_{16}$ dargestellt.

Zur Ermittlung der internen Befehlnummer wird deshalb die Array-Tabelle erneut indiziert, diesmal jedoch mit dem dritten Byte der Instruktion, statt mit dem ersten. Die so erhaltene Befehlsnummer muss dann noch entsprechend der Adressierungsart korrigiert werden. Dazu ist die Anordnung der internen Befehlsnummern wieder speziell ausgelegt worden, sodass man durch Addition von 1 bzw. 2 zu der Befehlsnummer, die entsprechende Nummer einer anderen Adressierungsart erhält. Abbildung 4.14 stellt das Vorgehen an einem Beispiel dar.

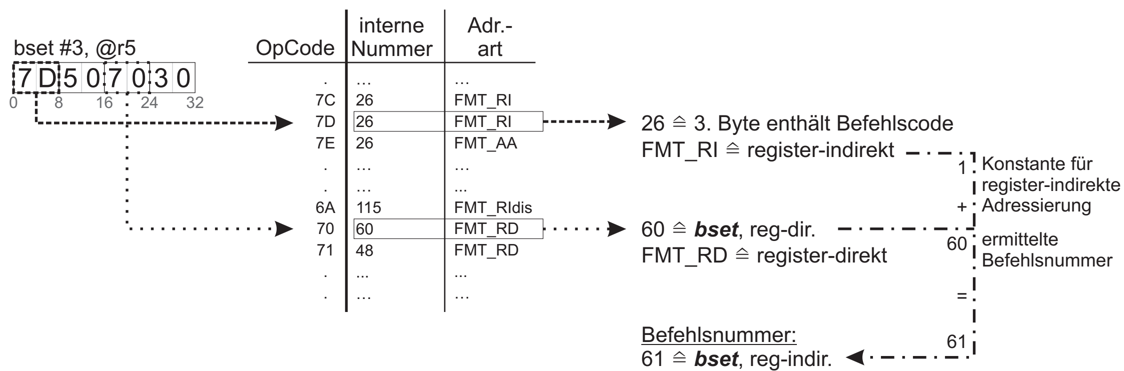


Abbildung 4.14: Trennung von Bitoperationen bei register-indirekter Adressierung

Damit wäre die Ermittlung der Befehlsnummer abgeschlossen, meistens tritt jedoch der oben beschriebene erste Sonderfall auf: Die zweite Array Indizierung bildet wieder zwei Befehle auf denselben Eintrag ab. Zur Lösung wird ein ähnlicher Weg wie oben beschritten. Da diesesmal jedoch nicht das erste Byte, sondern das dritte zur Ermittlung des Tabellen-Eintrags diente, muss nicht das 9. sondern das 23. Bit der Instruktion für weitere Untersuchungen verwendet werden.

Ist die Dekodierungsphase abgeschlossen, kehrt die decode-Methode mit einer eindeutigen Operationsnummer und den extrahierten Parametern zurück.

4.3.2.3 Ausführungs-Phase

Die Hauptschleife in der Klasse CPU wird daraufhin die klassen-eigene execute-Methode aufrufen. Diese soll den Befehl ausführen und alle nötigen Register entsprechend verändern. Die allgemeinen Register werden durch die Klassen Register, das Flagregister durch die Klasse CCR dargestellt. Kurze Beschreibungen dieser Klassen befinden sich in den anschließenden Abschnitten.

Zur Ausführung der Instruktion wird wieder eine switch-Anweisung durchlaufen, die für jeden Befehl den entsprechenden Java-Code bereithält. Alle diese Abschnitte sind in etwa gleich

aufgebaut. Zuerst werden, mit Hilfe der dekodierten Parameter die Operanden bestimmt, anschließend wird die Operation ausgeführt und deren Ergebnis schließlich gespeichert.

Bei arithmetischen Funktionen sind oft zusätzlich zum Ergebnis noch Informationen im Flagregister einzutragen. Bei einer Berechnung in Hardware fallen diese quasi als Nebenprodukte mit an. Im Java-Code jedoch müssen die Informationen, wie zum Beispiel die Carry-Flags, erst erzeugt werden. Damit der Code trotzdem übersichtlich bleibt, werden solche Arbeiten nicht in der switch-Anweisung erledigt, sondern sind in Hilfsfunktionen ausgelagert.

Am Ende der execute-Methode muss dann noch der Befehlszähler aktualisiert werden. Damit ist auch die letzte Phase abgeschlossen und die Kontrolle wird wieder an die Hauptschleife zurückgegeben, die einen neuen Zyklus starten wird.

4.3.2.4 Register-Klasse

Für den Zugriff auf die acht allgemeinen Register steht ein Objekt der Klasse Register mit den Methoden setR und getR zur Verfügung (siehe Abbildung 4.10). Diese Methoden erlauben es ein 16-bit Register zu bearbeiten oder nur einen Teil als 8-bit Wert anzusprechen.

Intern wird jedes Register durch eine Integer Variable dargestellt. 16-bit Zugriffe werden direkt auf der entsprechenden Variable durchgeführt. Für 8-bit Aktionen muss hingegen ein Byte der Variable extrahiert werden, da nicht der gesamte Wert verändert bzw. zurückgegeben werden darf.

4.3.2.5 CCR-Klasse

Für die Interaktion mit dem Flagregister ist die Klasse CCR vorgesehen. Für jedes der acht Bit werden von ihr drei Funktionen zur Verfügung gestellt. Diese sind: getX, um ein Bit abzufragen, setX und clearX, um es zu verändern. X steht dabei für ein Kürzel des jeweiligen Bits, so zum Beispiel Z für das Zero-Flag (siehe Abbildung 4.9(b)). Im Befehlssatz des H8/300 gibt es allerdings auch Befehle um das gesamte Flagregister zu bearbeiten (`ldc` und `stc`). Für diese Befehle wurden die Methoden getCCR und setCCR implementiert, die nicht ein einzelnes Bit, sondern das gesamte Register bearbeiten.

Die interne Darstellung des Flagregisters erfolgt durch eine Integer Variable. Als Folge davon müssen die meisten Methoden der Klasse ein Bit aus dieser Variable extrahieren, da sie nur ein Bit des Registers bearbeiten. Alternativ hätte man auch jedes Bit durch eine boolesche Variable darstellen können. Dadurch würden die meisten Methoden auf Zuweisungen reduziert. Jedoch müssten die Methoden getCCR und setCCR eine komplettes Flagregister zusammensetzen.

4.3.3 Ausnahme-Behandlung

4.3.3.1 Überblick

Zu den Aufgaben eines Prozessors gehört auch die Ausnahme-Behandlung (engl. Exception-Handling). Unter diesem Begriff versteht man im Allgemeinen die Reaktion auf Ausnahmesituationen. Beim Hitachi Mikrocontroller ist damit vor allem die Bearbeitung eines Interrupts

gemeint. Interrupts werden im Allgemeinen erzeugt, um den Prozessor über etwas zu informieren, auf das unter Umständen sofort reagiert werden muss. Sie können zum Beispiel durch Peripheriegeräte generiert werden, welche die Aufmerksamkeit der CPU benötigen. Das kann beispielsweise die serielle Schnittstelle sein, die Daten empfangen hat, die nun von der CPU abgeholt werden müssen.

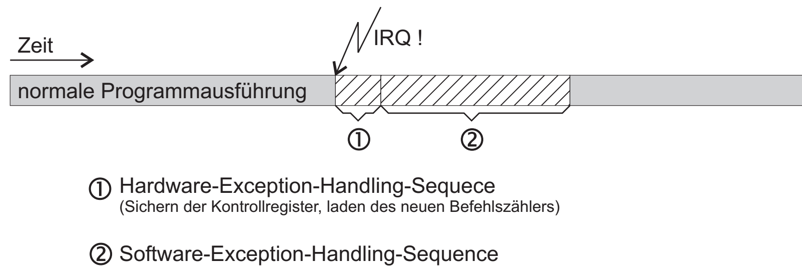


Abbildung 4.15: Reaktion auf einen Interrupt

Wird der CPU ein Interrupt mitgeteilt, so wird der sequentielle Programmfluss unterbrochen und die so genannte Hardware-Exception-Handling-Sequenz ausgeführt, die den aktuellen Zustand sichert und eine Software Behandlungsroutine aufruft (vgl. Abbildung 4.15). In dieser kann nun auf den Grund der Unterbrechung eingegangen werden. Im vorherigen Beispiel könnte hier der empfangene Wert aus dem Register der Schnittstelle in einen Speicherbereich im RAM kopiert werden. Ist die Interruptroutine abgeschlossen, wird das ursprüngliche Programm an der Stelle fortgesetzt, an der es unterbrochen wurde.

Nr.	**Quelle**	**Name**	**Beschreibung**
3	extern	NMI	nicht abschaltbarer externer IRQ
4		IRQ0	externe Interruptleitung (zur “Run” Taste)
5		IRQ1	externe Interruptleitung (zur “On-Off” Taste)
6		IRQ2	externe Interruptleitung
12	16-bit Zähler	ICIA (Input capture A)	Zählerstand wurde nach ICRA gesichert
13		ICIB (Input capture B)	Zählerstand wurde nach ICRB gesichert
14		ICIC (Input capture C)	Zählerstand wurde nach ICRC gesichert
15		ICID (Input capture D)	Zählerstand wurde nach ICRD gesichert
16		OCIA (Output compare A)	Zählerstand stimmt mit OCRB überein
17		OCIB (Output compare B)	Zählerstand stimmt mit OCRB überein
18		FOVI (Overflow)	Überlauf
19	8-bit Zähler 0	CMI0A (Compare-match A)	Zählerstand stimmt mit TCRA überein
20		CMI0B (Compare-match B)	Zählerstand stimmt mit TCRB überein
21		OVI0 (Overflow)	Überlauf
22	8-bit Zähler 1	CMI1A (Compare-match A)	Zählerstand stimmt mit TCRA überein
23		CMI1B (Compare-match B)	Zählerstand stimmt mit TCRB überein
24		OVI1 (Overflow)	Überlauf
27	SCI	ERI (Receive error)	Fehler beim Empfang
28		RXI (Receive end)	Empfang beendet
29		TXI (TDR empty)	Daten werden abgeschickt
30		TEI (TSR empty)	Datenfluss unterbrochen
35	ADC	ADI (Conversion end)	A/D Konvertierung abgeschlossen
36	WDT	WOVF (WDT overflow)	Überlauf des Watchdog Timers

Tabelle 4.2: Interruptgründe des Hitachi Mikrocontrollers
(ICR = Input Capture Register; OCR = Output Compare Register; TCR = Timer Compare Register)

Die Hitachi Mikrocontroller-Familie kennt 23 Gründe für Interrupts. 19 davon haben interne Ursachen, die anderen 4 werden durch externe Ereignisse ausgelöst. Intern heißt, dass der Interrupt von einem integrierten Modul, auf dem Mikrocontroller selbst, erzeugt wird.

Die vier externen Interrupts können hingegen durch Ereignisse ausgelöst werden, die außerhalb des Mikrocontrollers stattfinden. Dafür sind die Interrupt-Leitungen über Anschluss-Pins nach außen geleitet. Eine Pegelveränderung an einem dieser Pins löst den entsprechenden Interrupt aus. Im RCX ist zum Beispiel eine dieser externen Interrupt-Leitungen mit dem "On-Off-Knopf" gekoppelt. Wird dieser gedrückt, so wird der IRQ1 ausgelöst. Wie später im Abschnitt 4.3.4 über die Energiesparmodi noch beschrieben wird, dient dies der Möglichkeit den Kontrollbaustein RCX wieder zu aktivieren, nachdem er ausgeschaltet wurde.

Soweit die internen Interruptgründe von Bedeutung sind, wird in späteren Abschnitten noch genauer auf sie eingegangen. In Tabelle 4.2 sind alle Interrupts, auf die der Mikrocontroller reagieren kann zusammengefasst. Hier sollen sie jedoch nicht näher betrachtet werden.

4.3.3.2 Implementierung

Die Implementierung orientiert sich nahe an der Arbeitsweise der Hardware des Hitachi-Mikrocontrollers und weicht somit erheblich vom MIPS R3000 Simulator ab. Auf eine gesonderte Beschreibung der Ausnahme-Behandlung im realen Mikrocontroller soll an dieser Stelle verzichtet werden, da dessen Funktionsweise im folgenden Abschnitt größtenteils deutlich wird. Eine genaue Beschreibung kann man jedoch in [Hita] finden.

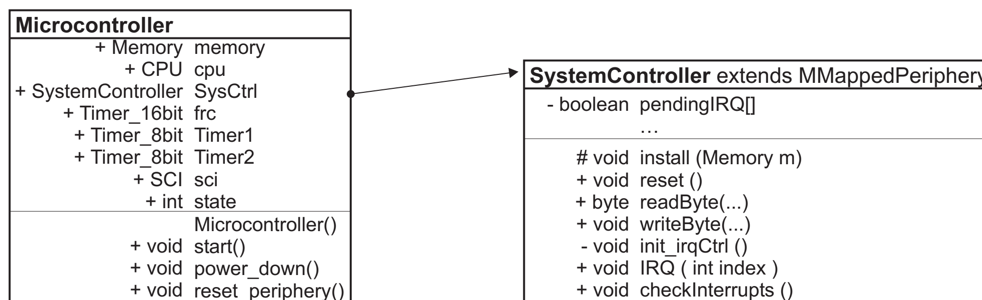


Abbildung 4.16: Die Klasse Systemcontroller

Soll ein Interrupt ausgelöst werden, so wird dem Interrupt-Controller ein Interrupt-Wunsch gemeldet. Im Simulator steht dazu die Methode IRQ bereit. Diese wird von der Klasse SystemController (siehe Abbildung 4.16) angeboten, in welcher der Interrupt-Controller realisiert ist.

Jeder Unterbrechungsgrund kann einzeln durch ein "Enable-Bit" ein- oder ausgeschaltet werden. Diese Bits befinden sich in den Registern der Module, die den entsprechenden Interrupt auslösen können. Da jedes Modul durch eine eigene Klasse simuliert wird, ist der Zugriff auf sie von einer zentralen Stelle nicht effektiv, zumal die Position der Bits keinem regelmäßigen Schema folgt. Aus diesem Grund prüft jedes Modul zuerst selbst sein entsprechendes Enable-Bit, bevor es die IRQ-Methode des Controllers aufruft. Nur die Enable-Bits der externen Interrupts werden von der Methode IRQ geprüft, da sich die entsprechenden Register in ihrem Zugriffsbereich befinden.

Ist der Interrupt aktiviert, so darf er an die CPU gemeldet werden. Der Interrupt-Wunsch wird dazu von der Methode IRQ in einem booleschen Array vorgemerkt.

In Flussdiagramm 4.17 ist die Aufteilung des Vorgangs in verschiedene Klassen bzw. Methoden dargestellt. Ein externer Interrupt-Wunsch kann dabei beispielsweise beim Drücken der "Run"-Taste, durch die Klasse RCXButton, erzeugt werden. Ein interner Interrupt kann von fast jedem integrierten Zusatzmodul ausgelöst werden (siehe auch Tabelle 4.2).

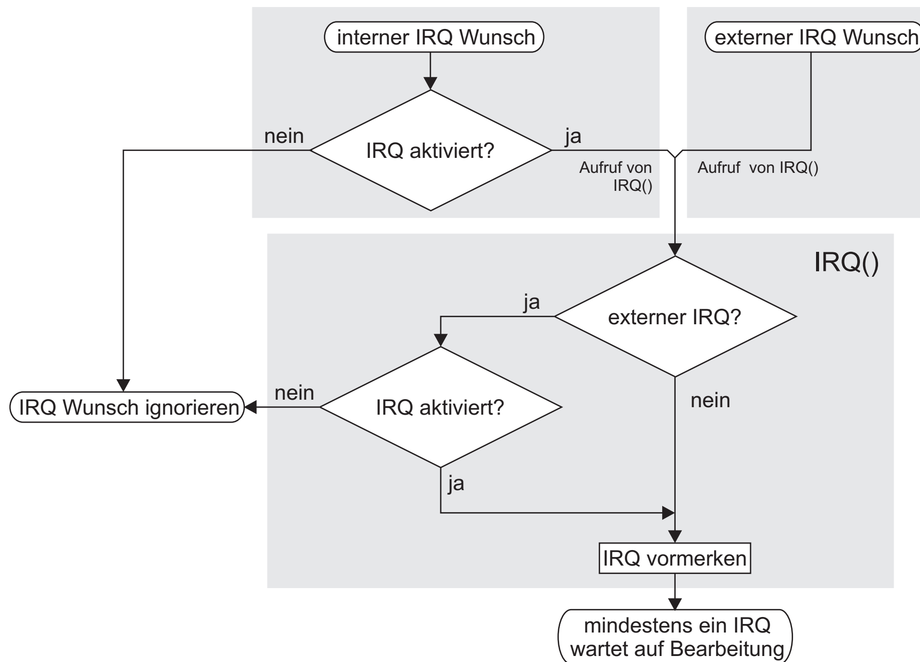


Abbildung 4.17: Generierung eines Interrupts

Der reale Interrupt-Controller würde nun, unter Berücksichtigung des *Interrupt-Mask*-Bits, den höchstprioren Interrupt-Wunsch der CPU melden. Der Prozessor würde den aktuellen Befehl noch beenden und anschließend eine Ausnahme-Behandlung durchführen. Im Simulator hingegen wird ein Interrupt nicht unaufgefordert an die CPU gemeldet. Stattdessen wird nach jedem Befehl das Vormerk-Array geprüft, ob ein Interrupt-Wunsch vorliegt. Flussdiagramm 4.18 zeigt das Vorgehen im Simulator.

Die Hauptschleife des Simulators ruft in jedem Durchgang die Methode checkInterrupts auf (siehe Flussdiagramm 4.11). Diese Methode befindet sich in der Klasse SystemController und prüft die vorliegenden Interrupt-Wünsche.

Sie prüft dazu zuerst das *Interrupt-Mask*-Bit im Flag-Register. Ist es gesetzt, so darf nur der *nonmaskable*-Interrupt (NMI) bearbeitet werden. Ist dieser nicht im Vormerk-Array eingetragen, so kehrt die Methode wieder zurück. Alle anderen Interrupts-Wünsche werden ignoriert und erst wieder bearbeitet, sobald das Maskierungs-Bit gelöscht ist.

Ist das Flag gelöscht, so wird das Vormerk-Array durchsucht. Falls dort ein Bit gesetzt ist, so wird die Methode Interrupt der Klasse CPU aufgerufen. Diese Funktion führt die Hardware-Exception-Handling-Sequenz durch, die die Ausführung einer Software-Behandlungsroutine vorbereitet. Als Parameter wird die Nummer des höchstprioren Interrupt-Wunsches mitgeteilt. Anschließend wird der Wunsch von der Vormerkliste gelöscht.

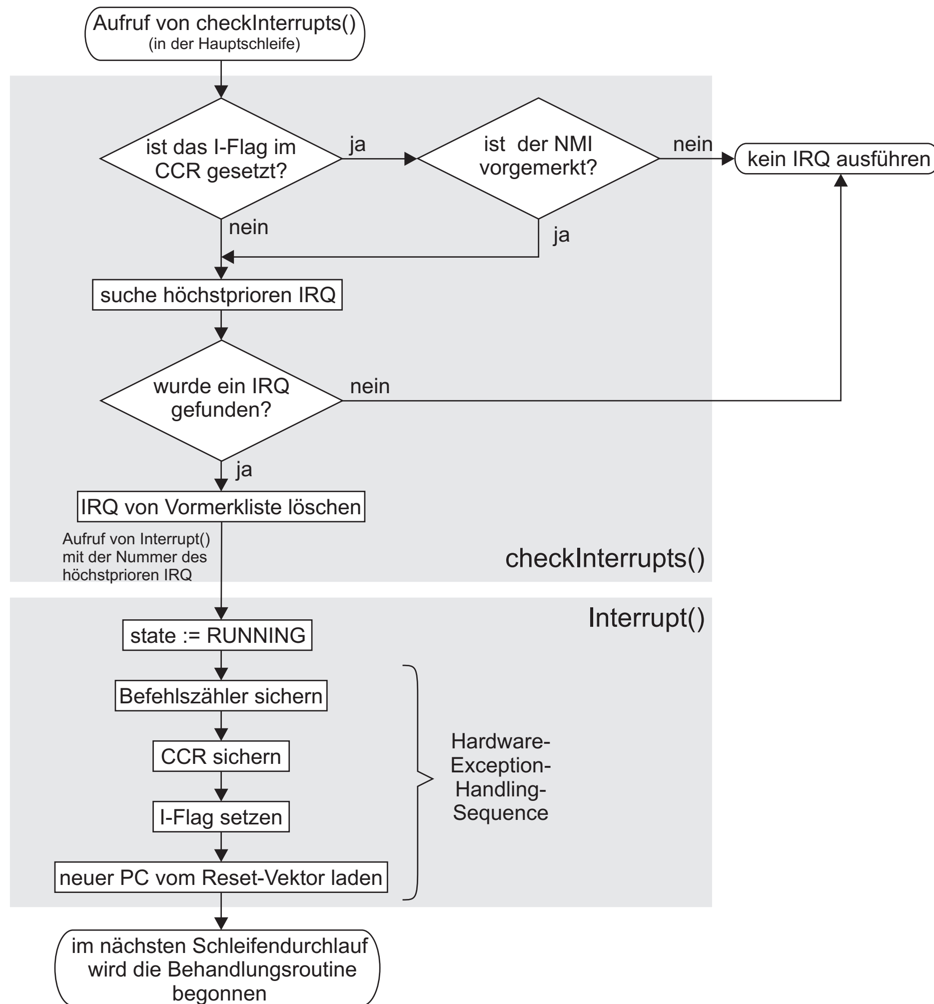


Abbildung 4.18: Bearbeitung eines Interrupt-Wunsches

Im Rahmen der Ausnahme-Behandlung werden zuerst der Befehlszähler und das Flagregister auf dem Stack gesichert. Dazu werden die schon bekannten Methoden writeShort, für den Speicherzugriff und getR bzw. setR zur Interaktion mit dem Stackregister verwendet. Dabei ist zu beachten, dass das Flagregister nur ein Byte groß ist, der Stack aber immer nur Worte aufnehmen sollte. Deshalb wird das Flagregister zweimal hintereinander auf dem Stack abgelegt.

Nachdem die Kontrollregister gesichert wurden, wird das *Interrupt-Mask*-Bit gesetzt. Damit soll verhindert werden, dass die folgende Behandlungsroutine durch einen Interrupt gestört wird. Ansonsten könnte es passieren, dass eine Behandlungsroutine, die sehr viel Zeit benötigt, durch ihren eigenen Interrupt immer wieder aufs neue unterbrochen wird, ohne dass sie jemals fertig wird. Dies hätte zur Folge, dass die Funktion mit jedem Interrupt neu aufgerufen wird. Nicht nur, dass der normale Programmfluss dann nie mehr aufgenommen wird, über kurz oder lang würde auch der Stack überlaufen, da dort bei jedem neuen Aufruf die Rücksprungadresse abgelegt wird.

Nachdem alle Vorbereitungen getroffen wurden, wird der Befehlszähler vom Interrupt-Vektor geladen. Der Interrupt-Vektor ist ein Speicherbereich, der wie ein Array organisiert ist, welches

für jeden Unterbrechungsgrund die Adresse enthält, an der die entsprechende Behandlungsroutine beginnt. Der Vektor ist unter der Adresse 0x6 bis 0x49 erreichbar, welche dem ROM des Mikrocontrollers zugeordnet sind (siehe Abbildung 4.3). In dem von Lego für das RCX entworfene ROM verweisen alle Werte des Vektors auf feste Speicherstellen im RAM. Der Code an diesen Stellen kann verändert werden, sodass die Reaktionen auf Unterbrechungen zum Beispiel durch die Firmware angepasst werden können. Um den neuen Befehlszähler zu laden genügt ein Speicherzugriff an die Stelle des Interrupt-Vektors, die durch die Interruptnummer bestimmt ist. Nachdem der neue Wert in die Variable, die den Befehlszähler repräsentiert, eingetragen wurde, kehrt die Methode Interrupt zurück.

Der nächste fetch-decode-execute Zyklus wird dann an der neuen Stelle mit der Befehlsausführung fortfahren, was dem Beginn der Software-Behandlungsroutine entspricht. Diese sollte in Normalfall zuerst alle Register, die sie verändern wird, sichern und am Ende wieder herstellen, sodass der normale Programmfluss nicht gestört wird. Beendet wird eine Ausnahme-Behandlungsfunktion durch den Maschinenbefehl `rte` (return from exception). Dieser sorgt dafür, dass der ursprüngliche Inhalt des Befehlszählers und des Flagregisters wieder hergestellt wird. Spätestens dadurch wird meist auch das *Interrupt-Mask*-Bit wieder gelöscht. Normalerweise war es vor der Unterbrechung nicht gesetzt, sonst hätte keine Interrupt-Behandlung stattfinden können.

Anforderungen an die Implementierung und Besonderheiten

Durch die oben beschriebene Implementierung werden alle Anforderungen an den zeitlichen Ablauf der Ausnahme-Behandlungs-Sequenz erfüllt. Zum einen soll der Prozessor erst seinen aktuellen Befehl beenden, bevor er sich der Ausnahmebehandlung widmet. Dies wird durch die Implementierung berücksichtigt, indem die Ankunft von Interrupt-Wünschen immer nur zwischen zwei Befehlen, in der Hauptschleife durch die Methode checkInterrupts, abgefragt wird. Ein Befehl kann deshalb niemals unterbrochen werden.

Die zweite, bisher noch nicht genannte, Anforderung ist, dass der beim Mikrocontroller in Hardware durchgeführte Teil der Ausnahmebehandlung, das Sichern der Kontrollregister und das Laden des neuen Befehlszählers, ununterbrechbar sein soll. Das heißt, dass während der Ausführung dieser Schritte kein neuer Interrupt bearbeitet werden darf. Beim Simulator wird dieser Teil der Ausnahme-Behandlung in der Methode Interrupt der Klasse CPU realisiert. Die Bearbeitung eines Interrupts kann nur durch die Methode checkInterrupts eingeleitet werden, welche wiederum nur in der Hauptschleife aufgerufen wird. Da zur Simulation nur ein Thread verwendet wird, kann dieser nicht gleichzeitig die Methode Interrupt und die Hauptschleife ausführen. Somit ist eine Unterbrechung der Hardware-Exception-Handling-Sequenz ausgeschlossen.

Ein weiterer Aspekt, der bei der Simulation beachtet wird, ist die Verzögerung eines Interrupts. So darf direkt nach der Resetsequenz und nach einigen speziellen Befehlen zur Veränderung des Flagregisters (`andc`, `orc`, `xorc` und `ldc`) kein Interrupt bearbeitet werden. Sinn und Zweck dieser Sonderfälle ist die Initialisierung des Stackzeigers nach einem Reset. Würde eine Ausnahme-Behandlungsroutine aufgerufen werden, bevor der Stack eingerichtet ist, könnte die Rücksprungadresse nicht gesichert werden. Das Betriebssystem soll deshalb nach einem Reset zuerst das Register `r7`, den Stackzeiger, initialisieren. Bevor dies geschieht, darf allerdings mit den oben genannten Befehlen das Flagregister verändert werden, um zum Beispiel die Interrupts ganz abzuschalten.

Die Simulation dieses Mechanismusses wird über die private Variable IRQs_deferred realisiert. Die genannten Befehle und die reset Methode setzen die Variable auf true. In der Hauptschleife verhindert diese Variable dann den Aufruf von checkInterrupts (siehe Flussdiagramm 4.11), sodass kein Interrupt gemeldet werden kann. Nach einem Schleifendurchlauf wird die Variable wieder auf false gesetzt, da die Interrupts nur einen Befehl lang ausgesetzt werden sollen.

4.3.4 Energiesparmodi

4.3.4.1 Überblick

Die H8/3297 Mikrocontroller-Familie hat drei verschiedene Energiesparmodi. Einer davon, der so genannte *Hardware Standby* Modus, ist nur "von außen" zu aktivieren indem an einem bestimmten Pin (STBY) ein Low-Pegel angelegt wird. Die anderen beiden, der *Sleep* Modus und der *Software Standby* Modus, lassen sich, wie der Name schon andeutet, durch einen `sleep` Befehl im Programm aktivieren. Ein Bit im *System Control* Register entscheidet dabei, in welchen der beiden Zustände gewechselt wird.

Der *Sleep* Modus ist der schwächste dieser drei Modi. Ein Wechsel in diesen Zustand findet statt, wenn die `sleep` Instruktion ausführt wird und dabei das *Software Standby* Bit im *System Control* Register nicht gesetzt ist. Alle integrierten Zusatzmodule bleiben von dieser Maßnahme unberührt, lediglich der Prozessor wird angehalten. Um den normalen Betrieb wieder aufzunehmen genügt ein beliebiger Interrupt. Benutzt wird dieser Modus zum Beispiel durch die Firmware legOS, deren "idle-Prozess" in diesen Zustand wechselt und vom nächsten Zähler-Interrupt wieder aufgeweckt wird.

Der *Software Standby* Modus schaltet nicht nur den Prozessor ab, sondern zusätzlich alle integrierte Peripherie inklusive des Taktgenerators. Um in diesen Zustand zu wechseln muss das *Software Standby* Bit im *System Control* Register gesetzt sein, wenn die `sleep` Instruktion ausgeführt wird. Verlassen wird dieser Zustand nur, wenn ein externer Interrupt auftritt, da interne Interrupts bei abgeschalteter Peripherie sowieso nicht mehr erzeugt werden können.

Anwendung findet dieser Modus immer dann, wenn die "On-Off" Taste am RCX gedrückt wird. Die im ROM des Mikrocontrollers gespeicherte Software bewirkt dann einen Wechsel in den *Software Standby* Modus. Um den Energieverbrauch noch mehr zu senken wird vorher auch das externe RAM in einen Energiesparmodus versetzt. Dies erfolgt durch einen Wechsel des Ausgangspegels an einem Pin des Port 5. Selbstverständlich bleibt der Inhalt des externen RAMs erhalten, während es sich im Energiesparmodus befindet. Das Reaktivieren des Lego-Bausteins erfolgt auch wieder über den "On-Off" Taster. Er ist mit einer externen Interruptleitung verbunden, sodass durch die Betätigung dieser Taste ein Interrupt ausgelöst wird und somit den normalen Betrieb wieder hergestellt werden kann.

Der *Hardware Standby* Modus hat das größte Energiesparpotential. Ist der Mikrocontroller in diesem Zustand, werden der Prozessor und alle Zusatzmodule inklusive der Ports abgeschaltet. Im Gegensatz zu den beiden vorherigen Modi geht hier sogar der Inhalt aller Prozessor-Register verloren, lediglich der Inhalt des internen RAMs bleibt erhalten. Das Verlassen dieses Zustandes ist nur durch eine Pegelveränderung am $STBY$-Pin oder durch das Auslösen eines Resets möglich. Da dieser Modus im Lego-Baustein nie verwendet wird, beschränkt sich die Simulation auf die beiden anderen Modi.

4.3.4.2 Implementierung

Trifft die execute-Methode der Klasse CPU auf das sleep Kommando, so wird die Methode power_down des Microcontroller-Objekts aufgerufen. Diese überprüft den Inhalt des *System Control* Registers und setzt die Variable state entweder auf ST_SLEEP oder auf ST_SW_STBY, je nachdem, ob das *Software Standby* Bit gelöscht ist oder nicht. Soll ein Wechsel in den *Software-Standby* Modus stattfinden, so wird zusätzlich die Methode reset_periphery aufgerufen, um die Zusatzmodule in den Startzustand zu versetzen.

Das Verändern der Variable state beeinflusst das Verhalten des Simulators. Wie in Flussdiagramm 4.11 zu sehen ist, steuert diese Variable die Aktionen in der Hauptschleife. Der fetch-decode-execute Zyklus wird zum Beispiel nur durchgeführt, wenn die Variable state auf ST_RUNNING steht. Ein Aufruf von power_down hat die Variable jedoch verändert, sodass die simulierte CPU jetzt gestoppt ist und keine Befehle mehr abarbeitet. Der *Sleep* Modus ist somit erreicht.

Soll in den *Software Standby* Modus gewechselt werden, so dürfen auch die Zusatzmodule nicht mehr arbeiten. Deren Aktivität ist dadurch gekennzeichnet, dass in jedem Schleifendurchlauf eine Funktion der entsprechenden Objekte aufgerufen wird. So zum Beispiel die Methode increment eines Zählers oder die Methode convert des A/D-Wandler Objekts. Werden diese Aufrufe unterdrückt, so sind die Zusatzmodule abgeschaltet. Genau das bewirkt die Zuweisung des Wertes ST_SW_STBY zu der Variable state.

Zurück in den normalen Betrieb führt jeder Aufruf der Methode Interrupt, da dadurch die Variable state immer wieder auf ST_RUNNING gesetzt wird (siehe Abbildung 4.18). Somit wird die normale Arbeit der Hauptschleife wieder hergestellt, wodurch alle Zusatzmodule wieder ihre regelmäßigen Aufrufe erhalten und auch der fetch-decode-execute Zyklus wieder ausgeführt wird.

Der Aufruf der Interrupt-Methode erfolgt, wie schon im Abschnitt 4.3.3 erklärt, nur durch die Methode checkInterrupts. Diese wird in jedem Schleifendurchgang aufgerufen, unabhängig vom Wert der Variable state, sodass auch während eines Energiesparmodus regelmäßig die Ankunft neuer Interrupts geprüft wird. Dass der *Software Standby* Modus nur durch externe Interrupts beendet werden kann, wird durch das Design auch berücksichtigt, da die gestoppten Zusatzmodule keine Interrupts auslösen können.

4.4 Ports

4.4.1 Überblick

Ein wichtiger Bestandteil aller Mikrocontroller sind die Ports. Mit dem Begriff Port bezeichnet man ein Gruppe von meist acht Ein-/Ausgangsleitungen. Über diese Schnittstellen kann der Mikrocontroller externe Prozesse überwachen und steuern, sofern entsprechende Sensoren und Aktoren angeschlossen sind. Oft sind die Schnittstellen nicht frei verwendbar, sondern von Zusatzmodulen wie beispielsweise dem A/D-Wandler belegt. Viele Mikrocontroller können zum Beispiel mithilfe interner Zähler an einem ihrer Ports Rechtecksignale erzeugen oder solche vermessen.

Der Hitachi H8/3292 Mikrocontroller bietet sieben Ports an. Bis auf Port 5 sind alle 8-bit breit. Wie schon erwähnt, werden die Verwendungsmöglichkeiten durch die Verknüpfung mit interner Peripherie bestimmt (vgl. Tabelle 4.3). Allerdings kann man die internen Geräte auch deaktivieren und somit die Schnittstellen frei nutzen.

Bei manchen Ports ist die Belegung auch vom Betriebsmodus des Mikrocontrollers abhängig. So sind die Ports 1 bis 3 als Erweiterung des Speicherbusses vorgesehen, wenn der Mikrocontroller in einem seiner beiden „expanded" Modi betrieben wird. Die verschiedenen Betriebsarten wurden im Abschnitt 3.2.2.2 kurz erläutert. Port 1 und 2, die je acht Leitungen zur Ein- oder Ausgabe von binären Signalen bereitstellen, sind dabei zur Ausgabe der Adresse gedacht. Port 1 beschreibt die unteren, Port 2 die oberen acht Bit des Adressbusses. Der Datenbus wird über den Port 3 nach außen geleitet. Da dieser nur 8-bit breit ist, werden 16-bit Zugriffe auf externe Geräte immer durch zwei 8-bit Zugriffe ausgeführt. Außerdem werden vier der acht Ein-/Ausgang-Pins des Port 4 für Steuerleitungen des Busses verwendet. Drei weitere Leitungen von Port 4 sind dem Interrupt-Controller zugeordnet, denn über diese Pins können externe Quellen einen Interrupt auslösen.

Port 5 ist die einzige Schnittstelle, die nur 3 Ein-/Ausgabeleitungen bereitstellt. Alle seine Pins werden durch die serielle Schnittstelle beansprucht, falls diese aktiviert ist. Die Pins des 8-bit breiten Port 6 werden von den Zählern benutzt. Sind diese aktiviert, ist es möglich Rechtecksignale zu erzeugen, ihre Funktionsweise wird in Abschnitt 4.6.1 erklärt. Schließlich gibt es noch einen 7. Port, welcher als einziger nur als Eingang zu verwenden ist. Die Eingabepins dieses Ports dienen auch dem A/D-Wandler als Eingänge.

Port	**Belegung**	**Port**	**Belegung**
1	Adressbus (low)	5	serielle Schnittstelle
2	Adressbus (high)	6	Zähler
3	Datenbus	7	A/D-Wandler
4	Steuer- und Interruptleitungen		

Tabelle 4.3: Verwendung der Ports durch interne Module

Bevor auf die Implementierung eingegangen wird, muss zuerst noch die Ansteuerung der Ports erklärt werden. Jeder Port hat ein Datenregister (DR) und bis auf Port 7 auch noch ein Richtungsregister, "data direction register" (DDR) genannt. Jedem Pin eines Ports ist je ein Bit eines Registers zugeordnet. Die Bits im Richtungsregister geben für jeden Pin an, ob er als Aus- oder als Eingang genutzt werden soll. Bei Port 7 fehlt dieses Register, da alle Pins nur als Eingang verwendet werden können. Die Bits des Datenregisters haben je nach Inhalt des Richtungsregisters verschiedene Funktionen. Ist ein Pin als Eingang konfiguriert, so zeigt das zugehörige Bit im Datenregister den logischen Pegel an diesem Pin an. Für einen Ausgangspin steuert das entsprechende Bit den Pegel, der dort erzeugt wird.

Port 1 bis 3 haben auch noch ein weiteres Register zur Steuerung ihrer pull-up-Transistoren. Für die Simulation ist das jedoch nicht von Bedeutung und wird nur zur Vollständigkeit erwähnt.

4.4.2 Implementierung

Jeder Port ist in einer eigenen Klasse implementiert. Um die Verknüpfung der Klasse mit den Adressen zu erreichen, wurde die im Abschnitt 4.2.3.3 vorgestellte Klasse MemoryLink entwickelt. Mit ihrer Hilfe werden jedem Port seine individuellen Adressen zugeteilt.

4.4.2.1 MMappedPeriphery-Klasse

Die Klassen aller internen Zusatzmodule, wie auch der Ports, sind von der abstrakten Klasse MMappedPeriphery abgeleitet. Diese ist eine Spezialisierung von MemoryRegion, die eine neue Implementierung der Zugriffs-Methoden read und write für Peripherie-Geräte bereitstellt. Da alle im Mikrocontroller integrierten Zusatzmodule nur über einen 8-bit breiten Datenbus mit der CPU verbunden sind (siehe Blockdiagramm 3.2), liefern die Zugriffs-Methoden für 16-bit und 32-bit Werte Fehler. Ist ein 16-bit Zugriff in Ausnahmefällen doch erlaubt, so werden die Zugriffs-Methoden für 16-bit Werte in Unterklassen neu implementiert.

Als Erweiterung der Klasse MemoryRegion werden die Methoden reset und install eingeführt, die in jeder abgeleiteten Klasse implementiert werden müssen (siehe Klassendiagramm B.3 und B.4 im Anhang). Ein Aufruf von reset soll das Gerät in den Grundzustand versetzen. Die Methode reset_periphery der Klasse Microcontroller nutzt diese Methode, um alle Geräte zurücksetzen. Zur Durchführung dieser Aufgabe existiert eine Liste, die Verweise auf alle Zusatzmodule enthält. Aufgebaut wird diese Liste durch die Methode install. Diese wird von der Klasse Microcontroller sofort nach der Erzeugung eines auf MMappedPeriphery basierenden Objekts aufgerufen. Außerdem soll ein Aufruf von install das entsprechende Gerät bei der Adressraumverwaltung anmelden.

Da die Anmeldung von jedem abgeleiteten Objekte individuell implementiert werden muss, der Aufbau der Liste jedoch allgemein erfolgen kann, ist der Code in zwei Methoden aufgeteilt. Beide heißen install, erwarten jedoch unterschiedliche Parameter. Die erste Methode fügt das aktuelle Objekt zu der reset-Liste hinzu und ruft anschließend die zweite auf. Diese soll in den abgeleiteten Objekten überschrieben werden um die Adresszuordnung vorzunehmen.

interne Klasse BinaryRegister

Alle integrierten Module enthalten Register, deren Bits jeweils als einzelne Schalter genutzt werden. Um die Implementierung dieser Register zu erleichtern, stellt die Klasse MMappedPeriphery die Klasse BinaryRegister für ihre Unterklassen zur Verfügung (siehe Klassendiagramm B.3 im Anhang). Diese Klasse speichert intern einen Wert und enthält Methoden um die einzelnen Bits dieses Wertes anzusprechen. Diese sind get, set und clear, die jeweils einen Parameter bekommen, der das gewünschte Bit spezifiziert. Selbstverständlich kann man den Wert auch als Ganzes bearbeiten. Dazu stehen die Methoden toInteger, assign und assign_with_mask zur Verfügung. Die ersten beiden Methoden dienen dazu, den Inhalt des Registers auszulesen bzw. neu festzulegen. Die dritte Methode verändert nur die Bits des Registers, die in einer zusätzlich angegebenen Maske gesetzt sind. Verwendung findet diese Methode, wenn dem Register ein neuer Wert zugeordnet werden soll, einige Bits davon aber unbeeinflusst bleiben sollen.

4.4.2.2 Port1 bis Port7

Die Ports sind in den Klassen Port1 bis Port7 implementiert (vgl. Klassendiagramm B.4 im Anhang). Aufbauend auf ihre Vaterklasse MMappedPeriphery, müssen diese Klassen lediglich den 8-bit Zugriff auf ihre Register, die durch Instanzen von BinaryRegister dargestellt werden, realisieren. Für einen lesenden Zugriff auf das Datenregister werden dazu alle Objekte abgefragt, die mit Pins verbunden sind. Mit jedem Port können dabei mehrere Eingabegeräte bzw. deren Objekte verbunden sein, wenn jedes Gerät nur wenige Pins benötigt. Ein Beispiel hierfür ist der Port 4, der mit zwei Tastern verbunden ist, die je nur eine Leitung verwenden.

Bei schreibenden Zugriffen wird eine Veränderung an alle Geräte, bzw. deren Objekte, weitergeleitet, die an diesem Port angeschlossen sind. Dafür müssen von den Klassen der externen Geräte entsprechende Methoden zur Verfügung gestellt werden. Da die externen Geräte jedoch sehr verschieden sind, ist die Verbindung zwischen Port-Objekt und Geräte-Objekt sehr individuell ausgeführt.

Die Klassen mancher Ports stellen jedoch keine Funktionalität zur Verfügung. Das ist dann der Fall, wenn an den entsprechenden Ports kein simuliertes Gerät angeschlossen ist. Zum Beispiel sind die Klassen Port1 bis Port3 nur Platzhalter, da der Hitachi Mikrocontroller im expanded Modus simuliert und somit der Busverkehr über diese Ports abgewickelt wird. Im Simulator wird jedoch nicht zwischen internem und externem RAM unterschieden. Das gesamte RAM ist deshalb über die Klasse Memory (siehe Abschnitt 4.2.2) ansprechbar und die Ports 1 bis 3 sind ungenutzt. Auch die Klasse Port5 stellt keine Funktionalität zur Verfügung, da die Pins der gleichnamigen Schnittstelle durch die integrierte serielle Schnittstelle verwendet werden.

Port 4, 6 und 7 werden hingegen simuliert, da bei diesen einige Pins als allgemeine Ein-/Ausgänge verwendet werden, um externe Peripherie anzuschließen. An Port 4 und 7 sind jeweils zwei der vier Tasten des Lego-Kontrollbausteins angeschlossen, über Port 6 findet die Kommunikation mit dem Display-Treiber statt.

4.4.2.3 externe Peripherie

RCXButton-Klasse

Zur Simulation einer Taste dient die Klasse RCXButton. Sie sorgt für die Bildschirmdarstellung eines Buttons und stellt die Methode get_state zur Verfügung. (siehe Klassendiagramm B.4 im Anhang). Diese Methode liefert den aktuellen Zustand des Buttons zurück und wird von den Klassen Port4 bzw. Port7 aufgerufen, sobald ein lesender Zugriff auf ihr Datenregister stattfindet.

Für die Bildschirmdarstellung ist die Klasse RCXButton von der Java-Swing Komponente JButton abgeleitet. Somit kann sich die Klasse zum Layout-Manager des Hauptfensters hinzufügen, um einen Button auf dem Bildschirm darzustellen. Damit man die vier Taster voneinander unterscheiden kann, übergibt man einem RCXButton-Objekt bei dessen Konstruktion einen Parametersatz. Dieser enthält Angaben über die Farbe, die Beschriftung und die Position des Buttons im Hauptfenster, sowie darüber, welcher Interrupt ausgelöst werden soll, sobald der Taster gedrückt wird.

Um die Methode get_state mit der richtigen Information zu versorgen, wird ein MouseListener installiert, der auf jede Mausaktion, die den Button betrifft, reagieren kann. Da man auch

die Möglichkeit haben muss zwei Tasten gleichzeitig drücken zu können, führt ein Anklicken eines Buttons mit der rechten Maustaste zu einer Arretierung des Buttons, die mit der linken Maustaste wieder gelöst werden kann. Zu beachten ist noch, dass die am Port 4 angeschlossenen "On-Off" und "Run"-Taster einen Interrupt auslösen können. Deshalb ruft der MouseListener bei einem mousePressed Ereignis die Methode IRQ des Interrupt-Controllers auf, falls bei der Konstruktion des Buttons eine gültige Interruptnummer angegeben wurde.

LCD_driver-Klasse

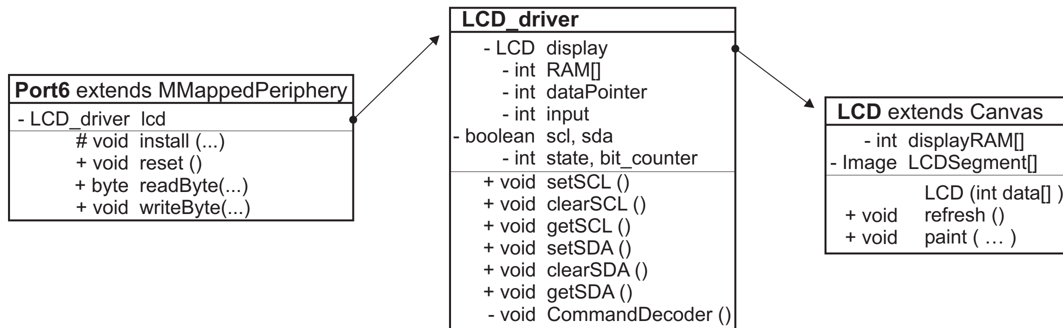


Abbildung 4.19: Die Klassen LCD_driver und LCD

Die Simulation des Display-Treibers findet in der Klasse LCD_driver statt (siehe Abbildung 4.19). Sie ist über zwei virtuelle Leitungen mit Port 6 verbunden. Die Leitungen haben die Namen "serial clock line" (SCL) und "serial data line" (SDA) und werden in der Klasse LCD_driver durch jeweils drei Methoden dargestellt. Die Methoden stellen die Verbindung mit der Klasse Port6 her. Bei einem lesenden Zugriff auf den Port 6 informiert sich dieser über den Zustand der Leitungen mithilfe der Methoden getSCL bzw. getSDA. Schreib-Zugriffe werden mit den Methoden setSCL bzw. setSDA und clearSCL bzw. clearSDA dem simulierten LCD-Treiber mitgeteilt.

Der Original-Display-Treiber Chip, bei dem es sich um einen Philips PCF8566 handelt, wickelt auf den beiden Leitungen das von Philips entwickelte I^2C-Bus Protokoll ab. Dabei handelt es sich um eine serielle Art der Datenübertragung, die zum Datenaustausch zwischen Mikrocontroller und Treiber-Baustein eingesetzt wird. Genaue Informationen über den Chip kann man dem zugehörigen Datenblatt [Phi98] entnehmen. Dort finden sich auch Details über das Bus-Protokoll, die über die hier gegebenen Informationen hinaus gehen.

Im Mikrocontroller wird das Protokoll in Software abgewickelt. Das heißt dass eine im ROM vorhandene Funktion die Kommunikation gemäß des Protokolls durchführt. Da eine Firmware die Ansteuerung des Displays jedoch auch ohne Hilfe der ROM-Funktion durchführen kann, indem sie das I^2C-Bus Protokoll selbst am Port 6 abwickelt, muss der simulierte Display-Treiber dieses Protokoll unterstützen.

Die Abwicklung des Protokolls wird in der Simulation durch die oben genannten Methoden setSXX, clearSXX und getSXX realisiert, wobei SXX für SCL und SDA steht. Wurde eine Byte empfangen, so wird anschließend die private Methode CommandDecoder aufgerufen, die entscheidet, ob es sich um ein Kommando oder um Daten handelt.

Bei einer Kommunikation über diesen Bus wird zuerst die Adresse des anzusprechenden Gerätes übertragen. Da sich beim Lego-Baustein jedoch nur ein Gerät am I^2C-Bus befindet, der

Display Treiber, wird die Adresse nicht überprüft. Anschließend folgen einige Kommandobytes, die dem Display-Treiber mitteilen, wie er das LC-Display ansprechen soll. Diese sind in der simulierten Version ebenfalls irrelevant, da der Treiber nur mit dem simulierten Display zusammenarbeiten und nicht für viele verschiedene LC-Displays geeignet sein muss. Lediglich das BLINK-Kommando könnte man beachten, welches die Blink-Frequenz festlegen soll. Allerdings unterstützt das simulierte Display diese Funktion nicht, da sie vom ROM des RCX nicht genutzt wird.

Nach den Kommandobytes folgen die Display-Daten. Diese werden in einem Byte-Array abgelegt und teilen dem Display mit, welche Segmente aktiv und welche inaktiv sein sollen. Dabei steuert jedes Bit dieser Daten ein Segment. Die I^2C-Bus Übertragung ist somit abgeschlossen und die Klasse LCD_driver ruft die refresh Methode ihres LCD-Objektes auf.

LCD-Klasse

Die Klasse LCD ist für die Simulation des Displays selbst verantwortlich. Die Verbindung mit dem Treiber erfolgt über eine Referenz auf das Daten-Array der Treiber Klasse, welche bei der Konstruktion eines LCD-Objekts angegeben wird. Die Klasse LCD ist eine Unterklasse der Java AWT Komponente Canvas. Deshalb kann sie beim Hauptfenster registriert werden und erhält dadurch ihren Zeichenbereich auf dem Bildschirm. Abbildung A.1 im Anhang zeigt das Erscheinungsbild des Displays.

Die paint Methode sorgt für die Segment-Darstellung, diese wird durch eine Überlagerung von Bildern erreicht, die im GIF-Format vorliegen. Jedes Bild stellt ein Segment dar und ist sonst transparent. Jedem Segment ist ein Bit im Daten-Array zugeordnet. Ist ein Bit gesetzt, so wird das entsprechende Segment-Bild angezeigt, ist es gelöscht, so wird das zugehörige Bild nicht angezeigt. Falls sich die Daten ändern, muss die Treiber-Klasse nur die Methode refresh aufrufen, die das Java Framework veranlasst den Bildschirminhalt neu zu zeichnen. Die paint-Methode, die für das Zeichnen des Bildschirminhaltes zuständig ist, berücksichtigt dann die neuen Daten, da sie durch eine Referenz auf das Array der Treiber-Klasse automatisch zur Verfügung stehen.

4.5 Motor-Treiber

4.5.1 Überblick

Die Motor-Treiber sind nicht, wie die anderen externen Geräte, an einen Port angeschlossen. Ihre Register werden stattdessen direkt in den Adressraum eingeblendet. Sie sind über die Adresse 0xF000 anzusprechen und steuern die Spannungsversorgung der Motoren. Der Wert, der an diese Adresse geschrieben wird, steuert somit die Bewegung der Motoren. Jedem der drei Motoren sind zwei Bits des Registers zugeordnet, daraus ergeben sich vier Funktionsarten für jeden Motor: vorwärts, rückwärts, frei drehend und gebremst.

4.5.2 Implementierung

Die Simulation ist auf zwei Klassen aufgeteilt (siehe Abbildung 4.20). Zum einen die Klasse Motor_driver für die Anbindung an den Adressbus, zum anderen die Klasse RCXMotor für die Bildschirmdarstellung. Beide werden im Folgenden kurz vorgestellt.

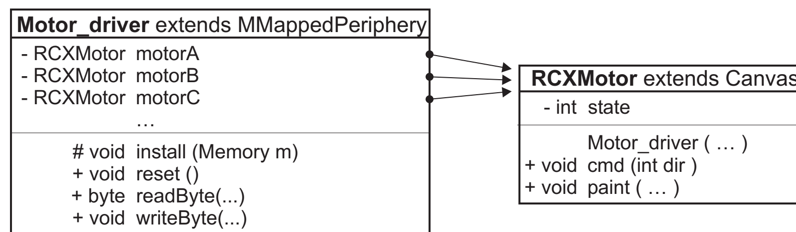


Abbildung 4.20: Klassendiagramm von Motor_driver und RCX_Motor

4.5.2.1 Motor_driver-Klasse

Die Klasse Motor_driver ist von MMappedPeriphery abgeleitet und verwaltet die Adresse, über die der Motor-Treiber anzusprechen ist. Um auf die Adresse reagieren zu können, muss ein Motor_driver-Objekt bei der Adressraumverwaltung registriert werden. Wie von der Vaterklasse vorgesehen, geschieht das in der Methode install, die nach der Konstruktion eines Objektes aufgerufen werden muss. Die Motoren werden durch drei Objekte vom Typ RCXMotor simuliert. Diese stellen die Methode cmd zur Verfügung, über die ihnen neue Kommandos mitgeteilt werden können.

Erfolgt ein lesender Zugriff auf die Adresse des Motor-Treibers, werden die zuletzt geschriebenen Daten zurückgegeben. Die Daten eines schreibenden Zugriffes werden gespeichert und an die simulierten Motoren weitergeleitet. Dazu werden jeweils die zwei Bits für einen Motor extrahiert und die cmd Methode des entsprechenden RCXMotor-Objekts aufgerufen. Dies geschieht bei jedem write-Aufruf für alle drei Motoren.

4.5.2.2 RCXMotor-Klasse

Die Klasse RCXMotor ist für die Bildschirmdarstellung der Motorbewegungen zuständig. Sie ist, wie die Klasse LCD, von der Java AWT Komponente Canvas abgeleitet. Die Position im Hauptfenster wird durch ein GridBagConstraints-Objekt festgelegt, welches bei der Konstruktion übergeben wird. Die Methode cmd, die von der Klasse Motor_driver aufgerufen wird, um den simulierten Motor zu steuern, überprüft die übergebene Steuerinformation und setzt die interne Variable state auf einen entsprechenden Wert. Unterscheidet sich der Wert von dem des vorherigen Aufrufes, so veranlasst sie eine Aktualisierung der Bildschirmausgabe, sodass die paint-Methode die Änderung darstellen kann.

Die Bildschirmausgabe, die im praktischen Teil dieser Studienarbeit implementiert wurde (siehe Abbildung A.1 im Anhang), zeigt nur den aktuellen Status der Motoren an. Dadurch ist sie schlecht geeignet, verschiedene Geschwindigkeiten eines Motors darzustellen. Diese werden erreicht, indem ständig zwischen frei laufend und angetrieben gewechselt wird. Die Trägheit eines Elektromotors sorgt dafür, dass es zu einer gleichmäßigen Drehung kommt, die umso schneller ist, umso länger die Einschaltphasen sind. Die Bildschirmausgabe einer mittleren Geschwindigkeit äußert sich deshalb in einem ständigen Wechsel von fahrend und stehend, verschiedene Geschwindigkeiten lassen sich daher nur schlecht voneinander unterscheiden.

Um Geschwindigkeiten besser darzustellen, könnte man einen Balken für jede Fahrtrichtung verwenden, dessen Länge sich mit der Geschwindigkeit ändert. Die Implementierung müsste

dann versuchen die Trägheit der Motoren auf den Balken zu übertragen. Dies könnte man zum Beispiel dadurch erreichen, dass der Balken länger wird, je länger die interne Variable `state` “fahrend” signalisiert. Wechselt sie auf “frei laufend”, so müsste der Balken wieder kürzer werden.

Da diese Art der Darstellung jedoch eine häufige Aktualisierung des Bildschirms zur Folge hätte und das viel Zeit in Anspruch nimmt, würde der Simulator langsamer werden. Deshalb wurde nur die oben beschriebene einfachere Darstellung implementiert.

4.6 Zähler

Der Mikrocontroller im RCX besitzt drei Zähler. Einen 16-bit Zähler, einen 8-bit Zähler mit zwei unabhängigen Kanälen und einen Watchdog-Timer. In den folgenden Abschnitten wird auf die Besonderheiten und Realisierung in der Simulation eingegangen. Diagramme der implementierten Klassen sind in Anhang B abgebildet.

4.6.1 16-bit Zähler

4.6.1.1 Überblick

Der 16-bit Zähler, auch „free-running timer” genannt, ist die größte und flexibelste Variante unter den Zählern. Sein Zählregister ist 16-bit breit und wird durch den Systemtakt oder durch einen externen Takt erhöht. Erreicht er einen benutzerdefinierten Wert, so können verschiedene Aktionen ausgelöst werden. Zu diesem Zweck sind zwei 16-bit breite so genannte *Output-Compare*-Register vorhanden, die ständig mit dem Zähler verglichen werden. Stimmt der Wert mit einem der beiden Register überein, so wird das durch ein Bit im Statusregister des Zählers vermerkt. Gleichzeitig können dadurch drei Reaktionen ausgelöst werden. Erstens kann ein Interrupt generiert werden, zweitens kann ein Pegelwechsel an einem Ausgang hervorgerufen werden und drittens kann der Zähler automatisch auf Null zurückgesetzt werden. Das automatische Rücksetzen des Zählers ist allerdings nur durch eine Übereinstimmung mit dem *Output-Compare*-Register A möglich. Alle Funktionen lassen sich einzeln und für jedes der beiden Compare-Register getrennt ein- und ausschalten.

Eine mögliche Anwendung des Zählers ist die Erzeugung eines Rechtecksignals. Dazu ändert man den Ausgangpegel bei Erreichen eines definierten Zählerstandes und setzt den Zähler auf Null zurück.

Eine Besonderheit besteht beim Zugriff auf die *Output-Compare*-Register. Beide sind über dieselbe Adresse zu erreichen. Welches bei einem Zugriff auf diese Adresse angesprochen wird, entscheidet ein Bit in einem der Kontrollregister.

Mit dem 16-bit Zähler hat man außerdem die Möglichkeit auf externe Signale zu reagieren, die an einem Pin des Zählers (Port 6) angelegt werden. Dazu stehen vier, ebenfalls 16-bit breite, *Input-Capture*-Register zur Verfügung, die im Falle einer positiven oder negativen Flanke des externen Signals, je nach Wunsch, den Inhalt des Zählregisters aufnehmen können. Dadurch ist es zum Beispiel möglich die Frequenz eines externen Signals zu bestimmen. Da diese Möglichkeiten im Lego-Kontrollbaustein aber nicht genutzt werden, wurden sie im Simulator nicht implementiert und werden deshalb auch nicht näher erklärt.

4.6.1.2 Implementierung

Die Simulation des 16-bit Zählers findet in der Klasse Timer_16bit statt (siehe Abbildung B.2 im Anhang). Sie ist wie alle Klassen interner Geräte von der Klasse MMappedPeriphery abgeleitet. Die Adresszuteilung in der Methode install erfolgt ohne Umweg über MemoryLink-Objekte, da die Register des Zählers einen zusammenhängenden Adressbereich bilden.

Die Funktionen des Zählers werden durch die Methode increment realisiert, die in der Hauptschleife des Simulators in jedem Durchlauf aufgerufen wird. In dieser Methode soll zuerst das Zählregister um eins erhöht werden. Deshalb wird überprüft, ob es dabei zu einem Überlauf kommen würde. Ist das der Fall, so wird ein entsprechendes Status-Bit gesetzt und unter Umständen ein Interrupt ausgelöst, falls das zugehörige Enable-Bit gesetzt ist. Das Auslösen eines Interrupts erfolgt durch den Aufruf der Methode IRQ des Interrupt-Controllers. Nach einem Überlauf wird das Zählregister auf Null gesetzt, sonst wird es erhöht.

Nachdem das Zählregister aktualisiert wurde, findet ein Vergleich mit den *Output-Compare*-Registern statt. Dazu dient die private Methode compare, die im Falle einer Übereinstimmung ein Bit im Status-Register setzt und auf Wunsch einen Interrupt auslöst. Anschließend muss überprüft werden, ob der Zähler zurückgesetzt werden soll. Dies darf jedoch nur bei einer Übereinstimmung des Zählers mit dem *Output-Compare*-Register A erfolgen. Da die Methode compare aber auch für den Vergleich mit dem *Output-Compare*-Register B eingesetzt wird, findet die Überprüfung außerhalb dieser Methode statt. Die Methode compare liefert dazu das Ergebnis des Vergleichs als boolescher Wert zurück. Bei einem positiven Vergleich mit dem Register A und gesetztem "Rücksetz-Bit" im Kontrollregister des Zählers wird dann das Zählregister gelöscht.

Die *Input-Capture*-Register wurden im Simulator aus den oben genannten Gründen nicht implementiert. Eine Simulation könnte man aber nach einem ähnlichen Prinzip realisieren, wie es bei der Simulation der Ports Anwendung findet. Die Klasse Timer_16bit könnte beispielsweise zwei Funktionen (clear und set) pro Eingangspin zur Verfügung stellen. Diese könnten dann von anderen Klassen, die externe Signale simulieren, aufgerufen werden um dem Zähler einen Pegelwechsel mitzuteilen. Die Reaktion auf eine Flanke könnte dann in den neuen Methoden implementiert werden.

Zwei Besonderheiten beim Zugriff auf die Register sollen hier noch erwähnt werden. Im realen Mikrocontroller ist das Auslesen eines 16-bit Registers der Peripherie nicht direkt möglich, da zu den integrierten Zusatzmodulen nur ein 8-bit breiter Datenbus vorhanden ist. Deshalb werden bei einem Zugriff auf die höherwertigen acht Bit die anderen in einem temporären Register gesichert. Der nächste Zugriff auf die unteren acht Bit wird dann mit dem Inhalt dieses Registers beantwortet. Somit wird verhindert, dass die beiden 8-bit Zugriffe auf verschiedenen Zuständen des 16-bit Registers arbeiten. Im Simulator ist ein direkter Zugriff auf die 16-bit Register möglich. Zwar wurden 16-bit Zugriffe allgemein durch die Klasse MMappedPeriphery gesperrt, doch durch eine neue Implementierung der Methoden readShort und writeShort in der abgeleiteten Klasse Timer_16bit werden sie wieder möglich.

Die zweite Besonderheit ist der Zugriff auf die *Output-Compare*-Register, die sich, wie schon erwähnt, eine Adresse teilen müssen. Um den Zugriff effizient zu realisieren, existiert im Simulator eine Referenz, die entweder auf das eine oder auf das andere Register verweist. Jeder Zugriff auf die Adresse wird über diese Referenz an das entsprechende Register geleitet. Wird

das “Auswahlbit” im Status-Register verändert, so wird nur die Referenz auf das neu gewählte Register eingestellt.

Vom Lego-ROM wird der 16-bit Zähler zur periodischen Erzeugung von Interrupts verwendet. Dazu werden die Register so konfiguriert, dass der Zähler einen Interrupt auslöst, sobald der Wert 500 erreicht ist und das Zählregister danach automatisch auf Null zurückgesetzt wird. Die Behandlungsroutine des Interrupts erledigt alle anfallenden Aufgaben, von der Steuerung der Motoren bis hin zur Verwaltung der Benutzerprogramme, also des Schedulings.

4.6.2 8-bit Zähler

4.6.2.1 Überblick

Der 8-bit Zähler stellt zwei voneinander unabhängige Kanäle zur Verfügung. Das heißt, dass es sich eigentlich um zwei Zähler handelt. Die grundsätzliche Funktionsweise jedes dieser 8-bit Zähler gleicht der des 16-bit Zählers, abgesehen davon, dass jedes der beiden Zählregister nur 8-bit breit ist und deshalb schon bei einem Wert von 255 sein Maximum erreicht hat.

Jeder Zähler hat zwei so genannte *Timer-Constant*-Register, welche dieselbe Funktion wie die *Output-Compare*-Register des 16-bit Zählers haben, nur dass hier jedes der insgesamt vier Register über eine eigene Adresse erreichbar ist. Bei der Übereinstimmung eines Zählregisters mit einem seiner Vergleichs-Register kann, wie beim 16-bit Zähler, ein Interrupt ausgelöst, der Zähler auf Null zurückgestellt oder der Pegel eines Ausgangspins verändert werden. Auf externe Signale kann der 8-bit Timer nicht reagieren, er wird deshalb hauptsächlich zur Erzeugung von Rechtecksignalen eingesetzt.

Im RCX wird der erste Kanal zur Erzeugung von Schwingungen genutzt, die über den Lautsprecher als Töne wiedergegeben werden. Der zweite Kanal wird von Lego-ROM dazu verwendet ein Trägersignal zu erzeugen, welches bei der Übertragung von Daten über die Infrarot-Schnittstelle benötigt wird.

4.6.2.2 Implementierung

Die beiden Kanäle des 8-bit Zählers werden durch Objekte der Klasse Timer_8bit simuliert. Der Konstruktor erwartet die Angabe, welcher Kanal durch dieses Objekt dargestellt werden soll. Die Simulation eines 8-bit Zählers erfolgt analog zu der des 16-bit Zählers. Jeder Zähler wird einmal pro Durchlauf der Hauptschleife des Simulators aktiviert, indem seine Methode increment aufgerufen wird. Dort wird analog zum 16-bit Zähler das Zählregister erhöht. Bei einem Überlauf wird ein Bit im Status-Register gesetzt und eventuell ein Interrupt ausgelöst. Anschließend werden die *Timer-Constant*-Register durch Aufruf der Methode compare mit dem Zählerstand verglichen. Bei Übereinstimmung wird, wie vom 16-bit Zähler schon bekannt, ein Flag gesetzt und durch die Methode IRQ ein Interrupt ausgelöst, falls das Enable-Bit gesetzt ist.

Im Gegensatz zum 16-bit Zähler können jedoch, je nach Einstellung im Kontrollregister, beide Vergleiche zu einer Rückstellung des Zählregisters führen. Trotzdem liefert die Methode compare nur das Vergleichsergebnis als boolescher Wert zurück, auf den dann die increment-Methode reagieren muss. Der Grund dafür ist nur in einem symmetrischen Aufbau der beiden Zähler-Klassen zu sehen.

4.6.3 Watchdog Timer

Der Watchdog-Timer wird nicht simuliert, da er vom Lego-ROM und von der bekannten Firmware nicht genutzt wird. Trotzdem soll seine Funktion kurz erklärt werden um auf eine mögliche Implementierung hinzuweisen.

Beim Watchdog-Timer handelt es sich um einen 8-bit Zähler, der bei Überlauf wahlweise einen einfachen Interrupt, den *nonmaskable* Interrupt oder ein Reset des Mikrocontrollers auslösen kann. Seinen Aufgabe ist es, Verklemmungen der Software zu erkennen und aufzulösen. Bei "normaler" Funktion muss die Software dazu das Zählregister des Watchdog Timers in regelmäßigen Abständen auf Null zurücksetzen. Erfolgt das nicht oder zu spät, so läuft der Zähler über. Dies ist ein Zeichen dafür, dass etwas nicht korrekt funktioniert. Das vermeintliche Problem wird dann je nach Konfiguration, beispielsweise durch einen Reset, gelöst. Alternativ kann man auch einen Interrupt generieren lassen, dessen Behandlungsroutine den Fall klären soll.

Die Implementierung könnte dem gleichen Prinzip folgen, welches schon den anderen Zählern zugrunde liegt. Die Hauptschleife könnte in jedem Durchgang eine increment-Methode aufrufen, die das Zählregister erhöht. Kommt es dabei zu einem Überlauf, wird eine Aktion ausgeführt. Soll ein Interrupt generiert werden, so müsste die Methode IRQ der Klasse SystemController aufgerufen und die entsprechende Nummer mitgegeben werden. Zur Durchführung der Resetsequenz wäre zum einen die Methode reset der Klasse CPU aufzurufen, um den Prozessor in den Startzustand zu versetzen. Zum anderen müsste die Methode reset_periphery der Klasse Microcontroller aufgerufen werden, um einen Reset der integrierten Peripherie zu erreichen.

4.7 Analog/Digital Wandler

4.7.1 Überblick

Zur Ausstattung eines Mikrocontrollers gehört meist ein A/D-Wandler um externe analoge Signale in eine digital verwendbare Darstellung zu bringen. Im Hitachi H8/3292 ist ein 8-Kanal, 10-bit A/D Wandler integriert. Er arbeitet nach dem Wägeverfahren und kann 8 Eingangsspannungen im Multiplexbetrieb bearbeiten. Zum Speichern seiner Ergebnisse stehen ihm vier 16-bit Register zur Verfügung. Da jede Transformation ein 10-bit genaues Ergebnis liefert, sind immer 6 Bit ungenutzt. Außerdem müssen sich jeweils zwei Eingänge ein Register teilen.

Der A/D-Wandler lässt sich in zwei Modi betreiben, zum einen der so genannte *Single*-Modus, in dem nur ein Eingang bearbeitet wird, zum anderen der *Scan*-Modus. In diesem Modus wird eine Gruppe von bis zu vier Eingängen zyklisch immer wieder nacheinander abgefragt. Ist ein Vorgang abgeschlossen, das heißt entweder die Transformation eines Wertes im *Single*-Modus oder ein Zyklus im *Scan*-Modus, so wird das durch ein Bit im Status-Register vermerkt. Gleichzeitig kann ein Interrupt ausgelöst werden, falls dies gewünscht ist.

Im Lego-Kontrollbaustein sind an den Eingangspins des A/D-Wandlers die drei Sensoreingänge, angeschlossen, die der Lego-Baustein bereitstellt. Ein weiterer Eingang wird genutzt um die Batteriespannung messen zu können. Die anderen vier Kanäle sind ungenutzt.

4.7.2 Implementierung

Die Simulation des A/D-Wandlers ist in zwei Klassen aufgeteilt (siehe Abbildung 4.21). Zum einen die Klasse ADConverter: sie ist das Interface zum Bus, verwaltet die Register und simuliert das Verhalten des Wandlers. Zum anderen die Klasse RCXSensor, welche die Verbindung mit dem Benutzer herstellt und einen Sensor simulieren soll. Von der Klasse ADConverter erzeugt das Microcontroller-Objekt eine Instanz (vgl. Abbildung B.2 im Anhang), diese wiederum enthält vier RCXSensor-Objekte, je eins für jeden benutzten Eingang.

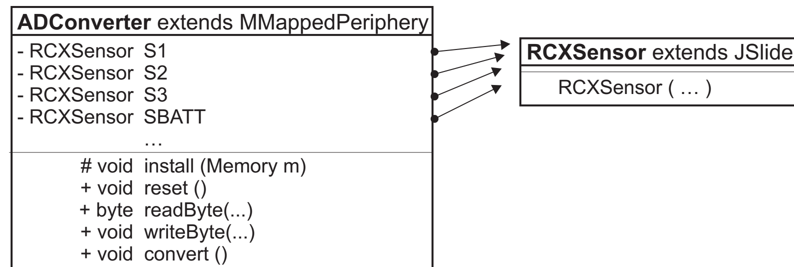


Abbildung 4.21: Die Klasse ADConverter und RCXSensor

Ein Sensor wird am Bildschirm als Schieberegler dargestellt (siehe Abbildung A.1 im Anhang). Dafür wurde die Klasse RCXSensor von der Java Komponente JSlider abgeleitet. Der Einstellungsbereich eines Schiebereglers erlaubt es einen Wert zwischen 0 und 1023 einzustellen. Dieser Wert entspricht dem 10-bit Ergebnis einer konvertierten Eingangsspannung. Eine Umrechnung von einem Spannungswert zu einem digitalen Wert ist somit nicht nötig.

Wie schon von den Zählern bekannt, wird auch eine Methode des A/D-Wandlers einmal pro Durchlauf der Hauptschleife des Simulators aufgerufen. Der Name dieser Methode ist convert. Ist der A/D-Wandler aktiv, so wird durch diese Methode eine interne Zählvariable erhöht. Erreicht diese Variable einen im Simulator festgelegten Wert, so werden die Ergebnisregister aktualisiert und ein Interrupt ausgelöst, falls dessen Enable-Bit gesetzt ist.

Dieses Vorgehen soll die Dauer eines Konvertierungsvorgangs simulieren. Die Zeit zwischen zwei Konvertierungen ist im Programm auf 133 Aufrufe festgesetzt. Ohne Verzögerung würde jeder Aufruf von convert zu einem Konvertierungsergebnis führen. Dadurch würden aber so viele Interrupts ausgelöst, dass das eigentliche Programm nicht mehr bearbeitet werden könnte.

Wird der A/D-Wandler im *Scan*-Modus betrieben, so wird nach Abschluss eines Konvertierungsvorgangs ein neuer gestartet, ist hingegen der *Single*-Modus gewählt, so wird das *Start*-Bit gelöscht. Das Löschen dieses Bits hat zur Folge, dass ein Aufruf von convert sofort wieder zurückkehrt, der simulierte A/D-Wandler ist somit deaktiviert.

Bei der Aktualisierung der Ergebnisregister weicht die Simulation von der realen Funktionsweise des Mikrocontrollers ab. Eigentlich sollte man über ein spezielles Register auswählen können, welche Eingangspins bei einer Konvertierung beachtet werden sollen. Die Simulation aktualisiert allerdings immer alle vier Datenregister mit den Werten der vorhandenen RCXSensor-Objekten. Dies sollte jedoch keine großen Auswirkungen auf das Gesamtverhalten haben, da zum einen die simulierte Dauer einer Konvertierung dadurch nicht länger wird, zum anderen beim Lego-Kontrollbaustein RCX nur vier Eingangspins des A/D-Wandlers mit Sensoren verbunden sind, sodass die Signalquellen für den A/D-Wandler nie wechseln können.

Außerdem wird die Zulässigkeit der Vereinfachung durch das Lego-ROM bestätigt, welches den A/D-Wandler immer im *Scan*-Modus betreibt, der zudem so konfiguriert ist, dass alle vier Eingänge abgearbeitet werden

4.8 serielle Schnittstelle

4.8.1 Überblick

Viele Mikrocontroller besitzen eine integrierte serielle Schnittstelle, um mit anderen Geräten Daten austauschen zu können. Sie wird meist auch genutzt, um die Software in den Speicher zu laden.

Auch der Mikrocontroller des Lego-Kontrollbausteins verfügt über eine serielle Schnittstelle. Sie ist mit der im Baustein eingebauten Infrarotsende- und -empfangseinheit verbunden. Die Funktionsweise der seriellen Schnittstelle wird durch die Infrarotübertragung nur insofern beeinträchtigt, dass vor dem Senden von Daten zuerst ein Trägersignal für die IR-Übertragung bereitgestellt werden muss. Wie schon erwähnt, wird dieses im RCX durch einen der beiden 8-bit Zähler erzeugt. Die Kontrollhardware der Infraroteinheit sorgt dann dafür, dass die eigentlichen Nutzdaten von der seriellen Schnittstelle auf das Trägersignal aufmoduliert und über die IR-Diode abgesendet werden.

Die serielle Schnittstelle beherrscht viele Übertragungsarten. Je nach Einstellung überträgt sie die Daten synchron oder asynchron, mit 7 oder 8 Datenbits, mit oder ohne Paritäts-Bit. Die Geschwindigkeit ist wählbar zwischen 110 und 38400 Baud, die Empfangseinheit ist unabhängig von der Sendeeinheit, sodass die Daten im vollduplex Betrieb übertragen werden können. Zusätzlich kann die Schnittstelle noch in einem speziellen Multiprozessormodus betrieben werden, der es erlaubt mehrere Mikrocontroller miteinander zu verbinden, die wie auf einem Bus über eine eindeutige Nummer identifiziert werden. Im Lego-Steuerbaustein wird dieser Modus jedoch nicht verwendet. Dort kommt nur eine normale asynchrone Datenübertragung zum Einsatz, weshalb sich auch die Simulation auf diese Art beschränkt.

Bevor die Schnittstelle aktiviert wird, müssen zuerst die gewünschten Einstellungen bezüglich der Übertragung und der Baudrate in die Steuer-Register der Schnittstelle geschrieben werden. Anschließend kann man durch das Setzen der entsprechenden Bits den Empfangs- und Sendeteil der Schnittstelle aktivieren.

Zum Absenden eines Bytes muss dann das Datenbyte in das *Transmit-Data*-Register geschrieben werden. Anschließend muss das "*Transmit Data Register Empty*"-Bit im Status-Register der Schnittstelle gelöscht werden, um ihr mitzuteilen, dass jetzt gültige Daten im Register vorliegen. Die Hardware kopiert dann dieses Byte in ein anderes Register und beginnt die bitweise Übertragung über die serielle Leitung. Da das Daten-Register nun wieder frei ist, wird das "*Transmit Data Register Empty*"-Bit wieder gesetzt. Gleichzeitig kann ein Interrupt ausgelöst werden, um der Software mitzuteilen, dass ein neues Datenbyte zum Versenden bereitgestellt werden kann.

Ist die Übertragung des Bytes abgeschlossen, überprüft die Hardware ob ein neues Byte bereitliegt, dies wird durch das oben beschriebene Bit festgestellt. Liegen neue Daten vor, so beginnt ein neuer Zyklus. Falls kein neues Byte vorhanden ist, so wird angenommen, dass die Übertragung beendet ist. Deshalb wird das *Transmit-End*-Bit im Status-Register gesetzt, um

der Software mitzuteilen, dass die serielle Schnittstelle jetzt inaktiv ist. Dieses Ereignis kann zusätzlich noch mit einem Interrupt gemeldet werden.

Das Empfangen von Daten geht nach einem ähnlichen Prinzip vonstatten wie das Senden. Ankommende Bits werden erst in einem internen Register gesammelt, bis ein vollständiges Byte empfangen wurde. Tritt dabei ein Fehler auf, zum Beispiel in Form eines falschen Paritäts-Bit, so wird das im Status-Register vermerkt und, falls aktiviert, ein Interrupt ausgelöst. Wurde ein Byte ordnungsgemäß empfangen, so wird es ins *Receive-Data*-Register kopiert. Außerdem wird das “*Receive Data Register Full*”-Bit im Status-Register gesetzt, um der Software anzuzeigen, dass ein neues Datenbyte empfangen wurde. Damit die Software nicht ständig dieses Bit prüfen muss, kann gleichzeitig ein Interrupt erzeugt werden. Die Behandlungsroutine dieses Interrupts kopiert normalerweise das empfangene Byte in einen anderen Speicherbereich. Anschließend muss durch das Löschen des oben genannten Bits noch bekannt gemacht werden, dass das Register jetzt wieder frei ist ein neues Byte zu empfangen. Die Hardware kann dann das nächste empfangene Byte in das Register kopieren. Wird das Daten-Register nicht rechtzeitig frei, bevor ein neues Byte eintrifft, so wird das alte Datenbyte überschrieben und ist somit verloren. Die serielle Schnittstelle vermerkt diesen Fall im Status-Register und kann auch wieder einen Interrupt auslösen.

4.8.2 Implementierung

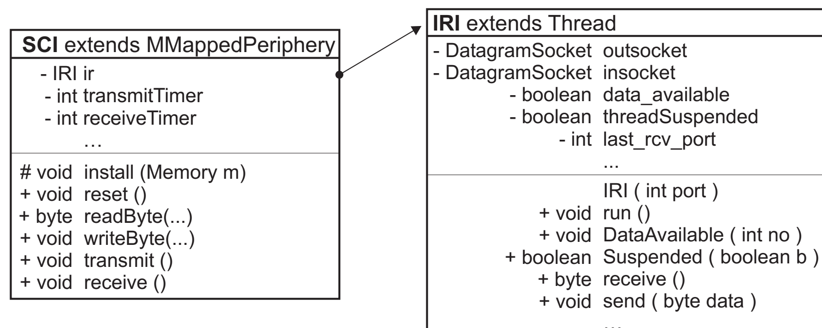


Abbildung 4.22: Die Klassen SCI und IRI

Die Übertragung der Daten zwischen dem Simulator und dem Gastsystem wird durch UDP-Pakete realisiert. Da die Übertragung somit nicht bitweise, wie es für eine serielle Übertragung üblich ist, sondern byteweise erfolgt, haben die Einstellungen bezüglich des Paritäts-Bits und der Größe der Datenbytes in der Simulation keine Bedeutung. Alle an einem benutzerdefinierten UDP-Port empfangenen Datenbytes werden für die Simulation als Ankunft über die serielle Schnittstelle dargestellt. Umgekehrt werden alle Daten, die über die serielle Schnittstelle abgeschickt werden in Wirklichkeit auf die Netzwerkverbindung umgeleitet.

Die Implementierung ist auch wieder in zwei Klassen aufgeteilt (siehe Abbildung 4.22). Zum einen die Klasse SCI, sie übernimmt die interne Repräsentation der seriellen Schnittstelle, indem sie die Register der Schnittstelle verwaltet. Zum anderen die Klasse IRI, sie stellt die Verknüpfung zu einem UDP-Port her.

4.8.2.1 Klasse SCI

Die Klasse SCI stellt die Register der seriellen Schnittstelle am Adressbus dar und ist von MMappedPeriphery abgeleitet. Die Anmeldung bei der Adressraumverwaltung wird deshalb durch die Methode install erledigt, die nach der Konstruktion eines neuen Objektes aufgerufen werden muss. Da der Adressbereich der Register nicht zusammenhängend ist, werden zwei MemoryLink-Objekte verwendet, um die Adresszuordnung vorzunehmen. Die Register werden durch Objekte vom Typ BinaryRegister dargestellt.

Die Funktionalität der Schnittstelle ist in den beiden Methoden transmit und receive enthalten, die von der Hauptschleife des Simulators periodisch aufgerufen werden. Die Methode receive simuliert das Empfangsteil der Schnittstelle, transmit übernimmt das Absenden der Daten.

Bei einem Aufruf von transmit wird zuerst geprüft, ob der Sendeteil der Schnittstelle aktiviert ist. Falls das entsprechende Bit, welches für die Aktivierung verantwortlich ist, nicht gesetzt ist, so kehrt die Methode sofort wieder zurück. Im anderen Fall wird anhand des "*Transmit Data Register Empty*"-Bits geprüft, ob Daten zum Versenden vorhanden sind. Ist das Daten-Register voll, so wird das Datenbyte an die Methode send des internen IRI-Objekts weitergegeben. Diese sorgt dafür, dass das Byte abgeschickt wird. Da das Daten-Register nun wieder frei ist, wird das "*Transmit Data Register Empty*"-Bit wieder gesetzt, anschließend wird die Methode IRQ des Interrupt-Controllers aufgerufen, um einen Interrupt zu erzeugen, falls das entsprechende Enable-Bit gesetzt ist.

Nun muss noch geprüft werden, ob das *Transmit-End*-Bit gesetzt werden muss. Das ist der Fall wenn die Übertragung abgeschlossen ist und kein neues Datenbyte bereit liegt. Im Simulator ist die Übertragung schon durch den Aufruf von send abgeschlossen. Das Daten-Register kann dann jedoch noch nicht wieder befüllt worden sein, da die Methode transmit nie die Kontrolle abgegeben hat. Somit hatte die Anwendungssoftware noch keine Gelegenheit, das Daten-Register neu zu befüllen. Um eine korrekte Simulation zu erreichen muss deshalb das Setzen dieses Bits verzögert werden. Zu diesem Zweck existiert eine private Zählvariable, die bei jedem Aufruf von transmit erhöht wird. Das Senden eines Bytes setzt die Variable wieder auf Null. Erfolgt lange keine Übertragung, wird dieser Zähler immer größer. Wird ein Schwellwert (im Simulator auf 300 festgelegt) überschritten, so wird das Ende der Transmission angenommen und im Status-Register vermerkt. Zusätzlich wird die Methode IRQ aufgerufen, falls der entsprechende Interrupt aktiviert ist.

Die Methode receive ist für die Bereitstellung von empfangenen Daten zuständig. Auch hier wird bei einem Aufruf zuerst geprüft, ob der Empfangsteil aktiviert ist. Bei positivem Ergebnis wird anschließend überprüft, wie lange der letzte Empfang zurückliegt. Ist erst kürzlich ein Datum empfangen worden, so muss der nächste Empfang verzögert werden, sodass die Software die Daten verarbeiten kann.

Der Mechanismus, der bei der Verzögerung verwendet wird, ist schon aus der Methode transmit bekannt. Eine interne Zählvariable wird beim Empfang von Daten auf Null gestellt. Jeder folgende Aufruf der Methode receive erhöht diesen Zähler und kehrt anschließend zurück, solange die Variable zu niedrig ist. Ist genügend Zeit verstrichen, das heißt der Wert der Zählvariable hat einen Schwellwert erreicht, so kann das nächste Byte empfangen werden. Der Schwellwert ist im Programm auf 160 Schleifendurchgänge festgelegt. Dieser Wert wurde experimentell so bestimmt, dass die Zeit meist ausreicht, um das empfangene Byte zu verarbeiten.

Für den Empfang des nächsten Bytes wird zuerst die Methode DataAvailable aufgerufen, um zu überprüfen, ob schon ein Datum vorhanden ist. Ist das nicht der Fall, so wird dem IRI-Objekt mitgeteilt, dass jetzt wieder ein Byte empfangen werden darf. Die Mitteilung erfolgt durch einen Aufruf der Methode Suspended mit dem Parameter false. Die Wirkung dieses Aufrufes wird später in der Beschreibung der Klasse IRI noch erläutert.

Ist doch ein Byte empfangen worden, so wird es vom IRI-Objekt abgeholt und im Daten-Register bereitgestellt. Anschließend wird geprüft, ob das "*Receive Data Register Full*"-Bit gesetzt ist. Dies wäre ein Zeichen dafür, dass das vorherige Datenbyte überschrieben wurde, ohne dass es von der Software abgeholt wurde. Ist das der Fall, so wird das Überlauf-Bit im Status-Register der Schnittstelle gesetzt und, falls gewünscht, ein Interrupt ausgelöst. Ist das "*Receive Data Register Full*"-Bit hingegen gelöscht, so wird es nun gesetzt, da wieder Daten für die Software vorhanden sind. Außerdem wird ein Interrupt simuliert, falls er aktiviert ist.

4.8.2.2 Klasse IRI

Die Klasse IRI realisiert die Datenübertragung zwischen Simulator und Gastsystem. Wie schon erwähnt erfolgt der Datenaustausch durch ungesicherte Datagramme (UDP).

Bei der Konstruktion eines IRI-Objekts übergibt man die Portnummer, an der auf Datagramme gewartet werden soll. Zur Verwaltung dieses Kommunikationsendpunktes wird eine Instanz der Java-Klasse DatagrammSocket eingerichtet. Um die Vollduplex-Fähigkeit der Schnittstelle zu gewährleisten wird ein weiteres Objekt dieser Klasse zum Senden von Daten verwendet.

Um Daten zu übertragen ruft das SCI-Objekt die Methode send der Klasse IRI auf und gibt ein Datenbyte mit. Dieses wird in ein UDP-Paket gepackt und dem DatagrammSocket-Objekt zum Versenden übergeben. Da für UDP keine Verbindung zwischen zwei Partnern aufgebaut wird, werden alle ausgehenden Pakete an den Sender des letzten empfangenen Pakets geschickt.

Für den Empfang wird die Klasse IRI als Thread ausgeführt. Dieses Vorgehen wurde gewählt, da die Methode receive der DatagrammSocket-Klasse, welche zum Empfang von Daten verwendet wird, solange blockiert, bis ein Datum empfangen wurde oder eine gewisse Zeit verstrichen ist. Der Simulationsthread sollte aber nicht blockiert werden um auf Daten zu warten.

In der run-Methode, welche die Aktivität des Threads definiert, wird in einer Endlosschleife auf ein Paket gewartet. Sobald Daten eingetroffen sind, wird die interne Variable data_available auf true gesetzt, um das Vorhandensein von Daten anzuzeigen. Gleichzeitig wird noch eine zweite interne Variable mit dem Namen threadSuspended auf true gesetzt. Danach wird der Aktivitätsträger angehalten, da ein neu eintreffendes Datum das alte überschreiben würde. Aktiviert wird der Thread erst wieder, sobald die Variablen data_available und threadSuspended wieder auf false stehen.

Die erstgenannte Variable wird durch einen Aufruf der Methode receive auf false zurückgesetzt, da durch diese Funktion das Datenbyte abgeholt wird. Die Variable threadSuspended dient dazu, den Empfang neuer Daten zu verhindern. Zum Verändern dieser Variable steht die Methode Suspended zur Verfügung.

Abbildung 4.23 zeigt, wie die Klasse SCI ein IRI-Objekt verwendet, um Daten zu empfangen. Zu Beginn wird angenommen, dass der Empfangsthread Daten empfangen hat.

Um festzustellen, ob Daten vorhanden sind, ruft die Klasse SCI zuerst die Methode DataAvailable auf, die den Wert der gleichnamigen Variable zurückliefert. Steht diese Variable

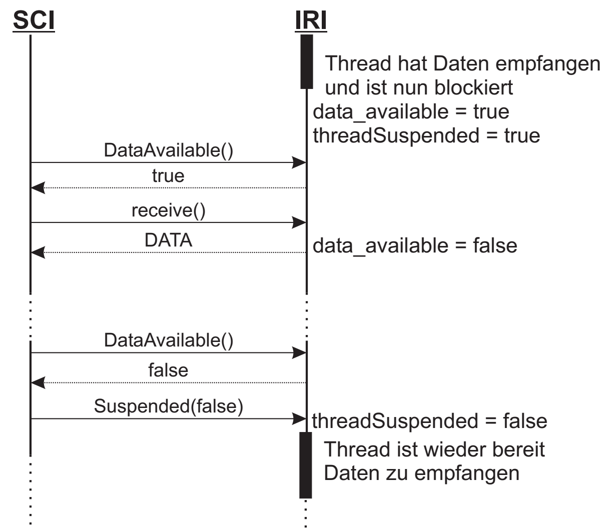


Abbildung 4.23: Verwendung der Klasse IRI durch die Klasse SCI

auf true, so ist ein Datenbyte vorhanden. Der Empfangsthread ist zu diesem Zeitpunkt sowohl durch die Variable threadSuspended als auch durch data_available blockiert.

Die Klasse SCI ruft dann die Methode receive auf, um das Datum abzuholen und im Daten-Register zur Verfügung zu stellen. Da die Variable data_available dadurch zurückgesetzt wurde, ist der Empfangsthread nun nur noch durch die Variable threadSuspended gestoppt.

Sobald ein neues Datum empfangen werden soll, wird wieder die Methode DataAvailable aufgerufen. Das Ergebnis ist jedoch negativ, da inzwischen keine Daten empfangen werden konnten, weil der Empfangsthread blockiert war. Deshalb wird die Methode Suspended mit dem Parameter false aufgerufen. Dadurch wird die private Variable threadSuspended auf false gesetzt, wodurch der Aktivitätsträger reaktiviert wird und somit wieder auf die Ankunft eines neuen Bytes wartet. Ein erneuter Aufruf von DataAvailable kann nun unter Umständen schon ein positives Ergebnis liefern.

Durch die Variable threadSuspended wird somit sichergestellt, dass keine UDP-Pakete empfangen werden können, wenn der Empfangsteil der seriellen Schnittstelle abgeschaltet ist oder sie auf die Abholung der Daten wartet.

Kapitel 5

Verbesserungsmöglichkeiten

In diesem Kapitel sollen einige Möglichkeiten aufgezeigt werden, wie der Simulator verbessert werden könnte. Die Hauptansatzpunkte sind dabei die Geschwindigkeit und die Simulationsgenauigkeit. Zusätzlich werden Bereiche genannt, um die man die Simulation erweitern könnte.

5.1 Verbesserung der Geschwindigkeit

5.1.1 Beschleunigung des LC-Displays

Um die Geschwindigkeit des Simulators zu verbessern könnte man zum Beispiel die Ansteuerung des Displays beschleunigen. Dieser Ansatzpunkt ist erfolgversprechend, da der Mikrocontroller einen großen Teil der Zeit damit verbringt das I^2C-Bus-Protokoll mit dem Display-Treiber abzuwickeln.

Zur Ansteuerung des Displays stellt das ROM eine Funktion zur Verfügung, welche die Daten aus einem festgelegten Speicherbereich, im RAM, zum Display-Treiber überträgt. Eine Beschreibung der Funktionen des LEGO-ROMs findet man bei [Pro98a].

Zur Beschleunigung könnte man eine Alternative zur Klasse LCD_driver entwerfen, die einen Aufruf von der oben beschriebenen Funktion abfängt und die Daten aus dem Speicherbereich direkt der Klasse LCD zuführt. Die Daten müssten dann nicht mehr über den simulierten I^2C Bus zum Display übertragen werden.

Nachteil dieser Methode wäre, dass der Simulator dann auf die Verwendung einer bestimmten Version des ROM-Codes angewiesen ist. Sollte in zukünftigen Versionen des LEGO-Kontrollbausteins neue, leicht veränderte ROM-Funktionen enthalten sein, so müsste der Simulator angepasst werden, um ein Abbild des neuen ROMs verwenden zu können.

Ein weiterer Nachteil der alternativen Klasse LCD_driver ist, dass Programme, die das Display direkt ansteuern, nicht mehr korrekt arbeiten würden. Um diesen Punkt zu entkräften, könnte man die besprochene Methode nicht als Alternative, sondern zusätzlich zur schon vorhandenen Klasse LCD_driver zu implementieren. Das simulierte Display könnte dann Daten sowohl über den simulierten I^2C Bus, als auch durch einen Aufruf der ROM-Funktion erhalten.

5.1.2 Beschleunigung der Speicherverwaltung

Ein anderer Ansatzpunkt, die Simulation zu Beschleunigen bietet die Adresszuordnung. Wie in Abschnitt 4.2.2 erklärt wird, führt jeder Zugriff auf eine Adresse zu einer sequentielle Suche durch alle registrierten MemoryRegion-Objekte. Da eine sequentielle Suche relativ viel Zeit in Anspruch nimmt, könnte man eine effektivere Strategie implementieren.

Eine Möglichkeit wäre der Aufbau eines Suchindex mit der Anfangsadresse als Primärschlüssel. Da ein MemoryRegion-Objekt niemals entfernt wird, bietet sich zum Beispiel ein B^{+}- Baum an. Allerdings ist der Einsatz eines komplexen Suchindex nur sinnvoll, wenn die Menge der Objekte relativ groß ist. Sind hingegen nur wenige Objekte vorhanden, so kann die durchschnittliche Such-Zeit sogar vergrößert werden. Welcher Suchindex sich für die hier vorliegende Situation eignet, müsste an konkreten Beispielen getestet werden.

Vorteil einer Optimierung an dieser Stelle ist jedoch, dass sich schon ein kleiner Geschwindigkeitsgewinn bei der Adressauflösung deutlich auf die Leistung des gesamten Systems auswirken kann, da jeder Befehl mindestens einen Speicherzugriff benötigt.

5.2 Erhöhung der Simulationsgenauigkeit

Die größte Abweichung der Simulation vom Original-Mikrocontroller stellt das zeitliche Verhalten der integrierten Module dar. So wird zum Beispiel jeder Zähler nach jedem Befehl erhöht. Dies entspricht aber nicht dem Original. Dort hat man die Möglichkeit die Geschwindigkeit in Abhängigkeit vom Systemtakt einzustellen.

Um diesem Rechnung zu tragen, könnte man versuchen den Systemtakt besser zu simulieren. Dazu wäre es nötig, die Ausführungsdauer eines Befehls zu berücksichtigen. Zu diesem Zweck könnte beispielsweise eine Tabelle angelegt werden, welche die Anzahl der Takte, die ein Befehl benötigt, enthält. In der Hauptschleife könnte somit nach einem fetch-decode-execute Zyklus die Anzahl der vergangenen Takte bestimmt werden. Diese Information könnte den Zusatzmodulen nach jedem Befehl mitgeteilt werden.

Ein Zähler könnte so das Zahlregister erst erhöhen, wenn genügend Zeit vergangen ist um so die gewählte Geschwindigkeit zu simulieren. Bei der seriellen Schnittstelle könnte man mit diesen Informationen die Baudrate simulieren, mit der die Datenübertragung stattfinden soll. Dazu bestimmt man aus einer angenommenen Taktfrequenz von zum Beispiel 16 MHz und der eingestellten Baudrate die Anzahl an Takten, die ein Byte bei der Übertragung benötigen soll. Zwischen dem Empfang bzw. dem Versenden von zwei Bytes sollte dann die errechnete Anzahl an Takten gewartet werden.

Einen völlig anderen Ansatz verfolgt die Idee, jedem Zusatzmodul einen eigenen Aktivitätsträger zuzuordnen. Sie wären dann nicht mehr von den Durchläufen der Hauptschleife abhängig, sondern könnten unabhängig davon ihre Aktionen durchführen.

Ein Nachteil dieses Ansatzes ist jedoch, dass die Module synchronisiert werden müssen. Das könnte zum Beispiel durch einen speziellen Thread erfolgen, der Informationen über den Takt zur Verfügung stellt. Soll ein Zähler beispielsweise mit dem halben Systemtakt betrieben werden, so müsste der Zähler-Thread sich mit diesem Zeit-Thread abstimmen, um eine genaue Simulation zu erreichen.

5.3 Erweiterung des Simulationsumfangs

Bei der Beschreibung der Simulation wurden an einigen Stellen Abstriche erwähnt. Die Abweichungen gegenüber dem Original lassen sich dabei in zwei Gruppen einteilen. Zum einen die Komponenten, die nicht implementiert wurden, da sie von der bekannten Software nicht genutzt werden. Zum anderen die Komponenten, die durch die Hardware des Lego-Bausteins nicht genutzt werden.

Vertreter der ersten Gruppe sind zum Beispiel der nicht implementierte Watchdog-Timer und die Abstriche bei der Fähigkeiten des Display-Treibers ein Segment blinkend darzustellen. Aber auch eine bisher nicht genannte Abweichung im Befehlssatzes der CPU. Drei Befehle des Original-Prozessors wurden für die Simulation nicht implementiert. Dies sind zum einen die Befehle `daa` und `das`, die zur Addition bzw. Subtraktion von BCD-Zahlen verwendet werden und zum anderen der Befehl `eepmov` um Daten in ein EEPROM zu schreiben.

Bei der zweiten Gruppe handelt es sich hauptsächlich um Module bzw. Fähigkeiten des Mikrocontrollers, die nicht benutzt werden können, da die entsprechenden Pins im Lego-Kontrollbaustein nicht belegt sind. Dazu zählt beispielsweise die Möglichkeit in den *Hardware-Standby* Modus zu wechseln, oder die Fähigkeit des 16-bit Zählers ein externes Signal zu einer benutzerdefinierten Zeit festzuhalten.

Eine sinnvolle Erweiterung der Simulation kann man vor allen durch die Implementierung von Vertretern der ersten Gruppe erreichen, da jede neue Firmware diese Möglichkeiten nutzen könnte. Eine Simulation von Komponenten der zweiten Gruppe ist nur sinnvoll wenn Lego die Hardware des Kontrollbausteins ändert, oder man eine vollständige Simulation des Mikrocontrollers benötigt.

Kapitel 6

Zusammenfassung

Ziel der vorliegenden Arbeit war es einen Simulator für den LEGO-Kontrollbaustein RCX zu erstellen, mit dessen Hilfe man in der Lage sein soll, verschiedene Betriebssysteme für das eingebettete System zu testen.

Der in der Arbeit entwickelte Simulator erfüllt diese Anforderung. Er interpretiert den Befehlssatz der Hitachi H8/300 CPU und kann somit alle für diesen Prozessor kompilierten Programme ausführen. Zusätzlich werden auch die integrierten Module des H8/3292 Mikrocontrollers, soweit sie vom LEGO-Kontrollbaustein verwendet werden, simuliert. Darunter fallen zum Beispiel die Zähler, die von der Firmware zur Erzeugung regelmäßiger Interrupts genutzt werden können. Auch die serielle Schnittstelle wird simuliert, um eine Firmware nachzuladen. Für die Simulation der Sensoren wurde der AD-Wandler implementiert, um die Werte der Software zur Verfügung zu stellen. Des Weiteren werden auch die zusätzlichen Komponenten des Kontrollbausteins simuliert. Dabei ist speziell das Display zu nennen. Es wurde dem Original nachempfunden und wird durch einen simulierten Display-Treiber angesprochen. Auch die Motor-Treiber werden simuliert um die Ansteuerung der Motoren zu erfassen.

Die Simulation auf Befehlssatzebene ermöglicht es somit jedes Programm für den LEGO-Kontrollbaustein ohne Änderungen auf dem Simulator ablaufen zu lassen. Der Simulator kann deshalb die gesamte Softwareentwicklung für den Baustein, von der Entwicklung eines neuen ROM-Codes bis zur Gestaltung eines Betriebssystems, unterstützten. Für die Entwicklung eines neuen ROMs ist zusätzlich die leichte Austauschbarkeit des ROM-Inhaltes durch Laden eines ROM-Abbildes von Vorteil.

Da der Simulator in JAVA implementiert wurde, ist er darüber hinaus relativ unabhängig von der Hardware des Gastsystems. Somit ist es möglich das simulierte System auf vielen unterschiedlichen Plattformen zur Verfügung zu stellen.

Allerdings ist der Simulator gegenüber dem realen System relativ langsam. Auf einer SUN Ultra SPARC 10 mit 300 MHz läuft ein Programm im Simulator etwa 20 mal langsamer ab als im Original-LEGO-Baustein. Ein Grund dafür ist die Art der Simulation, denn jeder Befehl des Hitachi-Mikrocontrollers wird durch mehrere Zeilen JAVA Code simuliert. Hinzu kommt, dass JAVA-Programme von einer Java Virtual Machine interpretiert werden. Der Simulator wird dadurch nochmals langsamer. Außerdem spielte die Geschwindigkeit eher eine zweitrangige Rolle bei der Entwicklung. Durch eine geschicktere Implementierung könnte man die Simulation daher sicherlich beschleunigen.

Anhang A

Bedienungsanleitung

A.1 Kompilierung des RCX Simulators

Zum Betrieb des RCX Simulators ist JAVA2 oder JAVA 1.1 mit zusätzlichen SWING Komponenten erforderlich. Das Kompilieren der Quellen sollte durch die Eingabe von

```
make
```

erfolgen. Fehlermeldungen wie zum Beispiel:

```
Package javax.swing not found in import.
```

oder

```
java.lang.NoClassDefFoundError: javax/swing/JButton
```

deuteten auf fehlende SWING Komponenten hin. Man sollte in diesem Fall die Installation der JAVA-Umgebung überprüfen.

A.2 Starten des RCX Simulators

Nach erfolgreicher Erstellung der Binärcode-Dateien kann man den Simulator durch Eingabe von

```
java Simulator -r ROM.srec -f firm0309.lgo
```

starten. Nach dem Parameter `-r` spezifiziert man dabei die Datei des ROM-Abbildes, nach `-f` sollte der Name des Firmware-Abbildes folgen, welches beim Starten des Simulators geladen werden soll. Diese Angabe ist optional, eine andere Möglichkeit die Firmware zu laden wird im Abschnitt A.4.2 erläutert.

Neben den schon erwähnten Parametern kennt der Simulator noch einige weitere:

Nach `-u` kann man den UDP-Port angeben, an dem der Simulator die Eingaben der seriellen Schnittstelle erwartet. Fehlt diese Angabe wird der Port 8000 verwendet.

Durch den Parameter `-p` kann man die Priorität der Simulationsthreads festlegen. Standardmäßig wird die niedrigste Priorität verwendet, da so das Java Framework genügend Rechenzeit bekommt um die Bildschirmausgabe korrekt darzustellen. Auf Wunsch kann man die Priorität jedoch erhöhen, muss dann aber mit einer falschen Bildschirmdarstellung rechnen.

Die Angabe von `-d` führt dazu, dass ein Debug-Fenster (siehe Abbildung A.1(b)) angezeigt wird. Dieses zeigt die Datenübertragung zum Display-Treiber an und eine Meldung, falls sich der simulierte Mikrocontroller in einen Energiesparmodus begibt. Außerdem kann man durch sechs Schalter verschiedene Debugausgaben auf der Standardausgabe hervorrufen. Das wäre zum einen eine Ausgabe des Befehls, welcher gerade abgearbeitet wurde, zusätzlich kann man den Inhalt aller Register anzeigen lassen. Da diese Daten ziemlich rasch über den Bildschirm wandern, sorgt ein weiterer Schalter dafür, dass nach jedem Befehl eine Taste gedrückt werden muss. Darüber hinaus kann man jeden Interrupt und alle Zugriffe zu MMappedPeriphery-Objekten sowie die Kommunikation über die serielle Schnittstelle am Bildschirm protokollieren lassen.

Jede dieser Optionen lässt sich auch von der Kommandozeile aus aktivieren. Dazu dienen die Parameter `-T`, `-V`, `-S`, `-I`, `-C` und `-P` (siehe auch Tabelle A.1).

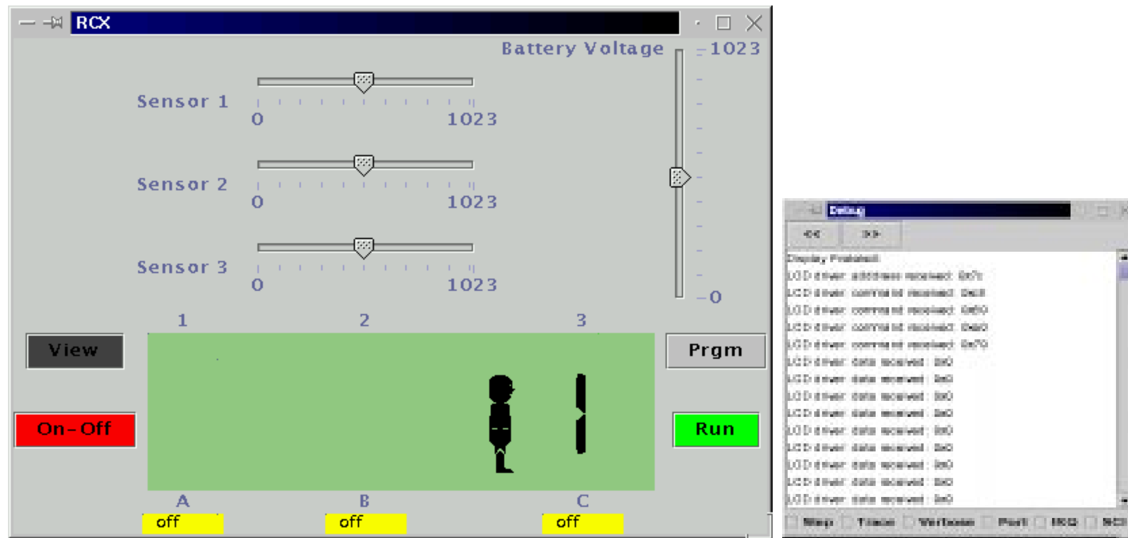
-r ROMFILE	Name der ROM-Abbildes festlegen
-f FIRMWARE	Name eines Firmware-Abbildes festlegen
-u PORT	Port der seriellen Schnittstellen Simulation festlegen
-p PRIORITY	Priorität des Simulationsthread festlegen
-d	Debug Fenster anzeigen
-T	Jeden Befehl anzeigen
-V	Registerinhalt nach jedem Befehl anzeigen
-S	nach jedem Befehl auf einen Tastendruck warten
-I	Interrupts anzeigen
-C	Protokoll der seriellen Schnittstelle anzeigen
-P	Zugriffe auf die Register der Peripherie anzeigen

Tabelle A.1: Kommandozeilenparameter des RCX Simulators

Um die Debugausgaben verändern zu können während das Debug-Fenster nicht angezeigt wird, akzeptiert der Simulator auch Tastatureingaben. Die Tasten zum Aktivieren der Debugausgaben entsprechen dabei den Buchstaben der Kommandozeilenparameter.

A.3 Bedienung des Simulators

Nach erfolgreichem Start des Simulators erscheint das Hauptfenster und ggf. auch das Debug-Fenster. Abbildung A.1 zeigt ein typisches Erscheinungsbild.



(a) Hauptfenster

(b) Debug-Fenster

Abbildung A.1: Erscheinungsbild des RCX Simulators

Das Hauptfenster zeigt als zentrales Element das Display des Kontrollbausteins. Links und rechts daneben befinden sich jeweils zwei Buttons. Die Funktion dieser Tasten ist von der Firmware abhängig. Für gewöhnlich dienen sie jedoch zum Auswählen und Starten des gewünschten Programms, zum Ein-/Ausschalten des Bausteins und zur Überprüfung der Sensoren.

Aktivieren lassen sich die Tasten durch Anklicken mit der linken Maustaste. Solange man die Maustaste gedrückt hält, ist der entsprechende Button auch für den Simulator gedrückt. Da es manchmal erforderlich ist, zwei Buttons gleichzeitig zu drücken, kann man durch Anklicken mit der rechten Maustaste einen Button feststellen. Er erscheint dunkel dargestellt und bleibt für die Simulation solange gedrückt, bis der Button mit der linken Maustaste wieder gelöst wird.

Über dem Display befinden sich drei Schieberegler. Diese dienen der Simulation der Sensoren, die am Kontrollbaustein angeschlossen werden können. Mit jedem Regler lässt sich ein Wert zwischen 0 und 1023 einstellen, der je nach erwartetem Sensor unterschiedliche Bedeutung hat. Erwartet das Programm im RCX zum Beispiel einen Lichtsensor, so entspricht der Wert der Lichtintensität. Für einen Tastsensor würde ein kleiner Wert bedeuten, dass der Taster nicht gedrückt ist, ein großer Wert würde einen gedrückten Tastsensor darstellen.

Auf der rechten Seite befindet sich darüber hinaus noch ein vierter senkrechter Schieberegler. Mit diesem Regler kann man den Wert beeinflussen, den der Kontrollbaustein erhält, wenn er die Batteriespannung überprüft. Die Original-Firmware beispielsweise schaltet den Kontrollbaustein bei zu geringer Versorgungsspannung ab.

Unterhalb des Displays sind drei Felder, die je den Zustand eines Motors anzeigen. Mögliche Zustände sind: off, forward, reverse und stop.

A.4 Laden der Firmware

Für das Laden eines Betriebssystems sind zwei Wege vorgesehen. Zum einen kann man ein Firmware-Abbild über die simulierte serielle Schnittstelle übertragen (siehe Abschnitt A.4.2). Dabei ist es erforderlich, dass das ROM diese Möglichkeit anbietet. Da dieses der übliche Weg für den Original-Baustein ist, ein Betriebssystem zu laden, erfüllt der Original-Lego-ROM-Code diese Anforderung.

Der zweite Weg ein Firmware-Abbild zu laden gleicht dem Laden des ROM-Abbildes. Dabei wird die Abbild-Datei beim Starten des Simulators angegeben (siehe Abschnitt A.2) und der Firmware-Code direkt in den simulierten Speicher gelesen. Allerdings ist auch hier zu beachten, dass der ROM-Code diesen Fall unterstützen muss, um die Firmware zu starten (siehe Abschnitt A.4.1).

A.4.1 Veränderungen am ROM

Für den normalen Betrieb des Simulators sind keine Änderungen am Original-ROM-Code nötig. Wenn man sich jedoch dafür entscheidet, gleich beim Starten des Simulators eine Firmware-Datei einzuladen, so muss man beachten, dass diese Firmware nie ausgeführt wird, wenn der Original-Lego-ROM-Code verwendet wird. Dieser möchte die Firmware immer über die serielle Schnittstelle laden. Damit die Firmware dennoch ausgeführt wird, müssen Veränderungen am ROM-Code vorgenommen werden. Der Patch `ROMPatch.diff`[1] führt eine Änderung des ROMs durch, sodass die Programmausführung nach dem Initialisieren der Geräte immer an der Speicherstelle 0x8000 (Startpunkt der Firmware) fortgesetzt wird.

Ein Aufruf von

```
patch ROM.srec ROMPatch.diff
```

wendet den Patch auf `ROM.srec` an.

A.4.2 Firmware Downloader

Um die Firmware über den "normalen" Weg in den Lego-Baustein zu laden, steht das Java Programm `FWdown` zur Verfügung[2]. Als Parameter wird der Name der Firmware Datei erwartet und eventuell noch die Nummer des UDP-Ports, an den die Daten gesendet werden sollen. Der Aufruf

```
java FWdown -f firm0309.lgo -p 8000
```

sollte zum Beispiel die Original-Lego-Firmware zum Simulator übertragen.

Gibt man noch `-d` an, so werden die übertragenen und empfangenen Bytes auf dem Bildschirm angezeigt.

[1] im Unterverzeichnis util\patches des Simulators
[2] im Unterverzeichnis util\FWdown des Simulators

Mit den Parametern `-s` und `-b` beeinflusst man die Übertragung der Firmware. Durch `-s` lässt sich die Zeit festlegen, die das Programm nach jedem übertragenem Byte warten soll, bevor das nächste übertragen wird. Wählt man einen zu niedrigen Wert, so können Daten verloren gehen, da der Simulator sie eventuell nicht schnell genug bearbeiten kann. Bei einer zu großen Pause kann es vorkommen, dass die Firmware im Simulator zu lange auf das nächste Byte warten muss und deshalb das Ende der Übertragung annimmt. Der Standardwert beträgt 300 Millisekunden.

`-b` legt die Größe der Blöcke fest, in die das Firmware-Abbild zur Übertragung segmentiert wird. Wird eine zu kleine Blockgröße gewählt, so steigt der Overhead, da jedes Paket von der Firmware bestätigt wird. Allerdings darf man auch nicht zu große Pakete wählen, da dann die Wahrscheinlichkeit steigt, dass ein Paket fehlerhaft übertragen wird. Die Standard Paketgröße beträgt 240 Bytes; bei Problemen sind kleinere Pakete zu empfehlen.

-f FIRMWARE	Name des Firmware-Abbildes festlegen
-p PORT	Portnummer des Simulators angeben
-d	Debug-Ausgaben anzeigen
-s DELAY	Zeit (in ms) zwischen zwei Übertragungen festlegen
-b SIZE	Blockgröße für Übertragung festlegen

Tabelle A.2: Kommandozeilenparameter des Firmware-Downloaders

Für das Nachladen von Verhaltensprogrammen, wenn es die Firmware unterstützt, ist noch kein Programm verfügbar. Es sollte jedoch prinzipiell möglich sein, ein vorhandenes Programm für den Betrieb mit dem Simulator anzupassen.

Will man Programme für legOS testen, hat man alternativ die Möglichkeit die Programme statisch mit der Firmware zu binden. Dazu kann man zum Beispiel eine ältere Version von legOS verwenden, die nur das statische Binden zulässt. Allerdings sollte man eine aktuelle Version von legOS bevorzugen, da zum Beispiel die Version 0.17 von legOS noch relativ viele Fehler enthält.

Bei aktuellen Versionen von legOS, die das Nachladen von Benutzerprogrammen unterstützen ist eine kleine Änderung nötig, um Benutzerprogramme statisch zubinden. So muss bei der Version 0.2 die Zeile `#define CONF_PROGRAM` in der Datei `boot/config.h` auskommentiert werden. Anschließend kann durch eine entsprechende Änderung der Datei `Makefile.kernel` ein Benutzerprogramm mit dem Kernel gebunden werden.

Anhang B

Software-Architektur des RCX Simulators

Auf den folgenden Seiten sind Diagramme der meisten Klassen abgebildet. In Abbildung B.1 sieht man die Klasse Simulator mit dem Hauptobjekt Microcontroller. Außerdem wird der Konstruktor der Klasse SRecordLoader dargestellt.

Abbildung B.2 zeigt das Microcontroller-Objekt und die wichtigsten Objekte, die durch dieses Klasse erzeugt werden. Einige Klassen sind aus Platzgründen nicht genau gezeichnet, in Abbildung B.4 wird das jedoch nachgeholt. Die Diagramme B.3 und B.4 zeigen einige von MemoryRegion abgeleiteten Klassen im Detail.

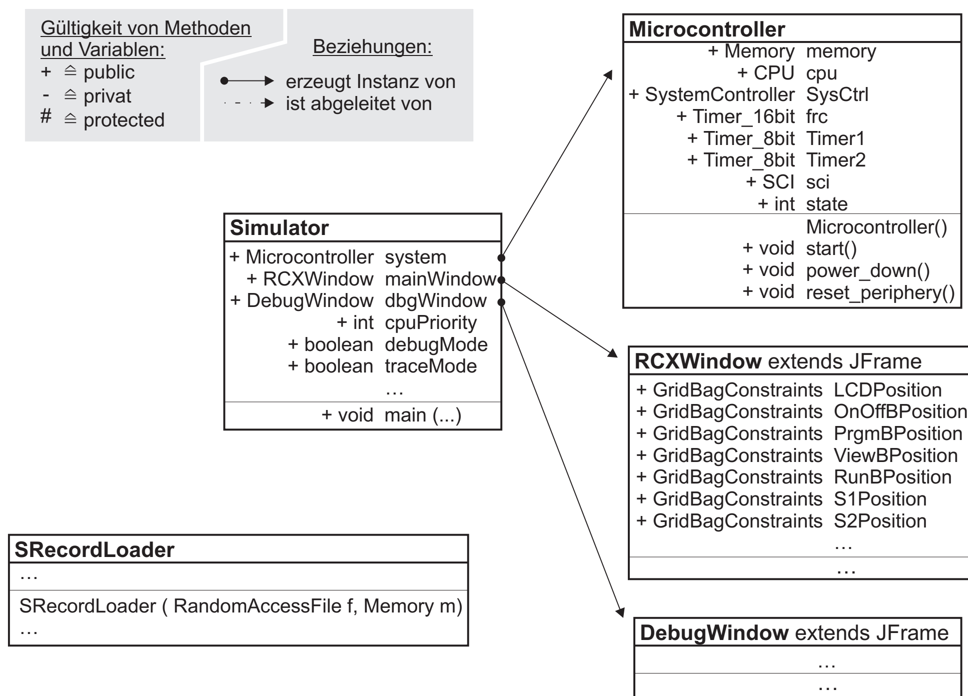


Abbildung B.1: Die Klasse Simulator und ihre Objekte

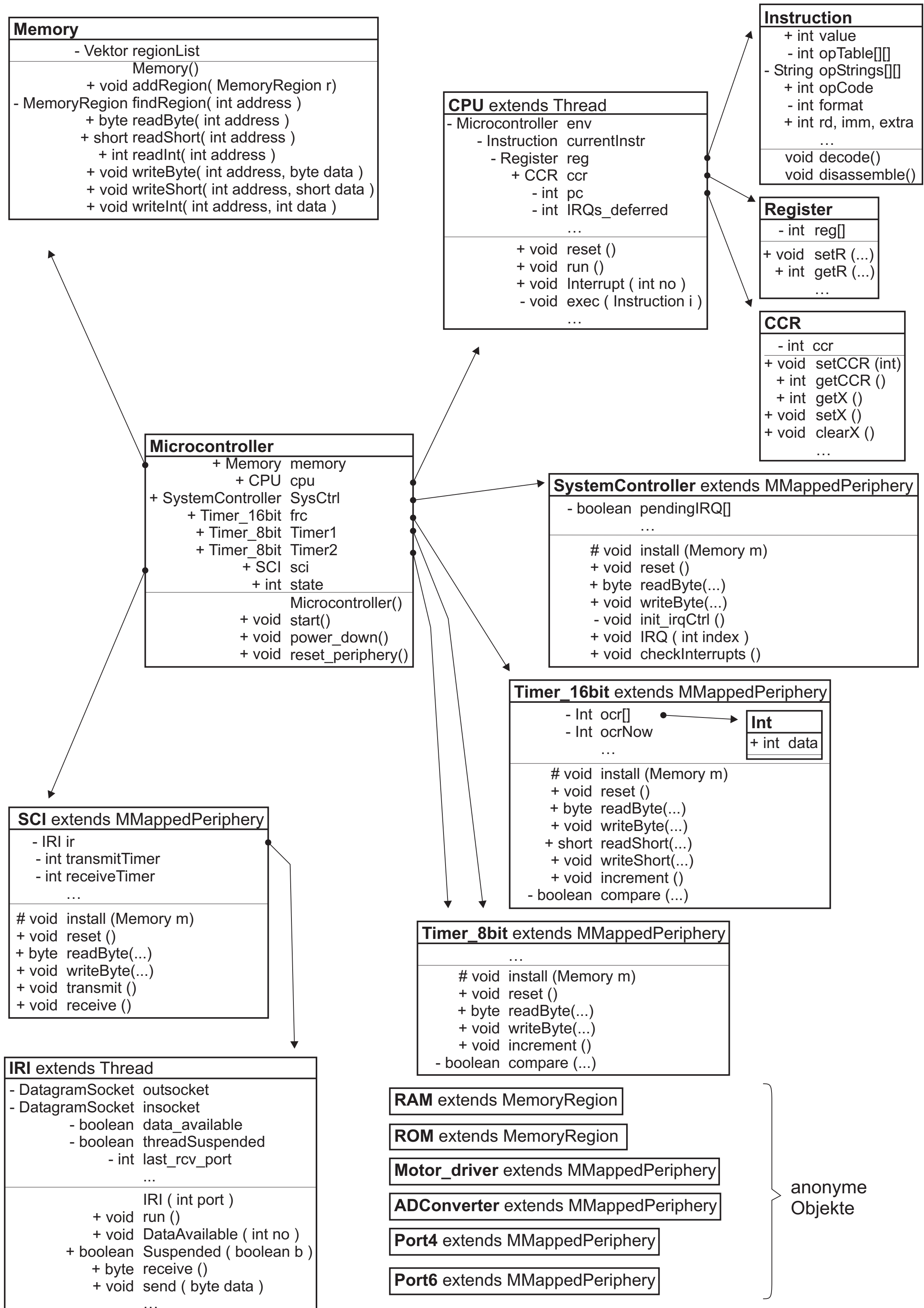


Abbildung B.2: Die Klasse Microcontroller und ihre Objekte

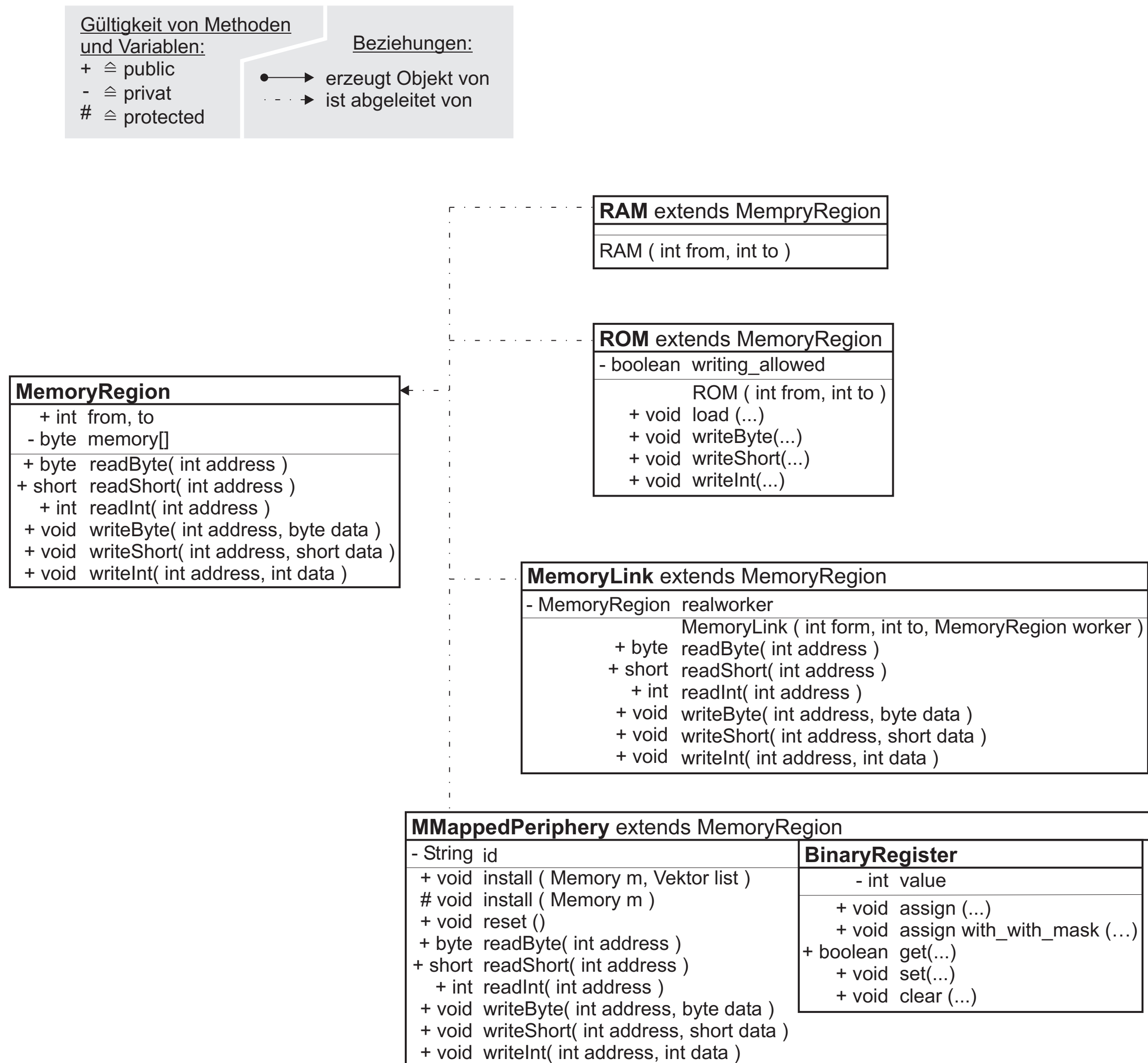


Abbildung B.3: von MemoryRegion abgeleitete Klassen

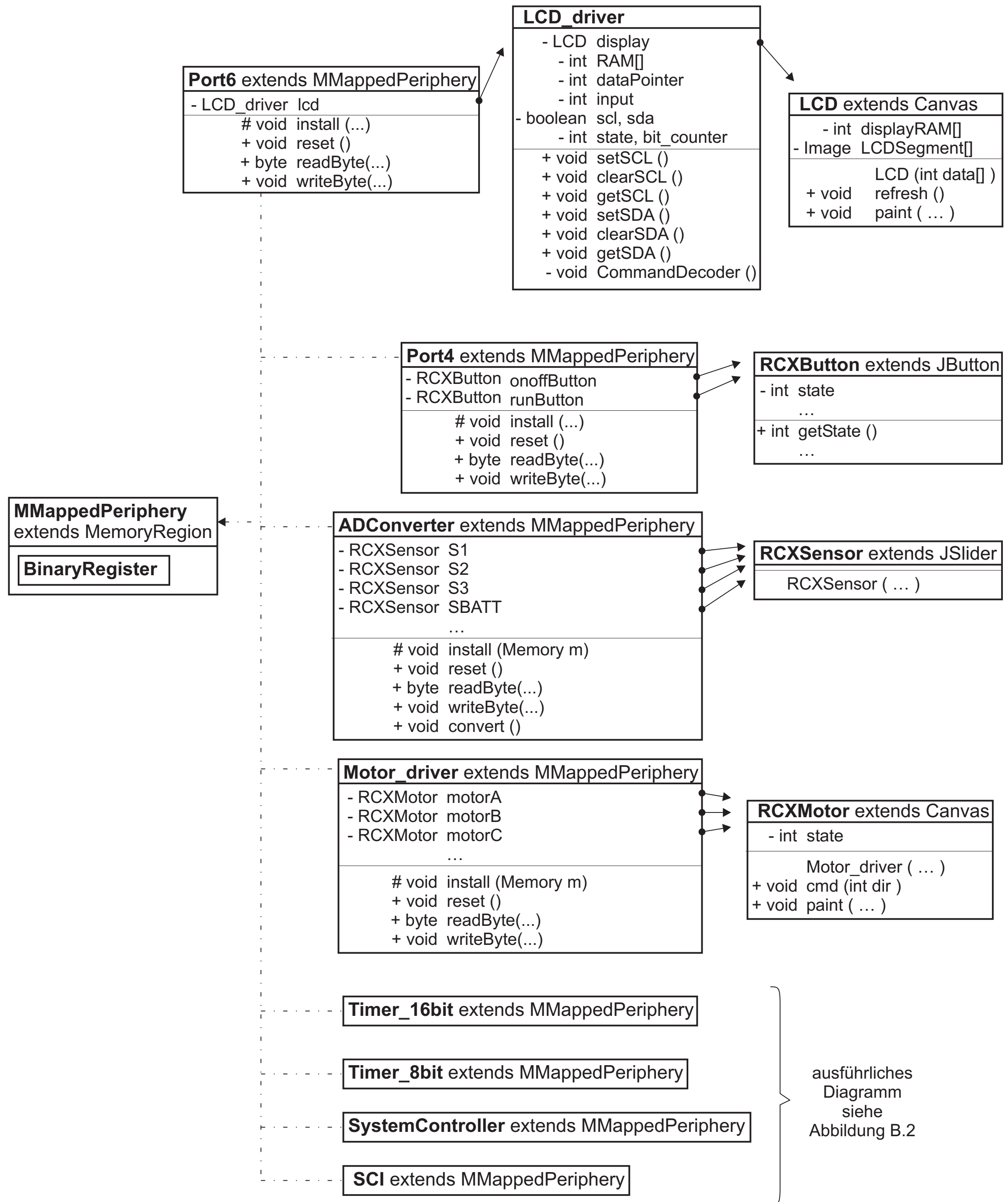


Abbildung B.4: von MMappedPeriphery abgeleitete Klassen

Anhang C

Erstellen eines ROM Abbildes

Zum Starten des Simulators benötigt man ein ROM-Abbild. Da von LEGO auf diesen Code ein Copyright besteht, ist es nicht im Umfang des Simulators enthalten, sondern muss getrennt beschafft werden. Eine Möglichkeit dafür ist, das ROM aus einem vorhandenen RCX zu extrahieren, indem es über die Infrarot-Verbindung zum Computer übertragen wird. Dazu findet sich bei [Pro] ein Anleitung, die im Folgenden wiedergegeben wird.

Man benötigt:

- die Original-LEGO-Firmware; auf der LEGO Mindstorms CD unter \firm\firm0309.lgo zu finden
- eine Möglichkeit die Firmware zum RCX zu transferieren. Zum Beispiel das Programm firmdl3, welches legOS beiliegt oder unter `http://graphics.stanford.edu/~kekoa/rcx/firmdl.c` separat zu finden ist
- das Programm send, als Quellcode zu finden unter `http://graphics.stanford.edu/~kekoa/rcx/send.c`
- das Perl Skript getrom.pl, von `http://graphics.stanford.edu/~kekoa/rcx/getrom.pl`
- einen Perl 5 - Interpreter

Zuerst muss man die Firmware etwas verändern[1]. Dazu öffnet man die Datei firm0309.lgo mit einem normalen Texteditor und sucht nach einer Zeile, die mit:

```
S11396D0...
```

beginnt. Diese muss nun ausgetauscht werden durch folgende Zeile:

```
S11396D0B2D6AEA0ACB6A7A49BB8BFD0B23CAE3055
```

[1] Die folgenden Änderungen sind auch durch die Anwendung des Patches getROM.diff durchführbar. (zu finden im Unterverzeichnis util\patches des Simulators)

Den gleichen Vorgang wiederholt man mit der Zeile, die mit

```
S11396F0 ...
```

beginnt. Sie muss gegen

```
S11396F0A998BFF0AD6AA21CAF50AABCA38899FA7E
```

ausgetauscht werden. Schließlich sucht man noch nach der Zeile, die mit

```
S113BFD0...
```

anfängt und tauscht sie und die nächsten beiden Zeilen durch

```
S113BFD06E7E001C6E76001D6B86BFC06E7E001EDA
S113BFE06E76001F6B86BFC2FECB5A00A1F600001E
S113BFF06B06BFC26DF66B06BFC06DF65A00BB9AE6
```

aus. Jetzt kann man die veränderte Firmware speichern und zum RCX übertragen.

Die Veränderungen haben zwei Befehle der Firmware umgeleitet und zwar StartSubroutine-Download und GetMemoryMap. Startet man das Skript getrom.pl, so benutzt es diese Befehle um die Firmware zu veranlassen, den gesamten ROM Inhalt über die Infrarot-Schnittstelle zum PC zu übertragen. Dort wird es auf der Standardausgabe angezeigt, leitet man diese in eine Datei um, so erhält man ein ROM-Abbild. Folgender Befehl könnte das bewirken:

```
./getrom.pl >ROM.srec
```

Zu beachten ist, dass das Perl-Skript das send-Programm aufruft, um mit dem Lego-Kontrollbaustein zu kommunizieren. Es muss dazu vorher kompiliert werden und in ausführbarer Form vorliegen. Damit es auch gefunden wird, sollte es sich im gleichen Pfad wie das Perl-Skript befinden oder im Suchpfad enthalten sein. Es soll noch erwähnt werden, dass das Perl-Skript für eine UNIX Umgebung geschrieben wurde, sodass man eventuell einige Änderungen vornehmen muss, um es auch unter anderen Betriebssystemen einsetzen zu können.

Anhang D

Erstellen eines Crosscompilers für den H8/300 Prozessor

Um Programme für den LEGO-Kontrollbaustein schreiben zu können, benötigt man einen Crosscompiler, der das Programm in H8/300 Maschinencode übersetzt. Im Folgenden soll eine Anleitung gegeben werden, die das Vorgehen zur Erstellung eines Crosscompilers beschreibt. Diese Anleitung ist für Linux und Solaris Systeme gedacht. Eine etwas ausführlichere Anleitung findet man zum Beispiel bei den legOS-HOWTOs [Vil99]. Dort erhält man auch Informationen um einem Crosscompiler für Windows-Systeme zu erstellen.

Man benötigt:

- die Quellen des GNU C Compiler (z. B. egcs-1.1.2.tar.gz[1]) und
- den Quellcode der GNU binutils (z. B. binutils-2.9.1.tar.gz[2]).

Zuerst muss man die GNU binutils für den Zielprozessor erstellen. Dazu wechselt man in das Verzeichnis der binutils und gibt folgende Zeile ein:

```
./configure --target=h8300-hitachi-hms --prefix=/usr/local/crossgcc
```

dadurch werden die Makefiles erstellt und man kann mit

```
make all
make install
```

die binutils erstellen und installieren. Das Installationsverzeichnis wurde dabei durch den Parameter `prefix` des `configure`-Aufrufes festgelegt. Nach erfolgreichem Kompilieren sollten die GNU binutils in den Pfad aufgenommen werden, damit diese beim Kompilieren des Crosscompilers gefunden werden.

[1]verfügbar z. B. bei: ftp://ftp.uni-bayreuth.de/pub/packages/languages/egcs/releases/egcs-1.1.2/

[2]verfügbar z. B. bei: ftp://ftp.uni-bayreuth.de/pub/packages/gnu/binutils/

```
export PATH=/usr/local/crossgcc/bin:$PATH
```

bzw.

```
setenv PATH /usr/local/crossgcc/bin:$PATH
```

erledigt das, je nach verwendeter Shell. Nun kann man mit der Konfiguration des Compilers selbst beginnen:

```
cd egcs-1.1.2
./configure --target=h8300-hitachi-hms
                --prefix=/usr/local/crossgcc --enable-target-optspace
```

Bevor man den Compiler erstellen kann, müssen noch einige Änderungen am Makefile vorgenommen werden. Dazu öffnet man es mit einem Editor und sucht die Zeile, die mit `CFLAGS` beginnt. An diese Zeile fügt man `-Dinhibit_libc` am Ende an.

Nun kann der Compiler erstellt werden. Dazu dient das Kommando:

```
make cross LANGUAGES="c c++"
```

Die Kompilierung des Compilers dauert einige Zeit und bricht mit der Fehlermeldung:

```
checking whether the C compiler
    (/usr/tmp/egcs-1.1.2/gcc/xgcc -B /usr/tmp/egcs-1.1.2/gcc/ -g -Os ) works
configure: error: installation or configuration problem:
                                          C compiler cannot create executables.
```

ab. Diese Fehlermeldung kann ignoriert werden. Sie tritt auf, weil die Standardbibliothek für das Zielsystem nicht gefunden werden konnte.

Jetzt kann der Crosscompiler installiert werden. Ein Aufruf von:

```
make LANGUAGES="c c++" install
```

kopiert alle benötigten Dateien in das Verzeichnis, welches oben unter `--prefix` angegeben wurde.

Literaturverzeichnis

[AG96] Arnold, K. und J. Gosling: *JAVA Die Programmiersprache.* Addison-Wesley, Bonn, 2. Auflage, 1996.

[Fan] Fankhauser, G.: *A MIPS R3000 Simulator in Java.* http://www.tik.ee.ethz.ch/ gfa/MipsSim.html.

[FBF] Ferrari, M., M. Beri und M. Falco: *emulegOS Homepage.* http://www.geocities.com/ marioferrari/emulegos.html.

[Gol99] Golm, M.: *Tutorial Object-Orientates Concepts in Distributed Systems*, 1999. Übungsfolien zur Vorlesung.

[Her98] Herrod, S. A.: *Using complete machine simulation to understand computer system behavior.* Dissertation, Stanford University, Februar 1998. http://www-flash.stanford.edu/ herrod/docs/thesis-2sided.pdf.

[Hita] Hitachi Ltd.: *H8/300L Series Programming Manual.* http://semiconductor.hitachi.com/products/pdf/h3ltp001d1.pdf.

[Hitb] Hitachi Ltd.: *Hitachi Single-Chip Microcomputer H8/3297 Series H8/3297 HD6473297, HD64333297 H8/3296 HD6433296 H8/3294 HD6473294, HD6433294 H8/3292 HD6433292 Hardware Manual*, 3. Auflage. http://semiconductor.hitachi.com/products/pdf/h33th014d2.pdf.

[Int94] Integrated Device Technology Inc.: *The IDTR3051, R3052 RISController Hardware User's Manual*, 1994. http://www.idt.com/docs/79R3051_MA_65105.pdf.

[Kla89] Klar, R.: *Digitale Rechenautomaten.* Sammlung Göschen. Walter de Gruyter, Berlin, 4. Auflage, 1989.

[LEG] LEGO Inc.: *official LEGO Mindstorms Homepage.* http://www.legomindstorms.com/.

[MDG$^+$98] Magnusson, P. S., F. Dahlgren, H. Grahn, M. Karlson, F. Larsson, F. Lundholm, A. Moested, J. Nilsson, P. Stenström und B. Werner: *SimICS/sun4m: A VIRTUAL WORKSTATION.* Usenix Annual Technical Conference, New Orleans, Juni 1998. http://www.simics.com/publications/usenix98-simics.ps.gz.

[Mül] MÜLLER, F.: *LegoSim Homepage.*
http://www.informatik.hu-berlin.de/ mueller/legosim/.

[Mor97] MORISON, M.: *JAVA 1.1 für Insider.* SAMS, 1997.

[Nel] NELSON, R.: *Lego Mindstorms Internals.*
http://www.crynwr.com/lego-robotics.

[Nog] NOGA, M.: *legOS-0.2 Homepage.*
http://www.noga.de/legOS/.

[PH96] PATTERSON, D. A. und J. L. HENNESSY: *Computer Architecture a Quantitative Approach.* Morgan Kaufmann Publishers, Inc., San Francisco, 2. Auflage, 1996.

[PH98] PATTERSON, D. A. und J. L. HENNESSY: *Computer Organization & Design.* Morgan Kaufmann Publishers, Inc., San Francisco, 2. Auflage, 1998.

[Phi98] PHILIPS SEMICONDUCTORS: *PCF8566 Universal LCD driver for low multiplex rates*, Mai 1998.
http://www.semiconductors.com/acrobat/datasheets/PCF8566_6.pdf.

[Pro] PROUDFOOT, K.: *Getting a ROM Image.*
http://www.crynwr.com/lego-robotics/rom-image.html.

[Pro98a] PROUDFOOT, K.: *RCX Internals*, 1998.
http://graphics.stanford.EDU/ kekoa/rcx/.

[Pro98b] PROUDFOOT, K.: *Reverse Engineering the LEGO RCX.* EE380 Colloquium, Stanford University Computer Systems Laboratory, 7. Oktober 1998.

[RBDH97] ROSENBLUM, M., E. BUGNION, S. DEVINE und S. A. HERROD: *Using the SimOS Machine Simulator to Study Complex Computer Systems.* ACM Transactions on Modeling and Computer Simulation, 7(1):78–103, Januar 1997.
ftp://www-flash.stanford.edu/pub/hive/TOMACS96-simos.pdf.

[Sima] *SimICS Homepage.*
http://www.simics.com.

[Simb] *SimOS Homepage.*
http://simos.stanford.edu.

[Tra] TRADER, W.: *S-Record Format.*
http://www.hitex.com/chipdir/oth/srecord.txt.

[Vil99] VILLA, L.: *LegOS HOWTO*, Juli 1999.
http://arthurdent.dorm.duke.edu/legos/HOWTO/HOWTO.html.

[VMW] *VMWare Homepage.*
http://www.vmware.com.

[WIN] *WINE Homepage.*
http://www.winehq.com.