Entwurf und Implementierung einer statischen Java-Laufzeitumgebung für den LEGO Mindstorms RCX

Studienarbeit im Fach Informatik

vorgelegt von

Jörg Domaschka

geb. am 10.08.1981 in Treuchtlingen

Angefertigt am

Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme)
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:	***Prof. Dr. Wolfgang Schröder-Preikschat***
	Dipl-Inf. Meik Felser
Beginn der Arbeit:	01. April 2004
Abgabe der Arbeit:	29. Oktober 2004

Ich versichere, dass ich die 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, 7. Dezember 2004 _____

Kurzzusammenfassung

In allen heutigen PCs und Großrechnersystemen kommen Prozessoren mit Hardware basiertem Speicherschutz zum Einsatz. Im Gegensatz dazu ist diese Einheit auf den meisten Mikrocontrollern nicht vorhanden und muss durch Software simuliert werden. Eine andere Möglichkeit, dies zu erreichen, liegt im Einsatz einer typsicheren Programmiersprache wie zum Beispiel Java, das auch vom Betriebssystem JX verwendet wird. JX realisiert jeglichen Speicherschutz ohne Hardware und stellt deshalb eine gute Ausgangsbasis für eine Portierung auf einen Mikrocontroller dar. Hierbei handelt es sich im vorliegenden Fall um den LEGO Mindstorms RCX. Da auf einem Mikrocontroller wenig Speicher zur Verfügung steht, kann das vorliegende System nicht vollständig portiert werden. Stattdessen werden benötigte Komponenten von JX auf den RCX angepasst, während andere verschwinden.
In einem ersten Arbeitsschritt wurde ein an JX angelehnter Mikrokern entwickelt, der grundlegende Funktionen wie Threads und Scheduling, Interruptbehandlung und das Ansprechen der Peripherie ermöglicht. Der zweite Teil der Arbeit bestand darin das Java-Laufzeitsystem von JX so anzupassen, dass die geringeren Speicherressourcen beachtet wurden. Aus diesem Grund sind die verwendeten Strukturen dieselben wie in JX, jedoch werden sie anders erzeugt: Während sie in JX im Laufe des Betriebes geladen werden, werden sie im entwickelten System vor dem Betrieb extern, auf einem leistungsstarken Rechner, erzeugt und anschließend mit dem Kern zusammen auf den Baustein geladen. Auf diese Weise lässt sich der Speicherplatz vor allem durch eingesparten Code besser ausnutzen.

Abstract

In all of today's PCs and mainframe systems processors are equipped with a hardware based memory protection mechanism. Those, however, do not exist in most microcontroller systems and thus they have to be simulated by software. One possibility of gaining this aim is to use type safe programming languages i.e. Java, that is used by the operation system JX, too. JX realizes all memory protection without the use of hardware and therefore is a good base for porting it on a microcontroller. The controller used in this case is the LEGO Mindstorms RCX. Since a microcontroller in general comes with little memory capacity, it is not possible to port JX directly. Instead of this necessary components are taken, while others are just thrown away.
In a first working step a microkernel similar to JX was developed whose job is to make possible to use basic functionality like threads and scheduling, interrupt handling or using the peripherals. The second part of the work was adapting the JX Java runtime system in a way that the reduced amount of memory was respected. While the used structures are the same as in JX, the way of producing them isn't for saving space. JX loads them during runtime, in opposite the new systems prepares them on a more powerful Computer and links them with the kernel. Doing this saves memory.

Inhaltsverzeichnis

Kapitel 1

Einleitung und Motivation

Die meisten aktuellen Mehrzweckprozessoren (*general purpose CPU*) besitzen eine Speicherverwaltungseinheit (*memory management unit,* MMU), die es ermöglicht den Prozessen einen virtuellen statt eines physischen Adressraums zur Verfügung zu stellen. Sie übernimmt die Aufgabe, virtuelle auf physische Adressen abzubilden, sowie zu überprüfen, ob ein Prozess legitimiert ist, auf eine bestimmte Adresse zuzugreifen. Damit wird ein Schutzmechanismus erzeugt, der vor allem in Mehrbenutzersystemen, aber auch auf allen anderen Systemen, auf denen Prozesse nebenläufig ablaufen können, unabdingbar ist, da so beabsichtigte Sabotage oder die unabsichtliche Beeinflussung anderer Prozesse erschwert oder unmöglich gemacht werden.

Während die MMU bei PC-Systemen, Workstations und Servern gang und gäbe ist, besitzen viele kleinere Prozessoren, Mikrocontroller und Spezialrechner diesen Schutz nicht und sind darauf angewiesen, darauf zu vertrauen, dass die auf ihnen operierenden Programmen keine Fehler enthalten und deren Entwickler keine bösartigen Ziele verfolgt haben. Eine Möglichkeit bei diesen Systemen einen Speicherschutz zu gewährleisten, besteht sicherlich darin, die im Speicher befindlichen Aktivitätsträger auf einen zu beschränken. Dies hätte allerdings den gravierenden Nachteil, dass bei einer fehlerhaften Programmierung, zum Beispiel bei einer Endlosschleife, das ganze System lahmgelegt wäre, was vor allem bei sicherheitskritischen Anwendungen katastrophale Auswirkungen haben könnte. Ganz zu schweigen von der Tatsache, dass auch auf Mikrokontrollern die nebenläufige Ausführung von Prozessen oder Threads in vielen Situationen unabdingbar sein kann, zum Beispiel in einem System, das sowohl eine Benutzer- als auch eine Maschinenschnittstelle anbietet.

Eine zweite Möglichkeit Speicherschutz zu ermöglichen wäre es, dies dem Betriebssystem zu überlassen, was in der Praxis jedoch bei jedem Speicherzugriff einen Sprung in den Systemkern bedeuten würde und somit sehr ineffizient wäre, da intern zu jedem Prozess Tabellen mit Rechten zum Speicherzugriff geführt, aktualisiert und kontrolliert werden müssten. Wäre dies in einer interpretierten Sprache eventuell noch machbar, wird es in einem System mit ladbaren, im Maschinencode vorliegenden Modulen wohl unmöglich zu realisieren sein.

Eine dritte, einfache Möglichkeit zu verhindern, dass sich Prozesse gegenseitig in ihrer Programmausführung stören, ist es dem Programmierer unmöglich zu machen, Fehler dieser Art zu begehen oder auch bewusst Sabotage zu betreiben. Einfach dadurch, dass Programmiersprachen eingesetzt werden, die typsicher sind, wie zum Beispiel Java oder C#, kann vermieden werden, dass ein Prozess jemals auf ein Stück Speicher zugreift, das er vorher nicht irgendwoher bekommen hat, vorausgesetzt, der Compiler verhält sich wie erwartet. Somit ist auch er vertrauenswürdig. Natürlich wird auch dies nicht ohne Unterstützung des Betriebssystems möglich sein, allerdings beschränkt sich dessen Funktionalität hier darauf eine Laufzeitumgebung zur Verfügung zu stellen, auf die höhere Anwenderschichten zugreifen können und müssen. Dies entspricht im Falle von Java einer *Java Virtual Machine* (JVM). Welche allgemeinen Möglichkeiten vorhanden sind um den Speicherschutz auf Software oder Programmiersprachen zu übertragen und welche Vor- und Nachteile daraus erwachsen, wird in [HvE98, BSP+95] ausführlich diskutiert.

Ziel der vorliegenden Arbeit war es eine derartige Laufzeitumgebung für einen Mikrocontroller zur Verfügung zu stellen. Ähnliches wurde bereits durch Manipulationen am Compiler in [Nik00] realisiert. Zusätzlich soll diese Laufzeitumgebung statisch, also bereits vor dem Starten des Codes weitestgehend erzeugt sein. Als Ausgangsbasis für die vorliegende Arbeit diente das typsichere Betriebssystem JX, sowie der in den LEGO Mindstorms Produkten verwendeten Mikrocontroller RCX, der einen Hitachi H8/300 Prozessor besitzt. Ein kurzer Überblick über JX wird durch [GFWK02] gewonnen. Für detailliertere Auskünfte sei auf [Gol00] verwiesen. [Ren03] ist die Dokumentation des Prozessors, während [Pro98] alle LEGO betreffenden Aspekte abhandelt. Grund für die Wahl des RCX waren die trotz der vielfältigen Einsatzmöglichkeiten geringen Beschaffungskosten. JX wurde gewählt, da seine Schutzmechanismen ausschließlich auf die Typsicherheit von Java basieren. In einem ersten Arbeitsschritt wurde ein an JX angelehnter Mikrokern entwickelt, der grundlegende Funktionen wie Threads und Scheduling, Interruptbehandlung und das Ansprechen der Peripherie ermöglicht. Da auf dem RCX viel weniger Speicherplatz zur Verfügung steht, als auf einem durchschnitt-

lichen IBM-kompatiblem System, war es nicht möglich das Java-Laufzeitsystem von JX komplett zu übernehmen. Zwar sind die verwendeten Strukturen dieselben wie in JX, die Art und Weise jedoch, wie diese entstehen und aufgebaut werden, hat sich geändert: Während sie in JX im Laufe des Betriebes geladen werden, ergibt sich auf dem RCX dadurch, dass diese Strukturen auf einem leistungsstarken Rechner erzeugt und anschließend mit dem Kern zusammen auf den Baustein geladen werden, großes Einsparungspotential in Form von Codezeilen Deshalb bestand der zweite Teil der Arbeit darin ein Programm zu entwickeln, das den später im System benötigten Java-Code und die dazugehörigen Metadaten so aufbereitet, dass sie zusammen mit dem restlichen Java-Code kompiliert und gelinkt werden können.

Kapitel 2 stellt den Mikrocontroller, sowie LEGO-spezifische Funktionalität vor. Außerdem werden Besonderheiten bei der Entwicklung von Mikrocontrollerprogrammen beleuchtet. Daran schließt sich in Kapitel 3 eine Betrachtung des Betriebssystems JX an. In Kapitel 4 wird gezeigt wie ein Mikrokern für den RCX aussehen kann und welche Schnittstellen er den darüberliegenden Schichten anbietet. Das Java-Laufzeitsystem mit allen seinen Strukturen und einigen wichtigen Funktionsabläufen wird in Kapitel 5 vorgestellt. Mit dem entwickelten externe Programm zum Aufbau der nötigen Verwaltungsstrukturen beschäftigt sich Kapitel 6 ausführlich. Im Anhang finden sich Beschreibungen zur Installation und Benutzung verschiedener Programme, die sich für das Entwickeln von Software für dem RCX notwendig sind oder sich als hilfreich herausgestellt haben.

Kapitel 2

RCX

Beim RCX, der in Abbildung 2.1 zu sehen ist, handelt es sich um einen in ein LEGO Gehäuse eingebauten H8/3292 Mikrocontroller von Renesas, ehemals Hitachi. Dieser ist mit einem H8/300 16-Bit-Prozessor ausgerüstet. Der H8/3292 lässt sich in drei verschiedenen Betriebsmodi konfigurieren, die sich jeweils in Speicherausstattung, *on-chip* RAM, Datenwortbreite und Nutzung des Daten- und Adressbusses unterscheiden. In den von LEGO verkauften Geräten ist der Modus fest eingestellt und nicht mehr änderbar. Alle folgenden Angaben beziehen sich deshalb ausschließlich auf diesen Modus. Abbildung 2.1 zeigt den RCX von außen, während Abbildung 2.2 einen Überblick über die Struktur des Mikrocontrollers gibt.

2.1 Prozessor, Speicher und Peripherie

Der Prozessor ist durch den internen Taktgeber mit 16 MHz getaktet. Er verfügt über ein acht Bit Statusregister (CCR, Abbildung 2.3 (a)), das die Flags zum Maskieren der Interrupts, zum Anzeigen eines Überlaufs, einer Null oder eines vier beziehungsweise acht Bit *carrys* besitzt. Außerdem sind zwei Bits für anwenderspezifische Signale reserviert. Neben dem Befehlszähler-Register arbeitet der Prozessor intern entweder mit acht 16 Bit (r0..r7) oder mit 16 acht Bit (r0l, r0h...r7l,r7h) Registern, die alle als Daten- und Adressregister dienen können. R7 wird als *Stack-Pointer* verwendet. Befehlsworte sind in der Regel 16, in machen Fällen aber auch acht oder 32 Bit breit. Der Befehlssatz des H8 ist verglichen mit Mehrzweckprozessoren sehr begrenzt, so werden keine Fließkomma und auch keine Fixpunktzahlen durch Operationen unterstützt. Alle Grundrechenarten sind ebenfalls davon betroffen: Addition und Division können nur dann auf 16 Bit

Abbildung 2.1: RCX Baustein mit Sensoren und Motoren (Quelle: [Fel00])

breiten Worten ausgeführt werden, wenn diese in Registern stehen; Multiplikationen sind generell auf zwei vorzeichenlose Acht-Bit-Operanden mit einem 16-Bit Ergebnis beschränkt; bei Division beziehungsweise Modulo-Arithmetik darf der Dividend nur acht Bit groß sein. Quotient und Rest haben ebenfalls lediglich acht Bit. Die Möglichkeit, Daten direkt im Speicher oder auf dem Stack zu manipulieren, ist nicht gegeben. Stattdessen muss jedes Datum aus dem Speicher in ein Register geladen werden, wo es verändert und von wo aus es anschließend wieder zurückgeschrieben wird. Wobei hier zu beachten ist, dass 16 Bit Daten immer mit *Alignment* geschrieben und auch gelesen werden, das heißt das letzte Bit einer 16 Bit Adresse wird immer automatisch auf *low* gesetzt. Im Gegensatz dazu werden alle gängigen Operationen zur Bitmanipulation und zur Rotation unterstützt. Abgesehen von *jsr* (Aufruf einer Funktion) und *jmp* (Sprung zu einer Adresse), die sowohl direkt, als auch indirekt angeboten werden, sind alle Sprungbefehle relativ zum Befehlszähler, also auch bedingte Sprünge. Es werden zwei verschiedene Rückkehrbefehle unterschieden, nämlich *rts* (*return from subroutine*) und *rte* (*return from exception*). Außerdem existiert eine spezielle *sleep* Instruktion, die den Prozessor in einen seiner drei Energiesparmodi versetzt, einige logische Operationen zur Manipulation des CCR-Registers, sowie ein *eepmov*-Befehl, der einen Datenblock im Speicher verschiebt. Für eine genauere und ausführlichere Beschreibung der Maschinenbefehle, der Adressierungsarten sowie der Arbeitszyklen des Prozessors sei auf [Ren03, infa] verwiesen.

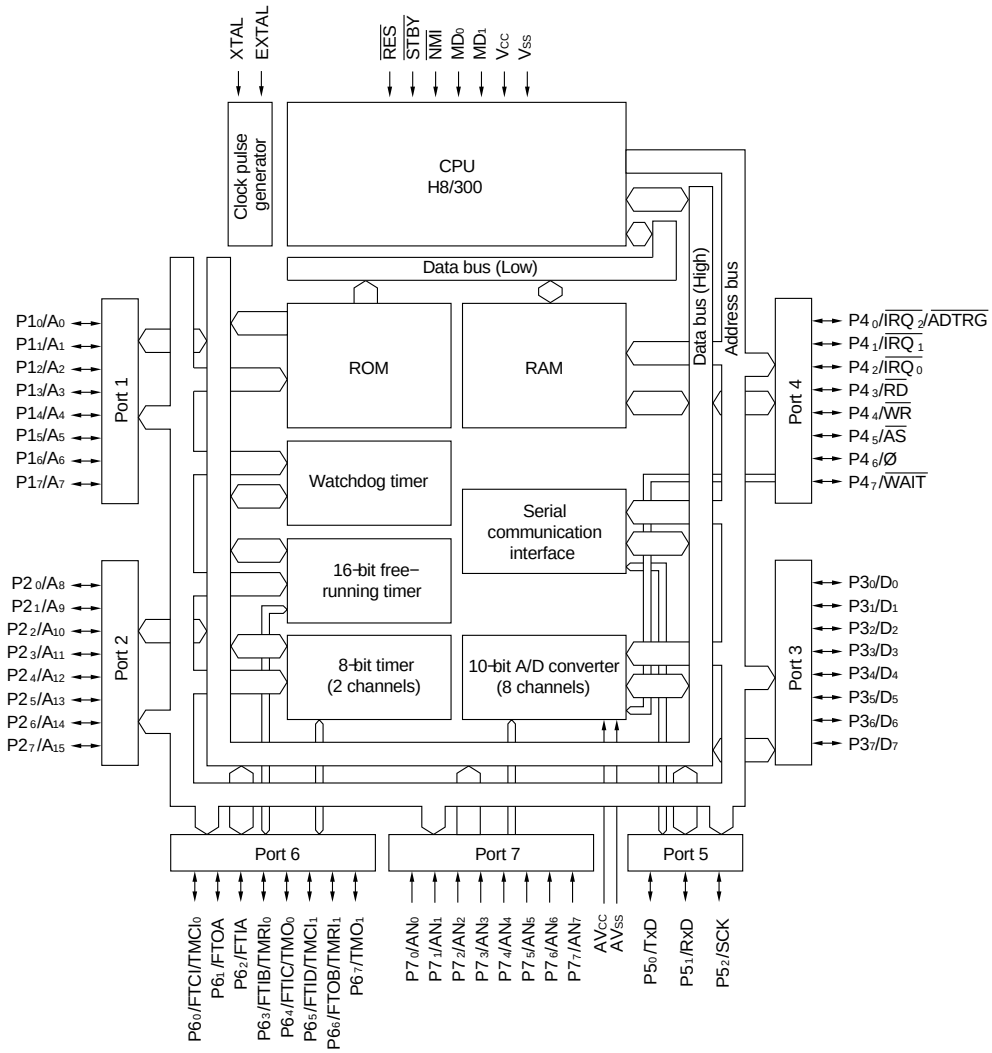
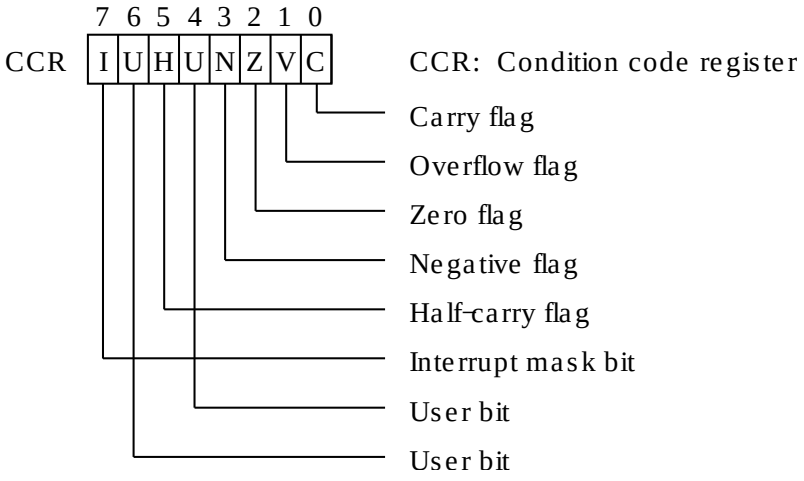


Abbildung 2.2: H8/3292 (Quelle: [Ren03])

Der Stack des H8/300 wächst von hohen hin zu niedrigen Adressen. Er ist prinzipiell acht Bit breit, jedoch verweist [Ren03] auf eine Reihe von unerwarteten Abläufen, die entstehen können, wenn er auch mit acht Bit Worten manipuliert wird. Aus diesem Grund wird empfohlen den Stack nur mit 16 Bit breiten Worten zu verwenden. Was auch erklärt, warum sowohl *push* als auch *pop,* die von den Entwicklern vorgesehenen Operationen zur Stackmanipulation ausschließlich auf Datentypen mit 16 Bit Breite operieren. Dennoch können acht Bit breite Worte



(a) CCR

Bit	7	6	5	4	3	2	1	0
	SSBY	STS2	STS1	STS0	XRST	NMIEG	–	RAME
Initial value	0	0	0	0	1	0	1	1
Read/Write	R/W	R/W	R/W	R/W	R	R/W	–	R/W

(b) SYSCR

Abbildung 2.3: einige Systemregister (Quelle: [Ren03])

mittels *mov.b* Anweisungen auf den Stack geladen und auch wieder davon entfernt werden.

Der Prozessor besitzt drei verschiedene Zustände: Ausführungszustand, Ausnahmezustand und Ruhezustand. Im Ausführungszustand befindet sich der Prozessor, wenn er in keinem anderen Zustand ist. In den Ruhezustand kann er durch den Maschinenbefehl *sleep* versetzt werden, in den Ausnahmezustand kommt er durch einen Interrupt (IRQ). Von diesen gibt es 23. Treten mehrere zur gleichen Zeit auf, entscheiden ihre Priorität darüber, welcher zuerst bearbeitet wird. Drei Interrupts können von externen Quellen kommen. Alle anderen 20 sind interner Natur. Bis auf den NMI (*non maskable interrupt*) kann jeder einzeln deaktiviert werden. Die CPU kann jedoch auch generell aufgetretene Interrupts mit Ausnahme des NMI ignorieren. Dazu muss das I-Bit im CCR (Abbildung 2.3) gesetzt sein. Sowohl der Ruhezustand als auch Ausnahmezustand werden in Kapitel 4 näher erläutert.

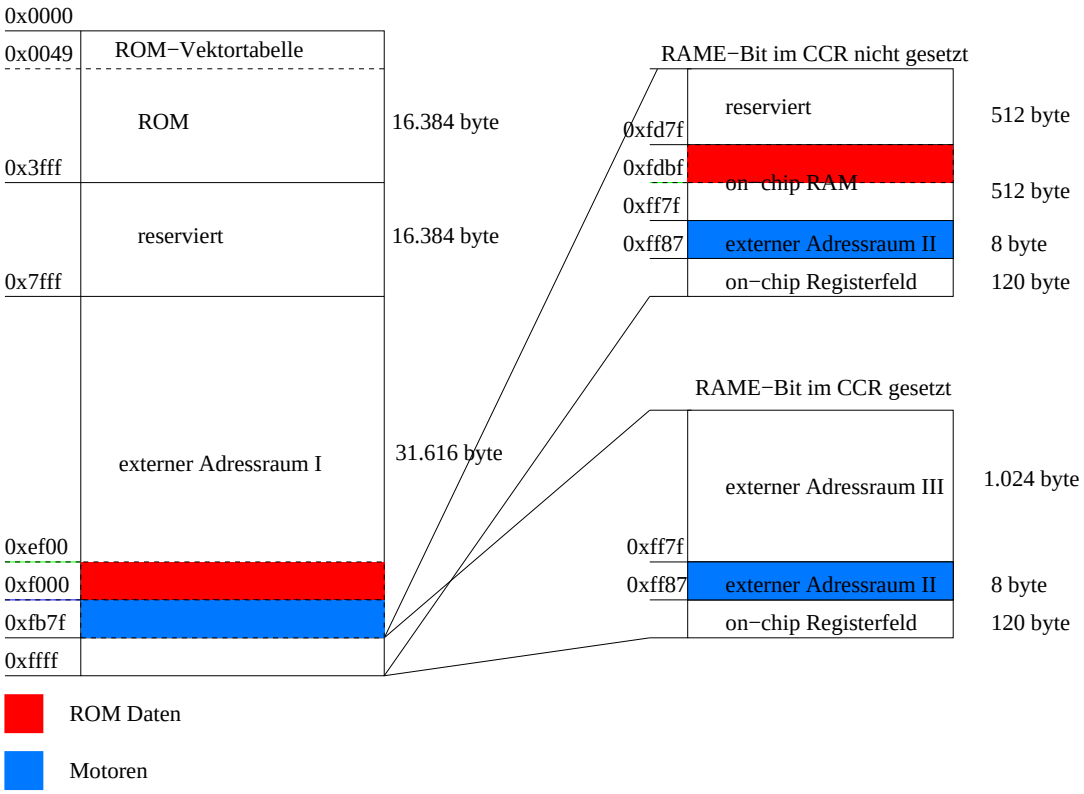


Abbildung 2.4: Speicher-Layout

Der Speicher hat in dem hier betrachteten Modus eine Adresswortbreite von 16 Bit, das heißt es können maximal 64 Kilobyte Speicher adressiert werden, deren Einteilung in Abbildung 2.4 gezeigt ist. Einem Entwickler steht hiervon jedoch nur ein Teil zur Verfügung, weil die Adressen zwischen 0x0000 und 0x3fff durch das ROM belegt sind. Die Bereiche von 0x4000 bis 0x7ffff, sowie 0xfb80 bis 0xfd7ff sind reserviert und alle internen Register auf Adressen jenseits von 0xff80 abgebildet. Der Großteil des Speichers ist externer Natur, lediglich 512 Bytes sind sogenanntes *on-chip* RAM. Das heißt dieser Speicher befindet sich zusammen mit dem Prozessor auf dem Controllerbaustein und ist nicht wie der externe Speicher auf einer externen Platine untergebracht. Es befindet sich zwischen 0xfd80 und 0xff7f. Daraus folgt, dass wenn man für den Betrieb auf das von LEGO vertriebene System aufsetzt, nur 29 kByte zur freien Verfügung hat, wobei man je nach Vorgehensweise auch mit weniger auskommen muss. Dieser Bereich lässt sich um 512 Byte erweitern, da man den Bereich des *on-chip* RAMS und den vorhergehenden reservierten Bereich durch externen Speicher überblenden kann (Abbildung 2.4).

Zum Ansprechen der Peripherie besitzt der H8/300 sieben *Ports* mit je acht Bit Wortbreite, die auf Adressen im *on-chip* Registerfeld über dem RAM Bereich abgebildet sind (Abbildung 2.4). Die *Pins* der Ports können einzeln als aus- oder eingehend konfiguriert werden. Hierzu besitzt jeder Port, bis auf Port sieben, der nur lesbar ist, ein acht Bit breites, nur schreibbares *data direction register* (DDR), dessen i-tes Bit die Richtung des i-ten Pins angibt. Zusätzlich hat jeder Port auch noch ein acht Bit breites Datenregister (DR), aus dessen i-tem Bit beziehungsweise in dessen i-tes Bit der Wert des i-ten Pins gelesen beziehungsweise geschrieben werden kann. In der LEGO-Konfiguration des H8/300 sind die Ports eins bis drei reserviert, da sie zum Ansprechen des Daten- und Adressbusses benötigt werden. Die genaue Pinbelegung der einzelnen Ports ist in Abbildung 2.5 dargestellt. Für weitere Informationen sei auf [Ren03, Pro98] verwiesen.

Port	Pins	Funktion
1	0 - 7	Adressbus niederwertiges Byte
2	0 - 7	Adressbus höherwertiges Byte
3	0 - 7	Datenbus
4	0	Reichweite des Senders
	1	on / off Knopf und IRQ1
	2	run button und IRQ0
5	0	Daten übertragen
	1	Daten empfangen
	2	externer Stromsparmodus für das RAM
6	0	Sensor2 Stromversorgung
	1	Sensor1 Stromversorgung
	2	Sensor0 Stromversorgung
	4	Ausgabe 8-Bit Timer0 (Sound)
	5	Ein-/Ausgabe Display
	6	Ein-/Ausgabe Display
	7	Ausgabe 8-Bit Timer1 (IR-Träger)
7	0	analog Eingabe 0 (Sensor0)
	1	analog Eingabe 1 (Sensor1)
	2	analog Eingabe 2 (Sensor2)
	3	analog Eingabe 3 (Batterie)
	6	view Knopf
	7	program Knopf

Abbildung 2.5: Pinbelegung der I/O-Ports

Weiterhin besitzt der RCX eine serielle Schnittstelle, die über Infrarotsignale mit einem an einen PC angeschlossenen Turm kommunizieren kann. Außerdem zwei acht Bit, sowie einen 16-Bit-Zeitgeber und einen *Watchdog*, der allerdings auch als zusätzlicher *Timer* verwendet werden kann. Eine ausführliche Darstellung findet sich wiederum in Kapitel 4.

Zusätzlich besitzt der RCX drei Anschlussmöglichkeiten für handelsübliche LEGO-Motoren und drei Steckverbindungen, an die verschiedene Sensoren angeschlossen werden können. Während die Sensoren über den Analog-Digital-Wandler ausgelesen werden, sind die Motoren beziehungsweise deren Controller auf die Adressen 0xf000 bis 0xfb7f und 0xff80 bis 0xff87 abgebildet, wobei über beide Intervalle alle Motoren gesteuert werden können. Näheres dazu findet sich in Abschnitt 4.2.

2.2 ROM und LEGO-*Firmware*

Ursprünglich war der RCX von LEGO dazu entwickelt worden, um mit einer *drag&drop* Programmierumgebung einfache Programme zu entwickeln, die dann auf den Baustein geladen werden können. Um diese Funktionalität bereit zu stellen, enthält das ROM des Bausteins eine große Anzahl unterschiedlichster Funktionen, auf die im Folgenden etwas näher eingegangen werden soll.

War der Baustein von der Stromzufuhr getrennt, so übernimmt das ROM nach dem Einlegen der Batterien die Kontrolle, initiiert einige Werte und installiert Standardhandler für sämtliche IRQs. Danach wird darauf gewartet, dass ein PC über die Infrarotschnittstelle Verbindung zum Baustein aufnimmt und ein Programm überträgt. Dieses Programm wird Firmware genannt. Im normalen, von LEGO vorgesehenen, Ablauf wird die standardmäßig vorhandene LEGO-Firmware installiert, die eine Art Interpreter für vom Benutzer entwickelten Programme darstellt. Von diesen können maximal neun auf den Baustein geladen werden.

Da die original Firmware einige Einschränkungen für das zu interpretierende Programm beinhaltet, so ist zum Beispiel der Platz pro Programm mit 2 kByte ([Pro98]) nicht zu großzügig bemessen, ist es wenig erstrebenswert, komplexe Programmabläufe für diese Schnittstelle zu entwickeln. Im Folgenden werden deshalb die Begriffe ,,Firmware" und ,,Programm" synonym gebraucht und bezeichnen das Stück Code, das anstelle der LEGO-Firmware vom ROM in den Speicher des Bausteins geladen wird. Die Möglichkeit, dass von der LEGO-Firmware einzelne Codestücke fester Größe in einen festen, beschränkten Bereich des Speichers

geladen werden, wird nicht mehr weiter betrachtet. Aus diesem Grund entfällt auch eine genauere Betrachtung der Funktionalität der original Firmware. Für detailliertere Informationen sei auf [Pro98]verwiesen.

Wie schon erwähnt erhält das ROM, nachdem die Stromzufuhr hergestellt wurde, die Kontrolle über den Baustein. Es initialisiert die für den Adress- und Datenbus zuständigen Ports, setzt den *Stack*, löscht danach die Bereiche des RAMs, die auf die Existenz einer Firmware hindeuten und kopiert die ROM-Interrupthandler in das RAM. Hierzu muss man wissen, dass bei einem Interrupt eine Funktion im ROM aufgerufen wird, deren Aufgabe lediglich darin besteht r6 zu sichern und anschließend an eine feste Adresse im RAM zu springen. Nach der Initialisierung startet das ROM seine *main*-Funktion. Hier werden zunächst die Timer und der Analog-Digital-Wandler initialisiert sowie neue IRQ-Handler hierfür installiert. Danach erst werden Interrupts global zugelassen, indem das I-Bit im CCR-Register (Abbildung 2.3 (a)) gelöscht wird. Zusätzlich wird zum DDR jedes Ports ein Schattenregister im Speicher angelegt, das den aktuellen Wert des DDRs beinhaltet, da DDRs nur schreibbar, aber nicht lesbar sind. Versucht man dennoch ein DDR zu lesen erhält man immer den Wert 0xff, was wenig hilfreich ist, da manche Operationen den vorherigen Wert des Registers benötigen. Dies ist beim Setzen und Löschen einzelner Bits erforderlich, was zum Beispiel beim Steuern des Displays der Fall ist. Hier werden zuerst Bits an den Displaycontroller gesendet und anschließend auf ein Bestätigungsbit von diesem gewartet. Beide Operationen benutzen das gleiche Bit des gleichen Ports, so dass dieses zunächst als „ausgehend“ konfiguriert werden muss. Zum Empfangen der Bestätigung muss es auf „eingehend“ geändert werden, wobei alle anderen Bits des Registers jedoch nicht beeinflusst werden dürfen. Dies ist nur möglich, wenn der momentane Wert des Registers zugreifbar ist. Da dieser jedoch, wie erwähnt, nicht ausgelesen werden kann, ist es erforderlich ihn zwischenzuspeichern. Nach diesem Schritt werden Sensoren, Motoren, die Knöpfe und gesondert der An/Aus-Knopf initialisiert. Anschließend wird noch das Display zurückgesetzt. Danach werden die Register der seriellen Schnittstelle (SCI) gesetzt und Standardwerte in die Speicherbereiche der für die SCI-IRQs zuständigen Handler geschrieben. Das System ist jetzt prinzipiell in der Lage Pakete von einem PC zu empfangen und an diesen zurückzusenden.

Genau darin besteht im Weiteren die Aufgabe des ROMs: Sobald der An/Aus-Knopf gedrückt wird, kann vom PC aus eine Verbindung mit dem RCX-Baustein hergestellt werden. Passiert dies nicht, wechselt der Prozessor nach einiger Zeit in den Energiesparmodus. Gleiches passiert bei einem erneuten Drücken des

An/Aus-Knopfes. Beim Übergang in den Sparmodus werden alle beschriebenen Initialisierungen rückgängig gemacht. Erneutes Anschalten führt diese wieder aus.

Alle empfangenen Pakete werden als Teil einer Firmware in den Speicher ab Adresse 0x8000 geladen. Anschließend wird versucht das empfangene Programm frei zu schalten, das heißt, es wird zwischen 0x8000 und 0xef00 nach dem String „Do you byte, when I knock?“ gesucht. Nur wenn diese Suche erfolgreich ist, werden die empfangenen Daten als Firmware akzeptiert, andernfalls wird der Empfang einfach ignoriert und wieder auf eine neue Verbindung gewartet ([Pro98]). Nach dem Freischalten wird an die Startadresse des empfangenen Programms gesprungen. Die Daten, die zum Baustein übertragen werden, sind im *srec*-Format kodiert, das unter Anderem die Definition einer Startadresse zulässt, die vom ROM beim Starten der Firmware berücksichtigt wird.

Neben der Initialisierung des Systems und dem Laden der Firmware bietet das ROM noch zahlreiche andere Funktionen, zum Beispiel können verschiedene Tonsequenzen abgespielt, Sensoren ausgelesen und konfiguriert, Motoren gesteuert oder auch Daten gesendet oder empfangen werden. Nicht zuletzt werden auch Funktionen zum Schreiben des Displays angeboten, mit denen Zahlen geschrieben und auch fast alle auf dem Display verfügbaren Symbole an- und ausgeschalten werden können.

Trotz dieser in sich mächtigen Schnittstelle, die das ROM zu den Funktionen des Mikrocontrollers bietet, muss die Verwendung jeder einzelnen Funktion geprüft werden. Ein Einsatz der ROM-Funktionen kann sicherlich viel Code sparen, weil die Funktionalität für Treiber nicht ein zweites Mal implementiert werden muss. Dagegen spricht, dass die Benutzung des ROMs den vorhandenen Speicherplatz zerklüften würde, da die meisten ROM-Funktionen Daten verstreut über das RAM ablegen. Dies hat zur Folge, dass es schwer, wenn nicht sogar unmöglich wäre, ein Programm ab einer bestimmten Größe geschlossen in den Speicher des RCX zu laden. Erschwerend hinzu kommt, dass es für alle ROM-Funktionen, die Parameter erwarten, eine Funktion geben muss, die dafür zuständig ist, die Konventionen für die Parameterübergabe zwischen der des Compilers und der LEGO-spezifischen zu konvertieren, da der Übersetzer die Parameter wahlweise auf dem Stack oder in den Registern r0 bis r2 übergibt, während das ROM hierfür die Register r4 bis r6 benutzt.

2.3 Besonderheiten bei der Programmentwicklung

Dadurch, dass die Zielplattform nicht mit der Entwicklungsplattform übereinstimmt, ergeben sich einige Unterschiede in der Art und Weise, wie entwickelte Programme geladen und getestet werden können. Um überhaupt Maschinencode für die Zielarchitektur erstellen zu können benötigt man einen sogenannten *Cross*-Compiler, der in der Lage ist auf einer Plattform Code für einen andere Plattform zu erzeugen. Genauere Informationen wie ein *Cross*-Compiler konfiguriert und kompiliert werden kann, finden sich in Anhang A. Während der Bearbeitung des praktischen Teils zur Studienarbeit wurden zur Code-Erzeugung das GNU gcc Packet benutzt. Programmiert wurde ausschließlich in Assembler und der Sprache C. An der Schnittstelle dieser beiden Sprachebenen ist es wichtig darauf zu achten, dass ein einheitlicher Parameterübergabemechanismus vorliegt. Standardmäßig benutzt der GNU-Compiler hierbei die Konvention, dass die ersten drei Parameter in den ersten drei Registern r0 bis r2 liegen, alle weiteren befinden sich auf dem Stack. Der Rückgabewert liegt in r0. Die Assemblerfunktionen wurden so entwickelt, dass sie mit dieser Konvention zurechtkommen. Alternativ gibt es noch die Möglichkeit, den Übersetzer dazu zu veranlassen, alle Parameter auf dem Stack zu übergeben, was den Code zwar größer macht, aber in Hinblick auf die Anbindung der Java-Schicht als sinnvoller Alternative erscheint.

Eine Besonderheit des RCX ist sicherlich die Tatsache, dass der Code zum Einen im *srec*-Format vorliegen muss und zum Anderen erst ab Adresse 0x8000 liegen darf. Drittens muss die oben erwähnte, als Schlüssel dienende Zeichenkette in einem bestimmten Speicherbereich liegen und viertens ist der Zugriff auf viele Register nötig, die in den Speicher abgebildet sind und im Programmcode durch Variablen repräsentiert werden müssen. Alle diese Bedingungen können durch die Benutzung eines Linkerscripts gelöst werden. Ein Linkerscript erlaubt es verschiedene Bereiche im Adressraum zu definieren und zu entscheiden, welcher Teil des Codes in welchem Bereich abgelegt wird. Im Folgenden soll ein kurzer Überblick gegeben werden, wie die Möglichkeiten eines Linkerscripts im vorliegenden Fall genutzt wurden. Ein Bereich besteht aus einer Startadresse und der Länge des Bereichs. Eine Dokumentation mit weiterführenden Angaben findet sich in [infb].

Für den betrachteten Fall werden folgende Bereiche definiert:
rom mit einer Startadresse bei 0x0000 und einer Länge von 0x8000 (ca. 32 kByte), *ram* mit Startadresse 0x8000 und Länge 0x7b80 (31 kByte), *on-chip* mit Startadresse 0xfd80 und Länge 0x0040 (64 Byte), sowie *registers*, das bei 0xff80

beginnt mit einer Länge von 0x0080 (128 Byte) bei 0xffff endet. *motor* beginnt bei 0xf000 und hat eine Länge von 0x0002. *rom* wird angegeben um dem Linker zu sagen, dass in diesem Bereich weder Daten noch Code abgelegt werden darf. Diese müssen vollständig im *ram* liegen. *on-chip* ist das *on-chip* RAM des Mikrocontrollers. In diesem Speicherbereich liegt zu Beginn der Ausführung der Stack, sowie die Adressen zu denen die IRQ-Handler des ROMs springen. Der Speicherbereich *registers* ist der Bereich des Speichers, auf den die Adressen der Peripherieregister abgebildet sind. *motor* müsste prinzipiell nur ein Byte lang sein, weil jedes Byte im Speicherbereich 0xf000 bis 0xfb7f bei der Motorsteuerung gleichberechtigt ist. Auf Grund des Alignments ist es jedoch günstiger mit zwei Byte breiten Werten zu arbeiten. Mit Hilfe der Anweisung *ENTRY("<STARTFUNKTION>")* kann nun im Linkerscript eine Startfunktion definiert werden. Außerdem kann mit *OUTPUT_FORMAT("<FORMAT>")* das Format der Ausgabedatei gewählt werden. Hier stehen zwei Formate zur Auswahl: *coff-h8300* und *srec*. Obwohl für die Übertragung zum RCX eigentlich das *srec*-Format benötigt wird empfiehlt es, sich doch vorerst für *coff-h8300* zu entscheiden, da in diesem Format sämtliche Symbolinformationen enthalten bleiben und diese somit später im *Debugger* verwendet werden können. Die zur Übertragung notwendige Umwandlung in das *srec* Format kann mit Hilfe des Programms *objcopy* durchgeführt werden, das allerdings für den Hitachi H8/300 verfügbar sein muss. Die Angabe der Zielarchitektur ist nur dann optional, falls der Compiler nur für ein Ziel installiert wurde. In jedem anderen Fall muss *OUTPUT_ARCH("<ARCHITEKTUR>")* verwendet werden. Um sicherzustellen, dass alle Elements des Codes im Bereich *ram* landen, gibt es im Linkerscript explizite Anweisungen, mit denen die Code-Sektionen *.text, .data* und *.bss* jeweils in den richtigen Bereich geschrieben werden. Die drei aufgeführten Sektionen sind dabei in jedem Code enthalten, da sie der Compiler einfügt. *.text* ist dabei Code, also Funktionen, sowie Daten, die nur lesbar sind, also zum Beispiel Konstanten oder Zeichenketten. Man sollte darauf achten, dass die Zeichenketten nicht allzu spät in den Maschinencode eingefügt werden. So stellt man sicher, dass die vom ROM gewünschte Zeichenkette im richtigen Bereich steht. In *.data* stehen globale und statische Variablen, die bereits initialisiert sind. In *.bss* ebenfalls Variablen, jedoch uninitialisiert. Die Register müssen in den *registers* Bereich geschrieben werden. Das wird erreicht, indem jedem Register, das man benutzen möchte einen Symbolname zugewiesen und dem Linker der Abstand zum Bereichsanfang von der Stelle, an der dieses Symbol später stehen soll, mitgeteilt wird. Wie groß ein Symbol ist spielt für den Linker keine Rolle. Ob es im Code als ein, zwei oder vier Byte-Wert angesprochen wird, ist für ihn völlig unerheblich. Um ein Register unter seinem symbolischen Namen im Code ansprechen zu können, muss der Name als *extern* deklariert werden. Zusätzlich

empfiehlt es sich, Variablen, die für Register stehen, als *volatile*, also flüchtig, zu kennzeichnen, damit der Compiler weiß, dass es sich um Speicher handelt, dessen Wert sich ständig ändern kann und der deshalb jedes Mal neu ausgelesen werden muss und nicht zwischen zwei Zugriffen in einem Register zwischengespeichert werden kann. Ein Auszug aus dem Linkerscript sowie der daraus folgende C-Code findet sich in Abbildung 2.6.

```
:
.finalize : {
              _finalizer = .;
              }>ram
.motor : {
              _motors = 0x00;
              } > motor
:
```

```
volatile extern byte motors;
volatile extern short* finalizer;
```

Abbildung 2.6: Auszug aus dem Linkerskript und dazugehörige C Variablen

Völlig anderes als beim Entwickeln für eine Anwendung auf einem laufenden Betriebssystem ist es, wenn keine Softwareschichten unter der eigenen Software vorhanden sind, also keine Betriebssystemfunktionalität und auch keine Standardbibliotheken. Diese Funktionalität muss, soweit sie denn benötigt wird, selbst implementiert, mit dem Rest des Codes verlinkt und auf den Baustein geladen werden. Auch das Laden des Codes unterscheidet sich grundlegend vom Entwickeln für Desktop Anwendungen. Da es mit einigen Minuten sehr viel Zeit in Anspruch nimmt ein — mit ca. 25 kByte relativ kleines — Programm auf den RCX zu laden, wird es nicht möglich sein, eine Änderung schnell zu testen und danach die Fehler zu suchen, weil allein das Laden meist mehr Zeit in Anspruch nimmt, als das Suchen und Finden mehrerer kleinerer Programmierfehler.

Umstellung erfordert auch die Frage, ob denn das Programm überhaupt das getan hat, was es tun sollte oder ob es vorher in eine Endlosschleife gelaufen oder durch einen fehlerhaften Sprung einen falschen Wert im Befehlszähler stehen hat. Denn die Möglichkeiten einer Rückmeldung sind sehr begrenzt. Zwar gibt es ein Display, auf dem sich auch Meldungen ausgeben lassen, da dieses jedoch über einen zusätzlichen *Controller* angesprochen wird, erscheint eine Ausgabe meistens mit Verzögerung oder auch gar nicht, wenn der Prozessor, noch bevor die Ausgabe gestartet wurde, wegen eines fehlerhaften Befehlswortes in den Ausnahmezustand

übergegangen ist und dieser das ganze System anhält. Als zweite Möglichkeit bietet sich die serielle Schnittstelle an, über die Daten ausgegeben werden können. Bedingung hierfür ist, dass man bereits einen funktionierenden Treiber hat, der über eine der string.h-Bibliothek ähnliche Schnittstelle angesprochen werden kann. Problematisch an der Infrarotübertragung ist jedoch, dass diese Verbindungsart sehr unzuverlässig und empfindlich ist. Deswegen muss eine nicht empfangene Zeile nicht auch gleichzeitig bedeuten, dass nichts gesendet wurde.

Als komfortabler Ausweg hat sich jedoch der in [Fel00] entwickelte und vorgestellte Simulator herausgestellt, der die Entwicklung erleichtert und dessen Verwendung zu empfehlen ist. Der Simulator benutzt die gleiche ROM Initialisierungsroutine wie der echte RCX, lädt also den Code über die per UDP simulierte Infrarotübertragung. Man kann also hier gleich erkennen, ob der Code vom ROM zurückgewiesen oder akzeptiert wird. Diese Vorgehensweise ist zwar schneller als das Laden auf den echten RCX, allerdings immer noch weit davon entfernt eine normale Arbeitsgeschwindigkeit zu erreichen. Deshalb unterstützt der Simulator noch die Möglichkeit den Code als Ganzes in den Speicher zu laden, wobei jedoch der in der *srec*-Datei angegebenen Startadresse des Codes keine Beachtung mehr geschenkt, sondern stattdessen direkt an die Adresse 0x8000 gesprungen wird. Dieser Betriebsmodus des Simulators ist äußerst komfortabel, erfordert jedoch unbedingt eine Änderung des Linkerskripts dergestalt, dass der Code der Startfunktion direkt am Anfang des Programms steht. Die Startfunktion in eine eigene Datei auszulagern und das *.text*-Segment dieser durch das Linkerscript an den Anfang schreiben zu lassen, ist wohl der einfachste Weg dieses Ziel zu erreichen. Ein weiterer Vorteil der Verwendung des Simulators ist die Unterstützung des GNU *Debuggers*, durch den fehlerhafte Programmabläufe oft leichter festgestellt werden können als durch Ausgaben über die serielle Schnittstelle. Vor allem bei der Entwicklung einer Programmierschnittstelle für das SCI ist er die einzige Möglichkeit den Programmfluss zu überwachen. Jedoch ist auch die Verwendung des Simulators keineswegs ungefährlich, da nicht sicher ist, bis zu welchem Grad sein Verhalten mit dem Verhalten des realen Bausteins übereinstimmt. Er eignet sich zum Beispiel nicht zum Testen des Displays, da für dieses das I^2C Protokoll benötigt wird, für welches das zeitliche Verhalten eine Rolle spielt, welches jedoch nicht vollständig simuliert wird. Auch die Funktionalität des Analog-Digital-Wandlers scheint nicht vollständig unterstützt zu werden.

Kapitel 3

Das Betriebssystem JX

JX ist ein typsicheres Betriebssystem, das seine Schutzmechanismen aus den Spracheigenschaften von Java schöpft. Diese verhindern, dass auf nicht allokierten Speicher zugegriffen werden kann, weil die Sprache das Konzept der Zeiger nicht unterstützt. Da Java die einzige Programmiersprache ist, die auf Anwendungsebene benutzt wird, kann somit auch eine Isolation von Prozessen und Programmen erreicht werden.

Dem in C implementierten Kern des Systems muss vertraut werden, während diese Eigenschaft für jedes andere Programm nicht erfüllt sein muss. JX verwendet nach [Gol00] fünf verschiedene Abstraktionen, nämlich Domain, Thread, Komponenten, Portale und Dienste, auf die im Folgenden etwas näher eingegangen werden soll.

3.1 Domain

Eine Domain stellt die zentrale Sicherheitseinheit des Systems dar. Domains sind strikt voneinander isoliert in Bezug auf Ressourcenverwaltung und Datenzugriff. Deshalb läuft sie in einer *Sandbox*-Umgebung ab, das heißt, sie hat ausschließlich Zugriff auf ihre eigenen Daten oder auf Schnittstellen, die von anderen Domains explizit zum Aufruf angeboten werden. Tritt während des Ablaufs ein Fehler auf, so muss eine Domain beendet oder neu aufgesetzt werden, ohne dass andere Domains dadurch in ihrer Konsistenz beeinflusst werden. Zu guter Letzt kann sie vom System zugesicherte Ressourcen verwenden ohne andere Domains zu beeinträchtigen. Da eine Domain eine isolierende Einheit ist, muss sie gleichzeitig auch die Ablaufumgebung für Aktivitätsträger, Threads, sein. Neben den Threads

verfügt sie über einen Heap, auf dem Objekte abgelegt werden. Auch der Stack der Threads kann von diesem Heap genommen werden (Alternativ ist es auch möglich den Stack eines Threads gesondert vom Systemspeicher zu allokieren.) Dieser Heap wird von einem *Garbage Collector* (GC) verwaltet, der dafür zuständig ist den Speicher von nicht mehr referenzierten Objekten zu löschen. Außerdem kann eine Domain über Portale und dazugehörige Dienste verfügen. Erzeugt wird eine Domain durch den *DomainManager*, der gleichzeitig auch einen initialen Thread für diese Domain erzeugt. Zerstört werden kann eine Domain explizit durch den Aufruf eines Kerndienstes. Geschieht dies nicht, wird eine Domain automatisch zerstört, wenn sie über keine Threads mehr verfügt, keine Dienste mehr anbietet, keine Referenzen mehr auf externe Threads besitzt und keine registrierten IRQ-Handler mehr vorhanden sind, weil in einer solchen Konstellation keinerlei Aktivität mehr von ihr ausgehen kann.

Alle Domains beinhalten ausschließlich Java-Code. Eine Ausnahme davon ist die Domain Zero, in der die Funktionalität des Kerns repräsentiert ist und die komplett in C implementiert ist. Sie nimmt weitestgehend die Aufgaben der *Java Virtual Machine* (JVM, VM) wahr und fungiert somit als Schnittstelle zwischen der Java-Schicht und der Hardware.

3.2 Portale und Dienste

Portale sind der einzige Weg, auf dem Domains kommunizieren können. Kommunikation ist nötig, um eine totale Isolation zu vermeiden. Sie muss auch zwischen zwei Domains möglich sein, die sich potenziell misstrauen. Deshalb sollen von vornherein DoS (*denial of service*) Attacken ausgeschlossen werden können. Ebenso soll ein direkter Durchgriff auf lokale Daten einer anderen Domain unmöglich sein. Um die Kommunikation für den Programmierer weitestgehend transparent zu gestalten, soll sie einem lokalen Methodenaufruf gleichen. Ob und welcher Nachrichtenaustausch nötig ist, um den Aufruf zu erfüllen, entscheidet das darunter liegende System. Passiert der Nachrichtenaustausch lediglich über fest definierte Schnittstellen, existiert nur eine kurze zeitliche Abhängigkeit zwischen zwei Domains, nämlich beim Start und bei der Rückkehr des Methodenaufrufs. Ein zusätzliches Designziel war es die Erweiterung auf ein verteiltes System einfach zu halten.

Ein Objekt innerhalb einer Domain, auf das von außen zugegriffen werden kann, heißt *Service Object*. Es ist mit einem oder mehreren *Service Threads* verbunden. Ein Objekt, das in einer anderen Domain dieses *Service Object* repräsentiert

heißt Portal. Ein Portal besitzt die gleiche Schnittstelle wie das ihm zugeordnete Objekt. Wird nun eine seiner Methoden aufgerufen, blockiert sich der aufrufende Thread und einer der Service Threads wird aktiviert. Sowohl die Parameter der Methode als auch der Rückgabewert werden vom Heap der einen auf den der jeweils andere Domain kopiert. Das stellt sicher, dass keine Domain auf den Heap der anderen zugreifen muss. Generell erleichtert es diese Vorgehensweise sowohl die Sicherheit als auch den Informationsfluss zu kontrollieren ([Gol00]).

Neben den eigentlichen Portalen existieren im JX-System noch die sogenannten *Fast Portals* (FP). Diese werden ausschließlich von der Domain des Kerns angeboten, der Domain Zero. FP unterscheiden sich insofern von anderen Portalen, als dass der Methodenaufruf durch den aufrufenden Thread bearbeitet wird. Dies ist zum Einen schneller als ein Threadwechsel, zum Anderen aber auch in manchen Situationen unumgänglich, zum Beispiel dann, wenn der aktuelle Thread den Kern davon in Kenntnis setzt, dass er den Prozessor abgeben möchte. Da die Funktionen des Kerns in C realisiert sind, besitzt jedes FP eine vom Kern aufgebaute vtable, die auf die jeweiligen Funktionen im Kern zeigt. Die Funktionen müssen gewisse Eigenschaften aufweisen, um als FP-Funktionen geeignet zu sein ([Gol00]):

- die Position ihres Codes darf sich nicht ändern
- die Domain, die sie beherbergt, darf nie beendet werden
- sie müssen unabhängig vom Kontext sein, in dem sie ausgeführt werden

Weiterhin haben sie die Möglichkeit die Domainisolation zu umgehen. Wegen dieses Sicherheitsrisikos und der oben genannten Eigenschaften ist es sinnvoll, wenn ausschließlich der Kern solche FP anbieten darf.

Anwendung finden sie zum Beispiel in Form von *Memory* Objekten, die ein Stück Arbeitsspeicher repräsentieren. Dieses Objekt kann zum Beispiel von verschiedenen Domains geteilt werden und somit *shared memory* realisieren. Auch die in den Speicher abgebildete Peripherie kann durch ein solches Objekt repräsentiert werden, so dass die Geräte von der Java Schicht aus ansprechbar und steuerbar werden.

3.3 Speicher

Die Speicherverwaltung in JX ist zweistufig: Auf globaler Ebene müssen große Objekte, wie die Heaps der Domains verwaltet werden. Die zweite Stufe ist domain-

lokal. Auf dieser Ebene sind eher kleine Speichereinheiten wie zum Beispiel Objekte zu verwalten. Es werden zwei verschiedene Arten von domain-lokalem Speicher unterschieden: verschiebbarer und fester. Verschiebbarer Speicher wird vom GC verwaltet, während im festen Speicher Strukturen verwaltet werden, die nicht verschoben werden können, wie zum Beispiel Code.

Eine weitere Aufgabe der Speicherverwaltung entsteht durch die Tatsache, dass eine MMU selbst dann nicht benutzt wird, wenn sie vorhanden ist. Aus diesem Grund müssen ihre Aufgaben von der Software wahrgenommen werden. Dazu zählen das Erkennen von Stapelüberläufen und das Entdecken von *NULL*-Zeigern.

3.4 Komponenten und Übersetzer

Der ganze Code innerhalb von JX ist in Form von Komponenten organisiert. Das Ziel von Komponenten ist es, die Wartung und Erweiterung des Systems zu erleichtern. Indem man den Code, der innerhalb der Domains abläuft, in Komponenten unterteilt, erleichtert man die Konfiguration und Verwaltung des Systems. Komponenten bestehen aus einer Menge von zusammengehörigen Klassen. Zwischen den Komponenten können Abhängigkeiten existieren, wenn ihre Klassen voneinander abhängig sind. Komponenten können in drei verschiedene Kategorien eingeteilt werden ([Gol00]):

Bibliotheken: Bei einer Bibliothek handelt es sich um eine Sammlung von Klassen und Schnittstellen, die von anderen Komponenten benutzt werden. Sie beinhalten wiederverwendbaren Code und Datenstrukturen.

Dienste: Eine Dienstkomponente beinhaltet die Implementierung eines bestimmten Dienstes, wie zum Beispiel Treiber.

Schnittstellen: Schnittstellenkomponenten beinhalten alle Schnittstellen, die nötig sind um einen Dienst zu nutzen. Sie beinhalten meist die Portal Schnittstelle (*Interface*), sowie Klassen- oder Schnittstellendefinitionen der Parameter. Wenn eine Schnittstellenkomponente ausschließlich *Interface*s beinhaltet, muss keinerlei Code importiert werden. Dies ermöglicht es einen Dienst zu benutzen, dem man nicht vertraut ohne Code verwenden zu müssen, dem man nicht vertraut.

Da jede Komponente aus Klassen besteht und eine Klasse aus Code, konstanten Daten und veränderbaren Daten besteht, ist es möglich sowohl den Code, als

auch die konstanten Daten zwischen verschiedenen Domains zu teilen, während die veränderbaren Daten domain-lokal sein müssen.

Komponenten werden von einem Java-zu-Maschinencode-Übersetzer ([Waw01]) erzeugt, der entweder auf einem JX-System in einer eigenen Domain oder auf einem anderen System läuft. Im ersten Fall lädt der Übersetzer die Komponente in den Speicher, im zweiten Fall wird sie zusammen mit dem Kern während des Boot-Vorgangs in den Speicher geladen.

Kapitel 4

Mikrokern

Um eine Laufzeitumgebung für eine höhere Programmiersprache wie Java auf einem Mikrokontroller umsetzen zu können, ist es zunächst erforderlich eine erste Zwischenschicht aufzubauen, die zwar noch nicht die Ausführung von Java-Code unterstützt, aber dennoch gewisse Schnittstellen für die Programmausführung und für die Peripherie anbietet, so dass nebenläufige Programmausführung und auch eine Kontrolle der *Hardware*-Funktionalität möglich ist. Ein großes Problem war hier einen guten Mittelweg zwischen einer hohen Abstraktionsebene, die ein einfaches Ansprechen der *Hardware* ermöglicht, auf der einen und einer platzsparenden Implementierung, mit weniger mächtiger Schnittstelle, auf der anderen Seite zu finden.

Dieses Kapitel zeigt zunächst, wie das System sinnvoll hochgefahren werden kann, bevor danach mit Motoren und Sensoren die Aktoren und Sensoren des Systems vorgestellt werden, wobei es sich hierbei um LEGO- und nicht Hitachi-spezifische Peripherie handelt. Wie Interrupts auf den Programmablauf einwirken und wie sie steuerbar gemacht werden können, wird im darauf folgenden Abschnitt gezeigt, bevor anschließend darauf eingegangen wird, auf welche Art und Weise Benutzer mit dem System oder auch das System mit Benutzern kommunizieren kann. Da für die Entwicklung des Kerns überwiegend die Programmiersprache C benutzt wurde, für diese aber keinerlei RCX-spezifischen Bibliotheken zur Verfügung stehen, müssen diese erst erstellt werden. Wie und in welchem Umfang das realisiert wurde, wird nach der Beschreibung der Kommunikationsschicht aufgezeigt. Anschließend wird ein Blick auf Maßnahmen geworfen, die das *Debugging* ermöglichen und erleichtern. Die darauf folgenden Abschnitte beschäftigen sich mit den für die nebenläufige Programmausführung notwendigen Konstrukten Domains, Threads, den dafür notwendigen Scheduler sowie Semaphoren. Zum

Abschluss des Kapitels wird ein Überblick über die Unterschiede zwischen dem entstandenen System und dem als Vorbild dienenden JX gegeben.

4.1 Startfunktion

Wie schon bei der Erläuterung des Linkerscripts bemerkt, macht es aus Gründen der Handhabbarkeit Sinn, eine einfache Startfunktion in einer eigenen Datei zu platzieren. Möchte man den Simulator ohne den Ladevorgang benutzten ist es sogar unumgänglich. Ein Linkerscript bietet zwar die Möglichkeit die Reihenfolge, in der Dateien in die Programmdatei geschrieben werden sollen, anzugeben, nicht jedoch die Abfolge der Funktionen in dieser Datei. Eine Startfunktion müsste deshalb immer am Anfang der ersten Datei stehen, die vom Linker verarbeitet wird. Deswegen ist es am komfortabelsten, wenn man die Aufgaben, die zu Beginn des Programmablaufs nicht umgangen werden können und die in der Regel auch sehr hardwarenah sind in eine kleine Assemblerdatei packt und am Ende des Programmflusses die eigentliche Startfunktion aufruft. Gleichzeitig teilt man dem Linker über das Script mit, dass diese Datei als erste verarbeitet werden soll. Das verhindert, dass man aus Versehen die Reihenfolge der Funktionen vertauscht und Fehler sucht, die eigentlich keine sind.

Dinge, die während des Startvorgangs erledigt werden können sind zum Beispiel das einzelne Deaktivieren aller Interrupts, sowie das Setzen des I-Bits im CCR, so dass man nicht durch unvorhergesehene Interrupts und nicht installierte IRQ-Handler überrascht wird. Eine Entscheidung, die zu diesem Zeitpunkt getroffen werden muss, ist, ob man das *on-chip* RAM durch externen Speicher überblenden möchte oder nicht. Ein Grund sich für das *on-chip* RAM zu entscheiden ist, dass sein Inhalt in allen Energiesparmodi erhalten bleibt, während dies bei externem Speicher nicht immer der Fall ist. Da der Speicher beschränkt ist und somit auch der Stack nur in kleinen Grenzen wachsen kann, dieser jedoch nach den Programmabläufen im ROM, die zum Laden der Firmware geführt haben, nicht mehr leer ist, kann er an dieser Stelle zurück- oder auch neu gesetzt werden.

4.2 externe Peripherie

Wie jeder Mikrocontroller bietet auch der RCX die Möglichkeit Werte aus seiner Umgebung zu messen beziehungsweise auf seine Umgebung einzuwirken. Dies ist jedoch nicht beliebig möglich, da er als Spielzeug konzipiert wurde und der

Benutzer sich mit den von LEGO angebotenen Geräten zufrieden geben muss. Als Aktoren verfügt der RCX dabei über drei Anschlüsse für Motoren. Sensoren stehen in einer Vielzahl von unterschiedlichen Ausprägungen zur Verfügung. Jedoch können lediglich drei von ihnen zur gleichen Zeit angeschlossen und benutzt werden. Über [Pro98] sind Informationen erreichbar, wie mehr als drei Sensoren gleichzeitig benutzt werden können.

Motoren

Abbildung 4.1: Byte zur Motorsteuerung

Die Steuereinheit der Motoren ist wie in der Abbildung über die Speicherbelegung (Abbildung 2.4) zu sehen in den Speicher abgebildet. In den markierten Adressbereichen ist jedes Byte gleichberechtigt, das heißt, es ist durch jedes Byte möglich die Motoren zu steuern. Wie in Abbildung 4.1 gezeigt sind die acht Bits eines Bytes in vier logische Einheiten zu je zwei Bits gruppiert, von denen drei je einen Motor steuern. Das vierte Paar ist ungenutzt. Jedes der Bitpaare kann vier verschiedene Werte annehmen, die auch alle von den Motoren genutzt werden. Haben beide Bits den Wert 0 oder 1 ruhen die Motoren. Sind sie dagegen 01 oder 10 drehen sie sich vor- beziehungsweise rückwärts. Da die Drehrichtung der Motoren auch von der Art und Weise abhängig ist, wie die Stromstecker auf den jeweiligen Anschlüssen angebracht sind, ist die Festlegung auf vor- und rückwärts eher willkürlich.

Sensoren

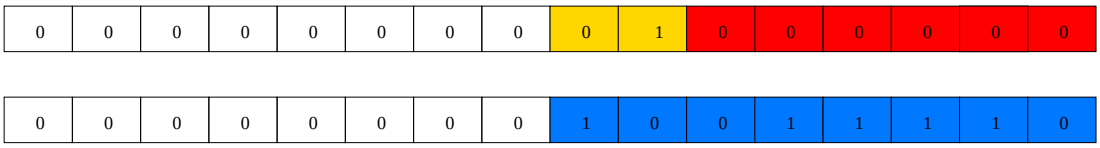
Das Auslesen der Sensoren erfordert die Umwandlung von analogen Signalen in digitale Werte, was Aufgabe des im Controller enthaltenen Analog-Digital-Wandlers ist. Dieser kann zwei Gruppen von je vier analogen Signalen zu 10-stelligen digitalen Werten umwandeln. Dazu benutzt er die acht Pins von Port sieben, von denen im RCX nur die ersten vier Pins sinnvoll verwendet werden können. Diese vier entsprechen gleichzeitig der ersten Gruppe. Mit diesen Eingängen sind die drei Sensorausgänge verkabelt. Der verbleibende vierte Pin ist mit der Batterie verbunden, so dass es möglich wird die aktuelle Spannung auszulesen und es dem Benutzer mitzuteilen, wenn die Batterie schwach wird. Der

Analog-Digital-Wandler kann in zwei Modi betrieben werden, einem Einzel- und einem Scanmodus. Im Einzelmodus wird jeweils der Wert eines spezifizierten Pins für eine gewisse Zeit überwacht und daraus ein 10-stelliger Binärwert gewonnen, während im Scanmodus eine aufsteigende Folge von Pins aus einer Gruppe umgewandelt wird. Also zum Beispiel Pin0, danach Pin1 oder aber Pin4, gefolgt von Pin5 und darauf Pin6. Das ROM verwendet den Scanmodus, während im vorliegenden Fall der Einzelmodus verwendet wird, da dieser schneller zu konfigurieren war und dadurch keinerlei Nachteile entstehen. Der gewandelte Wert jedes Pins wird in ein 16-Bit-Register abgelegt, dessen untere sechs Bits immer als null gelesen werden (Abbildung 4.2 (a)). Dabei teilen sich jeweils zwei Eingänge, jeder aus einer Gruppe, ein Register, das heißt es gibt vier 16-Bit-Register. Soll nun der Wert aus den Registern ausgelesen werden, so muss dies in zwei Schritten passieren, da die Register nur durch einen acht Bit breiten Bus an den Prozessor angebunden sind. Hat man die Werte der beiden Registerhälften in je eine 16-Bit-Variable ausgelesen, muss der Inhalt des höherwertigen Bytes um sechs Bit nach rechts, das niederwertige um zwei Bits nach links geschoben werden (Abbildung 4.2 (b) und (c)). Nach einer Addition der beiden Variablen erhält man den Sensorwert. Wie dieser Sensorwert interpretiert wird bleibt bisher noch der Anwendung überlassen, da lediglich eine Schnittstelle zum Auslesen der rohen Sensorwerte angeboten wird. Jedoch ist es denkbar ähnlich wie das ROM eine Schnittstelle zu schaffen, die den rohen Wert in eine für den jeweiligen Sensor sinnvolle Darstellung umwandelt. Als Sensoren kommen Druck-, Licht-, Rotations- und Wärmesensoren in Frage. Die ROM Funktion wird bisher nicht verwendet, jedoch kann diese Möglichkeit genutzt werden, wenn man bereit ist auf den von ihr benötigten Speicherplatz zu verzichten.

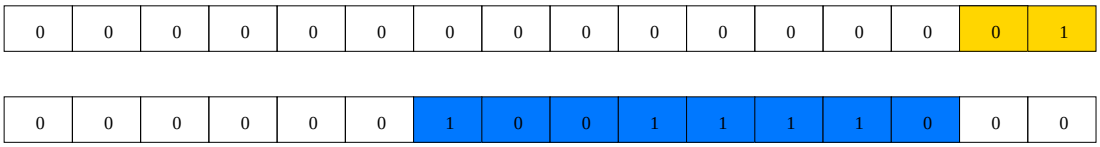
4.3 Interrupthandling

Wie schon mehrfach erwähnt gibt es im H8/300 23 prioritätengesteuerte Interrupts, von denen drei externer Natur sind. Im Falle des RCX scheinen nur zwei der drei externen IRQs belegt zu sein, zumindest findet sich keinerlei anders lautende Dokumentation. Bei den beiden belegten IRQ-Quellen handelt es sich um den An/Aus-Knopf auf IRQ1 und den *run*-Knopf auf IRQ0. Jeder dieser drei Interrupts kann einzeln aktiviert und deaktiviert werden. Zusätzlich kann gewählt werden, ob der Interrupt durch eine fallende Flanke oder durch ein anliegendes *low* Signal aktiviert wird. Im Fall von Knöpfen empfiehlt es sich den IRQ durch eine fallende Flanke zu aktivieren, da in der Regel niemand einen Knopf länger drückt.

(a) Registerwert nach A/D Wandlung



(b) oberes und unteres Byte



(c) shiften der Bytes

(d) Addition der Werte

Abbildung 4.2: Auslesen des A/D-Wandlers

Der Interrupt mit der höchsten Priorität ist der NMI, der als einziger nicht vom Anwender deaktiviert werden kann. Der Prozessor deaktiviert ihn, während einer Neustartphase. Danach folgen die drei externen Interrupts IRQ0 bis IRQ2. Alle Interrupts sind vektorisiert, das heißt, dass jedem Vektor eine Adresse im Speicher gehört, die wiederum die Adresse seines Interrupthandlers enthält. Der Interruptvektor beim RCX beginnt an der Adresse 0x0000 und erstreckt sich bis inklusive 0x0049. Der Prozessor kann also maximal 36 Interruptquellen besitzen, da sich in der ersten Adresse der *reset*-Vektor steht, der benötigt wird um das System neu zu starten. Tatsächlich liegen jedoch in dieser Version des Prozessors einige Interruptadressen brach, weil es zu wenig Peripherie gibt.

Die Interrupts selbst werden von einem Interrupt-*Controller* gesteuert, der als Mittler fungiert, falls zwei IRQ-Wünsche gleichzeitig gemeldet werden. Sind nun die Interrupts nicht durch das I-Bit im CCR gesperrt und liegen dem IRQ-*Controller* mehrere Wünsche vor, so wählt der den nach Priorität höchsten aus und veranlasst die CPU, nach Beendigung des aktuellen Maschinenbefehls in den Ausnahmezustand überzugehen. Alle anderen Wünsche bleiben vorgemerkt. Das Vorbereiten des Ausnahmezustands bedeutet für den Prozessor lediglich, dass er den aktuellen Befehlszähler, der die nächste Anweisung, die nach der Ausnahmebehandlung auszuführen ist, beinhaltet und das CCR auf dem Stack sichert. Da das CCR lediglich acht Bit groß ist, wird es doppelt auf den Stack geschrieben, um die Wortgrenzen von 16 Bit einzuhalten. Anschließend werden alle anderen IRQs durch das Setzen des I-Bits im CCR maskiert, die Adresse des zum IRQ gehörigen *Handlers* aus dem entsprechenden Vektoreintrag gelesen und zu ihr verzweigt.

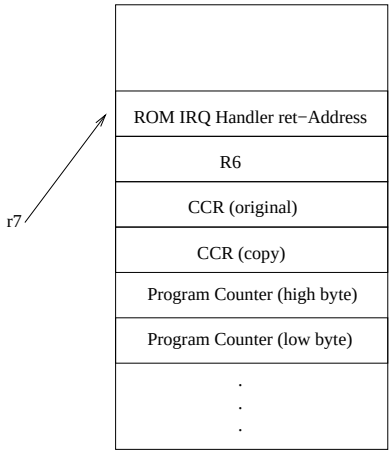


Abbildung 4.3: Stack nach auftreten eines Interrupts und ausführen des ROM-IRQ-Handlers

Besonderheiten beim RCX entstehen dadurch, dass schon durch das ROM eine Indirektionsstufe für die Interruptbehandlung eingebaut wird, weil die im ROM befindlichen, fest verdrahteten IRQ-Routinen, lediglich das Register r6 sichern, anschließend aber die Adresse der frei wählbaren Interruptroutine aus einer festen Adresse auf dem *on-chip* RAM lesen und zu dieser springen. Diese Konfiguration gewährt eine gewisse Flexibilität und ermöglicht es erst den RCX mit neuer, nicht LEGO-spezifischer, Firmware zu programmieren. Denn wären die LEGO-IRQ-Handler fest im ROM und somit nicht änderbar, müsste eine mögliche neue Firmware ohne Interrupts auskommen, das heißt präemptives Scheduling wäre

von vorherein ausgeschlossen, ebenso wie die unmittelbare Reaktion auf gedrückte Knöpfe und Ähnliches.

Um eine Verwaltung von Interrupts zu ermöglichen, benötigt jeder IRQ eine Funktion um ihn an-, und eine um ihn auszuschalten. Da in JX Java-Interruptfunktionen vorgesehen sind, muss dort nach jedem Auftreten eines Interrupts der Kontext gesichert werden und der Kontext des Handlers wiederhergestellt werden. Da für den RCX bisher noch keine in Java implementierten Interruptroutinen benutzt werden, wird diese Funktionalität jedoch vorerst für den RCX-Baustein nicht benötigt und ist aus diesem Grund auch nicht implementiert.

```
struct  irqfunctions {
   char *irqName;
   void (*enable)();
   void (*disable)();
   void (*ack)();
};
void installIRQhandler(short irqnr, void (*handler)());
void enable_irq(short irq);
void disable_irq(short irq)
```

Abbildung 4.4: Schnittstelle der IRQ-Verwaltung

Für jeden IRQ wird bei der Initialisierung der IRQ-Verwaltung die Adresse eines Handlers an die entsprechende Adresse im *on-chip* RAM geschrieben. Diese Funktion fungieren als Wrapper für benutzerdefinierte Interrupthandler, die durch die Funktion *installIRQhandler* (Abbildung 4.4) bei der IRQ-Verwaltung registriert werden können und von dieser in ein Funktionszeigerarray *ihandler* eingetragen wird, das die Adressen der Handlerfunktionen speichert. Jede der Wrapper-Funktionen schreibt die Nummer des korrespondierenden IRQs ins zuvor gesicherte r0-Register und verzweigt danach zu einer Funktion *_hwint*, die aus *ihandler*, die Adresse des eigentlichen IRQ-Handlers ausliest und die jeweilige Funktion aufruft. Bei der Initialisierung der Interrupts werden zunächst alle IRQs deaktiviert und in alle Felder von *ihandler* die Adresse einer *dummy*-Funktion eingetragen, die nichts tut außer zurückzukehren. Die oben genannten interruptspezifischen Funktionen zum Ein- und Ausschalten sind in einem Array vom Typ *struct irqfunctions* gebündelt, wobei diese Struktur pro Interrupt dessen Namen und die jeweiligen Funktionen enthält (Abbildung 4.4. Die Funktionen *enable_irq, disable_irq* greifen auf dieses Array zu, um die gewünschte Wirkung zu erzielen. Der Vorteil dieser zweistufigen beziehungsweise dreistufigen, wenn die Stufe im

ROM hinzugezählt wird, IRQ-Behandlung ergibt sich aus der Tatsache, dass die eigentlichen Handler in einer Hochsprache implementiert werden können, weil es jetzt nicht mehr nötig ist ihn mit *rts* zuverlassen. Das nötige *rte* veranlasst die Wrapper-Funktion.

Eine Sonderstellung nimmt der IRQ des 16-Bit-Timers ein, weil sein Auftreten ein essentieller Bestandteil des Schedulings und somit des Multitaskings ist. Tritt er auf, wird der Kontext des aktuellen *Threads* in der dem *Thread* zugeordnete Datenstruktur gesichert. Anschließend wird die Funktion *schedule()* des Schedulers aufgerufen, die den nächsten lauffähigen Thread aus einer Warteschlage auswählt und zu diesem wechselt. Eine genauere Beschreibung dazu findet sich in den Abschnitten 4.7 beziehungsweise 4.8, die sich mit Threads beziehungsweise dem Scheduler beschäftigen.

4.4 Kommunikation

Wichtig für die Rückmeldung des aktuell laufenden Programms an den Entwickler oder an ein auf dem PC laufendes Kontrollprogramm ist eine funktionierende Kommunikation vom RCX an den PC und umgekehrt über die serielle Schnittstelle. Ein zusätzliches Instrument um Statusmeldungen an einen Benutzer auszugeben bietet sich durch das Display des RCX. Beide Arten um Meldungen auszugeben werden im Folgenden betrachtet. Um die serielle Schnittstelle benutzen zu können, ist es nötig einen der beiden acht Bit Zähler zu programmieren, da durch ihn die Frequenz für die Infrarotschnittstelle gesetzt wird.

Serielle Schnittstelle

Wie schon erwähnt stehen zwei Zähler mit jeweils acht Bit breiten Zählregistern zur Verfügung. Einer von beiden wird dazu benutzt die Modulationsfrequenz von 38 kHz für die serielle Schnittstelle bereitzustellen. Der Andere ist mit dem Lautsprecher des RCX verbunden, so dass über ihn Töne nach außen gegeben werden können. Durch Ändern der Zählgeschwindigkeit und des Zählintervalls kann auch die Tonhöhe verändert werden, so dass, wie in der ROM-Funktion *play_sound_or_set_data_pointer* realisiert, auch Melodien abgespielt werden können.
Um wirklich Daten über die serielle Schnittstelle übertragen zu können, ist es notwendig, dass der zweite acht Bit Timer eine Rechteckspannung mit der Frequenz 38 kHz erzeugt, die als Infrarotträgerfrequenz benutzt wird. Der Timer

wird so konfiguriert, dass das Ausgangssignal jedes Mal von '0' auf '1' oder '1' auf '0' wechselt und der Zähler auf null zurückgesetzt wird, wenn der Zählerwert mit dem Vergleichsregister übereinstimmt. Eine Frequenz von 38 kHz bedeutet, dass sich der logische Wert des Ausgangs 76,000 mal pro Sekunde ändern muss. Das ROM erreicht dies dadurch, dass als Quelle $\frac{1}{8}$ des internen Takts gewählt wird. Rechnerisch muss der Zähler in diesem Fall immer dann zurückgesetzt werden, wenn er den Wert 27.7 hat. Im ROM wird allerdings der Wert $0x1a = 26$ gewählt, was darauf schließen lässt, dass es sich hierbei wohl um den richtigen Wert handelt und die Trägerfrequenz nicht exakt bei 38 kHz liegt. Nähere Informationen zur Programmierung der Timer finden sich in [Ren03].

Die serielle Schnittstelle (SCI) verfügt über eine Sende- und Empfangseinheit, die getrennt voneinander arbeiten können. Jedoch scheint dies durch die Infrarotschnittstelle unmöglich gemacht zu werden, da beim Senden eines Zeichens dasselbe Zeichen auf Grund von Reflexionen mehrmals empfangen wird. Das durch den LEGO-Tower festgesetzte Übertragungsprotokoll arbeitet mit einer Rate von 2400 baud. Gesendet werden acht Bit, gefolgt von einem Paritäts-Bit, das ungerade Parität herstellt, sowie einem Stopp-Bit. Die logische '1' wird als 417 μs Schweigen, eine logische '0' als 417 μs langes Senden einer 38 kHz Frequenz kodiert ([Pro98]).

Intern betrachtet können durch die serielle Schnittstelle vier verschiedene Interrupts ausgelöst werden: Empfangsfehler (ERI), Datenempfang beendet (RxI), Übertragung erfolgt (TxI), Übertragung beendet (TEI). Die Reihenfolge entspricht der Priorität der Interrupts von „hoch“ nach „niedrig“.

Das SCI unterstüzt eine Vielzahl von Betriebsmodi. Da der LEGO-Turm, der zur Kommunikation mit einem PC notwendig ist, einen asynchronen Modus mit acht Bit Wortlänge, Paritäts- und einem Stoppbit erwartet, müssen diese Attribute durch Setzen der entsprechenden Bits in den verschiedene Registern ([Ren03]) des SCI eingestellt werden. Spielraum besteht dagegen bei der Baudrate, mit der die Kommunikation abläuft. Bisher konnte Datenaustausch sowohl mit $2400\frac{bit}{s}$ als auch mit $4800\frac{bit}{s}$ erfolgreich hergestellt werden. Dabei können Überlauf-, Paritäts- und Rahmenfehler erkannt werden.

Alle erforderlichen Attribute lassen sich durch Setzen und Löschen von Bits in diversen Registern einstellen. Genauere Informationen zur Hardware-Programmierung finden sich in [Ren03]. Sowohl die Bereitschaft zum Senden als auch zum Empfangen von Daten kann einzeln aktiviert und deaktiviert werden. Um zu senden werden die Daten byteweise in ein Puffer-Register geschrieben und der Hardware der Schnittstelle durch Löschen eines Bits mitgeteilt, dass Daten vorlie-

gen. Anschließend wird das jeweilige Byte von der Hardware in ein Sende-Register kopiert und von dort aus bitweise gesendet. Bereits mit dem Kopieren wird der Software durch Setzen eines Bits mitgeteilt, dass die Übertragung erfolgreich war und das nächste Byte in das Puffer-Register geschrieben werden kann. Zusätzlich zum Setzen des Bits kann ein Interrupt ausgelöst werden. Wurden keine neuen Daten in den Puffer geschrieben, wenn alle Bits im Sende-Register gesendet sind, betrachtet die Hardware die aktuelle Übertragung als beendet. Auch dies wird der Software durch ein gesetztes Bit mitgeteilt. Hier kann ebenso zusätzlich ein Interrupt ausgelöst werden.

Das Empfangen funktioniert analog zum Senden. Hier gibt es ein Empfangs- und ein Puffer-Register, wobei das Empfangs-Register Bits empfängt, diese zu Bytes zusammensetzt, die anschließend in das Puffer-Register kopiert werden. Wurde ein Byte fehlerfrei empfangen, wird dies der Software durch ein gesetztes Bit mitgeteilt. Gleiches gilt, wenn während der Übertragung ein Fehler auftrat. In beiden Fällen können ebenfalls Interrupts ausgelöst werden.

Die Möglichkeiten, die von der seriellen Schnittstelle angeboten werden, sind äußerst vielseitig. Mit ihnen lassen sich sowohl blockierende als auch nicht-blockierende Sende- und Empfangsoperationen realisieren. Für den angestrebten Zweck lediglich Meldungen an den PC zu senden, war jedoch eine einfache, blockierende Aufrufsemantik ausreichend. Hierzu existiert eine Funktion *void put(char)*, die ein einzelnes Zeichen über die serielle Schnittstelle überträgt. Sie wird auch von der C-Bibliothek benutzt. Das Empfangen von Zeichen stellte sich als nicht triviale Angelegenheit heraus. Ürsprünglich war beabsichtigt den Datenempfang interruptbasiert abzuwickeln. Dies funktioniert jedoch nur, wenn auf ein gleichzeitiges Senden verzichtet wird, da der Baustein die von ihm gesendeten Zeichen selbst mehrmals wieder empfängt. Das gleiche Problem entsteht, wenn das Empfangen durch blockierende Funktionen durchgeführt werden soll. Bisher wurde keine befriedigende Lösung für dieses Problem gefunden. Um nicht-blockierende Aufrufe zu ermöglichen, müssen weiterhin Synchronisationsmechanismen eingeführt werden, die verhindern, dass Zeichen verloren gehen oder nicht in der richtigen Reihenfolge übertragen werden.

Display

Das Display ist kein Bestandteil des eigentlichen Mikrocontrollers, sondern ein von LEGO hinzugefügtes Peripheriegerät. Es hat auch keinerlei Bedeutung für die Java-Laufzeitumgebung. Allerdings eignet es sich um aktuelle Systemzustände anzuzeigen, wie es auch schon bei der LEGO-Firmware der Fall ist. Deshalb

wird an dieser Stelle ein kleiner Überblick über gegeben, wie der Prozessor mit dem Display kommunizieren kann. Wie schon in Abschnitt 2.3 angedeutet geschieht dies über das serielle I^2C Protokoll. Weil es seriell ist, werden für die Kommunikation zwischen Prozessor und dem für das Display zuständigen Controller nur zwei Pins des sechsten Ports benötigt. Einer der beiden übernimmt dabei die Rolle des Takts, der andere die Rolle der Datenleitung. Will man eine Bit übertragen muss dessen Wert so lange an der Datenleitung anliegen, bis der Takt von '0' auf '1' und wieder zurück gewechselt ist. Nach dem achten Bit sendet der Display-Controller ein Status-Bit, das darauf schließen lässt, ob die Übertragung erfolgreich war oder wiederholt werden muss. Wechselt der Pegel des Datenleitung, während der Takt auf '1' steht, so wird dies je nach Richtung des Wechsels entweder als Beginn oder Ende einer Übertragung gewertet.

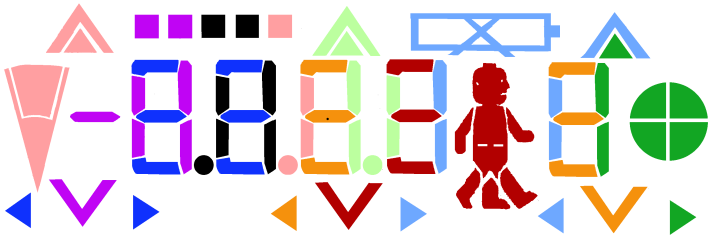


Abbildung 4.5: gruppierte Segmente des Displays

Das Display selbst besteht aus 70 Segmenten, die in neun Gruppen zu je acht Bit angeordnet sind (Abbildung 4.5, Gruppen farblich gekennzeichnet), wobei in zwei dieser Gruppen jeweils ein Bit keinerlei Auswirkung auf das Display besitzt. Das Protokoll ermöglicht es sowohl den Wert aller Segmente sequentiell, wie auch einzelne Segmente oder Gruppen von Segmenten zu schreiben. Das Startsegment wird in einem Protokollheader angegeben.

```
void cputc(char, int);
void cputs(char*);
```

Abbildung 4.6: Schnittstelle des Displays

Um das Display komfortabel ansprechen zu können wurden einige Funktionen von legOS ([Nog]) übernommen. Somit wird für das Display die in Abbildung 4.6 gezeigte Schnittstelle zur Verfügung gestellt. *cputc* gibt ein Zeichen an einer der fünf Stellen des Displays aus. *cputs* dagegen erwartet eine Zeichenkette und gibt deren vier erste Buchstaben auf dem Display aus.

4.5 Funktionen der C-Standard-Bibliothek

Da mit der Programmiersprache C gearbeitet wurde, es aber keine auf den H8/300 abgestimmten Standardbibliotheken gibt, wurden diese zum Teil neu geschrieben. Das ist nötig, da einige Befehle aus diesen Bibliotheken, wie zum Beispiel das Kopieren von Arrays oder Strings, sowie die Ausgabe von Zeichenketten oft verwendet und auch von der darrüberliegenden Java-Schicht benötigt werden. Da sich jedoch das Spektrum der benötigten Funktionen auf ein Minimum beschränkt, würde es wenig Sinn machen alle in [KR90] definierten C-Bibliotheken nach zu programmieren. Stattdessen wurde auf Grundlage der Vorauswahl, die bereits für JX getroffen wurde, der Funktionsumfang noch einmal reduziert. Grund hierfür war vor allem der Platzmangel.

Bevor die eigentlichen Funktionen implementiert oder von JX übernommen werden konnten, mussten zunächst die in Abschnitt 2.1 erwähnten Unzulänglichkeiten bei den Rechenoperationen beseitigt werden, da sonst keinerlei Rechenoperationen mit Zeigern möglich ist, die jedoch vor allem zum Kopieren von Speicherstellen oder zur Berechnung von Adressen benötigt werden. Hier wurde auf die GCC-Standardbibliothek zurückgegriffen, wobei die betreffenden Funktionen für vorzeichenlose und vorzeichenbehaftetet Zahlen mit einer Breite von 16 und 32 Bit daraus kopiert und zu einer Bibliothek gebunden wurden. Allerdings traten bei der Arbeit mit dieser Bibliothek Unzulänglichkeiten bei Rechenoperationen mit großen Zahlen auf, die oft Endlosschleifen erzeugten. Umgangen werden kann dies durch eine andere Befehlsfolge im Programmcode oder durch Verwendung der im ROM vorhandenen Funktionen, was auch zu einer leichten Platzersparnis führen kann.

Die letztlich implementierten Funktionen der C-Bibliothek sind zur Ausgabe und Manipulation von Zeichenketten, zur Konvertierung von Zeichenketten in Ganzzahlentypen, sowie zum Kopieren von Speicher oder Zeichenketten bestimmt. Während bei den meisten davon die einzigen Änderungen zu JX daraus bestanden aus 32-Bit-Datentypen 16-Bit-Datentypen zu machen, waren die Änderungen bei *int printf(const char *fmt,...)* beziehungsweise *int vprintf(const char *fmt, va_list args)* etwas größer, da in der JX-Version davon ausgegangen wurde, dass alle Datentypen, die als Argumente in Frage kommen vier Bytes groß sind. Da jedoch auf dem H8/300 Zeiger und *int* nur zwei Bytes groß sind, *long* allerdings vier Bytes, es jedoch keine Situation gibt, in der es vorstellbar wäre, dass unbedingt Daten vom Typ *long* dargestellt werden müssen, wurde die Unterstützung für Datentypen mit Längen von mehr als zwei Byte entfernt. Gleiches gilt für sämtliche Formatierungen, die *printf* nach [KR90]

unterstützen soll. Diese Reduktion auf das Wesentliche reduziert die Größe der Ausgabefunktionen auf ein gerade noch akzeptierbares Maximum, von ca. einem Kilobyte, was immer noch relativ viel ist, wenn man sich vor Augen hält, dass es sich hierbei nicht um eine Kernfunktionalität, sondern um einen Zusatz handelt, der nur dazu gedacht ist, eine Möglichkeit zu bieten, die Vorgänge auf dem Baustein zu visualisieren.

4.6 Domains

Eine Domain ist nach wie vor die Einheit der Datenkapselung. So hat jede Domain weiterhin einen Speicherbereich zur Verfügung, den sie auf die Threads aufteilen kann, eine eindeutige ID, sowie einen Zustand. Der komplett verfügbare Speicher wird vorerst initial auf eine Anzahl von Domains aufgeteilt, wobei diese Anzahl nicht fest ist, sondern sich aus dem noch verfügbaren Speicherplatz und des pro Domain vorgesehenen Speichers berechnet. Dieses Schema wurde gewählt, weil es auf Grund der zeitlichen Beschränkungen am praktikabelsten scheint. Jedoch ist es ratsam dies so zu erweitern, dass der verfügbare Speicherplatz vom Benutzer variabel auf die benötigt Anzahl von Domains aufgeteilt werden kann. Wovon jedoch abgesehen werden sollte ist eine, wie in JX vorhandene, über den Domains stehende Speicherverwaltung, da deren Platzbedarf größer ist, als ihr Vorhandensein Flexibilität verspricht, da sich dadurch die potentiell mögliche Menge an Anwendercode stark verringert. Nimmt man zum Beispiel an, dass den Benutzerprogrammen zehn Kilobyte zur Verfügung stehen und verwaltet man diese als Bitliste mit einem Bit pro Byte, gehen dadurch 1.25 Kilobyte verloren. Selbst bei einer Verwaltung von zwei oder vier Byte pro Bit wäre der Platzbedarf zu hoch, da die Bitliste allein nicht ausreicht, sondern zusätzlich noch Code für ihre Verwaltung benötigt wird.

Die Speicherverwaltung ist also ein elementarer Bestandteil der Domains geworden. Kann bei Initialisierung der Domainverwaltung auf Grund von Speichermangel keine Domain erzeugt werden, wird der Programmablauf abgebrochen, weil eine Fortführung ohne Domain Zero keinen Sinn machen würde. Jede Domain ist für die Verwaltung ihres Speichers selbst verantwortlich, das heißt es existieren Funktionen, die es ermöglichen Speicher von einer bestimmten Domain zu allokieren oder allokierten, nicht mehr benötigten Speicher wieder an diese zurück zu geben. Hierzu wurden die Speicherverwaltungsalgorithmen aus [KR90] so angepasst, dass sie in den benötigten Kontext passen. Um initial zu wissen, ab welchem Bereich des Speichers keine Daten und kein Code mehr stehen,

wird im Linkerscript als letzter Eintrag in den Bereich *ram* ein Symbol *finalizer* (Abbildung 2.6) eingefügt, durch dessen Adresse der Beginn des freien Speichers dargestellt wird.

Zusätzlich zur Speicherverwaltung ist eine Domain auch Ursprung für eine gewisse Anzahl von Threads, für deren Verwaltung sie auch zuständig ist. Realisiert wird dies durch eine doppelt verkettete Liste, in die alle laufbereiten Threads der Domain eingehängt werden. Die Schnittstelle ist aus Abbildung 4.7 ersichtlich.

```
void Domain_registerThread(Thread*);
Thread *Domain_removeThread(Thread*);
Thread *Domain_getNext();
Thread *Domain_schedule();
void *domainMalloc(Domain*, int);
void domainFree(Domain*, void*);
```

Abbildung 4.7: Schnittstelle der Domains

Domain_schedule bestimmt mit Hilfe von *Domain_getNext* den nächsten lauffähigen Thread aus der Warteliste, entfernt ihn daraus und liefert ihn zurück. Gibt es keinen lauffähigen Thread, ist die Warteschlange also leer, so wird der aktuell laufende Thread untersucht. Ist es ein Thread dieser Domain, erhält die aufrufende Funktion ihn, andernfalls *NULL*, zurück.

Zusätzlich zu all dem, enthält eine Domain für jede im System vorhandene Bibliothek einen lokalen Teil, so dass der ausgeführte Java-Code zum Beispiel in jeder Domain den domainspezifischen Wert eines statischen Feldes einer Klasse sieht. Wäre dieses Feld global, also für alle Domains gleich, wäre die Kapselungseigenschaft der Domains verletzt.

4.7 Threads

Threads sind nach wie vor die Aktivitätsträger des Systems. Jeder Thread ist eindeutig einer Domain zugeordnet, die ihm auch den Speicherplatz für seinen Stack und seinen Kontrollblock zur Verfügung stellt. Die Schnittstelle ist in Abbildung 4.8 aufgeführt.

createThread erzeugt einen neuen Thread in der angegebnen Domain mit dem angegebenen Namen. Die Programmausführung des Threads wird an der Adresse *thread_start* beginnen, wobei diese Funktion die Argumente bekommt, auf die *argv* zeigt. Beim Erzeugen wird zunächst von der Domain Speicherplatz für den Thread

```
Thread *createThread(struct Domain *domain, char *name,
                     unsigned short *thread_start,
                     unsigned short *argv);
void thread_pause();
void thread_exit();
void thread_kill(Thread* thread);
void thread_block(Thread *thread);
void thread_unblock(Thread *thread);
```

Abbildung 4.8: Schnittstelle der Threads

angefordert und die entsprechenden Werte im Kontrollblock vorinitialisiert. Abschließend wird der Thread in seiner Domain als ablaufbereit registriert und somit in die Warteschlange eingetragen. *thread_pause* gibt freiwillig die Kontrolle über den Prozessor ab, während *thread_exit* den aktuell laufenden Thread beendet und die von ihm belegten Speicherressourcen wieder frei gibt. *thread_kill* beendet einen anderen Thread, nimmt ihn aus allen Warteschlangen und gibt die von ihm belegten Speicherressourcen wieder frei.

thread_block und *thread_unblock* dienen im Wesentlichen dazu blockierende Aufrufsemantiken zu realisieren, zum Beispiel für Semaphoren. Falls die block-Funktion mit dem aktuell laufenden Thread aufgerufen wird, setzt sie seinen Status auf blockiert und ruft den Scheduler auf. Andernfalls entfernt sie ihn aus der Domainwarteschlange. Die unblock-Funktion reiht den Thread wieder in die Warteschlange seiner Domain ein und setzt seinen Status auf lauffähig.

4.8 Scheduling

Das Scheduling in JX ist präemptiv, was in dieser Form übernommen wurde. Deshalb wird ein Taktgeber benötigt, der in regelmäßigen Abständen einen Interrupt auslöst und es so dem Kern ermöglicht zum Beispiel den aktuellen Thread zu wechseln. Nach dem Scheduler soll noch der Schlafmodus des RCX betrachtet werden, der dazu geeignet sein kann Energie zu sparen, indem er den *idle*-Thread ersetzt.

Scheduler

Um dem System in regelmäßigen Abständen die Kontrolle zurück zu geben und somit präventives Scheduling zu ermöglichen, wird ein Taktsignal benötigt, das in regelmäßigen Abständen Interrupts auslöst und den dazugehörigen Interrupthandler aufruft.
Neben viel zusätzlicher Funktionalität, die [Ren03] entnommen werden kann, stellt der 16-Bit-Timer einen Interrupt zur Verfügung, der Auftritt, wenn der aktuelle Wert des 16 Bit breiten Zählerregisters mit dem Inhalt eines bestimmten Vergleichsregisters übereinstimmt. Gleichzeitig mit dem Auslösen des Interrupts wird der Zähler auf null zurückgesetzt. Das Taktsignal kann auf verschiedene Bruchteile des Systemtakts gesetzt werden. Im vorliegenden Fall wurde dazu der Faktor $\frac{1}{32}$ gewählt, was bedeutet, dass der Zähler etwa alle $500\mu s$ um eins erhöht wird.

Für den 16-Bit-Timer steht als Schnittstelle lediglich die Funktion *void set_clock_intervall(unsigned short ms)* zur Verfügung, mit der die Länge einer Zeitscheibe für den Scheduler gesetzt werden kann. Das Argument wird in Millisekunden angegeben und innerhalb der Funktion in den entsprechenden Zählerwert umgerechnet. Dieser wird anschließend in das Vergleichsregister geschrieben.

Im Scheduler werden Unterschiede zu JX sichtbar. Das zweistufige Scheduling wurde zwar nicht ganz aufgehoben, aber mit Rücksicht auf die beschränkten Platzgegebenheiten stark vereinfacht, jedoch durchaus so, dass es keine allzu großen Mühen bereiten sollte es wiederherzustellen. Der Scheduler verwaltet eine Warteschlange von Domains, die laufbereite Threads besitzen. Jede Domain, die diesen Zustand hat, trägt sich nachdem sie diesen Zustand erlangt hat, selbstständig in diese Liste ein und entfernt sich auch wieder daraus, falls sie ihn verlieren sollte.

Wie bereits im Abschnitt über die Interruptbehandlung angedeutet wird beim Auftreten eines Interrupts eine Interruptbehandlungsroutine aufgerufen. Diese heißt im Falle des 16-Bit-Timer-Interrupts *_ocia*. Sie sichert sämtliche Register und verschiebt die bereits hardwareseitig gesicherten Befehlszähler und CCR vom Stack in den Kontrollblock des Threads. Der Stack selbst wird so aufbereitet, dass darauf keinerlei Informationen mehr darrüber enthalten sind, ob der Scheduler von einem Interrupt aufgerufen wurde oder aber ob ein Thread den Prozessor freiwillig abgegeben will. In diesem Fall wird die Funktion *thread_pause* aufgerufen, in deren Ablauf ebenfalls alle CPU-Register, sowie das CCR gesichert werden. Zusätzlich müssen jedoch noch die Interrupts durch Setzen des I-Bits im CCR

deaktiviert werden. Dagegen sind keine Manipulationen am Stack nötig, weil kein Interrupt aufgetreten ist. Anschließend wird in beiden Fällen die Funktion *schedule* des Schedulers aufgerufen, die wiederum mit Hilfe von *choose_next* den nächsten lauffähigen Thread auswählt. Wird keiner gefunden, startet sie den *idle*-Thread, der bei der Initialisierung des Schedulers angelegt wird und als einziger Thread keiner Domain zugeordnet ist. Ist ein neuer Thread gefunden und ist der aktuelle Thread nicht der *idle*-Thread, wird der neue aus der Warteschlange seiner Domain entfernt, ebenso wie die Domain aus der Warteschlange des Schedulers. Der aktuell laufende Thread wird, falls es sich nicht um den *idle*-Thread handelt und der Thread nicht durch einen *thread_block* Aufruf die Kontrolle über den Prozessor verliert, in die Warteschlange seiner Domain eingefügt, woraufhin sich die Domain selbst wieder beim Scheduler registriert. Stammen der aktuelle Thread und der nächste Thread aus derselben Domain, muss diese wieder aus der Schedulerwarteschlange entfernt werden. Wird der aktuelle Thread blockiert und stammt der nächste Thread aus einer anderen Domain, so muss die Domain des aktuellen Threads in die Warteschlange des Schedulers eingefügt werden, falls sie noch einen wartenden Thread besitzt. Die Domain kann diese Aufgabe in diesem Fall nicht erledigen, weil *Domain_registerThread* nicht aufgerufen wird.

Choose_next soll den nächsten lauffähigen Thread finden und zwar so, dass jede Domain gleich oft den Prozessor bekommt, gleichgültig wie viele Threads sie besitzt. Da der Aufwand beim Wechsel einer Domain größer ist, als beim bloßen Umschalten zwischen Threads der gleichen Domain, empfiehlt es sich, die Kontrolle, falls möglich, für längere Zeit, das heißt, über mehrere Timerinterrupts hinweg, bei der gleichen Domain zu lassen. Hierzu besitzt *choose_next* einen Zähler, der die domaininternen Threadwechsel mitzählt und, sobald ein bestimmter Schwellenwert überschritten wird nach lauffähigen Threads in der nächsten Domain sucht. Wird keiner gefunden, bleibt die Kontrolle bei der aktuellen Domain, und der Zähler wird erhöht, andernfalls wird die Domain gewechselt und der Zähler auf null zurückgesetzt. Solange sich der Zähler unter dem Schwellenwert befindet, wird zunächst in der aktuellen Domain nach einem lauffähigen Thread gesucht. Findet sich hierbei keiner, weil zum Beispiel die Domain nur einen lauffähigen Thread, den Aktiven, hat, dieser jedoch gerade dabei ist sich zu blockieren oder zu beenden, wird in einer anderen Domain gesucht. Die Behandlung des Zählers unterscheidet sich in diesem Fall nicht vom vorhergehenden.

Um den nächsten Thread zu finden, wird die von der Domainschnittstelle angebotene Funktion *Domain_schedule* benutzt, die in etwa das darstellt, was in JX, die zweite Stufe des Schedulers ist. Unterschiedlich ist jedoch, dass in den Domains keine verschiedenen Arten von Schedulingverfahren eingesetzt werden können,

sondern alle nach einem einfachen Round-Robin-Verfahren arbeiten. Möchte man dies ändern um jeder Domain zu ermöglichen ein eigenes Verfahren zu installieren, so kann man den Kontrollblock einer Domain um einen Wert erweitern, der die Adresse einer Funktion enthält, die das gewünschte Verfahren realisiert. Werden dann sämtliche Aufrufe von *Domain_schedule* dorthin umgeleitet, hat jede Domain die Möglichkeit ein alternatives Schedulingverfahren zu realisieren.

Nach der Auswahl des Nachfolgethreads, wird die Assemblerfunktion *void thread_switch(Thread*)* aufgerufen, die den Kontext des Zielthreads wiederherstellen soll. Dies hat sich beim H8/300 als durchaus nicht unproblematisch herausgestellt, da es keinerlei Maschinenbefehl gibt, der ausschließlich in der Lage ist, Daten direkt vom Stack in das CCR zu bewegen. Das Problem das sich hieraus ergibt, wird einsichtig, wenn man sich vor Augen hält, dass man dafür ein Register belegen muss. Daraus folgt, dass der Inhalt dieses Registers erst dann wieder hergestellt werden kann, wenn bereits das CCR wiederhergestellt wurde, also eventuell auch die Interrupts wieder freigegeben sind und somit der Wiederherstellungsvorgang potentiell unterbrechbar wird, obwohl er nicht unterbrochen werden darf. Das hat den Grund, dass sich während der Ausführung der Funktion zwar die richtigen Registerwerte im Kontrollblock des Threads befinden, mindestens eines davon jedoch benötigt wird, um die Adresse des Kontrollblocks zu speichern. Ein weiteres muss reserviert werden, um den Wert des CCRs zwischenzuspeichern, das heißt, dass nach dem Zurückschreiben noch mindestens zwei Register falsche Werte enthalten. Werden nun durch das Zurückschreiben des CCR-Registers die Interrupts freigegeben und tritt zum Beispiel ein Timer-IRQ auf, werden die Werte, die sich aktuell in den Registern befinden in den Kontrollblock des Threads gesichert, das heißt die falschen Werte der Registers überschreiben die richtigen Wert im Kontrollblock.

Um diese Situation zu vermeiden hat man zwei Möglichkeiten. Zum Einen kann man in der Interruptfunktion des Timers prüfen, ob die Adresse von *thread_switch* auf dem Stack liegt. Da man hierfür jedoch wegen der Indirektionsstufen der Interrupts mehr als einen *frame* auf dem Stack nach oben gehen muss, empfiehlt es sich eher weniger. Eine andere Möglichkeit besteht darin den Stack so aufzubauen, wie in Abbildung 4.9 gezeigt. Gibt man dem Stack die gleiche Struktur, die er nach dem Auftreten eines Interrupts hat, kann man mit dem *rte*-Befehl gleichzeitig Werte vom Stack in das CCR schreiben und zu einer Rücksprungadresse zurückkehren, um die Ausführung des Threads fortzusetzen, wie ebenfalls aus Abbildung 4.9 zu ersehen.

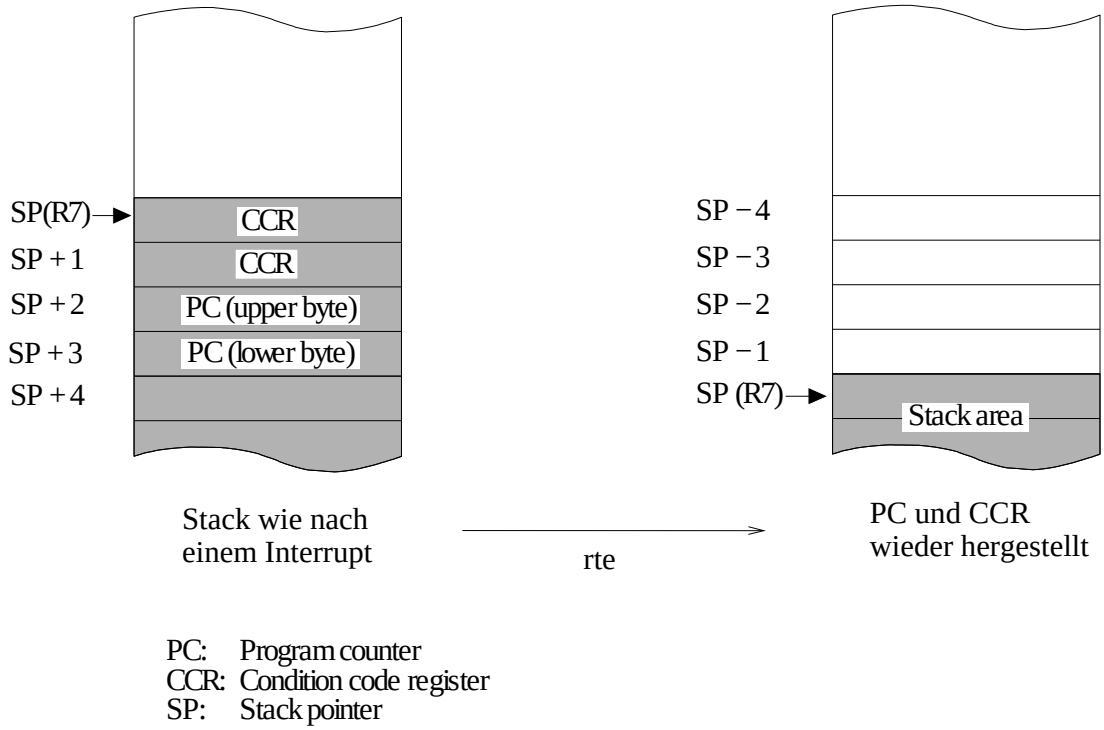


Abbildung 4.9: Wirkung von rte (nach [Ren03])

Schlafmodus

Wie bereits erwähnt kann sich der Prozessor in einem von drei Zuständen befinden. Einer davon ist der Ruhezustand, beziehungsweise einer von drei Ruhezuständen. Der Wechsel in einen Ruhezustand kann als Alternative zum Aufruf des *idle*-Threads genutzt werden. Von den drei Ruhezuständen kommt dabei nur der Schlafmodus in Frage, bei dem es sich um den schwächsten Ruhezustand handelt. Hierbei wird ausschließlich die CPU angehalten. Alle Register behalten ihren jeweiligen Wert. Erreicht wird er durch den Maschinenbefehl *sleep*, wenn gleichzeitig das SSBY-Bit im *system control register* (SYSCR, Abbildung 2.3 (b)) gelöscht ist. Die Register der Peripherie sind davon nicht betroffen. Verlassen wird dieser Zustand durch einen Interrupt, genau wie der *idle*-Thread ausschließlich nach dem Auftreten eines Ereignisses wieder abgelöst werden kann, das einen blockierten Thread wieder aktiviert. Alle anderen Ruhezustände sind nicht sinnvoll, da in diesen Fällen die Register der Peripherie gelöscht werden und somit alle Geräte erst wieder initialisiert werden müssen, bevor sie genutzt werden können.

4.9 Semaphoren

Bei einer Semaphore handelt es sich wie in [SGG03, Hof02, SP03] nachzulesen um eine nicht-negative ganze Zahl. Sie dient als Mittel zur Synchronisation von Threads und zur Realisierung von gegenseitigem Ausschluss. Auf ihr sind zwei Operationen definiert: *void P()* und *void V()*, wobei P das Betreten sowie V das Verlassen eines kritischen Abschnitts bedeutet. Im vorliegenden System ist eine Semaphore als eine Struktur realisiert, die eine Ganzzahl und einen Zeiger auf einen Thread besitzt. Die Ganzzahl wird mit einem nicht-negativen Wert initialisiert, der Zeiger mit *NULL*. Durch jede P-Operation wird die Ganzzahl erniedrigt, wenn ihr Wert positiv ist. Ist das nicht der Fall wird der aktuelle Thread in eine Warteliste eingehängt, deren Startelement durch den Thread-Zeiger erreicht wird. Die V-Operation dagegen erhöht die Ganzzahl, wenn in der Warteschlange keine Threads mehr eingehängt sind. Ist dies dagegen der Fall, wird der erste Thread daraus entfernt und in die Warteschlange des Schedulers eingehängt.

4.10 Unterschiede zu JX

Eine strikte Portierung von JX auf den RCX war aus Platzgründen und auch wegen der zu unterschiedlichen Peripherie von RCX und Intel-basierten Rechnern unmöglich. Im Folgenden sollen deshalb die veränderten Datenstrukturen dargestellt werden. Die oben genannten Datenstrukturen, Threads und Domains, sind zusammen mit den in Kapitel 5 und 6 beschriebenen Strukturen alles, was das neu entwickelte System, noch mit JX gemeinsam hat. Viele Besonderheiten des Java-Betriebssystems fielen sowohl den Begrenzungen der Zeit, als auch denen des Speicherplatzes zum Opfer.

Portale existieren im bisherigen Entwicklungsstadium nicht. Hierfür war vor allem der Mangel an Zeit ausschlaggebend. Dies stellt bisher kein Hindernis dar, da sich Kommunikation über Domaingrenzen hinweg vorerst durch ein Aufweichen der Isolation erreichen lässt; um mit der Domain Zero kommunizieren zu können werden keine Portale benötigt, sie lässt sich auch über *Fast Portals* realisieren. Jedoch stellt eine Einbindung der Portale in das System ein Ziel dar, da sie ein Charakteristikum von JX sind und weiterhin einen grundlegenden Beitrag zur Prozessisolation leisten.

Der Garbage-*Collector* (GC) fehlt aus dem Grund, weil die Entwicklung des Systems nicht so weit vorwärts gedrungen ist, um beurteilen zu können, ob sich

seine Einführung eher positiv oder negativ auswirken könnte. Prinzipiell wäre seine Existenz äußerst wünschenswert, da er dafür sorgen kann und auch soll, dass der ohnehin knapp bemessene Speicherplatz hin und wieder gesäubert wird. Allerdings wird es auch hier ein Abwägen geben müssen mit welcher Mächtigkeit er auf dem Baustein realisiert werden kann. Ebenso wird man sich nach seiner Implementierung die Frage stellen müssen, ob der Platz, den er benötigt, durch seine Funktionalität wieder eingespart werden kann. Da bisher die meisten Daten eher statischer Natur sind, also keine Objekte, die erzeugt und danach wieder gelöscht werden, sondern entweder Daten, die schon zur Kompilierzeit erstellt werden oder solche die zwar während der Initialisierung dynamisch angelegt werden, aber für das Funktionieren des Systems unabdingbar sind, hätte ein GC im Moment wenig Nutzen. Wie sich der Bedarf ändert, wenn in Zukunft Java-Code auf dem RCX abläuft, wird man abwarten müssen. Eine Alternative, die sich bietet, sobald sich über das Kommunikationssystem komplexere Protokolle realisieren lassen, wäre ein externerer GC, beziehungsweise eine ausschließlich externe Speicherverwaltung, die auf einem mit dem RCX kommunizierenden PC abläuft und auf Anfrage von Seiten des Bausteins die zur Verwaltung notwendigen Algorithmen ausführt oder im Falle des GC selbstständig den Speicher säubert. Natürlich wäre zu diesem Zweck ein Synchronisierungsmechanismus zwischen Baustein und PC nötig. Es bleibt dennoch festzuhalten, dass bereits einige Funktionen des GC Verwendung finden, so zum Beispiel die Funktion *allocObjectInDomain*, die wie der Name schon sagt, Speicher für ein Java-Objekt in einer Domain reserviert und diesen initialisiert.

Ein kleinerer und wahrscheinlich mit geringerem Aufwand als die oben angesprochenen Punkte zu behebender Unterschied ist die Tatsache, dass auf dem RCX weder Thread- noch Domainkontrollblöcke durch Java-Code angesprochen werden kann. Wobei dies nicht heißt, dass Java-Code keine Referenz auf ein Stück Speicher halten kann, das einen Thread oder eine Domain repräsentiert. Jedoch sollte dann nicht versucht werden darauf Operationen auszuführen, da dies mindestens zu einem falschen Ergebnis, aber wahrscheinlicher zu einem inkonsistenten System führen wird, weil keiner der Kontrollblöcke über eine *vtable* verfügt, über die Methoden erreicht werden könnten. Vorbeugen kann man dem dadurch, dass die Referenz auf Thread oder Domain auf Java-Ebene durch ein *Interface* repräsentiert, das keinerlei Methoden besitzt. Die einzige Gefahr, die in diesem Fall noch existiert, besteht darin, dass jemand versuchen könnte, die Methoden, die das Objekt *java.lang.Object* zur Verfügung stellt, an dieser Referenz aufzurufen.

Eine Fähigkeit von JX, die auf dem RCX ebenfalls fehlt, weil kein Bedarf dafür vorhanden war, ist die Mehrprozessorunterstützung, die auf einem Uniprozessorsystem wenig Sinn macht. Zwar ist es denkbar, dass sich mehrere RCX-Rechner zu einem Robotersystem zusammenschließen lassen, jedoch werden dies dann autonome, miteinander kommunizierende Maschinen sein und nicht ein Betriebssystem, das sich über mehrere Rechner erstreckt.

Schließlich gibt es im JX-System ein ganze Reihe von Schnittstellen, die der Java-Schicht als *Interface*s angeboten werden, in der Domain Zero jedoch mit C-Code realisiert sind. Hierzu zählen vor allem der CPU-Manager, der Domain-Manager und Memory-Objekte sowie einige andere, wobei ohne die drei genannten der Java-Code in seiner Funktionalität stark eingeschränkt sein wird, da er ohne Domain-Manager keine neuen Domains starten kann und sich ohne Memory-Objekte keine Treiber auf Java-Ebene realisieren lassen.

4.11 Zusammenfassung

In diesem Kapitel wurde der erstellte Mikrokern vorgestellt. Er bietet Schnittstellen zu vielen der im RCX vorhandenen Peripheriegeräte an und soll auch Unzulänglichkeiten der Arithmetik des RCXs bei Operationen mit Zahlen deren Länge 16 oder 32 Bit beträgt beseitigen. Dies ist in den meisten Fällen möglich, allerdings entstehen bei Berechnungen mit Zahlen deren führendes Bit gesetzt ist des öfteren Endlosschleifen, zumindest, wenn man auf die Bibliotheken des GNU Compilers zurückgreift. Motoren können gesteuert und Sensoren ausgelesen werden. Die für die Motoren angebotene Funktionalität dürfte für einen Betrieb ausreichend sein. Bei den Sensoren wäre es, wie bereits angedeutet, sicherlich sinnvoll weitere Funktionen anzubieten, die den vom Analog-Digital-Wandler zurück gelieferten, rein numerischen Wert in einen sensorspezifischen, interpretierten Wert umwandeln. Die zur Kommunikation mit dem Benutzer notwendigen Abläufe lassen sich teilweise über das Display am RCX realisieren. Eine mächtigere Schnittstelle zur Kommunikation stellt das SCI dar. Während die Funktionen des Displays für die meisten Anwendungen ausreichend sind, stellt sich die Situation beim SCI als äußerst unbefriedigend dar, da keine bidirektionale Kommunikation möglich ist. Hier sollten weiterführende Arbeiten ansetzen, da dies der Schlüssel ist, um Funktionen wie die Speicherverwaltung auf externe Rechner mit höherer Leistung auslagern zu können. Das Konzept der Domains wurde weitestgehend aus JX übernommen, auch wenn auf einige Felder der Domainkontrollblöcke verzichtet wurde. Gleiches gilt für die Threads und deren

Kontrollblöcke. Das Scheduling wurde reduziert. Hierfür war vor allem die Ersparnis an Code das ausschlaggebende Argument. Zum Ende des Kapitels wurde auf einige Unterschiede zwischen JX und dem entstandenen System hingewiesen, von denen einige durch die unterschiedliche Hardware, andere durch den Mangel an Platz und wiederum andere durch zeitliche Gründe bedingt sind.

Der entstandene Mikrokern kann mit geringen Modifikationen als allein stehender Kern benutzt werden, der nicht auf die in den folgenden Kapiteln dargestellte Java-Laufzeitumgebung angewiesen ist. Hierzu können die Dateien des Kerns als Bibliothek gebunden werden. Diese führt alle notwendigen Initialisierungen durch und verzweigt anschließend zu einer Funktion des Anwenders mit festgelegtem Namen.

Kapitel 5

Java-Laufzeitsystem

Ziel und damit Hauptteil der vorliegenden Arbeit war die Entwicklung einer statischen Java-Laufzeitumgebung für den RCX. Das Hauptaugenmerk lag hier vor allem in deren statischem Charakter, da das Laufzeitsystem an sich bereits durch JX vorgegeben war. Bevor jedoch im folgenden Kapitel auf die statischen Eigenschaften eingegangen wird, soll hier ein Überblick gegeben werden, wie sich das entstandene System zur Laufzeit verhält. Dazu werden auch alle benutzten Datenstrukturen vorgestellt und der Startvorgang des Systems bis hin zum Start von Java-Code beschrieben.

5.1 Angestrebtes Verhalten

Zur Ausführung gebracht werden sollen eine Reihe von Java-Klassen. Diese sind wie bereits aufgezeigt in Form von Komponenten zusammengefasst. Der Ausdruck Bibliotheken wird im Folgenden synonym zu Komponenten gebraucht. Die in den Bibliotheken vorhandene Information muss dabei ins System gebracht werden. Hierbei muss beachtet werden, dass Bibliotheken voneinander abhängen können, also zusammen mit einer abhängigen Bibliothek auch die Bibliothek vorhanden sein muss, von der sie abhängig ist. Die in den Bibliotheken enthaltene Information ist innerhalb des Systems in zahlreichen Strukturen abgelegt, die in den folgenden Absätzen beschrieben werden. Generell lässt sich jedoch zwischen statischen und dynamischen Strukturen unterscheiden. Dieser Unterschied begründet sich im Verwendungszweck der einzelnen Strukturen und dadurch auch in der Dynamik, mit der sie erschaffen und auch wieder zerstört werden können. Die Dynamik der statischen Strukturen beschränkt sich in JX dabei dadurch, dass sie lediglich einmal erzeugt und danach nie wieder verändert werden. Sie

sind global, also für jede Domain in der gleichen Form verfügbar. Geeignet für statische Strukturen sind zum Beispiel Programmcode und Zeichenketten, aber auch Vererbungsbeziehungen zwischen Klassen oder *Interface*s oder Implementierungsbeziehungen zwischen Klassen und *Interfaces*.
Dynamische Strukturen dagegen sind domainspezifisch und repräsentieren unter anderem Java-Objekte. Da Domains während des Programmablaufs erzeugt und auch wieder zerstört werden können und die angesprochenen Strukturen auf eine Domain angewiesen sind, können diese erst erzeugt werden, wenn die Domain existiert. Ebenso müssen sie spätestens dann wieder zerstört werden, wenn die Domain zerstört wird. Ebenfalls zu den dynamischen Strukturen zählen die auf Java-Ebene erzeugten Objekte, deren Lebenszeit in der Regel weit geringer als die einer Domain ist.

In JX wird nach dem Initialisieren der Hardware die erste Domain gestartet. Bei dieser handelt es sich um die Domain Zero, die als einzige Domain in C implementiert ist. Während ihres Initialisierungsvorgangs werden die von ihr benötigten Bibliotheken geladen und daraus die statischen Datenstrukturen erzeugt. Anschließend werden die dynamischen Datenstrukturen für die Domain selbst erzeugt. Beim Erzeugen einer neuen Domain müssen nur noch Bibliotheken geladen werden, die noch nicht im System vorhanden sind und deren statische Strukturen somit noch nicht erzeugt sind. Dynamische Strukturen dagegen müssen alle erzeugt werden, wenn eine neue Domain erzeugt wird, egal, ob die Bibliothek vorher bereits geladen war oder nicht.
Da dieses Vorgehen wegen der Vorgaben an Platz für den RCX schlecht geeignet ist, war eine der Zielsetzungen bei der Entwicklung des vorliegenden Systems die statischen Datenstrukturen für alle Domains bereits fertig erzeugt in das System einzubringen. Das Erzeugen der Strukturen soll dabei auf einem externen Rechner vonstatten gehen. Das Resultat davon wird verschiedenen C-Dateien abgelegt, so dass es möglich wird es mit dem Quellcode des in Kapitel 4 beschriebenen Mikrokerns zu kompilieren.

In diesem Kapitel werden zunächst alle wichtigen im System vorhandenen Datenstrukturen vorgestellt. Begonnen wird mit den statischen Strukturen. Im Anschluss daran werden Domain Zero Strukturen vorgestellt. Dabei handelt es sich um die gleichen programmiersprachlichen Strukturen, wie bei statischen Strukturen, jedoch ist der Verwendungszweck etwas anders. Als drittes folgen die dynamischen Strukturen. Den Abschluss des Kapitels bildet die Darstellung des Startvorgangs, der nun anders abläuft als in JX.

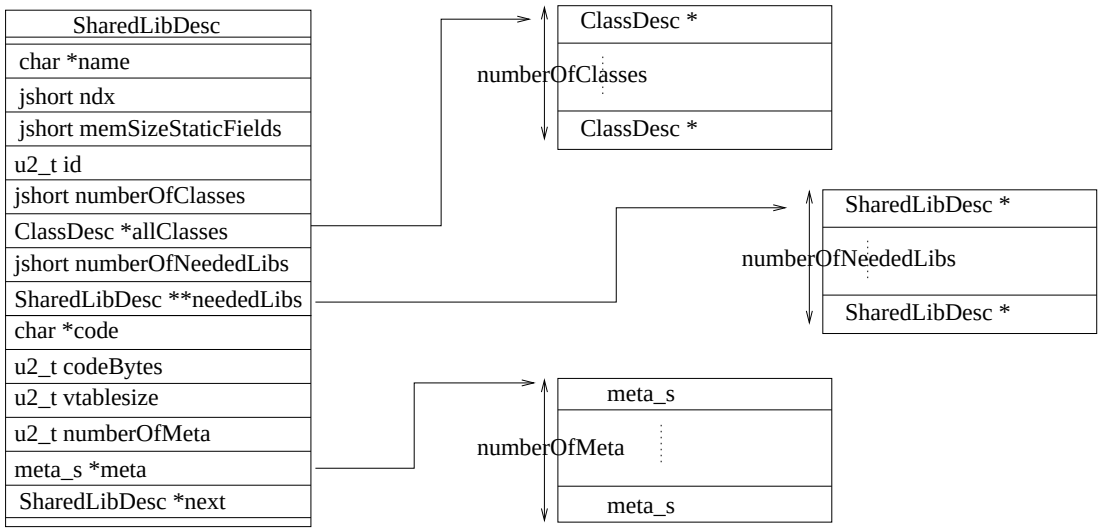


Abbildung 5.1: Bibliotheksdeskriptor

5.2 statische Datenstrukturen

In diesem Abschnitt werden alle relevanten statischen Datenstrukturen vorgestellt. Am seinem Ende findet sich mit Abbildung 5.4 eine Übersicht, die verdeutlichen soll wie die Strukturen zusammenhängen.

Bibliotheken

An statischen Datenstrukturen sind in JX zunächst die Bibliotheken selbst vorhanden. Zusammen mit den aus ihnen hervorgehenden *SharedLibDesc*-Strukturen, die, wie in Abbildung 5.1 gezeigt, alle Teile einer Bibliothek verwalten, die von allen Domains zusammen genutzt werden können. Dabei handelt es sich vor allem um den Code der Methoden und Zeiger auf die in der Bibliothek vorhandenen Klassendeskriptoren (*ClassDesc*). Weiterhin enthält jede Bibliothek ein Feld von Zeigern, auf alle anderen von ihr benötigten Bibliotheken. Da Bibliotheksdeskriptoren dynamisch erzeugt und in einer verketten Liste verwaltet werden, enthält ein *SharedLibDesc* zusätzlich noch einen Zeiger auf die folgende Bibliothek. Als weitere Felder sind noch Name und Identitätsnummer (*id*) zu nennen. Alle anderen Felder sind entweder für statistische Zwecke oder zur Optimierung der Geschwindigkeit vorgesehen.

Klassen

Die in Abbildung 5.2 gezeigten Klassendeskriptoren beschreiben die in den Bibliotheken enthaltenen Java-Klassen mit ihren Attributen und Methoden. Auch sie verfügen über einen Namen, sowie einen Klassentyp, das heißt vor allem eine Unterscheidung zwischen *Interface*s und *Class*es. Gibt es eine Basisklasse, was mit Ausnahme von *java.lang.Object* bei allen Klassen der Fall ist, zeigt das Feld *superclass* auf diese. Zusätzlich hierzu sind alle von dieser Klasse implementierten *Interfaces* durch ein Feld von Zeigern auf deren Klassendeskriptoren erreichbar. Zwar wird der Code einer Klasse zusammen mit der Bibliothek und nicht mit der Klasse geladen, dennoch besitzt eine Klasse mehrere Zeiger auf die von ihr realisierten Methoden. Diese sind durch die Elemente *vtable, methodVtable* und *methods* erreichbar.

Die Unterschiede der eben genannten Felder in der Bedeutung für den Klassendeskriptor und die Klasse sollen noch ein wenig genauer erläutert werden. Alle von einer Klasse direkt implementierten Methoden, also alle Methoden, die weder geerbt, noch abstrakt sind, finden sich in allen vier Konstrukten wieder. Alle geerbten Methoden finden sich in allen Feldern, mit Ausnahme von *methods,* da dieses lediglich auf die Methodendeskriptoren der direkt implementierten Klassen verweist. *methodVtable* verweißt ebenfalls auf Methodendeskriptoren, jedoch für alle Methoden, sowohl direkt implementierte als auch geerbte. Die Menge der zweitgenannten Methodendeskriptoren ist in fast allen Fällen eine Obermenge der erstgenannten, da eine Klasse durchaus Methoden von anderen Klassen wie zum Beispiel *java.lang.Object* erbt und diese nicht überschreibt, so dass deren Methodendeskriptoren erreichbar sein müssen. Das bedeutet auch, dass im zweiten Fall die Ziele der Zeiger im Speicher gestreut liegen können. Aus diesem Grund reicht es im ersten Fall aus einen Zeiger auf ein Array bereitzustellen, da die Methodendeskriptoren im Speicher aufeinander folgen, während im zweiten ein Zeiger auf ein Zeigerarray benötigt wird. Da jeder Methodendeskriptor einen Zeiger auf den zur Methode gehörigen Code besitzt, ist es möglich vom Klassendeskriptor zum Code jeder Methode zu gelangen, falls sie implementiert ist. Jedoch sind hierfür vier Indirektionsstufen nötig, um den gewünschten Code zu erreichen, falls man von einem Zeiger auf einen Klassendeskriptor ausgeht. JX wurde jedoch weitestgehend auf Geschwindigkeit optimiert. Aus diesem Grund verfügt jeder Klassendeskriptor über eine *vtable,* ein Array von Zeigern auf Funktionen. Ergänzt wird dieses Konstrukt durch eine dazugehörige Symboltabelle (*vtableSym*), die zu jeder Methode aus der *vtable* den Namen der sie implementierenden Klasse, den Namen der Methode und deren Signatur beinhaltet, was bedeutet, dass sie drei Mal so viele Einträge enthält wie die eigentliche *vtable*. Auf dem RCX

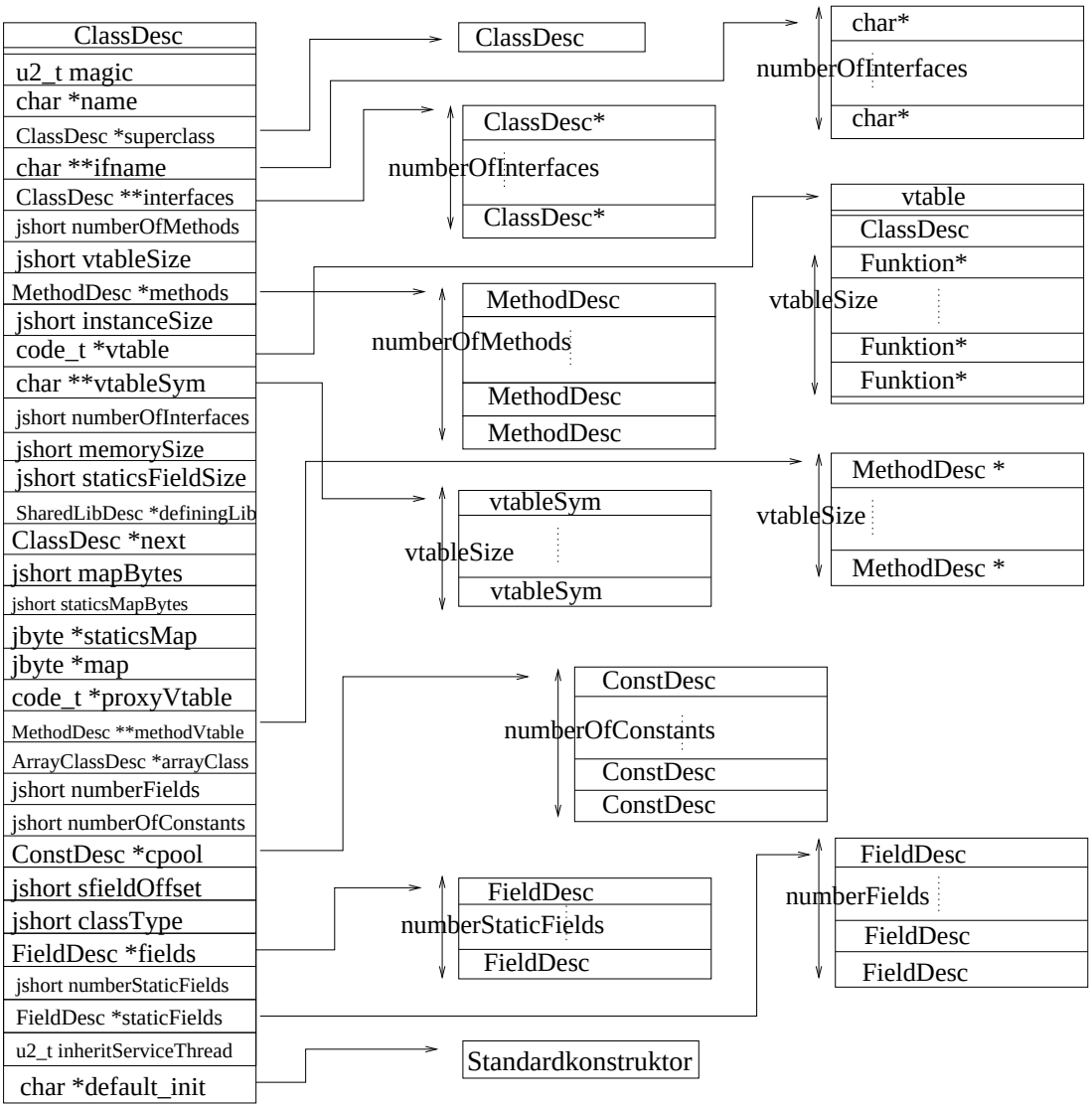


Abbildung 5.2: Klassendeskriptor

wäre es aus Platzgründen eventuell angebracht diese zusätzlichen Optimierungen einzusparen, jedoch wären die hieraus entstehenden Gewinne eher in kleineren Größenordnungen anzusiedeln. Zusätzlich unattraktiv wird diese Vorgehensweise, weil sie große Änderungen am Java-zu-Maschinencode-Übersetzer nach sich ziehen würde. Diese Aussage gilt generell für alle Optimierungen, die in JX bei der Java-Code Anbindung eingeführt wurden.

Des Weiteren besitzt ein Klassendeskriptor einen Zeiger zu seiner Bibliothek, sowie zu dem ihm in der Liste folgenden Deskriptor. Neben den Methoden stellen die Felder (*Fields*, Variablen) einer Klasse ihr prägendstes Charakteristikum dar, wobei hierbei zwischen statischen (Klassenvariablen) und nicht-statischen (Objektvariablen) Feldern unterschieden wird. Ein statisches Feld ist für alle Klassen einer Domain gleich, besitzt also gleiche Adresse und gleichen Inhalt. Ein nicht-statisches Feld ist objekt-lokal, das heißt, dass ein aus einer Klasse instanziiertes Objekt einen Speicherplatz für dieses Feld zugewiesen bekommt. Da es in diesem Abschnitt ausschließlich um statische Datenstrukturen gehen soll, Felder jedoch entweder auf eine Domain oder ein Objekt beschränkt sind, können sie noch nicht angelegt werden. Jedoch enthält jeder Klassendeskriptor zwei Arrays von Felddeskriptoren (*FieldDesc*), eines für die statischen, eines für die nicht-statischen Felder der Klasse. Weiterhin gibt es zu jedem Felddeskriptor eine *Bitmap,* die beschreibt, welche Attribute ein Feld besitzt, das heißt, welche Zugriffsrechte (*private, protected, package, public*) es besitzt. Jeder Klassendeskriptor besitzt außerdem noch einen Zeiger auf den Standardkonstruktor der Klasse.

Felddeskriptoren sind eine einfache Datenstruktur, in der lediglich festgehalten wird, welchen Namen eine Variable besitzt, ob es sich um einen primitiven Typ oder um ein Objekt handelt, sowie welchen Abstand es zum Beginn des Speicherbereichs hat. Letzteres ist nötigt, da zunächst der benötigte Platz für alle Felder einer Klasse beziehungsweise eines Objekts allokiert und später jedem Feld ein bestimmter Bereich in diesem Speicher zugewiesen wird. Da nicht jedes Feld gleich viel Speicher benötigt, reicht es nicht aus den allokierten Speicher als Array zu betrachten und einem Feld einen Index zuzuweisen. Stattdessen muss der *offset* in Bytes angegeben werden, an dem das Feld zu finden ist.

Methoden

Der Inhalt der in Abbildung 5.3 gezeigten Methodendeskriptoren beschreibt im Wesentlichen die von einer Methode angebotene Schnittstelle. Dabei handelt es sich vor allem um den Typ der Argumente, sowie die Startadresse des Codes.

Jedoch werden auch Informationen über die Größe der Methode, ihre Position im Quellcode, die Anzahl und Größe der lokalen Variablen, sowie die von ihr geworfenen Ausnahmen (*Exceptions*) zusammen mit einer Methodensignatur, die über den Rückgabewert Auskunft gibt, bereitgestellt. Ein wichtiges Element stellen außerdem die Symboldeskriptoren dar. Diese beschreiben Symbole, die im kompilierten Java-Code der Klassen enthalten sind. Notwendig werden sie, falls die Klasse zum Beispiel Referenzen auf andere Klassen besitzt, die zur Übersetzungszeit nicht aufgelöst werden können, weil es unmöglich ist vorher zu wissen an welcher Adresse sich die jeweils benötigte Klasse oder Methode oder aber auch die Instanz eines Objekts zur Laufzeit befindet. Weil jedoch nicht nur Klassen das Ziel von nicht auflösbaren Adressen sein können, sondern auch Funktionen des Kerns oder dessen VM, die Größe von Datenstrukturen oder auch der *Stack Pointer*, gibt es eine Vielzahl von Symboldeskriptoren, die oft unterschiedlich in ihrer Größe, also in der Menge ihrer Attribute, sind. Dennoch besitzen alle eine identische Grundmenge an Attributen, wobei es sich vor allem den Typ und die Position des zu ersetzenden Symbols handelt.

Abbildung 5.4 zeigt einen erweiterten Überblick über alle statische Datenstrukturen.

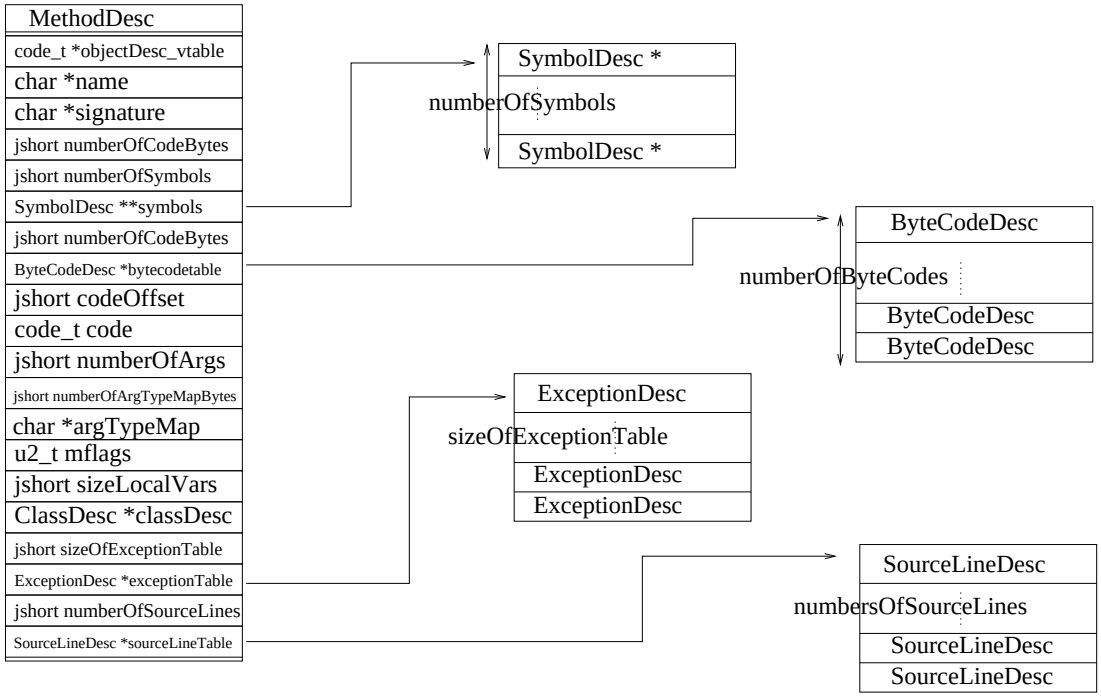


Abbildung 5.3: Methodendeskriptor

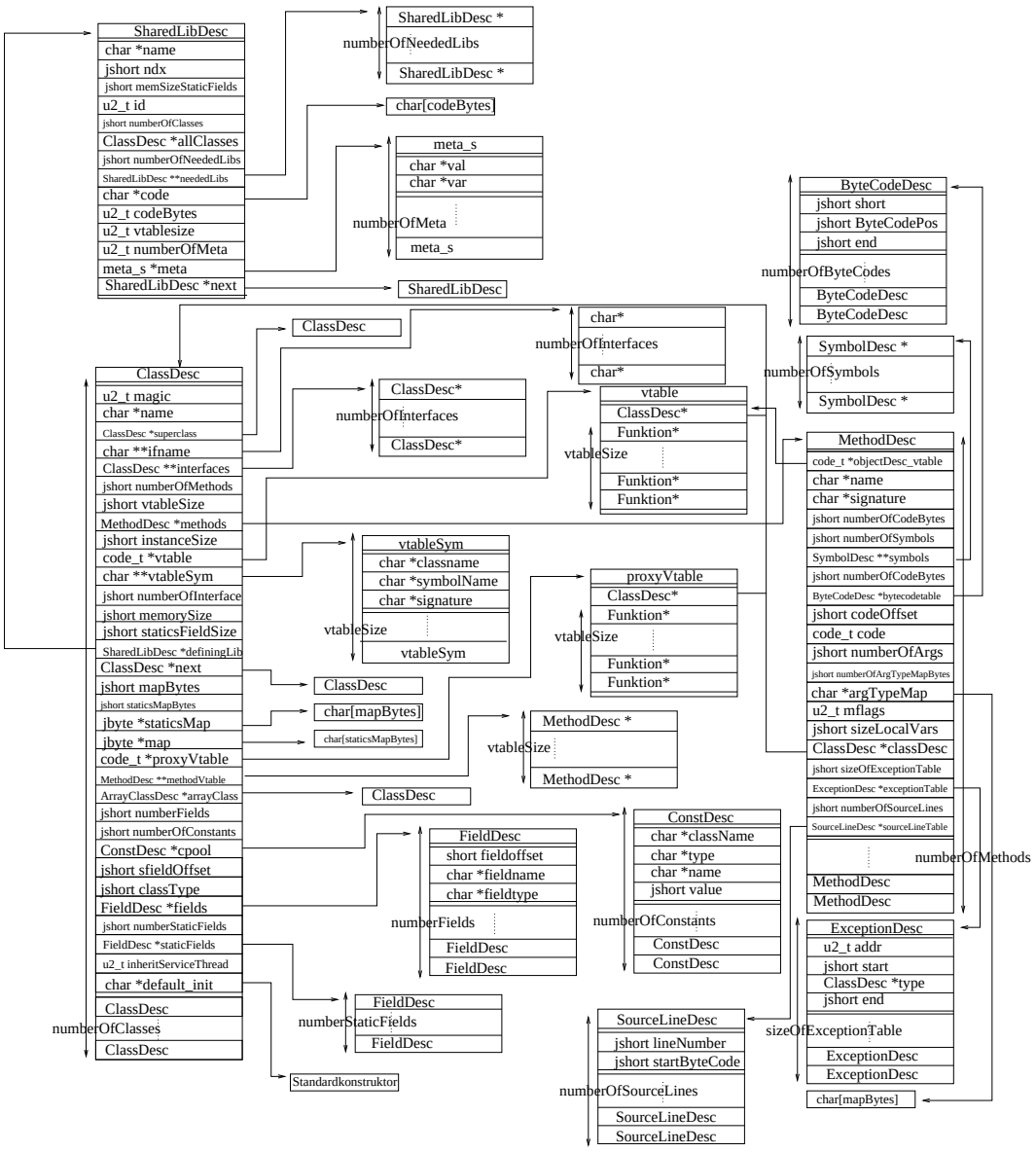


Abbildung 5.4: Abhängigkeiten zwischen statischen Strukturen

5.3 Domain Zero Strukturen

Der Großteil aller im System vorhandenen Strukturen wird, wie bereits angesprochen, aus Bibliotheken gewonnen. Einige davon werden jedoch anders erzeugt. Diese sind Teil der Domain Zero und in Folge dessen wie diese in C implementiert. Bevor nun im weiterhin Folgenden die dynamischen Datenstrukturen betrachtet werden, soll noch kurz aufgezeigt werden, welche Rolle diese Domain Zero

Strukturen spielen, wenn die statischen Strukturen aus den Bibliotheken erzeugt werden.

Absolut unabdingbar um später überhaupt die Strukturen für statische Bibliotheken und deren Klassen erstellen zu können, ist die Existenz der Basisklasse *java.lang.Object*, die als Grundlage für alle anderen Klassen dient. Sie muss deshalb vorhanden sein, weil die meisten anderen Klassen die von *java.lang.Object* zur Verfügung gestellten Methoden nicht überladen und deswegen die Adresse des Codes dieser Methoden im Speicher benötigen, um ihre eigenen *vtable*s richtig aufbauen zu können. Parallel dazu werden in Java auch Arrays als eigene Objekte behandelt, für die ebenfalls eine Grundfunktionalität gefordert wird, die durch eine eigene *vtable* realisiert und zur Verfügung gestellt wird. Da nun diese beiden Strukturen während der Erstellung der statischen Strukturen vorhanden sein müssen, also auch dann, wenn die Bibliotheken auf einem anderen Rechner aufbereitet werden, entsteht keinerlei Mehraufwand, wenn sie dem gleichen Prozess unterworfen werden, dem alle aus Bibliotheken gewonnenen Klassendeskriptoren unterworfen sind. Somit werden auch sie zusammen mit dem Kern fertig erzeugt auf den RCX geladen. Gleiches gilt ebenso für Domain Zero Klassen.

Eine weitere Struktur, die für die Funktionalität der Java-Schicht unabdingbar ist und aus diesem Grund bereits beim Laden der Bibliotheken vorhanden sein muss, stellt die *Virtual Machine* (VM) dar. Da es sich bei der von ihr bereitgestellten Struktur *vmsupport* (Abbildung 6.3) um ein Array von Funktionszeigern handelt, lässt es sich ähnlich wie die *vtable* von *java.lang.Object* behandeln.

5.4 dynamische Datenstrukturen

Die dynamischen Datenstrukturen bestehen, wie bereits im vorangegangenen Abschnitt angedeutet weitestgehend aus den domainspezifischen Teilen der Bibliotheken, der Klassendeskriptoren sowie den Java-Objekten selbst. Die dazugehörigen Datentypen werden im Folgenden näher erläutert.

- Bei einem *LibDesc* handelt es sich um eine Datenstruktur mit ähnlichen Informationen wie der dazugehörige geteilte Bibliotheksdeskriptor. Er verfügt jedoch über Klassen, statt über Klassendeskriptoren. Ansonsten besitzt er einen Zeiger auf sein statisches Pendant und einen auf die statischen Felder aller seiner Klassen, deren Platz bei seiner Erzeugung allokiert wird.
- Klassen (*Class*) besitzen einen Zeiger auf ihre Basisklassen, einen auf ihr statisches Gegenstück, den *ClassDesc*, sowie einen auf die in ihr enthaltenen

statischen Felder. Ihr Zweck besteht darin statische Felder einer Klasse domainintern zu verwalten, so dass es unmöglich ist, dass zwei Domains über statische Felder Nachrichten miteinander austauschen und so versuchen die gegenseitige Isolation zu umgehen.

- *ObjectDesc*s repräsentieren Java-Objekte. Sie sind von ihrer Grundstruktur einfach gestrickt, da sie lediglich aus einem Zeiger auf ihre vtable sowie einem Array vom Typ *short* bestehen, das eine Größe von mindestens eins haben muss. Allerdings ist diese Einfachheit auch gleichzeitig die Stärke der Objektdeskriptoren, weil durch sie alles ausgedrückt werden kann. Jede Datenstruktur, die eine vtable und darauf folgend Daten mit einer Länge von mindestens zwei Bytes besitzt, kann in einen *ObjectDesc* konvertiert und von Java-Code als Objekt behandelt werden.

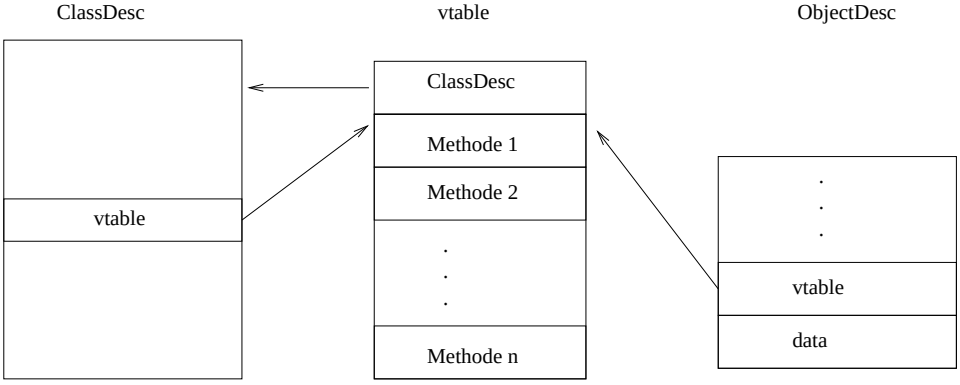


Abbildung 5.5: Klassendeskriptor, vtable und Objektdeskriptor

5.5 Startvorgang

Wird vorrausgesetzt, dass alle statischen Datenstrukturen bereits offline erzeugt wurden und sich somit bereits zusammen mit dem Kern auf dem Baustein befindet, sieht der Startvorgang des Systems aus wie im Folgenden beschrieben. Nach Ausführung der in Abschnitt 4.1 beschriebenen Startfunktion, wird eine Initialisierungsfunktion gestartet, die sämtliche in Kapitel 4 erläuterten Peripheriegeräte initialisiert, eine erste Domain (Domain Zero) sowie einen darin enthaltenen Thread (*InitialThread*) erzeugt und den Interrupt des 16-Bit-Taktgebers (OCIA) aktiviert. Gleichzeitig tritt sie dem Scheduler gegenüber als Thread auf, der sich nach Erfüllung seiner Aufgaben selbst zerstört. Deshalb kommt an dieser Stelle

der Scheduler zum Zug. In Folge seiner Aktivitäten wird das Interrupt-Bit des Prozessors deaktiviert, so dass ab diesem Zeitpunkt alle auftretenden Interrupts dem Prozessor auch signalisiert werden, wenn sie aktiviert sind. Der erste Thread ist dafür zuständig die Domain Zero zu initialisieren. Er führt nach seiner Aktivierung die Funktion *start_domain_zero* aus. Diese hat die Aufgabe, die von der Domain Zero benötigten Bibliotheken auf Domainebene zu initialisieren und die Klassenkonstruktoren der in ihnen enthaltenen Klassen aufzurufen. Dieses Vorgehen ist analog zu dem von JX, abgesehen davon, dass die Bibliotheken nicht mehr entpackt werden müssen, da die dort beim Ladevorgang entstehenden Strukturen bereits vollständig im Code vorliegen. Initialisierung einer Bibliothek auf Ebene der Domains bedeutet, dass alle domain-lokalen Strukturen der Bibliothek sowie der darin enthaltenen Klassen für diese Domain erzeugt und entsprechend verlinkt werden. Zu beachten ist, dass auch hier eine Abhängigkeitsrelation auf den Bibliotheken besteht, die unbedingt eingehalten werden muss, da es sonst durchaus vorkommen kann, dass zu einer Klasse keine Basisklasse gefunden wird.

Nachdem die dynamischen Konstrukte erzeugt sind, kommt im JX-System ein Schritt, in dem die Portale der Domain Zero ebenso wie andere Domain Zero Klassen initialisiert werden. Da keines dieser Konzepte im vorliegenden System bisher realisiert wurde, kann dieser Schritt ausgelassen werden. Jedoch ist davon auszugehen, dass eine Erweiterung wenig Probleme bereiten sollte, wenn es zum Beispiel nötig wird zum Beispiel einen Namensdienst zu installieren.

Im nächsten Schritt werden die Klassenkonstruktoren der eben erzeugten Klassen aufgerufen. Hierzu wird in jedem zu den Klassen korrespondierenden Klassendeskriptor nach einem Methodendeskriptor gesucht, der die Funktion „<*clinit*>“ repräsentiert. Die zu ihm gehörende Funktion wird aufgerufen.

Zu diesem Zeitpunkt ist die Domain Zero vollständig initialisiert und es kann eine erste Java-Domain gestartet werden. Dies wurde jedoch im beschriebenen System nicht getestet und bleibt weiterführenden Arbeiten überlassen.

Kapitel 6

Erzeugen der statischen Laufzeitumgebung

Nachdem in Kapitel 5 das Verhalten des Systems zur Laufzeit dargestellt wurde, soll in diesem Kapitel das Hauptaugenmerk auf den statischen Datenstrukturen, die den Großteil der Java-Laufzeitumgebung dartsellen, und vor allem deren Erzeugung liegen. Bei diesem Vorgehen wird von dem in JX üblichen Vorgehen abgewichen, da dieser Vorgang auf einem externen Rechner vor der Kompilierung des Kerns erzeugt und anschließend als C-Datei gespeichert werden. Die Gründe für diese veränderte Vorgehensweise wurden zum Teil bereits genannt, werden jedoch im nächsten Abschnitt nochmals erläutert. Die weiteren Abschnitte folgen dem Vorgehen beim Erzeugen der Datenstrukturen, das auch in Abbildung 6.1 dargestellt ist: Laden und Linken der statischen Datenstrukturen, sowie deren anschließende Serialisierung. Der letzte Schritt besteht aus dem Ausschreiben der serialisierten Strukturen in Dateien. Das Laden kann hierbei aus einer Bibliothek erfolgen oder auch der Aufbau einer Domain Zero Struktur bedeuten, die in Abschnitt 5.3 beschrieben wurden.

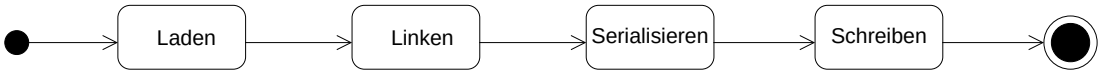


Abbildung 6.1: Schematischer Ablauf beim Erstellen und Verarbeiten der statischen Datenstrukturen

6.1 Motivation

JX ist so entworfen, dass es in der Lage ist dynamisch Code nachzuladen. Wie schon erwähnt wird der Java-Code zunächst in Maschinencode kompiliert und danach in .jll Dateien zu sogenannten Bibliotheken beziehungsweise Komponenten zusammengepackt. Diese werden zusammen mit dem Kern während des Bootvorgangs in den Speicher geschrieben. Jede Domain, die eine bestimmte Bibliothek benötigt, kann nun an den Kern die Aufforderung richten diese Bibliothek zu laden. Ist sie schon geladen, werden alle domainspezifischen Daten erzeugt, wie in Kapitel 5 erwähnt. Andernfalls wird die Bibliothek aus dem Speicher gelesen und entpackt. Danach werden die benötigten statischen Datenstrukturen erstellt und die Platzhalter der Symboladressen durch die richtigen Werte ersetzt (*patching*). Anschließend wird die vollständige geladene Bibliothek in eine Liste eingehängt. Zwar liegen bisher die Bibliotheken beim Start des Systems im Speicher, jedoch ist es durchaus denkbar, sie über das Netzwerk oder von Platte nachzuladen. Da sich die Größe einer Bibliothek in einer Größenordnung von einigen Kilobyte bewegt, wird es in einem auf dem RCX laufenden System nur eine geringe Anzahl von Bibliotheken geben können, so dass der vorhandene Platz vollständig ausgereizt ist und kein Bedarf beziehungsweise keine Möglichkeit mehr besteht andere Bibliotheken nachzuladen.

Lädt man den Kern zusammen mit den Bibliotheken auf den Baustein, werden diese nach dem Entpacken mehr als die doppelte Menge an Speicherplatz belegen, als die Bibliothek allein. Da man jedoch auf die entpackten Strukturen keinesfalls verzichten kann, schon aus dem Grund nicht, weil sie ausführbaren Code enthalten können, ist die einzige Möglichkeit wie diese Duplizität umgangen werden kann, die Existenz der gepackten Bibliotheken auf dem Baustein zu vermeiden. Ein dritter Grund, der dafür spricht das Entpacken vor das Laden auf den Baustein zu setzen ergibt sich dadurch, dass durch dieses Vorgehen die Menge an Code reduziert werden kann, die überhaupt auf den Baustein gelangen muss, da die ganze Funktionalität des Entpackens und des Aufbaus der Datenstrukturen nur als Vorbereitung zum Laden benötigt wird. Bei genauerer Betrachtung lässt sich weiterhin feststellen, dass sämtliche von der Domain Zero beim Starten des Systems erzeugten Datenstrukturen, bereits mit den statischen Strukturen erzeugt werden können, so dass sich der hierfür benötigte Code ebenfalls auf ein Minimum reduziert, wie bereits in Abschnitt 5.5 angedeutet.

Das für die Aufbereitung der Bibliotheken entwickelte Programm besitzt die Funktionalität eines *Loaders* und die eines *Linkers*. Als Grundlage für den benötigen funktionellen Code dienen zu einem Großteil Funktionen aus JX.

Probleme, die durch dieses Vorgehen entstehen, sind vor allem dadurch bedingt, dass Zeiger während des Erzeugungsvorgangs auf einem Intel-Rechner vier Bytes lang sind und in Zukunft bei einem 64-Bit Adressraum acht Bytes lang sein werden, während sie auf dem H8/300 durch zwei Bytes dargestellt werden. Diese Tatsache macht es erforderlich alle Werte, die während des Aufbaus der Datenstrukturen als Zeiger benutzt werden, vor dem Schreiben in Dateien mit C-Code durch zwei Byte breite Werte zu ersetzen. Weiterhin bietet es sich an, alle Zahlenwerte, die Speichergrößen darstellen ebenfalls durch 16 Bit große Werte zu ersetzen, da auf dem RCX keine Struktur eine Größe besitzen kann, deren Wert den durch zwei Bytes gegebenen Wertebereich überschreitet. Davon ausgehend ist es durchaus praktikabel, Zahlenwerte mit einer Größe von mehr als zwei Bytes generell zu untersagen, so dass alle primitiven Java-Zahlen auf den Zahlentyp *short* abgebildet werden. Im Gegensatz zur Umwandlung von Zeigern, was dynamisch erfolgen muss, kann die Abbildung der Zahlentypen bereits im Code des Offline-Programms erfolgen.

6.2 Erzeugen von *java.lang.Object*

Um *java.lang.Object* zu erzeugen, werden, wie schon erwähnt, die Funktionen von JX verwendet. Diese sind bis auf oben genannte Anpassungen unverändert übernommen. Um jedoch einen Einblick zu bekommen, was beim Erzeugen eines Klassendeskriptor geschieht wird im Folgenden das Vorgehen beim Erzeugen des *ClassDesc*s von *java.lang.Object* genauer erläutert.

Ausgegangen wird von den aus JX übernommenen Datenstrukturen, die aus einem Array vom Typ *MethodInfoDesc*, sowie einem fast leeren Methodendeskriptor-Array bestehen. Ein *MethodInfoDesc*-Array findet generell bei allen von der Domain Zero implementierten Klassen Verwendung. In ihm sind alle von dieser Klasse bereitgestellten Methoden repräsentiert, wobei eine Struktur, eine Methode repräsentiert und aus deren Namen, deren Signatur und einem Zeiger auf deren Code besteht. Da für das Bereitstellen der Datenstrukturen keine der Methoden ausgeführt werden muss, kann man sich darauf beschränken leere Funktionsrümpfe in den Code des Offline-Programms aufzunehmen. Wie deren Adressen später so umgewandelt werden, dass im erzeugten Quellcode die benötigten Funktionen referenziert werden, wird zusammen mit den anderen Verfahren zum Ersetzen von Zeigern in den Abschnitten 6.5 und 6.6 beschrieben.

Für das Erzeugen des Klassendeskriptors ist ausschließlich die Funktion *createObjectClassDesc()* zuständig, die sich jedoch anderer Funktionen bedient. In

ihrem Ablauf wird zunächst der notwendige Speicher in Größe eines Klassendeskriptors allokiert. Anschließend werden alle Felder belegt, sowie Speicherplatz für eine neue *vtable* allokiert und diese anschließend ausgefüllt. Eine Besonderheit stellt *java.lang.Object* insofern dar, als dass sich die gleiche Anzahl an Einträgen sowohl in *methods*, als auch in *methodVtable* und damit auch in der *vtable* findet. Außerdem handelt es sich um die einzige Klasse, die keine Basisklasse hat.

Nachdem *methodVtable, methods* und *vtableSym* angelegt wurden, wird im folgenden Schritt die *vtable* erzeugt und mit dem Klassendeskriptor verzeigert, mit einem Ergebnis wie in Abbildung 5.5 zu sehen. Daraufhin werden die Adressen der Funktionen eingetragen.

Für andere Domain Zero Klassen entspricht die Vorgehensweise in etwa der hier vorgestellten, auch wenn zusätzliche Aufgaben, wie das Installieren der *java.lang.Object*-Methoden in den oben beschriebenen Strukturen oder das Erzeugen der Methodendeskriptoren nötig sind, da bei allen anderen Klassen ausschließlich *MethodInfoDesc*s verwendet werden um die Methoden zu beschreiben. Alles weitere entsteht während der Initialisierung dynamisch. Da bisher jedoch keine Domain Zero Klassen zum Einsatz kommen, soll hierauf im Weiteren nicht näher eingegangen werden.

Dennoch handelt es sich bei *java.lang.Object* nicht um die einzige in C realisierte Klasse, die im System vorhanden ist. Neben ihr existieren noch die sogenannten primitiven Klassen, die die primitiven Java-Datentypen repräsentieren. Ihre Erzeugung besteht jedoch lediglich aus der Allokation von ausreichend Speicherplatz und der Vergabe eines Namens, da an primitiven Datentypen keine Operationen erlaubt sind. In beiden Fällen, sowohl bei *java.lang.Object* als auch bei den primitiven Klassen, ist es ebenfalls möglich die zu den zugehörigen *ClassDesc*s gehörenden *Class*-Konstrukte zu erzeugen. Das ist problemlos machbar, da in beiden Fällen keine Klassenvariablen vorhanden sind, die einer domain-internen Behandlung bedürften. Auf Grund dieser Tatsache können die *Class*-Stukturen ebenfalls offline erzeugt und mit auf den RCX geladen werden. Das Vorgehen beim Erzeugen orientiert sich wie im vorgestellte Fall an JX, die weitere Verarbeitung geschieht hierbei analog zur weiteren Vorgehensweise bei den statischen Elementen.

6.3 Laden der Bibliotheken

Nachdem aufgezeigt wurde, wie aus den in der Domain Zero abgelegten in C implementierten Methoden, die dazugehörigen Datenstrukturen erzeugt werden,

soll nun der Blick auf das Laden der Java-Klassen und den anschließende Aufbau der Deskriptoren gelenkt werden. Wie schon mehrmals erwähnt werden Klassen nicht elementar geladen. Stattdessen sind sie in Bibliotheken, die elementare Einheit beim Laden von Klassen, zusammengefasst. Bibliotheken werden durch den Java-Zu-Maschinencode-Übersetzer erzeugt und in Dateien zusammengefasst. In ihnen sind sowohl der Code der verschiedenen Klassen als auch eine Reihe von Metainformationen in gepackter Form enthalten. Auf den Inhalt wird im Folgenden noch eingegangen, ein grober Überblick über den Aufbau lässt sich jedoch durch Abbildung 6.2 gewinnen, wobei *Methode X.Y* für die Y-te Methode der X-ten Klasse steht. Im Gegensatz zu JX, liegen im vorliegenden Fall die Bibliotheken in Form von Dateien vor. Dies wird in der Betriebssystemversion von JX nicht unterstützt. Allerdings kann JX auch als konventionelles Programm in einer Linux-Umgebung ablaufen, in der es möglich ist Bibliotheken aus dem Dateisystem in den Speicher zu laden. Diese Funktionalität kann ohne Änderung übernommen werden, wenngleich auch der Kontext, aus dem die Funktionen aufgerufen werden, ein anderer ist: Im JX-System wird eine Bibliothek unabhängig von anderen Bibliotheken geladen, das heißt, wann eine Bibliothek geladen wird entscheidet das System. In der Regel wird dies der Fall sein, wenn eine Domain gestartet wird, die auf noch nicht geladenen Bibliotheken angewiesen ist. Dagegen werden im vorliegenden Fall alle Bibliotheken zur selben Zeit geladen. Ob sie gebraucht werden oder nicht, entscheidet der Benutzer.

```
Versionsinformation
benötigte Bibliotheken
Zeichenketten
Metadaten
VM-Informationen
Informationen Klasse 1
Informationen Methode 1.1
Informationen Methode 1.2
:
Informationen Klasse 2
:
Code
```

Abbildung 6.2: Schematischer Aufbau einer Bibliotheksdatei

Die Angabe welche Dateien geladen werden sollen, erhält das Programm als Parameter übergeben. Anschließend werden sie in der angegebenen Reihenfolge geladen. Hierzu wird zunächst die entsprechende Datei gesucht. Wird sie nicht

gefunden, werden einige Unterverzeichnisse durchsucht. Ist sie auch hier nicht zu finden, wird aufgegeben und zur nächsten Datei übergegangen. Ist es dagegen möglich sie zu öffnen, wird zunächst die Größe des von ihr benötigten Speichers aus ihr ausgelesen und diese Menge an Speicher allokiert. Danach wird die Datei am Stück in diesen Speicherbereich eingelesen. Der Begriff „auslesen“ muss im Folgenden unterschiedlich interpretiert werden, abhängig davon, ob es sich beim erwarteten Wert um eine Ganzzahl, also arithmetische Datentypen, oder um Zeichenketten handelt. Im ersten Fall wird eine Unterfunktion aufgerufen, die die folgenden n Bytes aus dem Speicherbereich ausliest und zu einer Integer-Zahl zusammenfügt, wobei n die Größe des entsprechenden Datentyps in Bytes angibt. Diese Zahl wird von der Funktion an den Aufrufer zurückgegeben. Im Fall von Zeichenketten gibt es mehrere Möglichkeiten. Zum Einen kann die Sequenz der nächsten n Bytes in einen Puffer kopiert werden. Hierzu ist es jedoch nötig zu wissen, wie groß n ist. Aus diesem Grund wird in vielen Fällen vor der eigentlichen Zeichenkette deren Länge gespeichert. So lässt sich durch ein Makro zuerst diese Länge auslesen und anschließend ein Puffer der richtigen Größe allokieren. In einem dritten Schritt kann dann die Zeichenkette in den Puffer kopiert werden. Letzen Endes existiert noch eine dritte Möglichkeit, an eine Zeichenkette zu kommen: Alle in einer Bibliothek vorkommenden Zeichenfolgen werden gesammelt und gemeinsam abgelegt. So verhindert man, dass unnötig Platz verschwendet wird, wenn etwa viele Klassen eine bestimmtes *Interface* implementieren, da der ganze Name nur einmal gespeichert wird. Da diese Sammlung von Zeichenketten auch am Stück in ein Array ausgelesen wird, ist es möglich den Index des Strings in diesem Array auf die Adresse des String-Arrays zu addieren um die Adresse der gewünschten Zeichenkette zu bekommen.

Der nächste Schritt im Ladevorgang ist der Test der Checksumme. Hierzu wird die Checksumme der Datei gebildet und mit der in ihr gespeicherten Summe verglichen. Anschließend wird geprüft, ob die Version der Datei mit der des Ladeprogramms übereinstimmt. Abschließend wird der Zielprozessor ausgelesen. Sind alle diese Tests erfolgreich verlaufen, kann damit begonnen werden zu der in der Datei befindlichen Bibliothek einen *SharedLibDesc* zu erstellen. Hierzu wird zunächst Speicherplatz für diesen und den Namen der Bibliothek reserviert und anschließend die Anzahl und die Namen der von dieser Bibliothek benötigten Bibliotheken ausgelesen und im Deskriptor vermerkt. Sind noch nicht alle benötigten Bibliotheken geladen, so wird zunächst der Ladevorgang für diese gestartet, bevor der eigentliche Ladevorgang fortgesetzt wird. In JX wird eine Bibliothek gleich nach dem Ladevorgang gelinkt, was auch das Ersetzten von Symbolen im übersetzten Java-Code beinhaltet. Dieser Schritt muss im vorliegenden Fall nicht

unmittelbar nach dem Laden der einzelnen Bibliotheken folgen, sondern kann, wie das Laden selbst, für alle Bibliotheken am Stück erfolgen. Dies erleichtert den Aufbau einiger Datenstrukturen, wie in Kapitel 6.4 ausgeführt wird.

Nun können alle noch benötigten Strukturen aus der im Speicher liegenden Datei ausgelesen werden. Begonnen wird hierbei mit allen von dieser Bibliothek verwendeten Zeichenketten, gefolgt von den zur Bibliothek gehörenden Metadaten. Daraufhin werden, falls vorhanden, die Symbolinformationen für die VM ausgelesen. Dies ist nur bei einer Bibliothek notwendig. Dazu muss beschrieben werden, wie die vom Kern bereitgestellten Funktionen der VM erreicht werden können: Informationen über sie befinden sich in einem Array *vmsupport* über dessen Definition Abbildung 6.3 Auskunft gibt. Der erste Eintrag beschreibt den Namen der Funktion, der letzte enthält einen Zeiger auf die Funktion selbst. Der Wert des mittleren Eintrags ist als Index zu verstehen. Er ist nötig, weil die Reihenfolge der Funktionen, die durch den Compiler festgelegt ist, nicht der entsprechen muss, die vom Programmierer des C Kerns gewählt wurde. Enthält deshalb die geladene Bibliothek einen Eintrag für eine VM-Funktion, die durch den Compiler an die i-te Position gesetzt wird, so wird deren Name ausgelesen und ihre Spalte j im oben genannten Array gesucht. Anschließend wird der Index der i-ten Spalte auf j gesetzt.

```
struct vm_fkt_table_t { char *name,
                        int index,
                        code_t fkt};
struct vm_fkt_table_t vmsupport[] {
  {''vm_unsupported'' , 0, &vm_unsupported},
  ...;
};
```

Abbildung 6.3: Typdefinition zur VM-Unterstützung

Das Ausfüllen der Funktionstabelle ist die vorerst letzte Aktion, die den Bibliotheksdeskriptor direkt betrifft, da darauf folgend die Attribute der zur Bibliothek gehörenden Klassen ausgelesen werden. Da sich in der Regel mehr als eine Klasse in einer Bibliothek befindet, geschieht dies iterativ, über alle vorhandenen Klassen. Gleichzeitig wird während dieser Rekursion die Größe der statischen Felder aller Klassen der Bibliothek aufsummiert und an ihrem Ende in den Bibliotheksdeskriptor eingetragen.

Da die meisten Werte in den Klassendeskriptoren genauso aus der Bibliotheksdatei ausgelesen werden, wie beim Aufbau der Bibliotheksdeskriptoren beschrieben,

soll im Folgenden nur noch auf die Besonderheiten eingegangen werden. Diese sind da zu finden, wo Querbezüge zu anderen Elementen hergestellt werden müssen, also bei der Basisklasse, bei implementierten *Interface*s und bei der Beschreibung der *vtable*-Elemente. Da ein Klassendeskriptor den Platz für einen Zeiger auf eine Basisklasse bereithält, dieser jedoch nicht aus der Bibliotheksdatei ausgelesen werden kann, weil seine Adresse nicht a-priori bekannt ist, ist dort zusätzlich der Name der Basisklasse gespeichert, der zunächst ausgelesen wird. Anschließend wird in den bereits geladenen Bibliotheken und deren Klassen nach dieser Basisklasse gesucht, wobei diese gefunden werden muss, da ja bereits alle Klassen, von denen die aktuelle Klasse abhängt geladen sind. Wird sie nicht gefunden, kann davon ausgegangen werden, dass nicht alle benötigten Bibliotheken zur Verfügung stehen. Aus diesem Grund wird der Ladevorgang und damit das Programm abgebrochen. Ähnlich wird auch bei den Informationen über die von der Klasse implementieren *Interface*s vorgegangen, wobei zunächst deren Anzahl ausgelesen wird. Anschließend werden zwei Zeiger-Arrays dieser Größe allokiert. Die Adresse des einen wird dem Eintrag *interfaces*, die des anderen *ifname* zugewiesen. Im nächsten Schritt werden die Namen der *Interface*s ausgelesen und die Werte in *ifname* entsprechend gesetzt. Das zum jeweiligen Namen gehörende *Interface* wird während der Linkerphase (siehe Abschnitt 6.4) gesucht und verlinkt. Bei den *vtable*-Symbolen verhält es sich ähnlich, wie bei den *Interface*s. Auch hier wird zunächst die Größe der *vtable* für das Feld *vtablesize* ausgelesen und anschließend Speicherplatz für die nötige Menge an Einträgen allokiert. Da jedoch wie bereits beschrieben, für jedem Eintrag der *vtable* drei Einträge im dazugehörigen Symboldeskriptor entsprechen, wird ein Zeiger-Array der dreifachen *vtablesize* allokiert. Darauf folgend werden für jeden vtable-Eintrag die entsprechenden Werte, Klasse, Name und Typ, ausgelesen und in den Symboldeskriptoren vermerkt. Enthält bereits der Eintrag der Klasse den Wert *'\0'*, bekommen alle anderen Einträge ebenfalls diesen Wert. In diesem Fall handelt es sich um eine abstrakte Methode.

Da während des Auslesevorgangs alle Strukturen rekursiv durchlaufen werden, besteht der nächste Schritt darin, für eine Klasse alle Methodendeskriptoren, der von ihr implementierten Methoden zu erstellen. Hierzu wird wie in den vorhergehenden Fällen, zunächst die Anzahl der Klassen im Feldelement *numberOfMethods* gespeichert und anschließend eine genügend große Menge an Speicher für diese Anzahl an Methodendeskriptoren allokiert. Der Zeiger auf dieses *MethodDesc*-Array wird im Feldelement *methods* des Klassendeskriptors hinterlegt. Da das Auslesen der Werte der Methodendeskriptoren weitestgehend analog zu Klassen- und Bibliotheksstrukturen verläuft, wird darauf im Weiteren nicht näher eingegangen. Einzig das Auslesen der Symboldeskriptoren, die zur Übersetzungszeit

nicht auflösbare Symbole des Java-Codes beschreiben, soll näher erläutert werden, da sie und ihre weitere Behandlung von essentieller Bedeutung für die Java-Code Anbindung an den Kern sind. Nicht näher beschrieben wird dagegen das Erzeugen von Byte-Code- und Exception-Table, da sie im bisherigen System eine untergeordnete Rolle spielen. Generell kann jedoch davon ausgegangen weden, dass ihre Erzeugung nach dem bereits beschriebenen Schema, auslesen der Größe, allokieren des Speichers, auslesen der Werte, erfolgt.

Da, wie in Abschnitt 5.2 erläutert, die Größe verschiedener Symboldeskriptoren nicht einheitlich ist, lässt sich hier das obige Schema Größe auslesen, Speicher allokieren, Werte auslesen nur dann anwenden, wenn es um einen Indirektionsstufe erweitert wird. Nachdem die Anzahl der benötigten Symboldeskriptoren bekannt ist, wird in diesem Fall kein Array von Symboldeskriptoren angelegt, sondern ein Array von Zeigern auf Symboldeskriptoren. Ersteres ist nicht möglich, weil es für das Anlegen eines Array erforderlich ist, dass alle seine Elemente die gleiche Größe besitzen. Trotz der Vielzahl an Symboldeskriptoren besitzen alle eine gemeinsame Grundmenge an Feldern, die zunächst einmal ausgelesen werden kann. Anhand dieser Werte ist es nun möglich den Typ des Deskriptors zu bestimmen und die entsprechende Menge an Speicher zu errechnen und zu allokieren. Anschließend werden alle weiteren Felder ausgelesen, und zusammen mit den vorher ausgelesenen und zwischengespeicherten in den neu angelegten Speicherblock geschrieben. Zuletzt wird dieser in das obige Zeiger-Array eingetragen.

Sind alle Methoden eine Klasse und alle Klassen einer Bibliothek erzeugt und besitzen alle ihre Felder den richtigen Wert, werden in einem vorletzten Schritt die in den Iterationen akkumulierten Werte in den Bibliotheksdeskriptor geschrieben und der komplette Code der Bibliothek auf einmal ausgelesen. Seine Adresse wird dem *Code*-Parameter des Deskriptors zugewiesen. Letztlich kann die neu geladene Bibliothek in die Kette der bereits vorhandenen Bibliotheken eingehängt werden.

6.4 Linken

Während des Ladevorgangs ist ein Gerüst aus den verschiedensten Deskriptoren entstanden, die alle auf die eine oder andere Weise verbunden sind. Die Speicherstrukturen sind also weitestgehend, wenn auch noch nicht ganz vollständig, erzeugt. Es fehlen noch einige Schritte, um das angestrebte Ziel zu erreichen. Zum Einen sind die oben erwähnten nicht-auflösbaren Symbole im Java-Code immer noch nicht aufgelöst. Außerdem besitzt keine der geladenen Java-Klassen eine

vtable, noch verfügt irgendeiner der Methodendeskriptoren über einen Zeiger auf den ihm zugeordeten Code.

Während für die letzten beiden Punkte im Folgenden Lösungen präsentiert werden, wird der erste Punkt erst in Abschnitt 6.5 behandelt. Hier gibt es im chronologischen Ablauf leichte Abweichungen zu JX, da hier alle drei offenen Punkte beinahe zur gleichen Zeit, nämlich während des Linkens behandelt werden. Warum es praktikabler ist mit dem Ersetzen mancher Symbole zu warten, soll ebenfalls später erläutert werden.

Nachdem nun alle Bibliotheken geladen sind, müssen sie verlinkt werden. Hierbei kann ebenfalls über alle Bibliotheken iteriert werden. Was während dieser Phase geschieht soll nun näher dargestellt werden. Der erste Schritt besteht darin, in allen Methodendeskriptoren die Startadresse des Codesegments zu setzen, das durch sie repräsentiert wird. Um diese zu bestimmen besitzt jeder Methodendeskriptor einen Wert, der den Abstand der Startadresse seines Codes zur Adresse des Codeblocks der Bibliothek angibt. Läuft man nun von einer Bibliothek ausgehend iterativ über alle deren Klassen- und weiter über alle Methodendeskriptoren dieses Klassendeskriptors, kann man für jede Methode den Startwert ihres Codes errechnen, indem man zur *code*-Wert der Bibliothek den eben beschriebenen Abstand addiert. Im gleichen Durchlauf kann man testen, ob es sich bei der Methode um den Standardkonstruktor einer Klasse handelt, das heißt, ob ihr Name mit „<init>“ übereinstimmt und ihre Signatur „()V“ lautet. Ist dies der Fall setzt man den Wert *default_init* des zugehörigen Klassendeskriptors auf diese Methode.

Im Gegensatz zu den Vorgängen beim Erzeugen von *java.lang.Object*, müssen bei allen anderen Klassen neben den *vtable*s der Klassendeskriptoren zusätzlich noch deren *methodVtable*s erzeugt werden. Vorerst wird für diese beiden Arrays lediglich Platz reserviert, ihre Felder jedoch noch nicht ausgefüllt. Dies geschieht zu einem späteren Zeitpunkt. Allerdings ist es jetzt möglich die *Interface*-Struktur jedes Klassendeskriptors auszufüllen, nachdem sie bereits beim Laden der Klasse erzeugt wurde. Zum Wert der i-ten Spalte der Struktur, wird der Klassendeskriptor gesucht, der die Klasse mit dem Namen der i-ten Spalte des *ifname*-Arrays des gleichen Klassendeskriptors repräsentiert. Wird zu einem Namen keine Klasse gefunden, kann wiederum davon ausgegangen werden, dass die geladenen Bibliotheken nicht vollständig sind. Aus diesem Grund wird das Programm in diesem Fall abgebrochen.

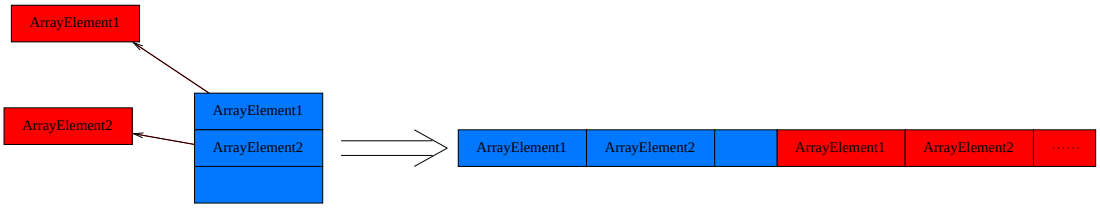
In einem finalen Schritt werden daraufhin *vtable*, sowie *methodVtable* mit Werten belegt. Hierzu sucht man zu jedem Eintrag, der während des Ladevorgangs

ausgefüllten *vtable*-Symbole (*vtableSym*), der nicht leer ist, den entsprechenden Methodendeskriptor, indem man wiederum zunächst über alle Bibliotheken und daraufhin über alle deren Klassendeskriptoren iteriert. Stößt man dabei auf einen Klassendeskriptor mit dem Namen, der in der *vtable*-Symboltabelle vermerkt ist, iteriert man über alle dessen Methodendeskriptoren. Trägt eine von ihnen den Namen und die Signatur, die in der Symboltabelle vermerkt ist, wird die Startadresse ihres Codesegments in die *vtable*, ihre Adresse in die *methodVtable* eingetragen. Auch hier wird das Programm beendet, wenn keine entsprechende Methode gefunden wird.

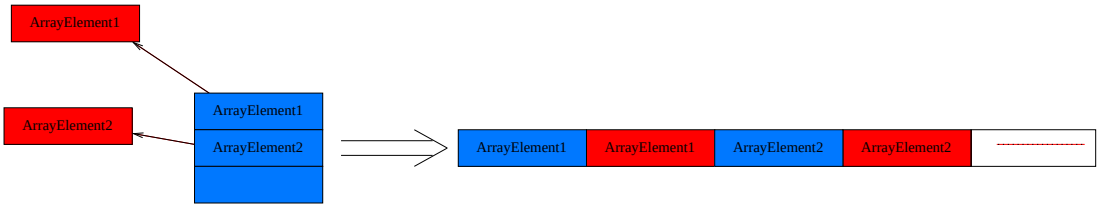
6.5 Serialisieren der Strukturen

Nachdem nun alle Datenstrukturen aufgebaut und entsprechend verlinkt wurden, könnte man mit ihnen arbeiten, indem man zum Beispiel Java-Code startet und die Funktionen verschiedener Klassen aufruft, wenn der Rechner, auf dem die Strukturen aufgebaut sind, mit dem identisch ist, der sie verarbeiten soll. Da dies hier nicht der Fall ist stellt sich die Frage, wie man das aufgebaute Netz der verschiedenen Deskriptoren geschlossen und persistent auf die Zielplattform bekommen kann. Eine Lösung, die sich anbietet, ist alle Daten ausgehend von den Bibliotheken in einen Datenstrom zu serialisieren und diesen in eine Datei zu schreiben, die dann wiederum mit allen anderen Quelldateien durch den Übersetzer geschickt werden kann. Probleme, die hierbei entstehen, ergeben sich vor allem aus der Existenz von Zeigern und dem Vorhandensein der in Abschnitt 6.2 angesprochenen leeren Funktionsrümpfe, die anstatt der im Kern vorhandene Methoden benutzt werden. Beide Probleme ähneln sich insofern, als dass jeweils eine *offline* vorhandene Adresse so abgelegt werden muss, dass sie später *online*, im Betrieb, wieder gefunden werden kann. Um dies zu ermöglichen ist es erforderlich, dass beim Aufbauen der Datensequenz weitere Datenstrukturen aufgebaut werden, die wiederum beim Schreiben in die verschiedenen C-Dateien abgearbeitet werden.

Da es, wie schon mehrmals erwähnt, möglich ist von der Kette der Bibliotheken ausgehend alle weiteren Daten zu erreichen, bietet es sich an mit dem Serialisieren der Bibliotheken zu beginnen. Generell ist es sinnvoll zunächst die in einer Datenstruktur vorhandenen Werte zu serialisieren, bevor man den in ihr enthaltenen Zeigern folgt. Weiterhin ist es notwendig im Falle eines Arrays zuerst alle Werte der im Array enthaltenen Datenstrukturen zu serialisieren. Erst danach kann den Zeigern gefolgt werden, die in diesen Datenstrukturen vorhanden sind.



(a) richtig



(b) falsch

Abbildung 6.4: Serialisiertes Array

Dieses Vorgehen wird in Abbildung 6.4 veranschaulicht. Würde man zuerst ein Datum serialisieren und gleich danach den Zeigern folgen, wie in Abbildung 6.4 (b), würde die Arraystruktur zerstört werden, so dass später falsche Werte das Ergebnis wären. Aus diesem Grund existiert für jede Struktur X eine Funktion *buildX(X *x)*, die sie serialisiert, sowie eine Funktion *buildXArray(int i, X *x)*, die zunächst i mal *buildX(&x[i])* aufruft und danach die im Array enthaltenen weiteren Datentypen serialisiert. Betrachtet man den Vorgang des Serialisierens auf kleinster Granularität, stellt man fest, dass vier verschiedene Arten von elementaren Datentypen zu serialisieren sind. Zum Einen, die Ganzzahltypen *long, short, char* und zum Anderen Zeiger, die auch auf Zeichenketten zeigen. Da der Typ *int* auf einem Intel-basierten System 32 Bit breit ist, auf dem RCX jedoch lediglich 16 Bit, vereinfacht es die Algorithmen, wenn die eindeutige Darstellung *long* für 32 Bit Datentypen und *short* für zwei Byte große Zahlen verwendet wird. Im vorliegenden Fall wurde das Vorkommen von 32 Bit Daten von vornherein unterbunden, da die meisten Zahlen dazu verwendet werden Speicherbereichsgrößen anzugeben, die beim RCX auf Grund des begrenzten Speichers niemals Größen annehmen können, die mehr als 16 Bit benötigen. Um die vorhin genannten niedrigsten Datentypen in den Datenstrom einzufügen, werden die in Abbildung 6.5 aufgezeigten Funktionen verwendet.

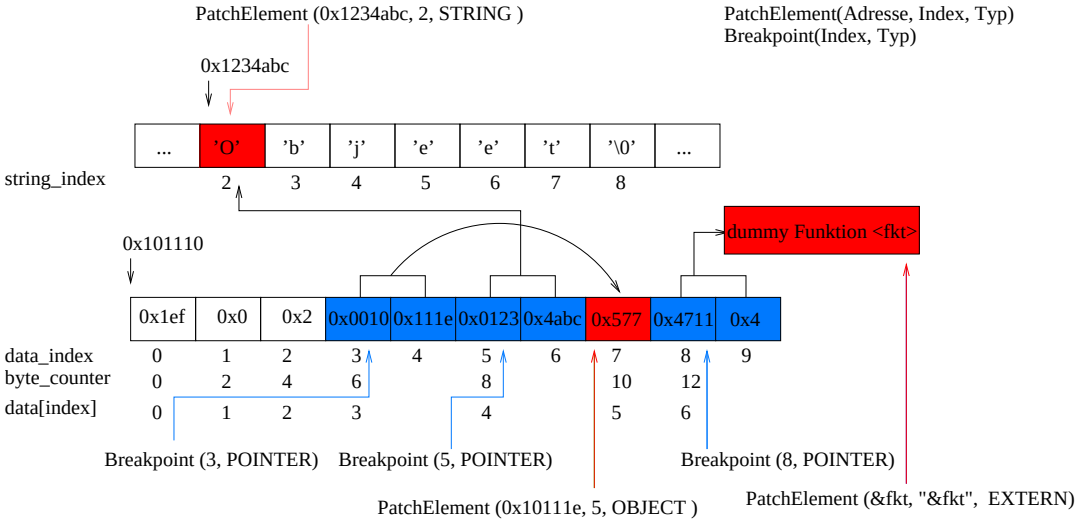
```
void insert_byte(char byte);
void insert_word(short word);
void insert_long(long l);
void insert_pointer(void *p);
long insert_string(char *s);
```

Abbildung 6.5: Schnittstelle zum Serialisieren der elementaren Daten

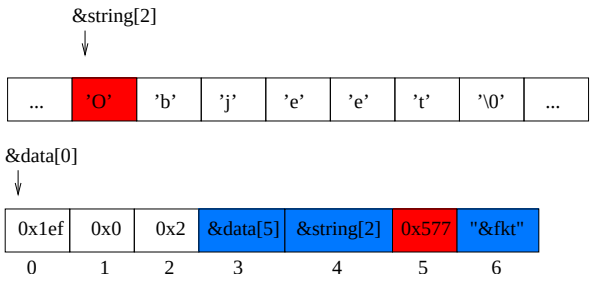
Damit Daten jeglicher Art überhaupt serialisiert werden können ist eine Datenstruktur nötig, die Daten linear aufnehmen kann. Ideal hierzu eignen sich Arrays, die sich dadurch auszeichnen, dass alle ihre Einträge hintereinander im Speicher liegen. Im vorliegenden Fall werden zwei Arrays benutzt, deren Größe sich bei Bedarf erweitern lässt. Eines der Arrays dient zum Zwischenspeichern der vorkommenden Zeichenketten und ist vom Typ *char[]*, das andere ist für alle anderen Daten und vom Typ *short[]*. Diese Teilung hat den Grund, dass, ähnlich wie in den JX Bibliotheken, vermieden werden soll, dass dieselbe Zeichenfolge mehrmals gespeichert und somit Platz zu verschwendet wird. Legt man alle bisher zwischengespeicherten Zeichenketten in einem eigenen Array ab, wird das Überprüfen, ob eine neu einzufügende Zeichenfolge bereits enthalten ist und demzufolge nicht mehr abgespeichert werden muss, erleichtert. Der Typ *short* des Daten-Arrays begründet sich durch den Umstand, dass es auch RCX-Zeiger enthalten können muss, die zwei Byte breit sind. Um diese Werte in ein *char*-Array zu schreiben muss ein größerer Aufwand betrieben werden, da zunächst der zwei Byte breite Zeiger in zwei ein Byte große Werte zerlegt werden muss. Ein weiterer Vorteil eines *short*-Arrays wird ersichtlich, wenn man bedenkt, dass auf dem RCX jeder Maschinenbefehl an einer Adresse beginnen muss, deren Wert ein Vielfaches von 16 ist (*alignment*). Da auch kompilierter Java-Code im Daten-Array enthalten ist, befindet er sich damit automatisch auf einer solchen Adresse.

Neben den Arrays, Container der eigentlichen Daten, sind auch eine Anzahl von Zählern für das Serialisieren notwendig: so besitzt jedes Array einen Zähler, der die Länge des Arrays und einen, der die aktuelle Einfügeposition angibt. Weiterhin gibt es einen *byte_counter*, der die Bytes zählt, die später in der Datenstruktur enthalten sein werden. Die Differenz aus der Schreibposition für das Datenarray und die Hälfte des *byte_counter*s gibt die Anzahl der Zeiger an, die in der Struktur vorhanden sind und später beim Ausschreiben in eine Datei ersetzt werden müssen. Dies wird ersichtlich, wenn man bedenkt, dass ein eingefügter Pointer den Zähler des Datenarrays um zwei erhöht, genau wie den des *byte_counter*s. Jedoch zählt Erstgenannter wortweise, während die Einheit des Zweitgenannten

Bytes darstellt. Bringt man beide auf die gleiche Einheit, nämlich Worte, bleibt eine Differenz von einem Wort, was genau dem Unterschied der Zeigerlängen auf dem RCX und einem x86-Rechner entspricht. Das Inkrementieren der jeweiligen Zähler bleibt den in Abbildung 6.5 gezeigten Funktionen vorbehalten.



(a) Speicher



(b) Datei

Abbildung 6.6: Datenstrom in Speicher und Datei

Wie sich nun das oben beschriebene Ziel, Adressen von Datenstrukturen in einer ganz anderen Umgebung wieder auffinden zu können, erreichen lässt, soll in den folgenden Zeilen dargestellt werden. Beim Serialisieren des aufgebauten Strukturnetzes tritt das Problem in drei verschiedenen Ausprägungen auf, von denen jetzt eine betrachtet wird. Später folgen die anderen beiden. Im ersten Fall soll ein Zeiger wiedergefunden werden, der zum Beispiel von einem Klassendeskriptor auf seine Bibliothek zeigt. Dazu ist es zunächst nötig zu wissen an welcher Stelle im

Daten-Array ein Zeiger steht. Weiterhin lässt er sich nur dann ersetzen, wenn man weiß, wohin er zeigt. Als Ziel eines Zeigers kommt prinzipiell jede der vorgestellten Strukturen in Frage, ebenso Zeichenketten. Eine dritte Möglichkeit stellen Platzhalterfunktionen dar. Um diese beiden Zielsetzungen, zu wissen, wo ein Zeiger im Daten-Array steht und wohin und ebenso auf was er zeigt, zu erfüllen, werden zwei Tabellen aufgebaut, wobei in der ersten alle *BreakPoints* und in der zweiten alle *PatchElements* gespeichert sind. Die Definitionen hierzu sind aus Abbildung 6.7 ersichtlich. Abbildung 6.6 (a) zeigt eine Beispielkonstellation mit den daraus entstandenen *BreakPoints* und *PatchElements*. Ein BreakPoint beinhaltet den Index, an dem ein Zeiger im serialisierten Datenstrom steht. Darüber hinaus beinhaltet er den Typ des Zeigers, der einen der beiden Werte *POINTER* oder *PATCH* annehmen kann, wobei in den hier betrachteten Fällen ausschließlich *POINTER* von Bedeutung ist. *PATCH* ist für das Ersetzen von Symbolen im Java-Code von Nöten und wird später erläutert. BreakPoints werden ausschließlich von der Funktion *insert_pointer* in die Tabelle eingefügt. Dadurch wird sichergestellt, dass deren Indizes streng monoton steigend sind. Diese Eigenschaft wird beim Schreiben des Datenstroms benötigt, auf das in Abschnitt 6.6 näher eingegangen wird. Nachdem nun die Position aller Zeiger im Datenstrom bekannt ist, fehlt noch die Information über das Ziel des Zeigers. Da es jedoch beim Serialisieren zu umständlich ist festzustellen, welche Adresse Ziel eines Zeigers ist, werden alle potentiellen Adressen, die Ziel eines Zeigers sein können, in die zweite Tabelle aufgenommen. Die Einträge der Tabelle sind vom Typ *PatchElement*. Sie beinhalten die Adresse des eingetragenen Objekts, sowie deren zukünftigen, über *byte_counter* ermittelbaren, Index im Datenstrom. Als drittes Attribut besitzt ein PatchElement einen Typ, der *NULLP, STRING, OBJECT, EXTERN* oder *CODE* sein kann. Die Bedeutung, des ersten Wertes wird weiter unten erklärt. *STRING* zeigt an, dass es sich bei diesem Element um eine Zeichenkette handelt, *OBJECT*, dass das Element eine Datenstruktur repräsentiert, die im Daten-Array zu finden ist. *EXTERN* schließlich gibt an, dass eine Platzhalterfunktion ersetzt werden soll. *CODE* ist semantisch äquivalent zu *OBJECT*. Es soll andeuten, dass das Ziel des Zeigers übersetzter Java-Code ist. Mit Hilfe dieser Datenstrukturen sind die beiden obigen Ziele erreicht: Das Wissen über die Stellen, an denen Zeiger im Datenstrom vorkommen, ist genauso vorhanden, wie darüber auf welche Stelle im Datenstrom ein Zeiger später zeigen kann.

An dieser Stelle angekommen, lässt sich feststellen, dass man einen Großteil der in Abschnitt 6.1 genannten Ziele erreicht hat. Viel Code zum Laden und Binden der Bibliotheken kann eingespart werden, ebenso Code, der zum Erzeugen

```
typedef struct{
        long address;
        long index;
        short type;
}PatchElement;
typedef struct{
        long index;
        short type;
} BreakPoint;
typedef struct{
        long index;
        long address;
        short type;
} SymbolElement;
```

Abbildung 6.7: Datentypen für das Patchen

von Domain Zero Klassen nötig ist. Jedoch enthält der geladenen Java-Code, wie mehrfach angedeutet, Symbole, die während des Übersetzens nicht aufgelöst werden können und in Form von Symboldeskriptoren in den jeweiligen Methodendeskriptoren vermerkt sind. Da auch zum Auflösen dieser Symbole viel Code nötig ist und gleichzeitig alle Bedingungen erfüllt sind, die es möglich machen diese Symbole zu ersetzen, gibt es keinen Grund dies nicht zu tun. Auch in diesem Fall muss herausgefunden werden, an welcher Stelle im Daten-Array das zu ersetzende Symbol steht. Durch von JX übernommenen Codesegmenten kann der Wert berechnet werden, der an diese Stelle geschrieben werden soll. Dies alles geschieht nach dem Serialsieren des Codes einer Bibliothek. Zu diesem Zweck werden alle Klassen dieser Bibliothek und anschließend alle Methodendeskriptoren dieser Klasse durchlaufen. Für jede dieser Methoden wird zunächst die Adresse ihres Codes als *PatchElement* eingetragen, da auch diese Ziel eines Zeigers ist. Anschließend wird die Methode *patchMethodSymbols* aufgerufen, die bereits in JX vorhanden ist und leicht verändert wurde. Sie läuft iterativ über alle Symbole eines Methodendeskriptors. Für jeden der darin enthaltenen Symboldeskriptoren wird zunächst dessen Typ ausgelesen und anschließend mit einer für diesen Type spezifischen Behandlung fortgefahren. Da zum Einen für das Funktionieren des vorgestellten Systems nicht alle Typen notwendig sind und zum Anderen die Vielzahl der verschiedenen Arten von Symbolen und deren Behandlungsarten den Rahmen der vorliegenden Arbeit sprengen würden, soll aufgezeigt werden,

wie es generell möglich ist die bisherigen Konstrukte so zu erweitern, dass wie auch das Ersetzen von Symbolen im Java-Code möglich wird.

Symbole, die das Einfügen einer Konstanten in den Java-Code erforderlich machen, können einfach ersetzt werden, da Konstanten keine Zeiger sind. Alle sonstigen, zu ersetzenden Symbole, sind Zeiger und müssen in die oben beschriebenen Tabellen eingefügt werden. Jedoch sind hierzu einige Änderungen an diesen notwendige. Eine, die schon erwähnt wurde, ist, dass der BreakPoint-Typ um das Element *type* erweitert wurde, das die Werte *POINTER* und *PATCH* annimmt. Während *POINTER* die Typen der bereits beschriebenen Szenarien repräsentiert, wird durch *PATCH* angezeigt, dass sich das zu ersetzenden Element im Java-Code befindet. Diese Unterscheidung ist nötig, weil Zeiger im Java-Code zwar ersetzt werden müssen, aber ihre Länge bereits zwei Byte beträgt, da der Java-zu-Maschinencode-Übersetzer „weiß“ wie groß ein Zeiger auf dem RCX ist. Es ist auch hier erforderlich BreakPoints in die Tabelle einzufügen, um die Zeiger im Java-Code zunächst zu finden und anschließend zu ersetzen. Die Linearität auf den Indizes in den BreakPoints muss dabei jedoch erhalten bleiben. Auch für die im Java-Code vorhandenen Zeiger sind zwei verschiedene Ziele denkbar: eine Routine, die sich im Code einer anderen Bibliothek befindet und deshalb nicht zur Übersetzungszeit durch eine relative Adresse ersetzt werden konnte. Oder eine Funktion des Kerns beziehungsweise der VM, deren Adressen zur Kompilierzeit ebenfalls noch nicht bekannt waren. Das Einfügen der BreakPoints erfolgt also analog zum Auflösen der bisher untersuchten Zeiger. Was sich in diesem Fall unterscheidet, ist das Auffinden des Ziels des Zeigers. Zwar muss sich das Ziel des Zeigers in der Tabelle mit den *PatchElements* befinden um letzten Endes aufgelöst werden zu können, jedoch kann die entsprechende Stelle im Java-Code zum Zeitpunkt der Serialisierung nicht ersetzt werden, da hier nur zwei Bytes für einen Zeiger vorgesehen sind, während auf der ausführenden Instanz vier Bytes benötigt würden. Aus diesem Grund bleibt als Alternative die Möglichkeit eine weitere Tabelle anzulegen. Diese enthält Strukturen vom Typ *SymbolElement*, der aus dem Index des zu ersetzenden Zeigers im Daten-Array, der Adresse des Zeigerziels und einem Typ besteht. Dieser ist jedoch bisher auf den Wert *SEARCH* beschränkt.

```
insert_patchElement((long) &fkt_name,
                    (long) "(short) &fkt_name",
                    EXTERN);
```

Abbildung 6.8: Erzeugen eines PatchElements mit Typ EXTERN

Bevor dargestellt wird, wie aus den aufgebauten Tabellen während des Schreibvorgangs Zeiger sinnvoll ersetzt werden können, muss noch ein Wort zu den in Abschnitt 6.2 vorgestellten Domain Zero Klassen und damit insbesondere zu *java.lang.Object* gesagt werden. Da weder sie, noch der Code ihrer Methoden oder besser gesagt deren Platzhalterfunktionen über eine Bibliothek geladen werden, werden sie im bisher vorgestellten Ablauf auch nicht serialisiert. Das ist einerseits gut, da die Platzhalterfunktionen nicht mit auf die Zielplattform gelangen sollen. Andererseits sind jedoch auch diese Strukturen Ziel von Zeigern, so dass für jede von ihnen ein Eintrag in der Tabelle mit den *PatchElements* vorhanden sein muss. Das Adressenattribut des jeweiligen *PatchElement*s bekommt den Wert der Adresse der Platzhalterfunktion, sein Typattribut ist *EXTERN*. Das dritte Attribut bezeichnet den Index, der durch die Adresse gekennzeichneten Struktur. Da es in diesem Fall keine Struktur gibt, kann es auch keinen echten Index geben. Stattdessen wird, wie in Abbildung 6.8 zu sehen, eine Zeichenkette benutzt, die in den Typ zu *long* gewandelt wird. Das gleiche Vorgehen wird für später eventuell vorhandene Domain Zero Klassen und ebenso, obwohl keine Klassendeskriptoren, für die Funktionen der VM nötig sein.

6.6 Schreiben des Datenstroms

Nach dem Serialisieren liegen also fünf Datenstrukturen vor, von denen zwei, nämlich das Zeichenketten- und das Daten-Array, in Form von Dateien persistent gemacht werden sollen, während die anderen drei dabei helfen sollen. Unproblematisch ist das Speichern des Zeichenketten-Arrays, das nach dem Öffnen einer Datei durch einfache *fprintf* Anweisungen in die Datei geschrieben werden kann. Da nicht nur der Inhalt des Array gespeichert werden soll, sondern die enthaltenen Zeichen im Kontext von gültigem C-Code stehen sollen, muss vor dem Ausschreiben der Daten noch ein char-Array angelegt werden, das die jeweiligen Werte aufnehmen kann. Ein mögliches Resultat ist in Abbildung 6.9 angedeutet.

```
char strings[846] = {
  "<init>\0()V\0java/lang/Object\0getClass...\0"
};
```

Abbildung 6.9: Ergebnis

Die generelle Vorgehensweise für das Datenarray ist ähnlich. Auch hier wird zu Beginn das für ein Array vom Typ *short* notwendige Präfix in die Zieldatei geschrieben. Ebenso werden alle Einträge in einer Schleife abgearbeitet. Jedoch

müssen hier die vorher über die Zeiger gesammelten Informationen verarbeitet werden. Deshalb wird zunächst aus der Tabelle mit den BreakPoints derjenige mit dem kleinsten Index ausgelesen. Bis zu diesem Index werden alle Werte des Arrays so in die Datei geschrieben wie sie sind. Ist er erreicht, kommt es auf seinen Typ an, wie weiter verfahren wird. Einen generellen Überblick zum folgenden Text liefert Abbildung 6.10.

Handelt es sich um einen *POINTER*-Typ, so werden die folgenden vier Bytes aus dem Datenarray ausgelesen und zu einer Zahl zusammengefügt. Diese Zahl repräsentiert eine Adresse, zu der anschließend das *PatchElement* aus der entsprechenden Tabelle gesucht wird. Dessen Typ entscheidet, wie weiter verfahren wird. Sowohl bei *STRING*, als auch bei *OBJECT* und *CODE*, wird der im *PatchElement* vorhandene Index genutzt. Im ersten Fall repräsentiert er einen Index im bereits geschriebenen Zeichenketten-Array, im zweiten und dritten einen im Datenarray. Was letzten Endes in die Datei geschrieben wird lässt sich aus Abbildung 6.6 (b) ersehen, wobei es sich bei *strings* um das Zeichenketten- und bei *data* um das Daten-Array handelt. Ist der Typ dagegen *EXTERN*, wird der Index des *PatchElement*s als Zeiger auf eine Zeichenkette interpretiert (siehe auch Abbildung 6.8) und diese Zeichenkette in die Zieldatei geschrieben. Entspricht der Typ einem *NULL*-Zeiger, wird eine *0* geschrieben. Nachdem der Zeiger durch seinen späteren Wert ersetzt wurde, muss auch der Laufindex zusätzlich zur Inkrementierung in der Schleife um eins erhöht werden, da aus den vier Bytes auf dem Zwischenrechner zwei Bytes auf der Zielmaschinen gemacht werden. Erhöht man den Zähler nicht, werden das dritte und vierte Byte des gerade behandelten Zeigers in die Zieldatei geschrieben.

Wenn dagegen der BreakPoint den Typ *PATCH* hat, wird zum aktuellen Laufindex ein *SymbolElement* gesucht. Dieses beinhaltet die Adresse des Elements, das an die aktuelle Position geschrieben werden soll und lässt sich auf die gleiche Art und Weise suchen wie im vorhergehenden Fall. Der Unterschied besteht lediglich darin, dass hier bei der Suche ausschließlich *PatchElement*e vom Typ *OBJECT* oder *CODE* auftreten können, da die Java-Ebene keine puristischen Zeichenketten kennt, sondern alles derartige in Form von String-Objekten verwaltet. Gleichermaßen darf kein *NULL*-Zeiger auftreten, da dieser beim Übersetzen des Java-Codes aufzulösen gewesen wäre. Danach wird der nächste BreakPoint ausgelesen und bearbeitet.

Neben den aus den Bibliotheken geladenen Datenstrukturen befinden sich wie mehrfach erklärt auch Strukturen, die zur Domain Zero gehören und von der Java-Schicht ansprechbar sind. Bisher wurden lediglich zwei Arrays in eine Datei geschrieben. Auf diesen Daten arbeitet die Domain Zero um sich selbst zu

```
1. BreakPoint.type == POINTER:

   (a) PatchElement.type == STRING
       ⇒ (unsigned short) &strings[index]
   (b) PatchElement.type == OBJECT ||
       PatchElement.type == CODE
       ⇒ (unsigned short) &data[index]
   (c) PatchElement.type == EXTERN
       ⇒ (char*) index
   (d) PatchElement.type == NULLP
       ⇒ 0x0

2. BreakPoint.type == PATCH
   ⇒ SymbolElement.type == SEARCH
   SymbolElement.address ⇒ PatchElement

   (a) PatchElement.type == OBJECT
       ⇒ (unsigned short) &data[index]
   (b) PatchElement.type == EXTERN
       ⇒ (char*) index
```

Abbildung 6.10: Typen und Ausgabe

initialisieren. Jedoch ist dies nicht ausreichend, da es keinen Einstiegspunkt in das Datenarray gibt und es somit bisher nutzlos ist. Deshalb müssen zusätzlich noch verschiedenen andere Einträge in die gleiche Datei oder in andere, neu erzeugte Dateien, erfolgen. Diese Einträge sind im später erzeugten Kern globale Variablen und für das Funktionieren des Gesamtsystems unabdingbar. Im Falle des oben erzeugten *java.lang.Object* ist diese Variable ein Klassendeskriptor und heißt *java_lang_Object*. Gleichzeitig ist die Adresse des auf dem Zwischenrechner erzeugten *java.lang.Object*s bekannt und befindet sich im Daten-Array. Dies ermöglicht es analog zur obigen Vorgehensweise nach dem Index zu suchen, den es auf dem Zielsystem haben wird und diese Information in die Datei zu schreiben.

6.7 Zusammenfassung und Bewertung

Natürlich entstehen durch das Vorgehen, alle Bibliotheken bereits vor dem Ausführen des Systems aufzubereiten, einige Nachteile. Wenn der Bedarf an Bibliotheken dynamisch, also zum Beispiel durch Verzweigungen im Programmfluss,

bestimmt wird, betreibt man unter Umständen einen erheblichen Mehraufwand, da *offline* Bibliotheken bearbeitet werden, die später eventuell nicht benötigt werden. Jedoch ist ebenso das Gegenteil der Fall, wenn sich während des Ablaufs Bedarf für eine Bibliothek ergibt, an die während der Vorbereitungsphase nicht gedacht wurde. In diesem Fall besteht keine Möglichkeit mehr die Bibliothek noch in den RCX zu laden und die Programmausführung wie gewünscht fortzusetzen. Sie muss entweder abgebrochen oder zumindest verändert werden, was beides nicht im Sinne des Benutzers sein kann.

Jedoch stehen dem die bereits als Motivation angeführten Pluspunkte gegenüber. Es werden einige hundert Codezeilen eingespart und somit ermöglicht, dass mehr benutzerspezifischer Code auf die Plattform geladen werden und zur Ausführung kommen kann. Ebenso müssen die oben genannten Nachteile nicht zwangsweise als solche interpretiert werden, wenn man eine hybride Form der Bibliotheksverwaltung betrachtet, in der es möglich ist, das Entpacken und Bearbeiten der Bibliotheken auf einem leistungsstärkeren Rechner erfolgen zu lassen. Jedoch nicht *offline*, vor dem Kompilieren und Laden auf das System, sondern *online*, während des Betriebs. In diesem Fall teilt der RCX dem anderen Rechner mit, welche Bibliotheken er benötigt. Nach diesen Angaben baut dann der Linker Strukturen auf und übermittelt diese auf den Baustein. Ebenso ist es denkbar, dass der RCX nicht mehr benötigte Bibliotheksstrukturen löscht um Platz für neue Anforderungen zu schaffen.

Jedoch bleibt festzuhalten, dass zum bisherigen Entwicklungszeitpunkt des Systems eine solche Dynamik nicht benötigt wird, da bisher lediglich eine Java-Domain mit einer Bibliothek getestet wurde. Bereits diese Konfiguration verschlingt so viel des vorhandenen Speichers, dass es durchaus fraglich erscheint, ob es mehr als eine Bibliothek gleichzeitig auf dem System geben kann.

Ein weitere Vorteil der statisch erzeugten Strukturen ergibt sich durch gewisse Optimierungsmöglichkeiten. So steht zum Beispiel die Menge an maximal verwendeten Bibliotheken bereits von vornherein fest, so dass bereits zum Zeitpunkt des Ladens einer Domain ausreichend Platz reserviert werden kann. Weiterhin werden viele der in den Deskriptoren existierenden Felder überflüssig, weil sie lediglich dazu dienen während des Ladevorgangs andere Strukturen zu finden und später nicht mehr gebraucht werden. Als Beispiel sei der *next*-Zeiger in den Bibliotheksdeskriptoren angeführt, der eingespart werden kann, wenn man die Bibliotheken nicht mehr als verkette Liste, sondern als Array ablegt, was nun durchaus möglich ist, weil sich ihre Anzahl nicht mehr verändert.

Kapitel 7

Zusammenfassung und Ausblick

In den vergangenen Kapiteln wurde ein Weg aufgezeigt, wie eine Java-Laufzeitumgebung für den LEGO Mindstorms RCX basierend auf dem JX-Betriebssystem erstellt werden kann. Hierzu wurde zunächst ein Mikrokernel erstellt, der es zulässt, die RCX-spezifische Hardware anzusprechen und zu steuern und auf die veränderte Unterbrechungsbehandlung Rücksicht nimmt. Weiterhin wurde in diesem Zusammenhang ein Thread-Konzept implementiert und die Domain-Schnittstelle von JX übernommen. Damit einhergehend wurde ein domainübergreifender Scheduler mit einer Round-Robin-Strategie erstellt. Zur Thread-Synchronisation werden blockierende Semaphoren eingesetzt. Problematisch im Zusammenhang mit der Realisierung hardwarenaher Software war die durchaus mangelhafte Dokumentation der LEGO-spezifischen Hardware, was vor allem Motoren, Sensoren und die Infrarotschnittstelle betrifft. Hier ist man weitestgehend auf Dokumentation aus privaten Quellen im Internet angewiesen, die nicht immer vollständig und in manchen Fällen sogar widersprüchlich ist. Während die Steuerung der Motoren und Sensoren nach einigen Versuchen realisiert werden konnte, war es bei der Infrarotschnittstelle bisher nicht möglich Daten sowohl zu senden als auch zu empfangen. Eine, für das *Debugging* notwendige, ausschließlich unidirektionale Datenübertragung vom Baustein auf den PC als auch umgekehrt konnte jedoch erreicht werden. Auf Grund des begrenzten Speicherplatzes war es jedoch oftmals notwendig, Konstrukte um- oder zurückzubauen oder sogar zu streichen, wie etwa die globale Speicherverwaltung, die nun domain-intern abläuft. Der JX-Monitor, der es erlaubt die Vorgänge auf dem System von außen über die serielle Schnittstelle abzufragen, wurde ganz herausgenommen. Es hat sich im Zusammenhang mit einer sinnvollen Speicherausnutzung bewährt, auf Textausgaben — wo möglich — zu verzichten und die Optimierungen des Übersetzers zu

nutzen, wenn die gewünschte Funktionalität in einem Entwicklungsschritt erreicht worden war.

Der zweite Teil der Arbeit bestand darin, die Java-Laufzeitumgebung von JX so umzubauen, dass alle während des Betriebs benötigten Konstrukte zur Verwaltung von Java-Code bereits vor dem eigentlichen Betrieb auf einem Drittrechner erzeugt und anschließend zusammen mit dem Kern auf den Baustein geladen werden können. Dadurch muss die Funktionalität, die dafür nötig ist, nicht auf dem Baustein beziehungsweise im Kern vorhanden sein, wodurch einige hundert Zeilen Code einspart und in Folge dessen mehr benutzer-spezifischer Code auf den RCX geladen werden kann. Die Informationen über den Java-Code und der Code selbst sind in sogenannten Bibliotheken verfügbar, die in JX während des Betriebs bei Bedarf in das System geladen werden. Im vorliegenden Fall übernimmt ein im Verlauf der Arbeit erstelltes Programm die Aufbereitung aller für den Betrieb notwendigen Bibliotheken. Das heißt, es verhält sich mit kleineren Veränderungen so wie JX, wenn Bibliotheken geladen werden müssen: Sie werden entpackt und die darin enthaltenen Informationen ausgewertet. Daraus werden verschiedene Konstrukte erzeugt. Diese sind miteinander verbunden und verweisen aufeinander, so dass sie einen Graph aufspannen. Um nun die Möglichkeit zu bekommen, die entstandenen Strukturen zu kompilieren und mit dem Kern auf den Baustein zu laden, muss der Graph zunächst serialisiert und die darin vorhandenen Informationen anschließend so aufbereitet werden, dass sie wieder einen Graphen ergeben; diesmal allerdings nicht im Speicher sondern als C-Code in einer Datei. Neben dem Inhalt der Bibliotheken sollten ebenfalls einige Strukturen der Domain Zero, der Teil des JX-Betriebssystem, der durch Java-Code ansprechbar ist, statisch erzeugt werden, da dies den hierfür nötigen Code einspart und auf diese Weise wiederum wertvollen Speicherplatz sparen hilft.

Im Gegensatz zu JX ist man nun nicht mehr darauf angewiesen vor dem Start der Domain Zero Bibliotheken zu entpacken und Strukturen aufzubauen. Stattdessen müssen lediglich noch einige globale Zeiger gesetzt werden, die zum Funktionieren der Domain Zero nötig sind. Nachdem dies geschehen ist, kann theoretisch Java-Code in einer eigenen Domain gestartet werden. Dies wurde jedoch nur rudimentär getestet, da der dafür notwendige Java-nach-H8/300-Compiler noch nicht die notwendige Reife aufwies, sowie der zeitliche Rahmen eng wurde. Das Ausführen von Java-Code und der Test des damit verbundenen Codes bleibt somit weiterführenden Arbeiten überlassen. Ein Ziel, das in einem solchen Rahmen in kurzer Zeit erreicht werden kann und dessen Umsetzung äußerst sinnvoll erscheint ist das Erstellen und Anbinden von Peripherietreibern auf Java-Ebene. Da die meisten Peripheriegeräte, vor allem auch die von den Anwenderschichten am

meisten genutzten Motoren und Sensoren, entweder in den Speicher eingebunden sind (Motoren, SCI) oder nur durch andere in den Speicher eingebundene Geräte ausgelesen werden können (zum Beispiel Sensoren durch den A/D-Wandler), kann ein Java-Treiber mit einem der in Kapitel 3 beschriebenen *Memory*-Objekten arbeiten. Alternativ können Treiber auch als Domain Zero Klassen, also in C, realisiert werden. Diese haben den Vorteil, dass der bisherige Code wahrscheinlich nur leicht verändert und um Initialisierungsroutinen erweitert werden muss. Trotzdem sind sie — bei der Definition geeigneter Schnittstellen — vom Java-Code nutzbar.

Letztendlich soll nicht verschwiegen werden, dass es trotz des Versuchs alle Anforderungen bestmöglich zu erfüllen in der vorliegenden Arbeit Stellen gibt, die für einen praxistauglichen Einsatz verbessert werden müssen: Die im bisherigen Betrieb unidirektional arbeitende Infrarotschnittstelle konnte trotz intensiver Bemühungen nicht so angesprochen werden, dass eine problemlose Datenübertragung in beide Richtungen möglich ist. Grund dafür sind Reflexionen, die dazu führen, dass ein gesendetes Zeichen mehrmals empfangen wird. Als Lösung kann ein Protokoll entwickelt werden, das mit solchen Erscheinungen zurecht kommt. Weitere Aufschlüsse kann ein Blick in das LEGO-ROM geben, in dessen Funktionen bereits eine zweiseitige Kommunikation realisiert ist. Ebenfalls verbessert werden sollten die vom Mikrokern angebotenen Schnittstellen für Peripheriegeräte, da bei deren Entwicklung in den meisten Fällen nur die Beherrschung der Geräte und weniger die Ziele und Bedürfnisse potenzieller Benutzer im Mittelpunkt standen. Deshalb wirken die meisten Schnittstellen eher willkürlich gewählt. Ein weiteres Manko stellen die trotz zahlreicher Bemühungen noch möglichen Optimierungen hinsichtlich des Speicherausnutzung dar. Der Versuch, einen Teil der aufgeblähten Interruptverwaltung in Assembler zu realisieren, was gut 200 Byte zusätzlichen freien Speicher zur Folge hatte, kann nur ein Anfang gewesen sein. Ansatzpunkte für eine weitere Optimierung durch Assembler stellen vor allem die Funktionen des Mikrokerns dar. Aber auch durch Veränderungen am externen Programm zur Bibliothsaufbereitung können Gewinne möglich sein. Jedoch ist hier für jeden Eintrag in den in Abbildung 5.4 gezeigten statischen Strukturen eine Einzelprüfung auch unter dem Gesichtspunkt zukünftiger Entwicklungen nötig. Weitere Optimierungsmöglichkeiten, auch zum Beispiel durch Nutzung der ROM-Funktionalität, wurden im Text erwähnt. Aus Mangel an verfügbarem Java-Code konnte das entwickelte Programm nur zu einem bestimmten Teil getestet werden. Deshalb ist es sicherlich noch nicht vollkommen ausgereift. Trotz dieser Liste an Schwachstellen wurde die Aufgabenstellung im Rahmen der verfügbaren Zeit vollständig erfüllt.

Literaturverzeichnis

[BSP+95] Bershad, Brian N., Stefan Savage, Przemyslaw Pardyak, David Becker, Marc Fiuczynski und Emin Gün Sirer: *Protection is a Software Issue*. Paper, Department of Computer Science and Engineering FR-35, University of Washington, 1995.

[Fel00] Felser, Meik: *Design und Implementierung eines Simulators für das LEGO Mindstorms Robotic Invention System*. Studienarbeit, Institut für Mathematische Maschinen und Datenverarbeitung (IV), Universität Erlangen-Nürnberg, 2000.

[GFWK02] Golm, Michael, Meik Felser, Christian Wawersich und Jürgen Kleinöder: *The JX Operating System*. In: *USENIX Association: General Track 2002 USENIX Annual Technical Conference*, Seiten 45–58, 2002.

[Gol00] Golm, Michael: *The Structure of a Type-Safe Operating System*. Dissertation, Universität Erlangen-Nürnberg, Institut für Informatik, 2000.

[Hof02] Hofmann, Friedolin: *Systemprogrammierung I*. Skript zur Vorlesung, Institut für Mathematische Maschinen und Datenverarbeitung (IV), Universität Erlangen-Nürnberg, 2002.

[HvE98] Hawblitzel, Chris und Thorsten von Eicken: *A Case for Language-Based Protection*. Paper, Department of Computer Science, Cornell University, 1998.

[infa] *GNU Assembler*. info Seiten. Version 2.14.

[infb] *GNU Linker*. info Seiten. Version 2.14.

[KR90] Kernigham, Brian W. und Dennis M. Ritchie: *Programmieren mit C*. Carl Hanser Verlag, München, zweite Auflage, 1990.

[Nik00] Nikander, Pekka: *An Operating System in Java for the LEGO Mindstorms RCX Microcontroller*. In: *2000 USENIX Annual Technical Conference*, 2000.

[Nog] Noga, Markus L.: *legos Homepage*. Website. www.legos.de.

[Pro98] Proudfoot, Kekoa: *RCX Internals*. Website, 1998. graphics.stanford.edu/~kekoa/rcx.

[Ren03] Renesas Technology Corp: *Hitachi Single-Chip Microcomputer H8/3792 Series H8/3297 HD6473297, HD6433297 H8/3296 HD6433296 H8/3294 HD6473294, HD 6433294 H8/3292 HD6433292 Hardware Manual*, dritte Auflage, 2003. documentation.renesas.com/eng/products/mpumcu/e602080_h83297.pdf.

[SGG03] Silberschatz, Abraham, Peter Bear Galvin und Greg Gagne: *Operating System Concepts, international Edition*. John Wiley & Sons, Inc., sechste Auflage, 2003.

[SP03] Schröder-Preikschat, Wolfgang: *Betriebssysteme*. Skript zur Vorlesung, Institut für Mathematische Maschinen und Datenverarbeitung (IV), Universität Erlangen-Nürnberg, 2003.

[Waw01] Wawersich, Christian: *Design und Implementierung eines Profilers und optimierenden Compilers für das Betriebssystem JX*. Diplomarbeit, Institut für Mathematische Maschinen und Datenverarbeitung (IV), Universität Erlangen-Nürnberg, 2001.

Anhang A

Erstellen des Cross-Compilers und Debuggers

Um Programme, die auf einem PC System für den RCX entwickelt werden übersetzen zu können benötigt man einen Cross-Compiler. Cross heißt er deshalb, weil er Code für eine andere Plattform erzeugt, als die auf der er abläuft. Gleiches gilt für den Debugger: er läuft zum Beispiel auf einem Linux-System, behandelt aber Programme, die auf einem RCX-System ablaufen. Im Folgenden soll kurz dargestellt werden, wie man aus den Quellen eines Compilers und denen eines Debuggers das jeweils lauffähige Programm beziehungsweise Programmpaket erstellt. In beiden Fällen sind die Produkte von GNU gewählt worden, da für sie neben ihrer Qualität auch die Tatsache spricht, dass sie umsonst beziehbar sind.

Beide Programme lassen sich als Bzip-Tar-Archive aus dem Internet herunterladen. Nach dem Entpacken und aus packen aus dem Archiv mit *tar -xjf <dateiname>* hat man jeweils einen Quellbaum. Da das Verfahren, das zu einem lauffähigen Programm führt ist in beiden Fällen relativ ähnlich ist, wird im Folgenden nur noch auf das Erstellen des Compilers eingegangen.

Als Leitfaden für die Installation dient die im Quellbaum enthaltene README-Datei, die sich im INSTALL-Verzeichnis befindet. In ihr sollte nachgelesen werden, wenn Installationen abweichend von der hier angegebenen gewünscht werden. Um überhaupt ein funktionierendes Programm erhalten zu können, ist es nötig einen aktuellen Compiler und aktuelle binutils zu besitzen. Die jeweils benötigten Versionen finden sich in obiger Datei. Sind die Programme nicht aktuell oder nicht auf dem jeweiligen Rechner vorhanden, kann man sie ebenfalls kostenlos aus dem Internet herunterladen. Der Installationsablauf ist ähnlich zu dem hier beschriebenen, nur dass alle zielplattformspezifischen Konfigurationen entfallen.

Sind alle Zusatzprogramme installiert, wird im ersten Schritt der Installation darin ein Verzeichnis anzulegen, aus dem heraus der Konfigurations- und Kompiliervorgang gestartet wird. Anschließend wechselt man in dieses Verzeichnis und führt das im Quellverzeichnis liegende Script *configure* aus:

% mkdir objdir

% cd objdir

% srcdir/configure [options] [target]

Optionen müssen keine angegeben werden. Jedoch empfiehlt es sich die Option *–program-prefix=<prefix>* zu verwenden. Diese stellt allen ausführbaren Datei das Präfix <prefix> voran, so dass verhindert wird, dass bereits vorhandene Dateien, überschrieben werden. <prefix> könnte zum Beispiel den Wert *h8300-hms* haben und somit im Falle des *gcc* verhindern, dass der Compiler, der den Namen *gcc* trägt und Programme für ein Linux System auf Intel-Basis erstellt überschriben wird, da das neue Programm *h8300-hms-gcc* heißen wird. Der vollständige *configure* Befehl sieht also folgendermaßen aus.

% srcdir/configure –program-prefix=h8300-hms –target=h8300-hms

Wenn dieser Aufruf erfolgreich ist, wird in aller Regel auch das Kompilieren erfolgreich verlaufen. Falls nicht lohnt sich auf jeden Fall ein Blick in die oben genannte Datei. Kompilieren und Installieren erfolgt durch zwei weitere Befehle:

% make

% make install

Wobei *make install* die Programme in */usr/local* installiert, falls *configure* keine anderen Parameter übergeben wurden.

Anhang B

Erstellen und Laden von Code

Das Erstellen von Code für den RCX unterscheidet sich nicht von der Code-Erstellung wie sie sonst abläuft. Der einzige Unterschied besteht im verwendeten Compiler. Was sich jedoch stark von der sonstigen Vorgehensweise unterscheidet ist die Art und Weise, wie Code zur Ausführung gebracht wird. Hierzu benötigt man zunächst zwei essentielle Bestandteile: einen RCX-Baustein und einen dazugehörigen LEGO-Tower. Weiterhin wird zu diesem Tower ein Treiber benötigt, wenn man die USB Version des Towers verwendet. Dieser Treiber ist ab Linux-Kernel Version 2.6 im Kern vorhanden, wobei er erst ab Version 2.6.7 funktioniert.

Mit diesem Equipment kann prinzipiell eine Verbindung zwischen dem Baustein und dem RCX hergestellt werden. Um eine neue Firmware aufzuspielen benötigt man zusätzlich ein Programm, das das hierfür nötige Protokoll unterstützt. Ein Programm, dass diese Eigenschaft erfüllt ist *firmdl*. Es ist frei im Internet verfügbar, benötigt jedoch einen ebenfalls frei verfügbaren Patch, falls ein USB Turm benutzt wird. *firmdl* kennt zwei Übertragungsgeschwindigkeiten, schnell und langsam. Dieser Parameter wird auf der Standardeingabe mit *-f* beziehungsweise *-s* angegeben, wobei nicht alle USB Treiber die *-f* Option unterstützen. Weiterhin muss dem Programm noch eine Schnittstelle auf der Kommandozeile übergeben werden, die im Falle des USB-Turms *–tty=usb*, in jedem anderen Fall *–tty=<device>*, lautet. Letztendendes muss noch eine srec Datei angegeben werden, die zum Baustein übertragen wird.

Nachdem der Code geladen wurde, kann es vorkommen, dass *firmdl* meldet, dass die Firmware nicht freigeschalten werden konnte. Das heißt eigentlich, dass der Schlüssel ,,Do you byte, when I knock?" nicht dort gefunden wurde, wo er erwartet wird. Jedoch ist das Programm in diesem Punkt äußerst unzuverlässig,

so dass man sich auf die akustische Rückmeldung des Bausteins verlassen sollte, die in Form einer Melodie erfolgt. Diese ist in jedem Fall eindeutig und sofort verständlich.

Anhang C

Debugger und Simulator

Der Simulator ist ein Java Programm, das beim Autor erhältlich ist. Da die meisten Optionen in [Fel00] und auch in der README-Datei des Simulators erläutert sind, soll im Folgenden nur auf die Anbindung des Debuggers eingegangen werden.

Da der Debugger die standardmäßig vom Compiler erzeugte *coff* Datei zum Auslesen der Symbole benötigt, der Simulator jedoch eine srec-Datei, muss zunächst aus der *coff* die *srec* Datei erstellt werden. Dazu kann das Programm *objcopy* mit folgendem Aufruf benutzt werden.

% h8300-hms-objcopy -I coff-h8300 -O srec <coff-datei> <srec-datei>

Anschließend wird der Simulator mit den Optionen *-f* und *-gdb* gestartet, so dass er gleich nach seinem Start darauf wartet, dass der Debugger über UDP-Port 9000 Kontakt mit ihm herstellt. Will man einen anderen Port benutzen kann man die Option *-gdbport <portnummer>* verwenden. Danach startet man den Debugger mit der *coff*-Datei als Parameter. Um die Verbindung mit dem Simulator herzustellen muss dem Debugger mitgeteilt werden, wo die ausführende Instanz zu finden ist. Dies geschieht mit *target remote localhost:9000*, wenn sowohl Debugger als auch Simulator auf dem gleichen Rechner laufeb und der Standardport benutzt wird. Ansonsten wird statt ,,localhost" der Rechnername, auf dem der Simulator läuft, sowie statt ,,9000" der Simulatorport eingesetzt. Die Programmausführung wird mit dem Befeht *,,c"* im Debugger gestartet. Der Programmablauf kann durch die Tastenkombination *,,Crtl-C"* unterbrochen und mit *,,c"* wiederum fortgesetzt werden. Über weitere Befehle gibt der Befehl *,,help"* Auskunkft.

Abbildungsverzeichnis