

## Zusammenfassung

Die *Realtime Specification for Java* beschreibt die Erweiterungen der *Java Language Specification* und der *Java Virtual Machine Specification*, die im Zuge des JSR – 000001, für den Einsatz in Echtzeitsystemen, gemacht wurden. Diese Arbeit beschäftigt sich mit dem Aufbau der API, die im Modul *javax.realtime.\** gekapselt ist, und zeigt mit Hilfe von zahlreichen kleinen Programmen, wie mit diesen Klassen gearbeitet wird und wo deren Stärken und Schwächen liegen. Anhand einer Robotersteuerung wird sie auf ihre Eignung für größere Projekte hin untersucht. Hierzu wurde eine Bibliothek erstellt, die im Anschluss von mehreren Anwendungen, die den Roboter klassische Aufgaben ausführen lassen, genutzt wird. Da es sich hierbei um ein Pilotprojekt handelt, lag auch einer der Schwerpunkte darin, sich mit den Entwicklungswerkzeugen und dem Entwicklungsprozess auseinander zu setzen. In diesem Zusammenhang wird detailliert auf die JamaicaVM und deren Tools, sowie auf das Echtzeitbetriebssystem RTEMS eingegangen.

## Inhaltsverzeichnis

1. Echtzeit.....	3
2. Java und Echtzeit.....	4
2.1 Java.....	4
2.2 RTSJ.....	6
3. Entwicklungsumgebung.....	58
3.1 Programmiergerät.....	58
3.2 Steuergerät.....	84
4. Robotersteuerung.....	92
4.1 Hardware.....	92
4.2 Software.....	101
5. Fazit.....	132
5.1 Arbeitshinweise.....	132
5.2 Schwächen.....	134
5.3 Stärken.....	136
5.4 Zukunft.....	137
6. Anhang.....	138
6.1 Literaturverzeichnis.....	138
6.2 Abkürzungsverzeichnis.....	141

## **1. Echtzeit**

Computer haben die Welt in den letzten Jahrzehnten maßgeblich beeinflusst und Einzug in fast alle Lebensbereiche gefunden. Angefangen bei den Mobiltelefonen, über den PC am Arbeitsplatz bis hin zu Steuerungen großer Industrieanlagen, sind sie nicht mehr wegzudenken. Dabei hat sich ihr Einsatzgebiet stark mit der zunehmenden Leistung und den sinkenden Preisen vergrößert. Während es zu Beginn nur einige wenige Großrechner gab, die viele oder komplizierte Berechnungen durchführen mussten und auf deren Ergebnisse der Nutzer getrost warten konnte, werden sie mittlerweile in Systemen eingesetzt, die eng mit realen Prozessen verzahnt sind. Bei diesen sind neben den rein logischen Anforderungen auch noch zeitliche zu beachten. Denn es ist wohl offensichtlich, dass ein Roboter in einer Schweißstraße nicht nur an der richtigen Position, sondern auch zum richtigen Zeitpunkt, mit der Arbeit beginnen muss.

## 2. Java und Echtzeit

Bisher wurden Echtzeitsysteme, wie beispielsweise die eingangs erwähnte Robotersteuerung, mit PERAL, Ada oder C erstellt. Doch im Zuge der zunehmenden Vernetzung und Integration hat man begonnen sich nach einer universelleren Sprache umzusehen. Diese soll um ein standardisiertes Modul, das die Programmierung solcher Systeme erlaubt, erweitert werden.

### 2.1 Java

Java spielt bei solchen Betrachtungen keine unbedeutende Rolle, zumal es in den letzten Jahren großen Zulauf gefunden hat. Dies ist vor allem auf die folgenden, teilweise einzigartigen, Eigenschaften zurückzuführen.

#### 2.1.1 Plattformunabhängig

Die wahrscheinlich Bekannteste von diesen ist unter dem Namen Plattformunabhängigkeit bekannt. So wird das Phänomen bezeichnet, bei dem eine Applikation, die einmal geschrieben wurde, ohne Veränderung auf allen anderen Systemen ausgeführt werden kann. Dies ist möglich, da die Anwendungen für eine abstrakte Plattform, dem JRE, geschrieben und erst zur Laufzeit von der VM auf die Reale abgebildet werden.

#### 2.1.2 Einfach

Ein weiteres herausragendes Merkmal von Java ist die Tatsache, dass es relativ leicht zu erlernen ist. Dies ist ein Verdienst des wohl durchdachten Designs, bei dem man sich im Vorfeld mit den Stärken und Schwächen bestehender Sprachen auseinander gesetzt hat. So bediente man sich der vertrauten Syntax und Semantik von C/C++, wobei fehleranfällige Konzepte, wie beispielsweise die Zeiger, abgeschafft wurden, während man bewährte Konzepte, wie das Exception-Handling, sogar noch ausgebaut hat. Darüber hinaus wurden noch neue Konzepte, wie beispielsweise RMI oder die automatische Speicherverwaltung, die dem Entwickler die Arbeit zusätzlich erleichtern, eingeführt. Abgerundet wird das Ganze durch den streng objektorientierten Ansatz, der dem Menschen von Natur aus vertrauter ist.

#### 2.1.3 Modular

Java ist streng modular aufgebaut und geht sogar so weit, dass der Entwickler gezwungen wird *packages*, so heißen diese Module, anzulegen. Das Herausragende an diesen ist aber nicht, dass sie sich sehr leicht erstellen und importieren lassen, sondern, dass sie dynamisch, also erst wenn man sie benötigt, eingebunden werden. Das hat den Vorteil, dass diese Module leicht und sogar zur Laufzeit ausgetauscht werden können. Im Großen und Ganzen reduziert dies den Wartungsaufwand und sorgt dafür, dass die Wiederverwendbarkeit von Code stark vereinfacht wird.

### **2.1.4 Mächtig**

Außerdem bietet Java neben dem Sprachkern noch eine Reihe standardisierter und sehr mächtiger Module an, mit denen vergleichbare Sprachen nicht mithalten können. Das Besondere an diesen ist, dass sie nur im Zuge des JCP aufgenommen werden. Was bedeutet, dass ein jeder den Bedarf an einem neuen Modul anmelden, an einer Umsetzung mitarbeiten und Einwände zu einer vorgeschlagenen Lösung machen kann. Damit ist sichergestellt, dass Java nur kontrolliert wächst und Probleme im Vorfeld vermieden werden.

### **2.1.5 Sicher**

Des weiteren zeichnet es sich durch sein ungeahntes Maß an Sicherheit aus. Diese resultiert auch aus der Tatsache, dass man sich auf der abstrakten Ebene des JRE bewegt. Der Entwickler hat dadurch keinen direkten Zugriff auf die HW, sondern kann nur auf die Services zugreifen, die ihm von der VM angeboten werden. Dadurch ist er zwar in seiner Entfaltung eingeschränkt, doch dafür ist sichergestellt, dass er sich nur innerhalb bestimmter Grenzen, der so genannten Sandbox, bewegen kann.

## 2.2 RTSJ

Java bietet, wie man sieht, eine Reihe von Vorteilen, die auch für Echtzeitsysteme von Interesse sind. Allerdings kann es nicht direkt für solche Anwendungen eingesetzt werden, da es dafür nicht konzipiert wurde und daher auch nicht allen Anforderungen, die diese an die VM und die API stellen, genügt. Es ist zwar immer möglich, mit Hilfe von JNI, native Code zu nutzen, um solche Systeme umzusetzen. Doch dabei gehen einige dieser Vorteile verloren.

### 2.2.1 Geschichte

Daher haben sich schon sehr bald, nach der Publikation von Java im Jahre 1995, verschiedene Gruppen und Unternehmen, unter anderem *Sun*, IBM und das *National Institute of Standards and Technology*, damit beschäftigt, wie eine ganzheitliche Lösung, die nicht auf native Code zurückgreifen muss, aussehen könnte. Im Zuge der Einführung des JCP, wurden alle diese Bestrebungen vereinheitlicht, da mit dem JSR – 000001 die einzige offiziell anerkannte Echtzeiterweiterung für Java geschaffen werden sollte.



Abbildung 1: Geschichte der RTSJ

Bereits Mitte 2000 veröffentlichte diese, mit der Version 0.9 der RTSJ, ihre ersten Ergebnisse. Diese umfassten sowohl eine Erweiterung der *Java Language Specification*, als auch eine der *Java Virtual Machine Specification*. Allerdings ließ die Referenzimplementierung, die einen festen Bestandteil des JCP darstellt, durch eine Reihe von Problemen bei IBM, auf sich warten. *Timesys* hat diese daraufhin übernommen, so dass die RTSJ im Jahr 2002 in der Version 1.0 abgenommen werden konnte. Dieses Unternehmen war auch das Erste, das eine Implementierung der RTSJ kommerziellen Nutzern, unter dem Namen *JTime*, anbieten konnte. Mittlerweile haben eine Reihe anderer Unternehmen und Institutionen nachgezogen. Somit stehen mehrere Alternativen zur Auswahl.

Zu den Bekanntesten zählen:

- JamaicaVM der aicas GmbH<sup>1</sup>
- OVM Project mehrerer Universitäten und Firmen<sup>2</sup>
- JRate von SourceForge<sup>3</sup>
- Mackinac von Sun<sup>4</sup>
- Aphelion von Apogee<sup>5</sup>

Dass man weiter an der Idee, Java in Echtzeitsystemen einzusetzen, festhält, verdeutlicht die Abnahme der Version 1.0.1 im Juni dieses Jahres, das Einrichten des JSR-282 und die Tatsache, dass im Moment eine Vielzahl an Projekten durchgeführt werden. Während einige Gruppierungen diese noch vorsichtig untersuchen und bewerten, gibt es auch eine Reihe von Firmen und Institutionen, die bereits von der RTSJ überzeugt sind und in millionenschwere Projekte investieren, um diese Technologie zu forcieren. Sie kommen vor allem aus den Bereichen Militär, Telekommunikation und Luft- und Raumfahrt.

Projekte<sup>6</sup>:

- *Golden Gate* – Luft- und Raumfahrt
- *DD(X)* – Militär
- *netcentric battlefield* – Militär
- *Mackinac* – Industrie

---

1 [www.aicas.com](http://www.aicas.com)

2 [www.ovmj.org](http://www.ovmj.org)

3 [www.jrate.sourceforge.net](http://www.jrate.sourceforge.net)

4 [research.sun.com/projects/mackinac](http://research.sun.com/projects/mackinac)

5 [www.apogee.com/aphelion.html](http://www.apogee.com/aphelion.html)

6 [www.automotivedesignline.com/news/159907195](http://www.automotivedesignline.com/news/159907195)

## 2.2.2 Eigenschaften

Dieses breite Interesse an der RTSJ ist darauf zurückzuführen, dass man Fachkräfte aus allen Bereichen herangezogen hat, um die Eigenschaften<sup>7</sup>, die diese auszeichnen und von bewährten Technologien abheben soll, festzulegen.

Als Schlüsselmerkmale wurden 7 Punkte ausgearbeitet:

- **Applicability to Particular Java Environments**

Die RTSJ muss für alle Java Environments gültig sein. Das bedeutet, dass Echtzeitanwendungen, unabhängig davon ob sie für Server, Desktops oder kleinere Systeme geschrieben wurden, alle die gleiche Bibliothek nutzen können.

- **Backward Compatibility**

Applikationen, die RTSJ nicht nutzen, müssen auf einer VM, welche die RTSJ unterstützt, auch laufen. Damit soll gewährleistet werden, dass bestehender Code wieder verwendet werden kann.

- **Write Once, Run Anywhere**

Die Bedeutung der Portabilität von Applikationen hat auch im Umfeld von Echtzeitsystemen ihre Gültigkeit. Allerdings kann und muss man, auf Grund der Vielzahl der unterschiedlichen Plattformen, möglicherweise Abstriche machen.

- **Current Practice vs. Advanced Features**

Damit die RTSJ auch Zukunft hat, muss sie so angelegt sein, dass sie sowohl aktuelle Standards unterstützt, als auch Platz für Erweiterungen bietet.

- **Predictable Execution**

Die höchste Priorität der RTSJ gilt der Sicherheit. Daher steht die Vorhersagbarkeit der Ausführungszeiten an erster Stelle, selbst wenn dies zu Lasten der Performance geht.

- **No Syntactic Extension**

Es gibt keine neuen Schlüsselwörter und Konstrukte, stattdessen nutzt die RTSJ nur die bekannte Java Syntax.

- **Allow Variation in Implementation Decisions**

Die RTSJ darf dem Entwickler nicht vorschreiben, wie er seine Anwendung umzusetzen hat.

---

<sup>7</sup> RTSJ\_latest.pdf - Kapitel 1 Seite 1ff



## 2.2.3 Klassenbibliothek

Auf Basis dieser Schlüsselmerkmale wurden folgende Lösungen, für die Probleme, die einem Einsatz von Java in Echtzeitsystemen bisher im Wege standen, erarbeitet. Die neuen Klassen wurden dabei im package *javax.realtime.\** abgelegt.

### 2.2.3.1 Time

Da Echtzeitsysteme eine Genauigkeit im Bereich von Nanosekunden fordern, kann man nicht, wie es bisher üblich war, mit *System.currentTimeMillis()* arbeiten, sondern muss ein völlig neues System einführen.



Abbildung 2: Interner Aufbau einer Zeitangabe in der RTSJ

Dieses speichert, wie bisher, die Anzahl der Millisekunden in einem *long* Wert. Allerdings wurde für die Nanosekunden zusätzlich ein *int* Wert eingeführt. Das hat den Vorteil, dass es mit dem bisherigen vollständig kompatibel ist, die gewünschte Genauigkeit erreicht wird und man zudem über ein System verfügt, das auf Grund des Speicherbereichs, auch mit großen Zeitangaben umgehen kann.

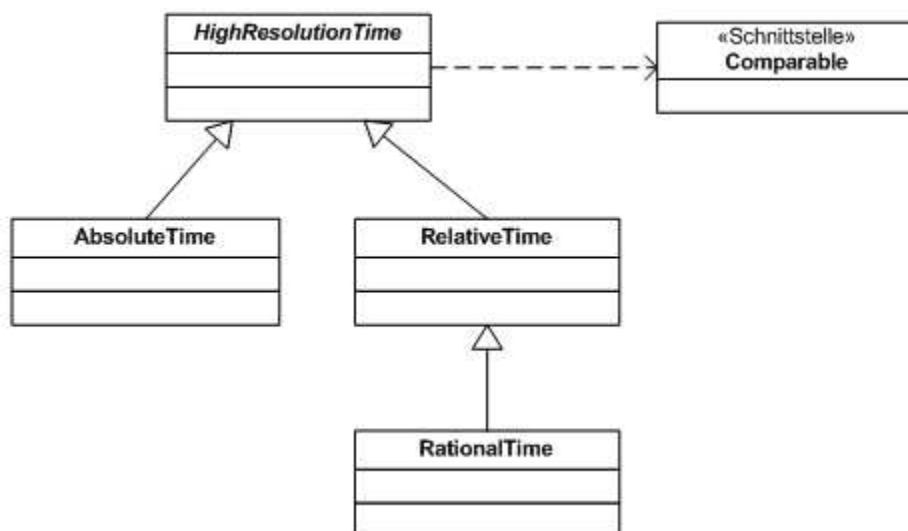


Abbildung 3: Klassenhierarchie für Zeitangaben in der RTSJ

## HighResolutionTime

Die abstrakte Klasse *HighResolutionTime* stellt dabei die Basis aller Zeitangaben der RTSJ dar. Sie bietet die Grundfunktionalität an, die allen Zeitangaben gemein ist. So ermöglicht sie es auf die einzelnen Werte für Milli- und Nanosekunden zuzugreifen und Zeiten komfortabel miteinander zu vergleichen, indem sie das Interface *Comparable* implementiert.

Wichtige Methoden:

*int compareTo(HighResolutionTime)* – Methode zum Vergleich zweier Zeiten

*long getMilliseconds()* – Liefert die Millisekunden

*int getNanoseconds()* – Liefert die Nanosekunden

*void set(long, int)* – Setzt die Anteile für Milli- und Nanosekunden auf die übergebenen Werte

## AbsoluteTime

Die Klasse *AbsoluteTime* repräsentiert Zeitpunkte, wie beispielsweise der 14.4.2005. Sie ist mit der Klasse *java.util.Date*, die bisher dazu herangezogen wurde, kompatibel.

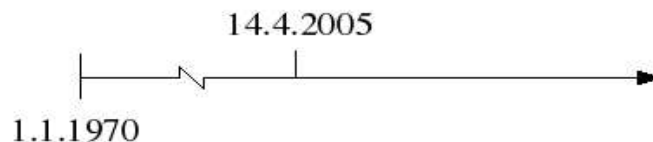


Abbildung 4: Absolute Zeitangabe

Der kleinste darstellbare Zeitpunkt ist, wie in der IT-Branche üblich, der 1.1.1970 um 00:00:00 Uhr GMT. Daraus folgt, unter Berücksichtigung der Speicherbereiche für Milli- und Nanosekunden, dass der größte darstellbare Zeitpunkt ca. 292 Millionen Jahre in der Zukunft liegt. Alle Zeitpunkte dazwischen lassen sich auf die Nanosekunde genau angeben.

Wichtige Methoden:

*Date getDate()* – Liefert das Datum dieses Zeitpunktes zurück

*AbsoluteTime add(RelativeTime)* – Addiert auf den gegebenen Zeitpunkt einen Zeitraum

*RelativeTime subtract(AbsoluteTime)* – Liefert die Differenz zwischen zwei Zeitpunkten

### RelativeTime

Zeiträume, beispielsweise 1 Tag oder 24 Stunden, werden durch die Klasse *RelativeTime* ausgedrückt. Sie werden in der RTSJ meist eingesetzt um das Zeitverhalten eines Objekts zu beschreiben.

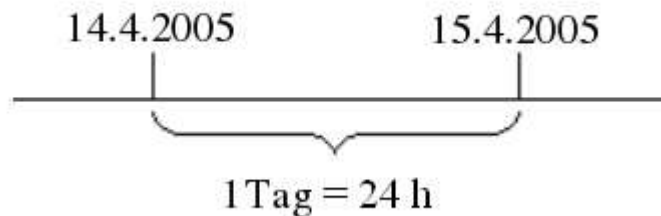


Abbildung 5: Relative Zeitangabe

Dabei beträgt das kleinste darstellbare Intervall 1 Nanosekunde, während das Größte ca. 292 Millionen Jahre lang ist.

Wichtige Methoden:

*RelativeTime add(RelativeTime)* – Addiert zwei Zeiträume miteinander

*RelativeTime subtract(RelativeTime)* – Subtrahiert einen Zeitraum von einem anderen

### RationalTime

Um periodische Ereignisse leichter auszudrücken zu können, besteht die Möglichkeit mit der Klasse *RationalTime* einen Zeitraum von einer Frequenz in gleich große Intervalle zu zerteilen.

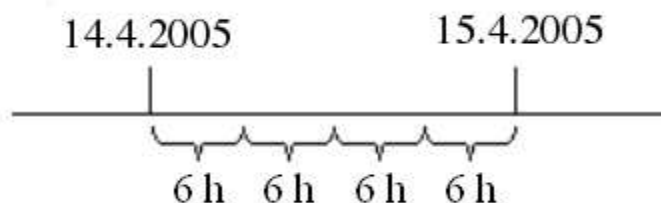


Abbildung 6: Relative Zeitangaben mit Hilfe von Frequenzen

In dem obigen Beispiel werden 24 Stunden in 4 gleichgroße Intervalle, zu jeweils 6 Stunden, aufgeteilt. Das besondere an diesen Intervallen ist, dass sie größtmögliche Genauigkeit bieten, indem die Bruchteile der Millisekunden auf die Nanosekunden übertragen werden.

Wichtige Methoden:

*int getFrequency()* – Liefert die Anzahl der Teile in die der Zeitraum unterteilt wird

*void setFrequency(int)* – Legt die Anzahl der Teile in die der Zeitraum unterteilt wird fest

*RelativeTime getInterarrivalTime()* – Liefert die Länge eines solchen Teils zurück

Hinweis:

In der kürzlich veröffentlichten Version 1.0.1 wurde diese Klasse als *deprecated* gekennzeichnet, da die Spezifikation keine geeignete Umsetzung des zu Grunde liegenden Konzepts erlaubt.

Schwächen:

- Automatische Normalisierung

Die strikte Trennung zwischen Milli- und Nanosekunden ist die Ursache für einige Effekte die eher kritisch anzusehen sind. So ist es in der JamaicaVM möglich mit Werten für Nanosekunden, die eigentlich außerhalb des darstellbaren Bereichs liegt, zu arbeiten.

z.B. 1000000 ns = 1 ms

Das hat zur Folge, dass sich beim Anlegen von Zeitangaben leicht Fehler einschleichen können. Deutlich wird dieses Problem, wenn man mit sehr großen Zeitangaben arbeitet und dabei auf die Werte *Long.MAX* und *Integer.MAX* zurückgreift. Hier wird erst zur Laufzeit im Zuge der Normalisierung eine Exception geworfen, da der Übertrag aus den Nanosekunden den Wertebereich der Millisekunden zum Überlaufen bringt.

- Keine Multiplikation / Division

Es besteht keine Möglichkeit Zeitangaben komfortabel zu vervielfachen oder aufzuteilen. Bisher konnte die Klasse *RationalTime* zu diesem Zweck missbraucht werden, indem man die Frequenz entsprechend änderte. Doch nachdem diese Klasse in der neuen Version 1.0.1 der RTSJ für *deprecated* erklärt wurde, fällt auch diese Option weg.

## Clock

Die Systemuhr wird durch die Klasse *Clock* repräsentiert. Damit kann zum einen der aktuelle Zeitpunkt und zum anderen die Genauigkeit dieser Uhr abgefragt werden. Letzteres ist insbesondere für zeitkritische Systeme von Interesse, da diese das kleinste Intervall angibt, mit dem sicher gearbeitet werden kann. Allerdings ist dieses in der Regel größer als eine Nanosekunde, so dass man zwar Zeitangaben, auf die Nanosekunde genau, machen kann, diese aber nicht eingehalten werden können, da die Uhr nicht die nötigen Voraussetzungen erfüllt.

Wichtige Methoden:

*Clock getRealtimeClock()* – Liefert eine Referenz auf die Systemuhr

*RelativeTime getResolution()* – Liefert die Genauigkeit der Systemuhr

*AbsoluteTime getTime()* – Liefert die aktuelle Uhrzeit der Systemuhr

Hinweis:

Da die Methode *Clock.getResolution()* beim Einsatz der JamaicaVM sowohl unter Linux als auch unter RTEMS auf verschiedenen Rechnern immer 20 ms zurück liefert, ist anzunehmen, dass dieser Wert fest im Profil gespeichert ist und nicht dynamisch ermittelt wird.

```
/*
Zunächst wird die aktuelle Uhrzeit in „now“ und ein Intervall der Länge 60 Sekunden in „aMinute“ abgelegt.
Letzteres wird auf „now“ aufgeschlagen, um den Zeitpunkt „inAMinute“ , welcher 1 Minute in der Zukunft
liegt, zu erhalten.
*/

...
AbsoluteTime now = null;
RelativeTime aMinute = null;
AbsoluteTime inAMinute = null;

//Create current AbsoluteTime
now = Clock.getRealtimeClock().getTime();

//Create RelativeTime
aMinute = new RelativeTime(60*1000,0);

//Create AbsoluteTime
inAMinute = now.add(aMinute);
...

/*
Beispielprogramme: RTSJ_MoreTimes, RTSJ_Times, ...
*/
```

Hinweis:

Auf der CD sind unter *Sourcen* alle Anwendungen. Dort ist auch eine Übersicht zu finden, die jedem Beispiel die eingesetzten Klassen der RTSJ zuordnet.

### 2.2.3.2 MemoryManagement

Bisher wurde der gesamte Speicher von Java automatisch verwaltet. In diesem Zusammenhang spielt die GC eine entscheidende Rolle. Sie sorgt dafür, dass unreferenzierte Objekte entfernt werden und deren Speicher wieder freigegeben wird. Außerdem verhindert sie, dass der Speicher fragmentiert. Das hat den Vorteil, dass der Entwickler sich auf die Anwendung konzentrieren kann und keine Fehler in der Speicherverwaltung macht. In Echtzeitsystemen stößt man mit diesem Konzept auf folgende Probleme. Zum einen kommt in diesen eine Vielzahl unterschiedlicher und spezieller Arten von Speicher, angefangen bei schnellem statischen bis hin zu DMA, zum Einsatz. Diese müssen, was bisher nicht möglich war, gezielt angesteuert werden, um von ihren teilweise einzigartigen Eigenschaften Gebrauch machen zu können. Zum anderen können damit keine Ausführungszeiten garantiert werden, da die GC weder unterbrochen noch gesteuert werden kann und somit die gesamte VM für unbestimmte Zeit lahmlegt. Um diese beiden Probleme zu lösen, teilt die RTSJ den Speicher in verschiedene Bereiche mit unterschiedlichen Eigenschaften ein.

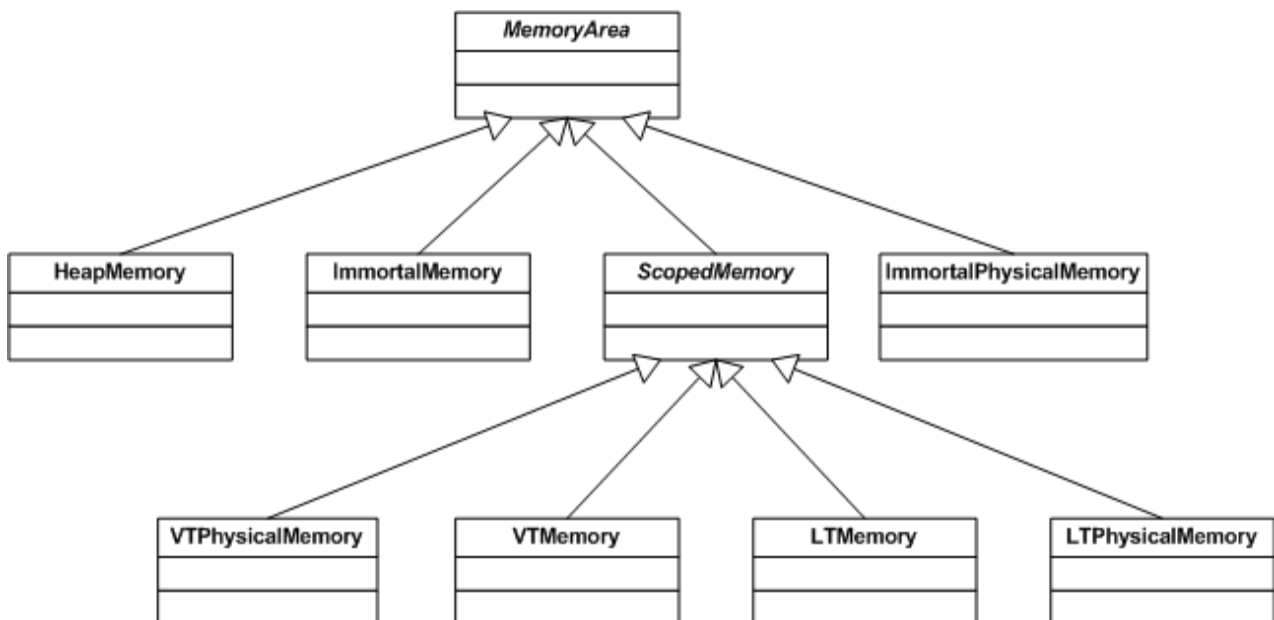


Abbildung 7: Klassenhierarchie MemoryArea

## MemoryArea

*MemoryArea* ist die abstrakte Basisklasse aller Speicherbereiche. Sie bietet die Grundfunktionalität an, die diese auszeichnet. Dazu zählen das Anlegen neuer Objekte, das Abfragen der Basisdaten und das Betreten des Speicherbereichs.

Wichtige Methoden:

*void enter()* – Betreten eines Speicherbereichs

*long memoryConsumed()* – Liefert den belegten Speicher in Byte

*long memoryRemaining()* – Liefert den freien Speicher in Byte

*java.lang.Object newArray(java.lang.Class, int)* – Legt eine neues Array im Speicher an

*java.lang.Object newInstance(java.lang.Class)* – Legt ein Objekt im Speicher an

*long size()* – Gibt die Größe des Speicherbereichs zurück

## HeapMemory

Das *HeapMemory* repräsentiert den Speicherbereich, der, wie bisher, automatisch verwaltet und von der GC aufgeräumt wird. Aus diesem Grund kann es auch nicht für harte Echtzeitanforderungen herangezogen werden. Es handelt sich dabei um ein Singleton mit statischer Größe.

Wichtige Methoden:

*HeapMemory instance()* – Liefert eine Referenz auf den Heap

```
/*  
Ein Objekt vom Typ Object wird im Heap Memory, auf die von Java vertraute Art, angelegt.  
*/  
  
Object o = new Object();  
  
/*  
Beispielprogramme: RTSJ_MemoryAccessExample, ...  
*/
```

## ImmortalMemory

Beim *ImmortalMemory* handelt es sich um einen Speicherbereich, der, wie der Name schon anklingen lässt, während der gesamten Laufzeit der Anwendung erhalten bleibt, und nicht von der GC betroffen ist. Daher kann er auch für Anwendungen, die harten Echtzeitanforderungen unterliegen, genutzt werden. Das Besondere am *ImmortalMemory* ist, dass dort abgelegte Objekte aus allen Speicherbereichen heraus angesprochen werden können. Wie auch schon beim *HeapMemory* handelt es sich dabei um eine Singleton mit fester Größe.

Wichtige Methoden:

*ImmutableMemory instance()* – Liefert eine Referenz auf den *ImmutableMemory*

*Object newInstance(Class)* – Legt ein neues Objekt an

*Object newInstance(Constructor, Object[])* – Erzeugt ein Objekt mit den gegebenen Parametern

*Constructor Class.getConstructor(Class[])* – Liefert den Konstruktor der Klasse mit der gegebenen Signatur

```
/*
Ein Objekt vom Typ im PriorityParameters wird im ImmutableMemory angelegt.
Dazu wird...
- eine Referenz auf das ImmutableMemory besorgt.
- eine Referenz mit der man auf das Objekt zugreifen will bereitstellt.
- eine Referenz auf den entsprechenden Konstruktor der Klasse beschafft.
- das Objekt mit Hilfe des Konstruktors und der Parameterliste erzeugt.
*/

...
//MemoryArea of the NoHeapRealtimeThread
ImmutableMemory immortal = ImmutableMemory.instance();
...
PriorityParameters nhrtThreadPriority = null;
...
try
{
    Object[] parameters = new Object[1];
    parameters[0] = new Integer(PriorityScheduler.getMaxPriority(nhrtThread));

    Class[] parameterTypes = new Class[1];
    parameterTypes[0] = int.class;

    Class classType = PriorityParameters.class;

    Constructor constructor = classType.getConstructor(parameterTypes);

    nhrtThreadPriority =(PriorityParameters)immortal.newInstance
    (
        constructor,
        parameters
    );
}
...

/*
Beispielprogramme: RTSJ_ThreadComparison, RTSJ_MemoryAccess, ...
*/
```



### Schwächen:

- **Neue Objekte**  
Das Anlegen neuer Objekte im *ImmortalMemory* hat nichts mehr gemein mit jener Vorgehensweise, die von herkömmlichem Java bekannt ist. Es kommt dem Entwickler sogar sehr umständlich und aufwendig vor. Insbesondere wenn er mehrere Parameter übergeben möchte. Um dies zu umgehen, versucht er zunächst einen default Konstruktor zu nutzen und erst anschließend die Attribute zu setzen. Damit nimmt er aber das Risiko in Kauf, dass bis dahin falsche Werte aus dem Objekt gelesen werden.
- **Memory leaks**  
Die Gefahr beim Einsatz von *ImmortalMemory* besteht darin, dass es keine Möglichkeit gibt ein Objekt zu entfernen. Dies kann bei falschem Umgang zu einem Effekt führen, der als memory leaks in C/C++ bekannt ist.

### ImmortalPhysicalMemory

Die Klasse *ImmortalPhysicalMemory* verhält sich in weiten Strecken wie der *ImmortalMemory*. Daher sollte sie immer dann eingesetzt werden, wenn ein Objekt im *ImmortalMemory* gespeichert werden soll, aber der Speicher über spezielle Eigenschaften verfügen muss. Es handelt sich dabei allerdings nicht um ein Singleton. Das bedeutet, dass man Speicherbereiche flexibler Größe reservieren muss, wobei man die gewünschten Eigenschaften mit Hilfe des *PhysicalMemoryManager* angibt.

### Wichtige Methoden:

keine – Attribute werden im Konstruktor übergeben

### Schwächen:

- **Eigenschaften**  
Zur Definition der Eigenschaften des Speichers wird eine Instanz der Klasse *Object* herangezogen. Damit ist nicht klar wie diese eigentlich aussehen müssen. Im *PhysicalMemoryManager* sind diese als *Strings* realisiert. Doch auf diese Weise lassen sich mehrere Eigenschaften nur schwer kombinieren, falls man nicht jede mögliche Kombination separat handhabt.
- **Fehlende Präzision**  
Insgesamt fehlt es diesem Lösungsvorschlag noch etwas an Präzision. So wurden nur einige wenige mögliche Speicherarten definiert. Desweiteren ist die Bedeutung der einzelnen Klassen für den Entwickler nicht zweifelsfrei geklärt. Es scheint zwar, dass der *PhysicalMemoryManager* die gesamte Organisation und Verteilung des Speichers übernimmt, während die VM individuelle Klassen, die das Interface *PhysicalMemoryTypeFilter* implementieren, zur Identifikation und Kategorisierung der Speicherbereiche mit besonderen Eigenschaften, nutzen kann.

### Hinweis:

Alle Klassen die auf Speicherbereiche mit besonderen Eigenschaften zugreifen wollen, werden bis dato nicht von der JamaicaVM unterstützt.

## ScopedMemory

Die abstrakte Klasse *ScopedMemory* ist die Basis aller Speicherbereiche mit begrenzter Lebensdauer. Daher kann man diese auch nur mit Objekten, die das Interface *Runnable* implementieren, betreten. Sie bleiben nur solange erhalten, bis das letzte *Scheduleable*, das diesen Speicherbereich genutzt hat, die *enter()*-Methode verlässt oder seine *run()*-Methode beendet. Danach wird er automatisch wieder freigegeben. Das bedeutet auch, dass alle Objekte, die innerhalb der *run()*-Methoden angelegt wurden, gelöscht werden.

Diese Verhalten ist der Grund für die folgende Restriktionen:

1. Referenzen auf Objekte, die im *ScopedMemory* angelegt wurden, dürfen nicht in Objekten, die sich im *HeapMemory* befinden, gespeichert werden.
2. Referenzen auf Objekte, die im *ScopedMemory* angelegt wurden, dürfen nicht in Objekten, die sich im *ImmortalMemory* befinden, gespeichert werden.
3. Referenzen auf Objekte, die im *ScopedMemory* angelegt wurden, dürfen nur in Objekten, die sich im gleichen *ScopedMemory* oder in einem *ScopedMemory*, das diesem untergeordnet ist, gespeichert werden. Ein Schachteln von *ScopedMemory* Bereichen kommt dann zustande, wenn eine *run()*-Methode, die innerhalb eines *ScopedMemory* ausgeführt wird, die *enter()*-Methode eines anderen *ScopedMemory* aufruft.

Diese Speicherbereiche sind ebenfalls nicht im Fokus der GC und können somit von alle Arten von Threads genutzt werden. Dazu müssen diese lediglich die *enter()*-Methode dieses Speicherbereichs aufrufen oder man muss diesen den Speicherbereich im Konstruktor, als deren *MemoryArea*, übergeben.

Die Subklassen des *ScopedMemory* erweitern nicht dessen Funktionalität, sondern machen Angaben darüber, ob dieser Speicherbereich noch über andere Eigenschaften verfügt (siehe *ImmortalPhysicalMemory*) und welches Zeitverhalten er bei der Allokierung zeigt. Linear bedeutet in diesem Fall, dass die Allokierung innerhalb bestimmter zeitlicher Grenzen, die abhängig von der Größe ist, erfolgt, während eine variable Allokierungszeit keine Aussagen darüber zulässt.

	Speicher mit besonderen Eigenschaften	Speicher ohne besondere Eigenschaften
Variable Allokierungszeit	VTPhysicalMemory	VTMemory
Lineare Allokierungszeit	LTPhysicalMemory	LTMemory

Tabelle 1: Kategorisierung der Subklassen von *ScopedMemory*

Wichtige Methoden:

*void enter()* – Die *run()*-Methode des zuvor übergebene *Runnable* wird im *ScopedMemory* ausgeführt

*void enter(Runnable)* – Die *run()*-Methode dieses *Runnable* wird im *ScopedMemory* ausgeführt

*void join(HighResolutionTime)* – Wartet max. das übergebene Intervall auf das Ende des *Runnable*

```
/*  
Ein Runnable wird im ScopedMemory ausgeführt.  
Dazu muss man...  
- sich eine Instanz eines der Subklassen von ScopedMemory besorgen  
- das ScopedMemory mit einem nebenläufigen Prozess, beispielsweise einem Thread, betreten  
  
Alle Objekte die innerhalb der run()-Methode in der Java typischen Syntax angelegt werden befinden sich  
nun nicht auf dem Heap sondern in diesem ScopedMemory.  
*/  
  
...  
//create ScopedMemory  
LTMemory scoped = new LTMemory(1024,1024);  
  
//Execute the given Runnable in this ScopedMemory  
scoped.enter(new scopeExecute());  
  
...  
/*  
Beispielprogramme: RTSJ_ScopedMemory, ...  
*/
```

Schwächen:

- Einsatz  
Der Einsatz des *ScopedMemory* erfolgt auf bisher unbekannte Art und Weise und ist daher gewöhnungsbedürftig.
- Verwaltungsaufwand  
*ScopedMemory* ist ein ausgeklügeltes und sehr mächtiges Konzept, doch sein Einsatz ist mit sehr viel Verwaltungsaufwand verbunden. Insbesondere wenn man beginnt mehrere Instanzen zu Schachteln. Das hat zur Folge, dass die Performance darunter leidet.

## SizeEstimator

Diese Einteilung des Speichers ermöglicht es dem Entwickler für jeden Anwendungszweck den geeigneten auszuwählen. Dabei muss man, vor allem bei allen Speicherbereichen, die nicht als Singleton ausgelegt sind, die Größe des Bereichs, den man reservieren möchte, mit angeben. Aus diesem Grund wurde die Klasse *SizeEstimator* eingeführt. Sie unterstützt den Entwickler bei der Ermittlung des Speicherbedarfs, indem sie die Größe der einzelnen Objekte bestimmt, mit deren Häufigkeit multipliziert und die Ergebnisse der einzelnen Objekte aufsummiert.

Wichtige Methoden:

*void reserve(java.lang.Class, int)* – Nimmt die gegebene Anzahl an Objekten in die Kalkulation mit auf  
*long getEstimate()* – Liefert den, für die betrachteten Objekte, benötigten Speicher in Byte

```
/*  
Eine Instanz eines SizeEstimator wird erzeugt.  
Daraufhin wird der Speicherbedarf eines RealtimeThread ermittelt und ausgegeben.  
*/  
  
...  
//Create SizeEstimator  
SizeEstimator sizeEstimator = new SizeEstimator();  
...  
//Reserve a RealtimeThread  
sizeEstimator.reserve(RealtimeThread.class, 1);  
...  
System.out.println  
(  
    "Estimated size after adding a Thread: "  
    +sizeEstimator.getEstimate()  
    +" bytes"  
);  
...  
/*  
Beispielprogramm: RTSJ_SizeEstimator  
*/
```

### 2.2.3.3 Scheduling

Wie bereits angesprochen ist der fehlende Determinismus in der zeitlichen Ausführung eines der größten Probleme, die den Einsatz von herkömmlichem Java in Echtzeitsystemen verhindert. Neben der GC spielt in diesem Zusammenhang das Scheduling, das ist die Art und Weise, wie die Threads aktiviert werden, eine entscheidende Rolle. An dieser Stelle wird auch besonders deutlich, dass Java nicht für die Programmierung von Echtzeitsystemen konzipiert wurde, denn es verhält sich an diesem Punkt ähnlich wie ein normales OS und aktiviert die Threads nicht, wie es in Echtzeitbetriebssystemen üblich ist, streng nach ihrer Priorität.

#### Scheduler

Aus diesem Grund führt die RTSJ Scheduler ein. Diese leiten sich von der abstrakte Klasse *Scheduler* ab. Sie stellt die Grundfunktionalität, die für eine solche Aufgabe notwendig ist, bereit und sorgt dafür, dass, neben dem *PriorityScheduler*, auch noch weitere Schedule-Algorithmen umgesetzt werden können.

Wichtige Methoden:

*boolean addToFeasibility(Scheduleable)* – Nimmt das *Scheduleable* in die Gültigkeitsprüfung auf

*boolean removeFromFeasibility(Scheduleable)* – Entfernt das *Scheduleable* aus der Gültigkeitsprüfung

*void fireScheduleable(Scheduleable)* – Aktiviert das *Scheduleable*

#### PriorityScheduler

Der gängigste Algorithmus wurde in der Klasse *PriorityScheduler*, der in jeder Implementierung der RTSJ vorhanden sein muss, umgesetzt. Dieser zeichnet sich durch sein *fixed priority preemptive scheduling* aus. Das bedeutet, dass den Threads feste Prioritäten zugewiesen werden, wobei er Threads mit höherer bevorzugt und dafür sorgt, dass jene mit einer niedrigeren Priorität die CPU entzogen bekommen, falls einer mit höherer darauf wartet.

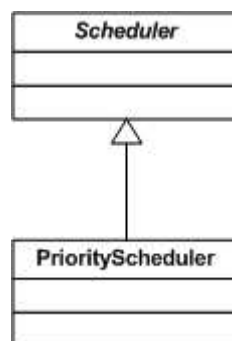


Abbildung 8:  
Klassenhierarchie  
Scheduler

Aus Sicherheitsgründen ist dieser als Singleton realisiert. Damit wird verhindert, dass mehrere Scheduler parallel arbeiten. Dies wäre nämlich eine höchst kritische Situation, da man dessen Verhalten nicht mehr vorhersagen könnte.

Wichtige Methoden:

*PriorityScheduler instance()* – Liefert den *PriorityScheduler* dieser VM zurück

*int getMaxPriority(Thread)* – Liefert die höchste Priorität, die diesem Thread zugewiesen werden kann

*int getMinPriority(Thread)* – Liefert die niedrigste Priorität, die diesem Thread zugewiesen werden kann

Schwächen:

- Definition der Prioritätsstufen

Die Anzahl der unterschiedlichen Prioritäten ist nicht fest vorgeschrieben. Laut RTSJ müssen nur mindestens 38 unterschiedliche Stufen vorliegen. Dies stellt eine höchst kritische Situation dar, denn eine Anwendung, die in einer VM fehlerfrei läuft, kann auf einer anderen plötzlich ein unerwartetes Verhalten zeigen.

```
...
RealtimeThread high = new RealtimeThread();
RealtimeThread low = new RealtimeThread();

//set Priority
high.setSchedulingParameters
(
    new PriorityParameters(PriorityScheduler.getMaxPriority(high) - 15)
);

//setPriority
low.setSchedulingParameters
(
    new PriorityParameters(PriorityScheduler.getMinPriority(low) + 15)
);
...
```

Der Ausschnitt beschreibt 2 RealtimeThreads, von denen der eine – *high* – fast mit höchsten und der andere – *low* – fast mit niedrigster Priorität arbeiten soll. Auf *JTime*, das 256 unterschiedliche Prioritäten kennt, arbeitet dieses Programm auch wie erwartet. Die Priorität von *high* wäre dabei 241, während die von *low* 26 betragen würde. Führt man es allerdings in einer VM, die nur über 38 Prioritätsstufen verfügt aus, dann kommt es zu einem interessanten Effekt. *high* hat hier eine Priorität von 23 während *low* seine behält. Damit wäre *low* nun wichtiger als *high*, was aber so vom Entwickler nicht beabsichtigt war.

## Scheduleable

Damit der Scheduler die nebenläufigen Prozesse gegeneinander abwägen kann, wurde das Interface *Scheduleable* eingeführt. Es sorgt dafür, dass die Objekte, die es implementieren, charakterisiert werden können. Dies erfolgt mit Hilfe der folgenden 4 Arten von Parametern.

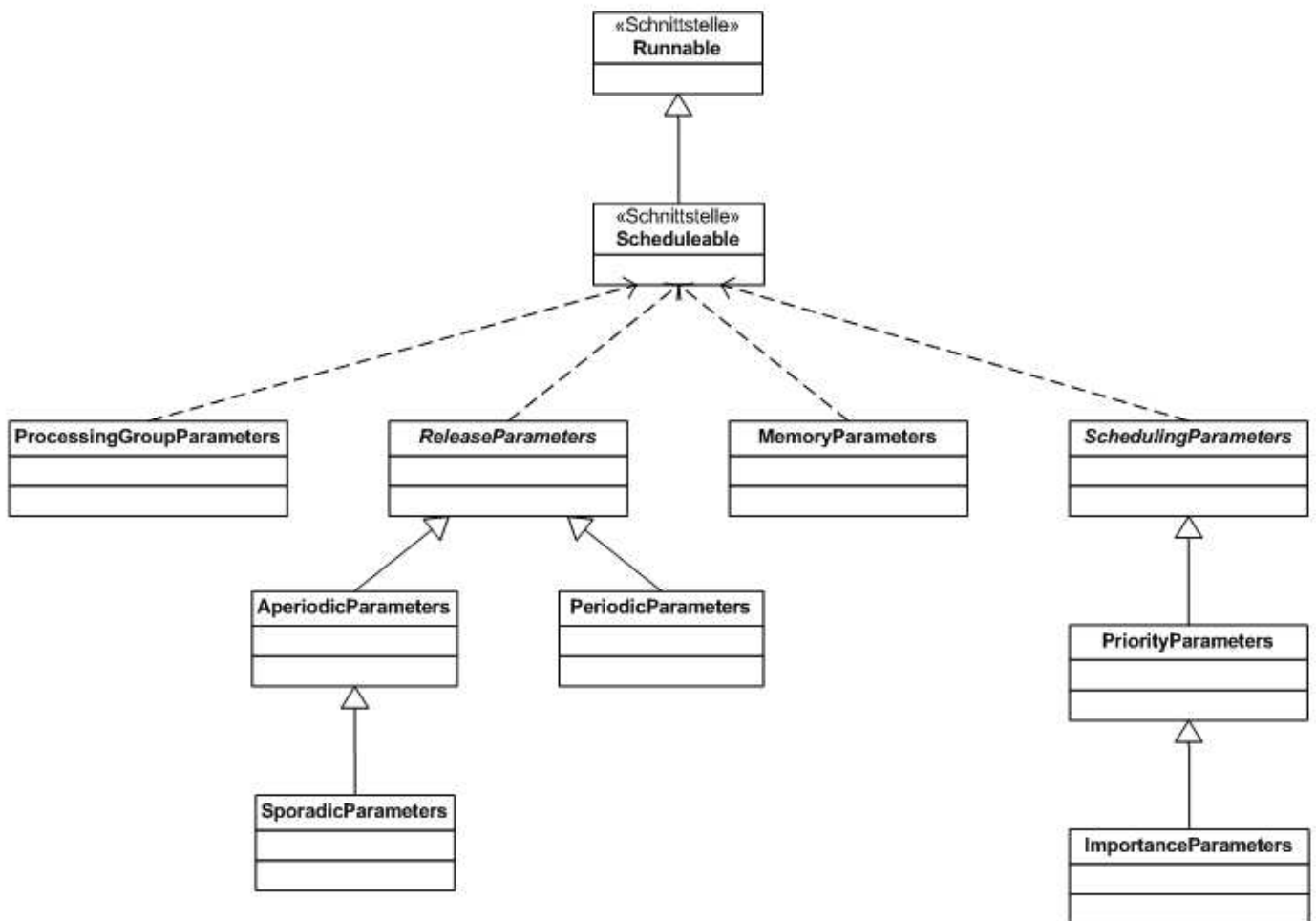


Abbildung 9: Klassenhierarchie *Scheduleable*

### ReleaseParameters

Diese abstrakte Klasse von Parametern beschreiben das Zeitverhalten eines *Scheduleable*. Dazu zählt insbesondere die Zeit in der es, nach seiner Aktivierung, abgearbeitet werden muss (*deadline*) und die Zeit, die es dabei maximal die CPU belegen darf (*cost*). Letztere muss aber nicht, wie es in den Schaubildern dargestellt ist, zusammenhängend sein.

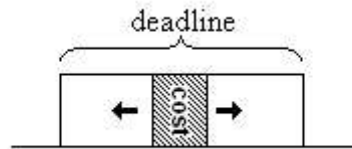


Abbildung 10: ReleaseParameters

Darüber hinaus besteht die Möglichkeit Maßnahmen vorzubereiten, die greifen, falls eines der beiden Intervalle überschritten wird. Die entsprechenden *AsyncEventHandler* werden entweder im Konstruktor übergeben oder nachträglich zugewiesen.

- Miss  
Der so genannte MissHandler wird aktiviert, falls das *Scheduleable* nicht innerhalb des vorgesehenen Intervalls abgearbeitet werden kann. (siehe Beispielprogramm RTSJ\_Miss)

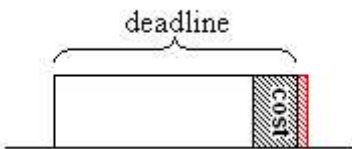


Abbildung 11: Miss Situation

- Overrun  
Der so genannte OverrunHandler wird aktiviert, falls das *Scheduleable* länger die CPU nutzt als das vorgesehen wurde. (siehe Beispielprogramm RTSJ\_Overrun)

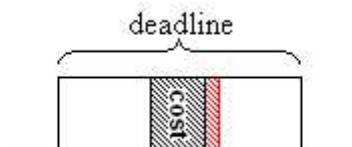


Abbildung 12: Overrun Situation

#### Hinweis:

In minimalen Systemen kann *cost* unberücksichtigt bleiben, während die *deadline* in jedem Fall beachtet werden muss.



Wichtige Methoden:

keine – Die Attribute werden in der Regel in den Konstruktoren der Subklassen gesetzt

### AperiodicParameters

*AperiodicParameters* kennzeichnen jene *Scheduleable* die zufällig auftreten. Daher können auch nur Aussagen über die Rechenzeit (*cost*) und das Intervall in dem sie abgearbeitet werden müssen (*deadline*) gemacht werden.

Da auch ein Aufruf erfolgen kann, während noch an einem vorhergehenden gearbeitet wird, wurde eine Warteschlange eingerichtet, die diese unverarbeiteten Ereignisse aufnimmt. Sollte diese überlaufen, dann wird nach der Strategie, die aus den folgenden 4 Alternativen ausgewählt wurde, verfahren.

EXCEPT = eine *ArrivalTimeOverflowException* wird geworfen  
SAVE = die Warteschlange wird verlängert (default)  
IGNORE = der Aufruf wird ignoriert  
REPLACE = der Neue ersetzt einen bestehenden Eintrag

Wichtige Methoden:

*boolean setIfFeasible(RelativeTime, RelativeTime)* – Setzt die beiden Parameter *cost* und *deadline*

### SporadicParameters

Um Aufgaben, die wiederholt, aber in unregelmäßigen Abständen, auftreten, effizient abbilden zu können, wurde die Klasse *SporadicParameters* eingeführt.

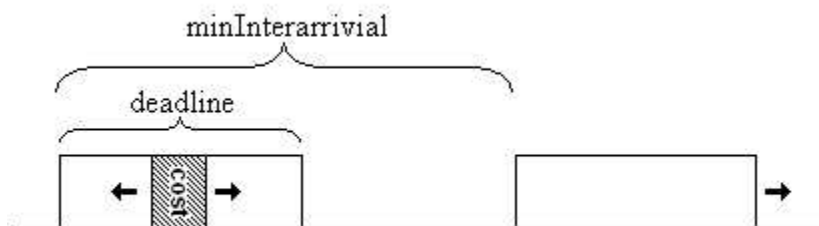


Abbildung 13: *SporadicParameters*

Neben dem Intervall, indem das *Scheduleable* abgearbeitet werden muss (*deadline*) und der Rechenzeit die es dafür maximal benötigen darf (*cost*), wird noch eine Aussage darüber gemacht, in welchem Intervall kein zweiter Aufruf erfolgt (*minInterarrival*).

## Evaluation der Realtime Specification for Java anhand einer Robotersteuerung

Sollte dennoch innerhalb dieses Intervalls ein Aufruf erfolgen, dann wird nach der Strategie, die aus den folgenden 4 Alternativen ausgewählt wurde, verfahren.

EXCEPT = eine *MITViolationException* wird geworfen  
SAVE = *minInterarrival* wird gar nicht beachtet (default)  
IGNORE = der Aufruf wird ignoriert  
REPLACE = der Neue ersetzt den vorhergehenden Aufruf

Auch hier ist eine Warteschlange notwendig. Diese entspricht jener, die in den *AperiodicParameters* Verwendung findet.

Wichtige Methoden:

*boolean setIfFeasible(RelativeTime, RelativeTime, RelativeTime)* - Setzt alle drei Intervalle

### PeriodicParameters

PeriodicParameters kennzeichnen jene Scheduleable, die in regelmäßigen Abständen auftreten.

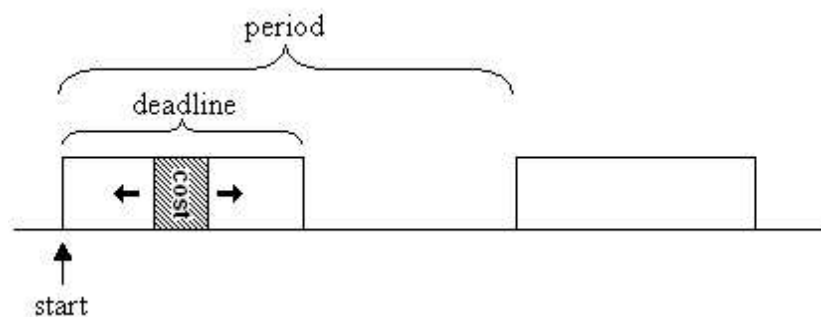


Abbildung 14: PeriodicParameters

Ausgehend von einem Starttermin (*start*) wird es in bestimmten Zeitabständen (*period*) aktiviert. Auch hier muss dessen Code innerhalb eines bestimmten Intervalls (*deadline*) abgearbeitet werden, wobei es ebenfalls nur eine bestimmte Zeit lang die CPU nutzen darf (*cost*). In diesem Zusammenhang ist die Methode *waitForNextPeriod()* der Klasse *RealtimeThread* von entscheidender Bedeutung. Diese pausiert nämlich die Ausführung des Codes so lange, bis eine neue Periode beginnt, so dass man sehr komfortabel mit einer Schleife arbeiten kann.

Wichtige Methoden:

*boolean setIfFeasible(RelativeTime, RelativeTime, RelativeTime)* – Setzt alle drei Intervalle

*boolean RealtimeThread.waitForNextPeriod()* – Verzögert die Ausführung bis zum Beginn einer neuen Periode

```
/*
Zunächst wird eine Instanz von SporadicParameter angelegt. Diese zeichnet sich dadurch aus, dass
zwischen zwei Aufrufen mindestens 200 ms liegen. Des weiteren darf das entsprechende Scheduleable nur
1 ms lang die CPU belegen und muss innerhalb von 5 ms abgearbeitet werden. Für das Überschreiten
dieser beiden Zeiten ist keine besondere Reaktion vorgesehen.
Abschließend wird die Strategie, mit der auf einen zweiten Aufruf innerhalb von 200 ms reagiert wird,
festgelegt.
*/
SporadicParameters sporadic = new SporadicParameters
(
    new RelativeTime(200,0), //minInterarrival
    new RelativeTime(1,0), //cost
    new RelativeTime(5,0), //deadline
    null, //overrunHandler
    null //missHandler
);

//which throws an exception if minInterval is undercut
sporadic.setMitViolationBehavior(SporadicParameters.mitViolationExcept);

/*
Beispielprogramme: RTSJ_SporadicParameter, RTSJ_PeriodicRealtimeThread, ...
*/
```

Schwächen:

- Minimale Implementierung  
Es gibt keinen Parameter der angibt, ob bei dieser Implementierung der Parameter *cost* berücksichtigt wird. Dies könnte im Zusammenhang mit der Portabilität für Schwierigkeiten sorgen.
- *Null* für *cost*  
Falls *null* als Wert für *cost* angegeben wurde, entspricht dieser der *RelativeTime(0,0)*. Damit dürfte dieses *Scheduleable* die CPU gar nicht nutzen. Dieses Verhalten war, bei Angabe dieses Wertes, sicherlich nicht beabsichtigt. Es wäre daher sinnvoller den Wert von *deadline* zu übernehmen oder die Überwachung von *cost* abzuschalten.

### ProcessingGroupParameters

Diese Klasse von Parametern dient im wesentlichen zur Verwaltung komplexer Systeme. Hiermit kann einer Gruppe von *Scheduleable* ein Zeitverhalten zugeordnet werden. Bei der Aufnahme eines solchen Objekts in die Gruppe wird automatisch geprüft, ob es, auf Basis der *ReleaseParameters* der einzelnen Mitglieder, möglich ist, die Gruppe in dem gestecktem Rahmen auszuführen. Dabei ist deren Aufbau, wie man unschwer erkennen kann, stark an dem der *PeriodicParameters* angelehnt.

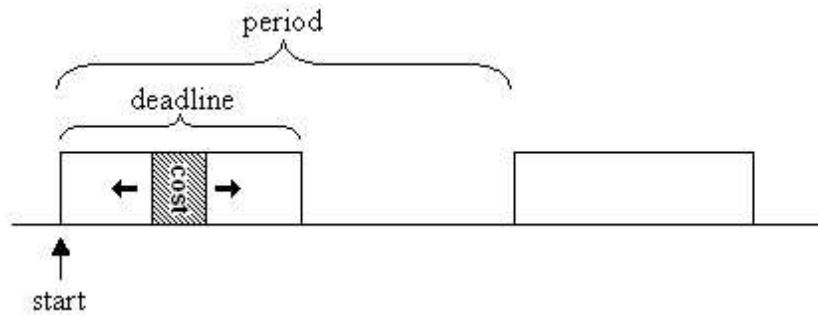


Abbildung 15: *ProcessingGroupParameters*

Allerdings werden hier, ausgehend von einem Starttermin (*start*) in gewissen Zeitabständen (*period*), allen Instanzen vom Typ *Scheduleable* bestimmte Zeitfenster für Rechen- und Arbeitszeit (*cost* und *deadline*) zur Verfügung gestellt.

Wichtige Methoden:

*boolean setIfFeasible(RelativeTime, RelativeTime, RelativeTime)* – Setzt alle drei Intervalle

*boolean Scheduleable.setProcessingGroupParametersIfFeasible(ProcessingGroupParameters)* – Nimmt ein *Scheduleable* in die Gruppe auf

```

/*
Eine Instanz von ProcessingGroupParameters wird mit den gegebenen Zeiten angelegt und anschließend
wir ein Thread in die Gruppe aufgenommen
*/
public final static int GROUP_PERIODE_MILLIS = 100;
public final static int GROUP_COST_MILLIS = 50;
public final static int GROUP_DEADLINE_MILLIS = 50;
...
//to ensure all threads would start at the same time
AbsoluteTime now = Clock.getRealtimeClock().getTime();

//Create ProcessingGroupParameters
ProcessingGroupParameters group = new ProcessingGroupParameters
(
    now.add(1000,0), //start
    new RelativeTime(GROUP_PERIODE_MILLIS,0), //period
    new RelativeTime(GROUP_COST_MILLIS,0), //cost
    new RelativeTime(GROUP_DEADLINE_MILLIS,0), //deadline
    null, //overrunHandler
    null //missHandler
);
...
RealtimeThread rtThread_1 = new RealtimeThread();
...
//Set PorcessingGroupParameters
if(!rtThread_1.setProcessingGroupParametersIfFeasible(group))
{
    ...
}
...
/*
Beispielprogramm: RTSJ_ProcessingGroupParameters
*/

```

Schwächen:

- Fehlermeldungen  
Die Gründe, weshalb ein Thread nicht in diese Gruppe aufgenommen werden konnte, werden nicht detailliert angegeben. Des weiteren ist es nicht möglich, die verbleibenden Ressourcen abzufragen.
- Überwachung von Gruppen  
Es ist nicht möglich mehrere Gruppen gemeinsam zu überwachen. Könnten diese Parameter geschachtelt werden, dann würde dies die Flexibilität und die Eignung für große Anwendungen deutlich erhöhen.

- Mangelnde Flexibilität

Die radikale und geradlinige Vorgehensweise im Falle einer Überschreitung von *cost* ist in diesem Zusammenhang ein Zeichen für mangelnde Flexibilität. Das Einführen von alternativen Strategien würde dem Entwickler mehr Komfort bieten.

### MemoryParameters

Diese Klasse beschreibt den Speicherbedarf eines *Scheduleable*, indem Aussagen über die maximale Anzahl an Bytes, die dieses im aktuellen *MemoryArea* und im *ImmortalMemory* belegen darf, gemacht werden. Des weiteren können sie vom Scheduler für zeitlichen Steuerung des GC benutzt werden, da man zusätzlich die gewünschte Allokierungsrate im *HeapMemory*, in Byte/s, angeben kann.

Wichtige Methoden:

keine – Die Werte werden in der Regel beim Aufruf des Konstruktors übergeben

```
/*
Eine Instanz vom Typ MemoryParameter wird angelegt. Es beschreibt den Speicherbedarf eines
Scheduleable, das nur das aktuelle MemoryArea nutzen darf.
*/
...
//MemoryParameters - describes the memory requirements
MemoryParameters memory = new MemoryParameters
(
    MemoryParameters.NO_MAX, //MemoryArea
    0 //ImmortalMemory
);
...
/*
Beispielprogramm: RTSJ_PeriodicRealtimeThread
*/
```

### SchedulingParameters

Die Subklassen der abstrakten Klasse *SchedulingParameters* dienen in erster Linie dem Scheduler. Mit deren Hilfe ist es ihm möglich, die Threads nach ihrer Bedeutung zu ordnen. Da dies auf die unterschiedlichste Art und Weise erfolgen kann, enthält sie keinerlei Vorgaben.

Wichtige Methoden:

keine – Diese Klasse dient nur als abstrakte Basis

### PriorityParameters

Die *PriorityParameters* stellen die bekannten Prioritätsstufen dar. Sie sollten für Scheduler verwendet werden, die, wie beispielsweise der *PriorityScheduler*, *int* Werte heranziehen, um die Threads gegeneinander abzuwägen.



Abbildung 16: *PriorityParameters*

Wichtige Methoden:

*int getPriority()* – Liefert die Priorität der *PriorityParameters*

*int PriorityScheduler.getMaxPriority(Thread)* – Liefert die höchste Priorität, die für diesen Thread möglich ist

*int PriorityScheduler.getMinPriority(Thread)* – Liefert die niedrigste Priorität, die für diesen Thread möglich ist

### ImportanceParameters

*ImportanceParameters* erweitern die herkömmlichen *PriorityParameters* um einen zweiten *int* Wert. Mit diesem ist es nun auch möglich Threads gleicher Priorität unterschiedlich zu gewichten.

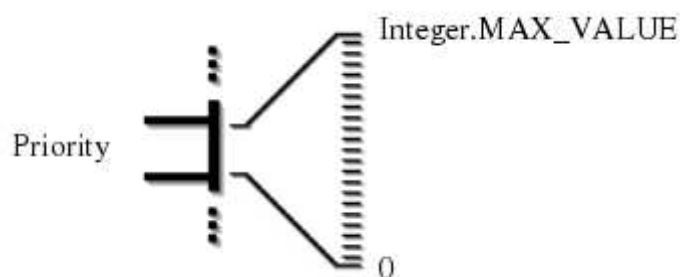


Abbildung 17: *ImportanceParameters*

Damit muss der Scheduler nicht länger nach dem FIFO Prinzip vorgehen, wenn er einen Thread einer Prioritätsstufe aktiviert, in deren Warteschlange mehrere warten.

#### Hinweis:

Der *PriorityScheduler* nutzt nur die Priorität.

Wichtige Methoden:

*int getImportance()* – Liefert die Gewichtung innerhalb der zugehörigen Prioritätsstufe

```
/*  
Es wird eine Instanz von PriorityParameters erzeugt. Dabei wird auf die maximale Priorität die der  
PriorityScheduler für diesen Thread anbietet zurückgegriffen.  
*/  
//Thread  
RealtimeThread rtThread = null;  
  
//SchedulingParameters - used for the Scheduler to activate the threads  
PriorityParameters priority = new PriorityParameters  
(  
    PriorityScheduler.getMaxPriority(rtThread) //Priority  
);  
/*  
Beispielprogramme: RTSJ_PeriodicRealtimeThread, ...  
*/
```

### Threads für Echtzeitsysteme

Die Ersten, die diese Parameter nutzen und damit vom Scheduler deterministisch gesteuert werden können, sind die neu eingeführten Klassen *RealtimeThread* und *NoHeapRealtimeThread*. Diese stellen eine Spezialisierung der bekannten Klasse *Thread* dar.

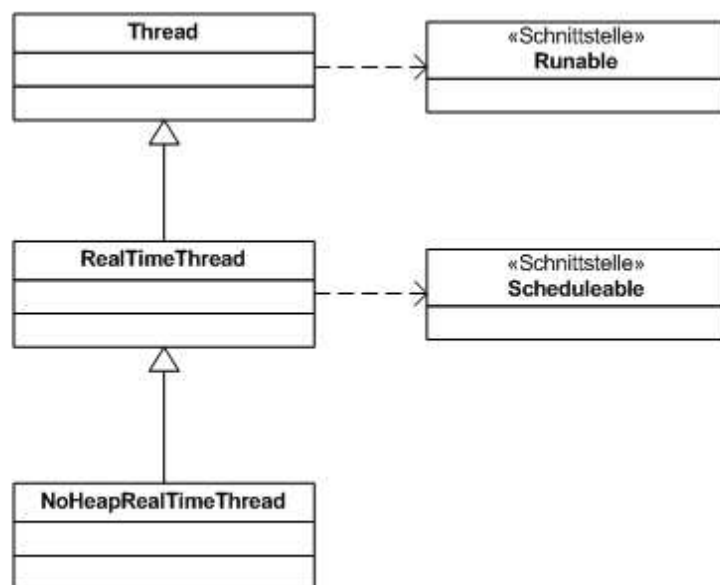


Abbildung 18: Klassenhierarchie *RealtimeThread*



### RealtimeThread

Ein *RealtimeThread* ist dabei nur für weiche Echtzeitanforderungen geeignet. Das sind Systeme, bei denen ein Überschreiten der deadline, innerhalb gewisser Grenzen, hingenommen werden kann, da es nur zu Störungen aber nicht zu großen materiellen Verlusten oder Personenschäden kommt. Ursache für diese Restriktion ist die Tatsache, dass zwar, mit Hilfe des Interface *Scheduleable*, ein Zeitverhalten vorgeschrieben werden kann, dieses aber nicht immer eingehalten wird. Das liegt daran, dass die GC, wenn sie erst einmal angelaufen ist, immer bis zu ihrem Ende abgearbeitet werden muss, um Inkonsistenzen zu vermeiden, und in dieser Zeit der *RealtimeThread* nicht zum Zuge kommt. Auch wenn dies eine große Einschränkung darstellt, hat diese Klasse dennoch ihre Berechtigung, da man mit ihrer Hilfe begrenzte Echtzeiteigenschaften erzielen kann, ohne dass man sich zu weit vom vertrauten Java entfernen und auf die neuen und ungewohnten Speicherbereiche ausweichen muss.

Wichtige Methoden:

*void start()* – Startet den Thread

*void run()* – Methode die der Thread nach seinem Start ausführt und nach dessen Abarbeitung er endet.

*RealtimeThread currentRealtimeThread()* – Gibt den aktuellen *RealtimeThread* zurück

*void sleep(HighResolutionTime)* – Sorgt dafür, dass der *RealtimeThread* eine Zeit lang inaktiv bleibt

*boolean waitForNextPeriod()* – Verzögert die Ausführung des Threads bis zum Beginn der neuen Periode

*boolean setIfFeasible(...)* – Ermöglicht das Setzen der einzelnen Parameter

*void interrupt()* – Ermöglicht es einen Thread zu unterbrechen

### NoHeapRealtimeThread

Der *NoHeapRealtimeThread*, eine Spezialisierung des *RealtimeThread*, kann dagegen bei harten Echtzeitanforderungen herangezogen werden, da er das vorgeschriebene Zeitverhalten unter allen Umständen garantiert, sobald man ihm eine Priorität, die höher als die der GC ist, zuweist. Denn ihm ist der Zugriff auf das *HeapMemory* untersagt, so dass er, im Gegensatz zu einem *RealtimeThread*, nun in der Lage ist, auch eine laufende GC zu unterbrechen.

Wichtige Methoden:

siehe *RealtimeThread*

```
/*
Ein RealtimeThread ohne besonderes Verhalten wird angelegt und gestartet.
*/
...
//create the first RealtimeThread
RealtimeThread rtThread = new RealtimeThread
(
    null, //SchedulingParameters
    null, //ReleaseParameters
    null, //MemoryParameters
    null, //MemoryArea
    null, //ProcessingGroupParameters
    new RealtimeThreadExample() //java.lang Runnable
);

//start the first RealtimeThread
rtThread.start();
...
/*
Beispielprogramme: RTSJ_RealtimeThread, RTSJ_NoHeapRealtimeThread_RTSJ, ...
*/
```

Schwächen:

- Aperiodische Ausführung  
Es ist bisher nicht möglich einen Thread komfortabel aperiodisch auszuführen, da die Methode *start()* kein zweitesmal aufgerufen werden kann und eine eigens konstruierte Lösung mit Aufwand und Overhead verbunden ist.
- Prioritäten  
Im Gegensatz zu den ersten Entwürfen, als die Prioritätsbereiche der Klassen *RealtimeThread* und *NoHeapRealtimeThread* noch getrennt waren, teilen sie sich mittlerweile den Gleichen. Das hat zwar den Vorteil, dass man auf diese Weise gemeinsam Ressourcen nutzen und eine mögliche Prioritätsinversion, wie sie in Kapitel 2.2.3.5 angesprochen wird, sehr leicht verhindern kann. Allerdings geht dies zu Lasten der Anwenderfreundlichkeit. Denn der Entwickler muss nun sehr genau aufpassen welche Prioritäten er vergibt. Beispielsweise ist es nun möglich einen *NoHeapRealtimeThread* durch die GC zu verzögern, indem man ihm einfach eine Priorität, die niedriger als die der GC ist, zuweist.

### 2.2.3.4 Physical memory access

Neben den Anforderungen an das zeitliche Verhalten, ist es für Echtzeitsysteme in viele Fällen auch von essentieller Bedeutung auf bestimmte physikalische Speicheradressen zugreifen zu können. Nur so ist es möglich Treiber und andere low-level Software zu schreiben. Da im herkömmlichen Java aber der Speicher völlig automatisch verwaltet wird, sind dem Entwickler die Hände gebunden.

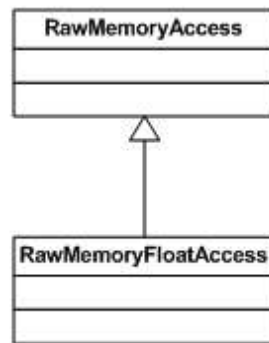


Abbildung 19:  
Klassenhierarchie  
*RawMemoryAccess*

Aus diesem Grund enthält das package *javax.realtime.\** die Klasse *RawMemoryAccess* und optional die Klasse *RawMemoryFloatAccess*.

### RawMemoryAccess

Ausgehend von einer Startadresse (*offset*) wird durch die Klasse *RawMemoryAccess* eine feste und zusammenhängende Anzahl an Bytes (*size*) für den direkten Zugriff freigegeben. Diese werden nun als *byte*, *short*, *int* oder *long* Werte interpretiert. Es besteht sogar die Möglichkeit direkt mit Arrays dieser Datentypen zu arbeiten. Des weiteren können, falls die Plattform *virtual memory* unterstützt, Speicherbereiche auf andere Adressen abgebildet werden.

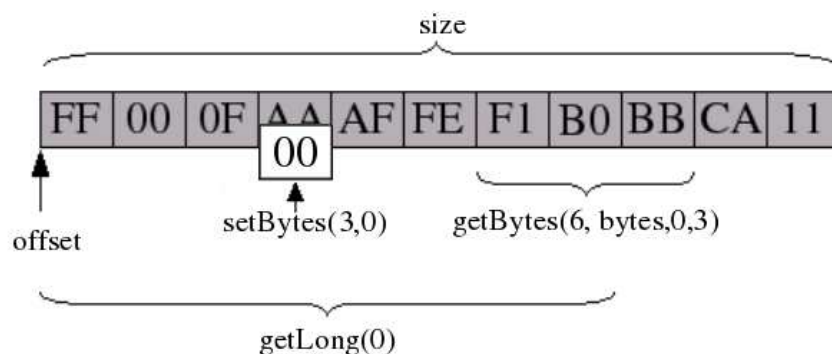


Abbildung 20: *RawMemoryAccess*

Wichtige Methoden:

*byte* *getBytes(long)* – Liefert das Byte an der gegebenen Position

*void* *setByte(long, byte)* – Setzt das Byte an der gegebenen Position auf den übergebenen Wert

### RawMemoryFloatAccess

*RawMemoryFloatAccess* gibt, wie auch die Klasse *RawMemoryAccess*, ausgehend von einer Startadresse (*offset*), eine feste und zusammenhängende Anzahl an Bytes (*size*) für den direkten Zugriff frei. Allerdings werden diese Bytes nun als *float* oder *double* Werte interpretiert.

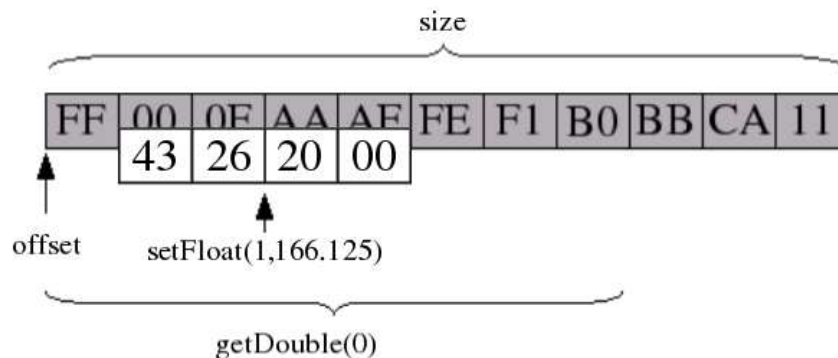


Abbildung 21: *RawMemoryFloatAccess*

Wichtige Methoden:

*double* *getDouble(long)* – Liefert den *double* Werte an der gegebenen Position

*void* *setDouble(long, double)* – Legt den *double* Wert an der gegebenen Position ab

#### Hinweise:

1. In diesem Zusammenhang spielt die Reihenfolge wie die Bytes im Speicher abgelegt werden eine entscheidende Rolle. Diese ist plattformabhängig und kann über den booleschen Wert `BYTE_ORDER` der Klasse *RealtimeSystem* abgefragt werden.
2. Diese Speicherbereiche können nicht dazu genutzt werden um Objekte abzulegen.
3. Die VM erlaubt nur Zugriff auf Speicherbereiche die außerhalb ihres Speichers liegen, damit es nicht zu Inkonsistenzen kommen kann.

```
/*
Es wird das Datenregister des LPT1 ausgelesen.
Dazu wird...
- der Type des Speichers, dessen Länge und Startadresse bestimmt
- Eine Instanz von RawMemoryAccess erzeugt
- die entsprechende Adresse gelesen
*/
...
Object type = PhysicalMemoryManager.IO_PAGE;
long address = 0x378;
long size = 3;

RawMemoryAccess rma = new RawMemoryAccess
(
    type, //memory type
    address, //starting address
    size //size
);

//read byte
byte data = rma.getBytes
(
    0 //offset from starting address
);
...
/*
Beispielprogramm: RTSJ_RawMemoryAccess
*/
```

### 2.2.3.5 Synchronization

Wie in allen nebenläufigen Anwendungen, muss auch bei Echtzeitsystemen der Zugriff auf exklusive Ressourcen geregelt werden. Java verfügt diesbezüglich bereits über ein ausgeklügeltes Monitor-Konzept. Bei diesem ist es möglich jegliches Objekt, unter Verwendung des Schlüsselwortes *synchronized*, als Monitor einzusetzen. Allerdings kommt es bei dem exklusiven Zugriff auf Ressourcen leicht zu einem Phänomen das unter dem Namen Prioritätsinversion bekannt ist.

Dabei wird ein Thread – HIGH – mit einer höheren Priorität an seiner Ausführung gehindert, da er auf eine exklusive Ressource warten muss. Diese wird nämlich im Augenblick von einem Thread – LOW – mit niedriger Priorität genutzt. Problematisch wird es erst, wenn ein Thread – MIDDLE – mit einer Priorität, die zwischen der von HIGH und LOW liegt, zu arbeiten beginnt. Denn dieser verdrängt LOW und verzögert somit die Ausführung von HIGH unnötig.

Priorityinversion

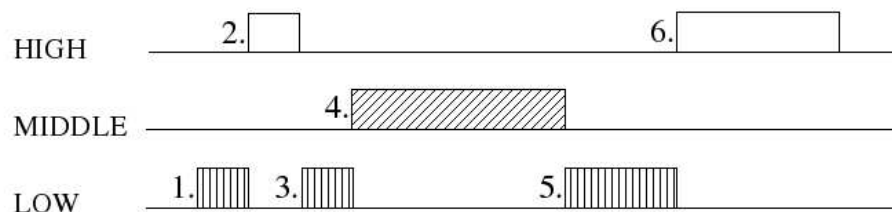


Abbildung 22: Prioritätsinversion

Ein solches Szenario lässt sich, wie die obige Grafik zeigt, auch sehr leicht konstruieren. (siehe Beispielprogramm RTSJ\_PriorityInheritance)

1. Man startet einen Thread – LOW – mit niedriger Priorität und reserviert so schnell wie möglich die exklusive Ressource.
2. Man startet nun einen Thread – HIGH – mit hoher Priorität. Dieser verdrängt zunächst LOW, bis er versucht die gleiche exklusive Ressource zu reservieren. Da diese aber bereits von LOW besetzt ist, gibt er die CPU wieder ab.
3. Da außer LOW keine weiteren Threads auf die CPU warten, wird dieser wieder aktiviert.
4. Wenn jetzt ein Thread – MIDDLE – mit einer Priorität die zwischen den Prioritäten von HIGH und LOW liegt, gestartet wird, dann verdrängt er LOW von der CPU. Das hat zur Folge, dass HIGH, in dieser Zeit, unnötig verzögert wird.
5. Erst wenn MIDDLE beendet ist, dann kommt wieder LOW zum Zuge. Denn nun ist er wieder der Thread mit der höchsten Priorität, der nur auf die CPU wartet. HIGH wartet ja weiterhin auf die Ressource.

6. Sobald LOW die Ressource wieder frei gibt, wartet HIGH nur noch auf die CPU. Da seine Priorität höher als die von LOW ist, bekommt er die CPU zugesprochen.

Bisher wurde in Java einfach über dieses Problem hinweg gesehen. Dass dies in Echtzeitanwendungen nicht mehr möglich ist, zeigt die berühmte Software-Panne der NASA im Zuge der *Mars Pathfinder Mission*<sup>8</sup> aus dem Jahre 1997. Bei dieser führte die Prioritätsinversion dazu, dass ein Watchdog immer wieder ein komplettes Reset des Mars Rover *Sojourner* durchführte, da ein wichtiger Thread nicht das gewünschte Zeitverhalten zeigte. Auf diese Weise gingen immer wieder wertvolle Daten verloren.

## Synchronisation

Aus diesem Grund erweitert die RTSJ, zur Synchronisation von Threads, dieses Monitor-Konzept und führt gleichzeitig die beiden gängigsten Strategien, die eine Priorityinversion verhindern, ein.

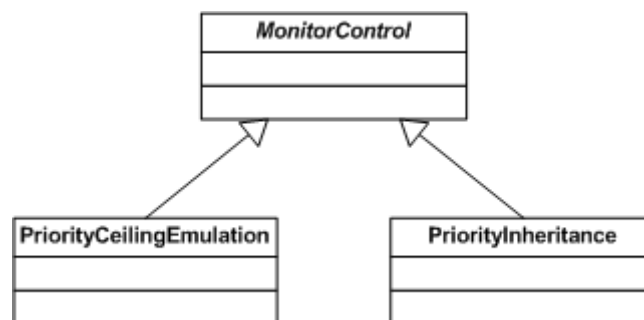


Abbildung 23: Klassenhierarchie MonitorControl

### MonitorControl

Um die Umsetzung weiterer Algorithmen zu ermöglichen wurde die abstrakte Klasse *MonitorControl* eingeführt. Sie ist die Basis aller Strategien und bietet nur die Möglichkeit den Monitoren eine der implementierten Strategie zuzuordnen bzw. die aktuelle Strategie abzufragen.

Wichtige Methoden:

*void setMonitorControl(java.lang.Object, MonitorControl)* – Weist die gegebene Strategie dem Monitor zu

*void setMonitorControl(MonitorControl)* – Legt die gegebene Strategie für alle Monitore fest

### PriorityInheritance

Bei der *PriorityInheritance* bekommt ein Thread, der eine Ressource exklusiv hält, auf die ein anderer Thread mit einer höheren Priorität wartet, dessen Priorität temporär zugewiesen. Sobald er die Ressource wieder frei gibt, wird er, nachdem seine Priorität wieder zurückgesetzt wurde, von dem Thread, dessen Priorität er geerbt hat, verdrängt.

Wichtige Methoden:

siehe MonitorControl

---

<sup>8</sup> [www5.in.tum.de/lehre/seminare/semsoft/unterlagen\\_02/marssojourner/website/mars/zusammen.htm](http://www5.in.tum.de/lehre/seminare/semsoft/unterlagen_02/marssojourner/website/mars/zusammen.htm)

### PriorityCeilingEmulation

Bei der *PriorityCeilingEmulation* wird der Ressource, die nur exklusiv genutzt werden kann, eine neue Priorität zugewiesen. Sobald ein Thread diese reserviert, wird seine Priorität, auf den Wert, der im Vorfeld der Ressource zugewiesen wurde, angehoben. Sobald er sie wieder frei gibt, wird auch seine Priorität auf den ursprünglichen Wert zurückgesetzt.

Wichtige Methoden:

keine – Die Priorität wird dem Konstruktor übergeben

Bei dem Einsatz einer der beiden Strategien ergibt sich für das obige Szenario nun folgendes Bild:

### PriorityInheritance / PriorityCeilingEmulation

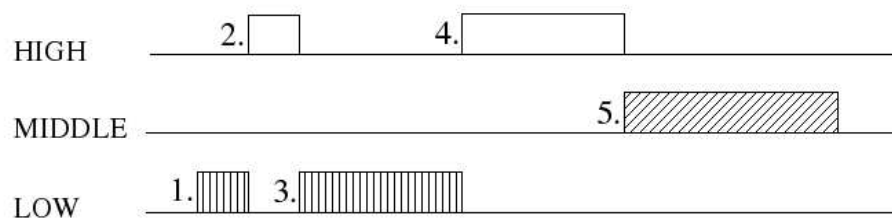


Abbildung 24: Prioritätsinversion (verhindert)

1. Man startet, wie schon zuvor, einen Thread – LOW – mit niedriger Priorität und reserviert die exklusive Ressource.
2. Nun startet man denn Thread – HIGH – mit der hohen Priorität. Dieser verdrängt zunächst LOW bis er wieder versucht die Ressource zu besetzen.
3. LOW wird wieder aktiviert, da kein weiterer Thread auf die CPU wartet. Allerdings läuft er nun mit einer neuen Priorität. Diese hat er entweder im Zuge der *PriorityInheritance* von HIGH geerbt oder im Zuge der *PriorityCeilingEmulation* direkt von der Ressource zugewiesen bekommen. Das ist der Grund weshalb der Start des Threads – MIDDLE – LOW nicht von der CPU verdrängt.
4. Sobald LOW diese Ressource wieder frei gibt, wird seine Priorität wieder zurückgesetzt und von HIGH verdrängt.
5. Sobald HIGH beendet ist wird MIDDLE aktiviert, da es sich dabei um den Thread mit der höchsten Priorität handelt der nur auf die CPU wartet.



## Datenaustausch

In diesem Zusammenhang stellen auch Objekte eine exklusive Ressource dar. Während Instanzen der Klassen *RealtimeThread* und *Thread* den *HeapMemory*, auf den Beide zugreifen können, nutzen, um Daten auszutauschen, steht der *NoHeapRealtimeThread* etwas isoliert dar. Denn er kann weder auf das *HeapMemory* zugreifen, noch kann er sich den Luxus erlauben durch die GC verzögert zu werden.

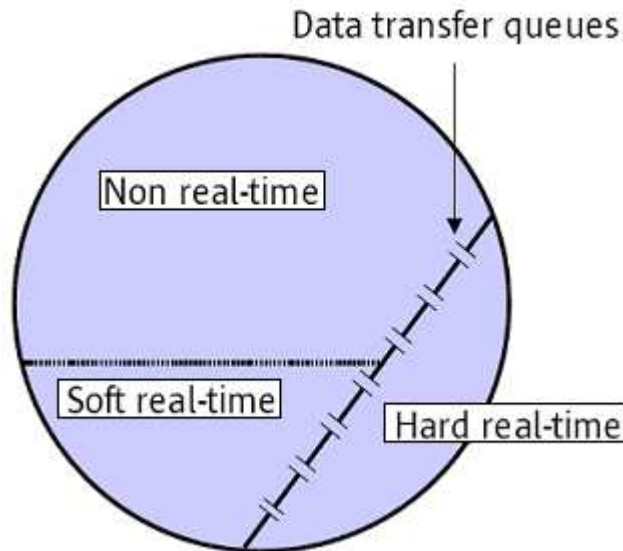


Abbildung 25: Einsatz von Warteschlangen zum Datenaustausch [CL S.14]

Aus diesem Grund musste ein neuer Mechanismus, in Form von Warteschlangen, eingeführt werden. Mit diesen ist es nicht nur möglich Daten zwischen einem herkömmlichen *Thread* und einem *NoHeapRealtimeThread* auszutauschen, sondern man kann unter allen Umständen ausschließen, dass es zu Verzögerungen kommt.

### Hinweis:

Bei diesen Warteschlangen ist aber immer darauf zu achten, dass der Thread, der den harten Echtzeitbedingungen unterliegt, immer das Ende ohne Wartezeiten zugeteilt bekommt.

### WaitFreeWriteQueue

In diese Warteschlangen kann ohne Verzögerung geschrieben werden, während das Lesen nur synchronisiert erfolgt. Daher sollte der *NoHeapRealtimeThread* nur auf das Schreiben beschränkt sein. Falls die Warteschlange voll ist, können mit Hilfe der Methode *force()* Werte überschrieben werden.

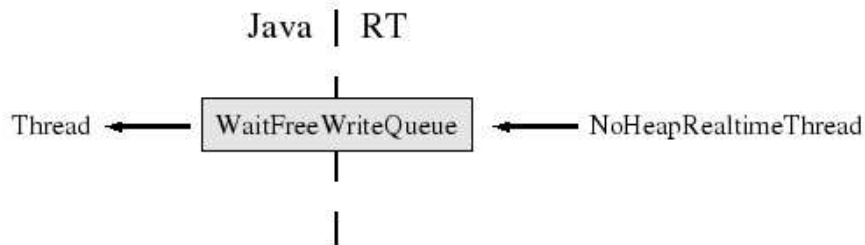


Abbildung 26: *WaitFreeWriteQueue*

#### Wichtige Methoden:

*boolean isEmpty()* – Gibt an ob die Warteschlange leer ist

*boolean isFull()* – Gibt an ob die Warteschlange voll ist

*java.lang.Object read()* – Synchronisiertes Lesen eines Wertes

*boolean write(java.lang.Object)* – Schreiben eines Wertes

*boolean force(java.lang.Object)* – Überschreiben des letzten Wertes

### WaitFreeReadQueue

Diese Warteschlangen erlauben dagegen nur ein synchronisiertes Schreiben, während das Lesen ohne Verzögerungen abläuft. Daher sollte der *NoHeapRealtimeThread* nur auf das Lesen beschränkt sein.

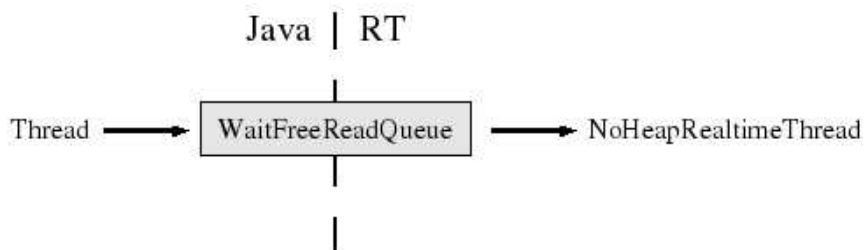


Abbildung 27: *WaitFreeReadQueue*

#### Wichtige Methoden:

*java.lang.Object read()* – Lesen eines Wertes

*boolean write(java.lang.Object)* – Synchronisiertes Schreiben eines Wertes

### WaitFreeDequeue

Diese Klasse besteht sowohl aus einer *WaitFreeWriteQueue* als auch aus einer *WaitFreeReadQueue*. Aus diesem Grund kann sie für einen bidirektionalen Datenaustausch genutzt werden.

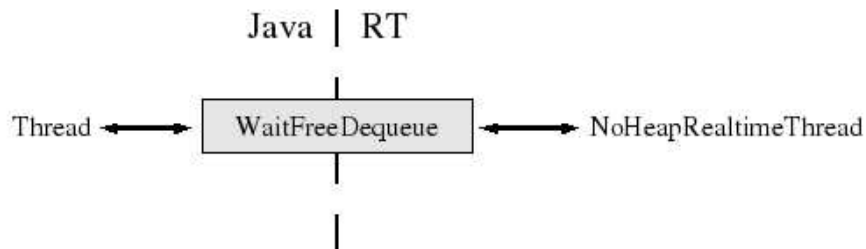


Abbildung 28: *WaitFreeDequeue*

Wichtige Methoden:

*boolean blockingWrite(java.lang.Object)* – Synchronisiertes Schreiben in die *WaitFreeReadQueue*

*boolean nonBlockingWrite(java.lang.Object)* – Schreiben in die *WaitFreeWriteQueue*

*java.lang.Object nonBlockingRead()* – Lesen aus der *WaitFreeReadQueue*

*java.lang.Object blockingRead()* – Synchronisiertes Lesen aus der *WaitFreeWriteQueue*

#### Hinweis:

In der kürzlich veröffentlichten Version 1.0.1 wurde diese Klasse als deprecated gekennzeichnet, da sie überflüssig ist und sich eine Synchronisation mit separaten Instanzen von *WaitFreeReadQueue* und *WaitFreeWriteQueue* einfacher gestaltet.

```
/*  
Es wird eine WaitFreeWriteQueue der Größe QUEUE_SIZE erzeugt. Von einem NoHeapRealtimeThread  
werden nun Integer-Werte geschrieben und parallel von einem Thread ausgelesen.  
*/  
...  
//Create queue  
this.queue = new WaitFreeWriteQueue(QUEUE_SIZE);  
...  
queue.force(new Integer(i));  
...  
Integer j = (Integer)queue.read();  
...  
/*  
Beispielprogramm: RTSJ_WaitFreeWriteQueue  
*/
```

Schwächen:

- Mehrfach Synchronisation  
Der Nachteil dieser Warteschlangen besteht darin, dass sich so nur zwei Threads synchronisieren lassen. Wollen beispielsweise mehrere Instanzen vom Typ *NoHeapRealtimeThread* eine *WaitFreeWriteQueue* nutzen, dann müssen diese separat synchronisiert werden.

### 2.2.3.6 Asynchronous event handling

Systeme die eng mit der Realität verknüpft sind müssen sowohl auf interne als auch auf externe Ereignisse schnell und in geeigneter Weise reagieren können. Das Problem ist, dass es sehr viele verschiedene Ereignisse mit den unterschiedlichsten Konsequenzen gibt. Erschwerend kommt hinzu, dass auf diese in geeigneter Art und Weise reagiert werden muss. Java bot für solche Fälle bisher keine komfortable Lösungen an. Man hätte zwar mit Exceptions arbeiten können, doch diese sollten nur bei echten Ausnahmen Verwendung finden. Aus diesem Grund führt die RTSJ einen neuen Mechanismus, das *asynchronous event handling* (AEH), ein.

#### Ereignisse

Dabei wird das eigentliche Ereignis strikt von der Reaktion getrennt. Erstere lassen sich dabei mit Hilfe der folgenden Klassen abbilden.

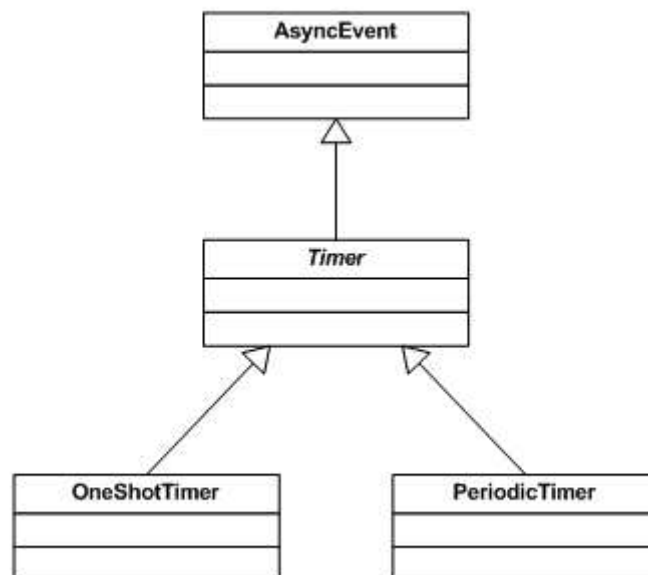


Abbildung 29: Klassenhierarchie AsyncEvent

#### AsyncEvent

Die Klasse *AsyncEvent* repräsentiert ein allgemeines Ereignis. Dabei wird zwischen internen Ereignissen, die der Entwickler in seinem Code definiert und die mit der *fire()*-Methode ausgelöst werden müssen, und externen Ereignissen, wie beispielsweise einem Interrupt, die man nur mit der Methode *bindTo(String)* an ein solches Ereignis knüpft, unterschieden. Da es dabei nur darauf ankommt das Auftreten eines Ereignisses zu signalisieren ist diese Klasse relativ einfach konzipiert und ermöglicht darüber hinaus nur das Registrieren von Reaktionen.

Wichtige Methoden:

*void addHandler(AsyncEventHandler)* – Meldet einen *AsyncEventHandler* an

*void fire()* – Aktiviert die angemeldeten *AsyncEventHandler*

*void bindTo(String)* – Knüpft das *AsyncEvent* an ein externes Ereignis, das über den *String* identifiziert wird

Schwächen:

- Die *Strings*, die von der Methode *AsyncEvent.bindTo(String)* genutzt werden um auch auf externe Ereignisse reagieren zu können, sind nicht vorgeschrieben. Ebenfalls ist nicht definiert, wo diese *Strings* abgefragt werden können.

### Timer

Die abstrakte Klasse *Timer* beschreibt die grundlegenden Eigenschaften die zeitlich gesteuerte Ereignisse auszeichnen. Der Aufbau erinnert dabei stark an einen Countdown. Sobald dieser abgelaufen ist wird dessen *fire()*-Methode ausgelöst. Eine Instanz der Klasse *OneShotTimer* wird in diesem Zusammenhang einmalig, nach Ablauf des Zeitintervalls, oder an dem eingestellten Zeitpunkt, ausgelöst, während ein *PeriodicTimer* zyklisch, in immer gleichen Abständen aktiviert wird.

#### Hinweis:

Mit Hilfe einer Instanz von *PeriodicTimer* lassen sich die gleichen Probleme lösen, für die man einen *RealtimeThread* mit *PeriodicParameters* heranziehen würde. Allerdings hat diese Variante den Vorteil, dass sie weniger Ressourcen benötigt.

Wichtige Methoden:

*void start()* – Startet den Countdown

*AbsoluteTime getFireTime()* – Liefert die Zeit zu welcher der Countdown ablaufen wird

*void disable()* – Deaktiviert den Countdown, so dass er bei „0“ nicht die *fire()*-Methode auslöst

*void enable()* – Reaktiviert den Countdown nachdem er deaktiviert wurde

*void reschedule(HighResolutionTime)* – Überschreibt den Countdown

*void setInterval(RelativeTime)* – Legt das Intervall fest, in dem der *PeriodicTimer* ausgelöst wird

## Reaktionen

Die Reaktionen auf ein solches Ereignis werden, wie bereits erwähnt, separat in einer der folgenden Klassen gekapselt.

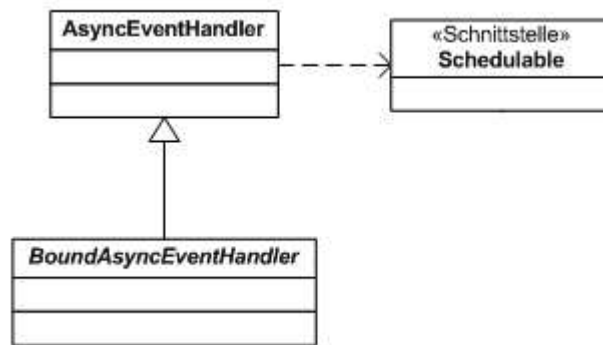


Abbildung 30: Klassenhierarchie `AsyncEventHandler`

### `AsyncEventHandler`

Die Klasse `AsyncEventHandler` ermöglicht es auf ein oder mehrere Ereignisse zu reagieren, indem sie ihre `handleAsyncEvent()`-Methode ausführt oder auf ein `Runnable` verweist. Wie das Interface `Schedulable` schon anklingen lässt, handelt es sich dabei um einen nebenläufigen Prozess, der einem Thread sehr ähnlich ist. Das hat zur Folge, dass sie nicht wie herkömmliche Handler, wie sie beispielsweise aus AWT bekannt sind, direkt ausgeführt, sondern parallel verarbeitet werden. Man hat sich für diese Architektur entschieden, da man davon ausgehen kann, dass es zwar sehr viele unterschiedliche Ereignisse gibt, aber gleichzeitig nur sehr wenige auftreten und diese häufig auch noch die gleiche Reaktion erfordern.

Der Aufbau eines solchen AEH ist relativ einfach und folgt immer dem gleichen Schema:

1. Nach dem Anlegen werden die `AsyncEventHandler` mit Hilfe der `AsyncEvent.addHandler()`-Methode an die Ereignisse geknüpft.
2. Das Ereignis tritt ein und es wird die `fire()`-Methode des `AsnycEvent` ausgelöst.
3. Alle registrierten `AsyncEventHandler` werden gestartet und führen ihre `handleAsyncEvent()`-Methode bzw. das übergebene `Runnable` parallel aus.

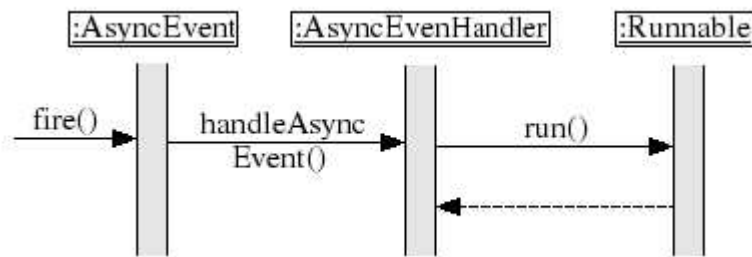


Abbildung 31: AsyncEventHandling

Darüber hinaus garantiert er, auch in kritischen Situationen, in denen mehrere Ereignisse in kurzer Abfolge auftreten, ein deterministisches Verhalten. So stellt er selbstständig sicher, dass er für jedes eingetretene Ereignis einmal die *handleAsyncEvent()*-Methode ausführt. Diese kapselt die eigentliche Reaktion, wobei sichergestellt ist, dass eine Reaktion nicht mehrfach parallel gestartet werden kann.

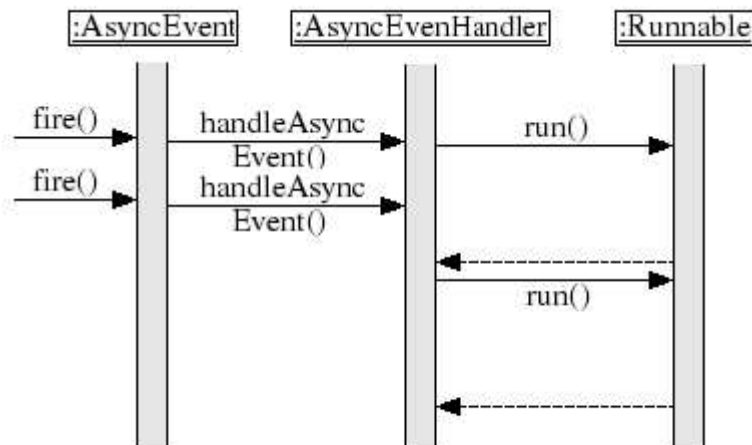


Abbildung 32: Verhalten in kritischen Situationen

Wichtige Methoden:

*void handleAsyncEvent()* – Wird beim Eintreten eines *AsyncEvent* aufgerufen

*boolean setIfFeasible(...)* – Ermöglicht das Setzen der einzelnen Parameter

### BoundAsyncEventHandler

Da der *AsyncEventHandler* dynamisch, also erst beim Auftreten des Ereignisses, für seine parallele Ausführung vorbereitet wird, startet er in der Regel mit einer kleinen Verzögerung. Falls dieses Verhalten für die Anwendung nicht tragbar ist, besteht die Möglichkeit einen *BoundAsyncEventHandler* einzusetzen. Dieser ist bereits ausführbar und startet daher entsprechend schneller.

Wichtige Methoden:

siehe *AsyncEventHandler*



```
/*
Gemäß dem Schema werden als erstes sowohl ein AsyncEvent als auch ein AsyncEventHandler angelegt
und dann über die addHandler()-Methode miteinander verknüpft. Abschließend wird das AsyncEvent mit der
fire()-Methode ausgelöst.
*/
...
//Create an AsyncEvent
AsyncEvent event = new AsyncEvent();
...
//Create an AsyncEventHandler with a specific logic (Runnable)
AsyncEventHandler eventHandler1 = new AsyncEventHandler
(
    new Runnable()
    {
        public void run()
        {
            ...
        }
    }
);
...
//Add those eventHandlers to that event
event.addHandler(eventHandler1);
...

//Fire the event
event.fire();
...

/*
Beispiele: RTSJ_AsyncEventHandler, RTSJ_BoundAsyncEventHandler, ...
*/
```

### POSIXSignalHandler

Beim *POSIXSignalHandler* handelt es sich um eine optionale Klasse. Das hängt damit zusammen, dass POSIX (Portable Operating System Interface for Unix) eine verbreitete und standardisierte API für Unix Systeme ist und damit auch nur auf jenen, die diese unterstützen, von Nutzen ist.

Wichtige Methoden:

*void addHandler(int, AsyncEventHandler)* – Bindet den *AsyncEventHandler* an das POSIX-Signal

### **2.2.3.7 Asynchronous transfer of control**

Viele Programmierer von Echtzeitsystemen sehen sich mit dem Problem konfrontiert, dass die Rechenzeit mancher Algorithmen nicht fest vorgegeben ist. Das kann damit zusammenhängen, dass diese rekursiv arbeiten und bei jedem Durchlauf das Ergebnis nur noch weiter verfeinern. Damit solche Algorithmen auch in zeitkritischen Systemen eingesetzt werden können, ist es üblich das Ergebnis erst dann abzurufen, wenn die deadline fast erreicht ist. Ein solcher plötzlicher Sprung aus einer Prozedur wird als *asynchronous transfer of control* (ATC) bezeichnet.

An diesen Mechanismus wurden im Rahmen der RTSJ folgende Anforderungen gestellt:

1. Die Möglichkeit einen ATC durchzuführen muss explizit gekennzeichnet werden. Fehlt diese Kennzeichnung, dann darf kein ATC durchgeführt werden, da nicht sichergestellt ist, dass der Code für ATC ausgelegt ist.
2. Selbst wenn ATC prinzipiell unterstützt wird (siehe 1), muss es die Möglichkeit geben, Bereiche, die nicht unterbrochen werden können, zu definieren.
3. Nach dem Auslösen einer ATC wird der Code nicht automatisch an der Stelle, an der er unterbrochen wurde, fortgesetzt. Falls dies erforderlich ist, kann man sich anderer Mechanismen, beispielsweise des AEH, bedienen.
4. Ein ATC muss gezielt auslösbar sein.
5. Das Auslösen eines ATC kann direkt, mit Hilfe eines Threads, oder indirekt über einen AsyncEventHandler erfolgen.
6. ATC muss es ermöglichen einen Thread sicher zu beenden.
7. ATC sollte bekannte Sprachkonstrukte nutzen und, falls es nicht eingesetzt wird, sollte es für den Entwickler keinen Mehraufwand bedeuten.
8. Das Schachteln von ATC muss möglich sein.

Java bietet diesbezüglich bereits in der Klasse Thread die *interrupt()*-Methode an. Diese sorgt dafür, dass eine *InterruptedException* geworfen und zum entsprechenden Exception-Handling gesprungen wird. Da dies aber nur für verzögerte Threads gilt und diese, wenn sie sich im Zustand *Running* oder *Runnable* befinden, nicht direkt unterbrochen werden, sondern nur ein Flag gesetzt wird, welches der Thread in regelmäßigen Abständen überprüft, kann man in zeitkritischen Systemen nicht damit arbeiten. Um dieses einfache Konzept dennoch beizubehalten und die gestellten Anforderungen zu erfüllen, wurde zum einen die *interrupt()*-Methode in der Klasse *RealtimeThread* überschrieben und zum anderen die *InterruptedException* erweitert.

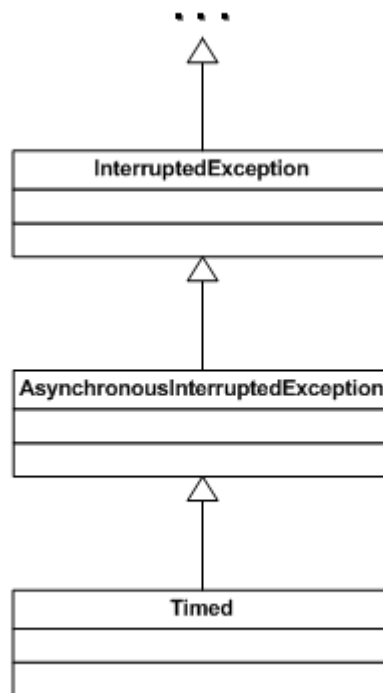


Abbildung 33: Klassenhierarchie AIE

### AsynchronouslyInterruptedException

Die *AsynchronouslyInterruptedException* stellt also eine Erweiterung der *InterruptedException* dar. Sie muss in einem separaten *catch*-Block abfangen werden, um zu signalisieren, dass man diesen Block durch einen ATC unterbrechen kann. Kritische Bereiche innerhalb dieses *try*-Blocks können dabei mit Hilfe des Schlüsselwortes *synchronized* vor einer Unterbrechung geschützt werden. Dieses sorgt nämlich dafür, dass, bei einer Unterbrechung, durch die *interrupted()*-Methode der Klasse *RealtimeThread*, erst nach Verlassen des kritischen Bereiches die AIE geworfen wird.

Damit könnte das eingangs beschriebene Problem wie folgt gelöst werden:

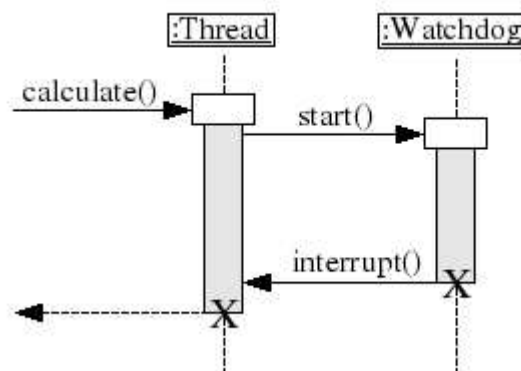


Abbildung 34: Lösung mit Watchdog

1. Für die Berechnung wird ein eigener Thread gestartet.
2. Ein vom Entwickler geschriebener Watchdog überwacht die Zeit, die der Berechnung zur Verfügung steht.
3. Mit Hilfe der *interrupt()*-Methode unterbricht dieser die Berechnung sobald das Ende der deadline naht.
4. Das letzte gültige Ergebnis wird, bevor die deadline überschritten wird, zurückgeliefert

Wichtige Methoden:

*boolean doInterruptible(Interruptible)* – Übergibt die Kontrolle an das entsprechende *Interruptible*

### Timed

Die Klasse *Timed* ermöglicht es den obigen Aufbau weiter zu vereinfachen, da sie die Implementierung eines eigenen Watchdog überflüssig macht, indem sie die zeitliche Steuerung übernimmt, während sie die Logik in ein *Interruptible* auslagert.

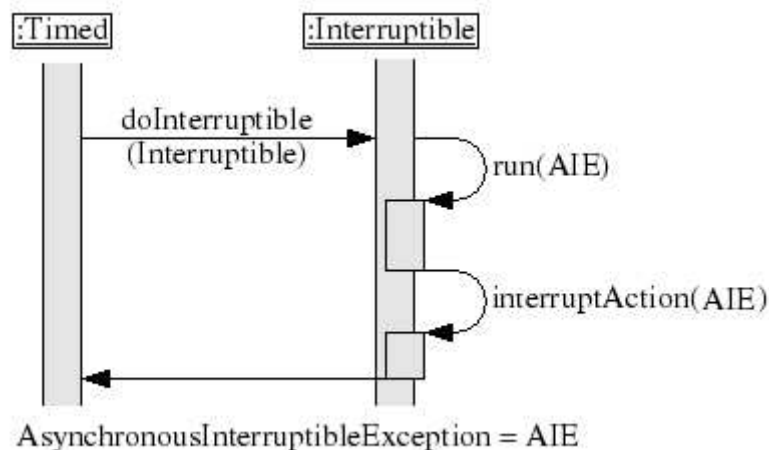


Abbildung 35: Lösung mit *Timed*

Dieses *Interruptible* bekommt genau die angegebene Zeit um seine Berechnung in der *run(AIE)*-Methode durchzuführen und wechselt dann automatisch in die *interruptedAction(AIE)*-Methode, in der das Ergebnis ermittelt werden kann.

Wichtige Methoden:

*void resetTime(HighResolutionTime)* – Legt ein Intervall für die Bearbeitung fest

*boolean doInterruptible(Interruptible)* – Startet ein neues Intervall

```
/*
Eine Berechnung, welche in der run(AIE)-Methode des Interruptible gekapselt ist, wird 1000 ms
durchgeführt. Anschließend wird automatisch in die interruptAction(AIE)-Methode gewechselt und das
Ergebnis ausgegeben
*/
...
new Timed(new RelativeTime(1000,0)).doInterruptible(new RTSJ_TimedExample());
...
public void run(AsynchronouslyInterruptedException e)
                                throws AsynchronouslyInterruptedException
{
    //computing result
    ...
}
public void interruptAction(AsynchronouslyInterruptedException e)
{
    //return result
    ...
}
...
/*
Beispielprogramme: RTSJ_AsynchronouslyInterruptedException, RTSJ_Timed, ...
*/
```

## Interruptible

Klassen die das Interface *Interruptible* implementieren können von Instanzen der AIE mittels der *doInterruptible(Interruptible)*-Methode aktiviert werden. Dadurch wird zum einen sichergestellt, dass vor und nach dem Aufruf deren *run(AIE)*-Methode das System nochmals aktiv werden kann, und zum anderen hat man die Möglichkeit auf eine Unterbrechung dieser Methode zu reagieren. In diesem Fall wird nämlich automatisch die *interruptAction(AIE)*-Methode aufgerufen.

Wichtige Methoden:

*void run(AIE)* – Wird beim Aufruf von *doInterruptible(Interruptible)* ausgeführt

*void interruptAction(AIE)* – Wird ausgeführt, falls die *run(AIE)* ein zweites mal aufgerufen wird, bevor sie beendet werden konnte.

### 2.2.3.8 Asynchronous thread termination

In Systemen die eng mit der realen Welt verknüpft sind, kommt es vor, dass auf gravierende Änderungen, wie beispielsweise einem Nothalt, reagiert werden muss. In diesen Fällen ist es nötig Threads vorzeitig zu beenden, da sie entweder nicht mehr von Nutzen sind oder sogar die Reaktion auf dieses Ereignis stören. Um nicht alle möglichen Fälle modellieren zu müssen, boten frühere Java Versionen die Methoden *stop()* und *destroy()* in der Klasse *Thread* an. Da aber beide Methoden kein sicheres Beenden eines Threads garantieren können, ist deren Einsatz nicht ratsam. Daher schlägt die RTSJ folgendes Verfahren vor, das auf einer Kombination der Beiden, zuvor behandelten Mechanismen, AEH und ATC basiert.

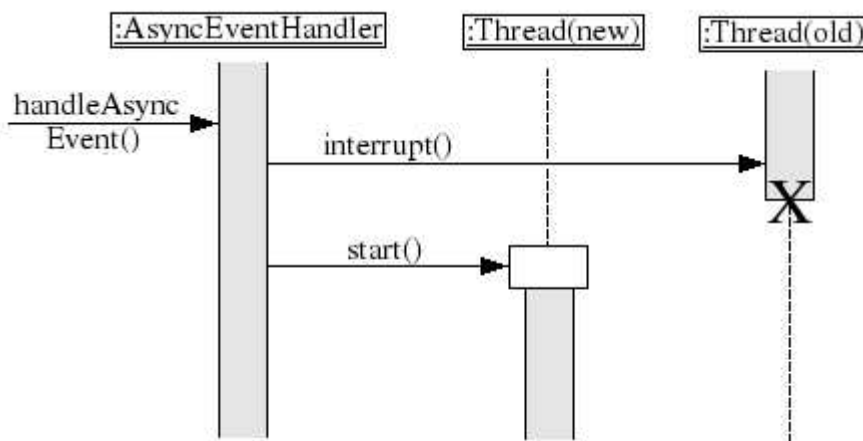


Abbildung 36: Sicheres Beenden eines Thread

Ablauf:

1. Das Ereignis, welches einen ATT erfordert, tritt ein und löst die *fire()*-Methode des entsprechenden *AsyncEvent* aus.
2. Der *AsyncEventHandler*, der den ATT durchführt, wird aktiv und ruft die *interrupt()*-Methode des zu beenden Threads auf. Das hat zur Folge, dass der Thread, sobald er sich in einem Bereich befindet bei dem er unterbrochen werden kann, seine aktuelle Arbeit einstellt und in die *catch*-Klausel der AIE wechselt. Dort hat er die Möglichkeit sein sauberes Ende vorzubereiten, indem er dafür sorgt, dass die *run()*-Methode normal beendet werden kann.
3. Der *AsyncEventHandler* startet nun einen neuen Thread, der mit dem Ereignis besser umgehen kann.

Dieses Verfahren hat darüber hinaus den Vorteil, dass der *AsyncEventHandler* an mehrere Ereignisse geknüpft werden kann. Damit müssen alle Ereignisse, die diese ATT erfordern, nur ein einfaches *AsyncEvent* umsetzen.

```

/*
Der Thread mit der folgenden run()-Methode kann asynchron beendet werden, da er die
AsynchronouslyInterruptedException explizit abfängt und die Schleife beendet.
*/
...
public void run()
{
    ...
    while(running)
    {
        try
        {
            ...
        }
        catch(AsynchronouslyInterruptedException e)
        {
            //leave run()-methode == secure finilization
            running = false;
        }
        catch(InterruptedException e)
        {
            //do nothing
        }
    }
    ...
}

/*
Die Subklasse SecureThreadTermination von AsyncEventHandler unterbricht nur den Thread, so dass er
sicher beendet wird, falls ein entsprechendes Ereignis auftritt.
*/
...
public void handleAsyncEvent()
{
    rtThread.interrupt();
}
...

/*
Beispielprogramm: RTSJ_SecureThreadTermination
*/

```

Schwächen:

- Fehlender Komfort

Es gibt nach wie vor keine bequeme Lösung um einen Thread sicher zu beenden. Es sind auch weiterhin besondere Vorkehrungen seitens des Entwicklers nötig.



#### **2.2.3.9 Sonstige**

Neben den bereits besprochenen Klassen, die aktiv eingesetzt werden, um die Schwächen von Java auszugleichen, führt die RTSJ noch eine Reihe weiterer ein. Dazu zählen neben den neuen Errors und Exceptions auch die Klassen *RealtimeSystem* und *RealtimeSecurity*. Letztere kümmert sich vorwiegend um den Zugriff auf physikalische Speicheradressen während die Klasse *RealtimeSystem* es ermöglicht grundlegende Einstellungen, wie beispielsweise die `BYTE_ORDER`, abzufragen und Zugriff auf die lokalen Instanzen *RealtimeSecurity*, *GarbageCollector* und *Scheduler* der VM gewährt.

### 3. Entwicklungsumgebung

Beim praktischen Arbeiten mit der RTSJ, wird in diesem Fall, wie es in der Automation üblich ist, zwischen einem Programmiergerät, auf dem die Anwendung entwickelt wird, und einem Steuergerät, auf dem die Anwendung produktiv eingesetzt wird, unterschieden. Diese Trennung hängt mit der Tatsache zusammen, dass die Anforderungen, die bei der Entwicklung und bei der Steuerung an ein System gestellt werden, derart unterschiedlich sind, dass es nicht nur aus finanzieller Sicht günstiger ist, diese separat zu handhaben. Allerdings kommt hier für beide Geräte die gleiche HW, ein Standard-PC, zum Einsatz, so dass sich diese nur durch die hier aufgeführten SW-Komponenten unterscheiden.



#### **Programmiergerät**

Suse 8.2 Professional

J2SDK

eclipse

RTEMS cross development environment

Jamaica Toolset



#### **Steuergerät**

RTEMS

GRUB

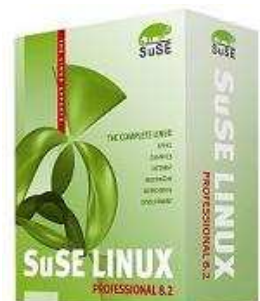
Aus diesem Grund ist die Trennung zwischen diesen beiden nicht physikalischer Natur, sondern wird durch das Laden der entsprechenden Konfiguration erreicht.

#### 3.1 Programmiergerät

Da das Programmiergerät, neben der reinen Implementierung der Anwendung, auch noch eine Reihe weiterer Aufgaben, wie beispielsweise die Dokumentation, erfüllen muss, wird ein Standard-PC herangezogen und mit den, für die Entwicklung, notwendigen SW-Komponenten ausgestattet.

##### 3.1.1 Suse 8.2 Professional

Als Basis dient hierfür die Linux Distribution Suse 8.2<sup>9</sup> der SuSE Linux AG. Sie zeichnet sich durch einen stabilen Kernel, eine gute Dokumentation, eine einfache Installation und Konfiguration und der Tatsache, dass sie frei im Netz verfügbar ist, aus. Darüber hinaus liefert sie auch noch eine Vielzahl von Anwendungen, die den Entwicklungsprozess unterstützen, mit. Dazu zählen die GCC, die Office Anwendungen von *OpenOffice.org*, sowie eine Reihe kleiner Tools.



<sup>9</sup> [www.novell.com/linux/suse/](http://www.novell.com/linux/suse/)

Hinweis:

Da es sich bei Linux nicht um ein RTOS handelt, kann es auch beim Einsatz der RTSJ kein Echtzeitverhalten garantieren. Es werden zwar Erweiterungen von Linux, wie beispielsweise Linux/RT angeboten, doch diese werden im Moment noch nicht von der JamaicaVM unterstützt.

### 3.1.2 J2SDK



Neben diesen Programmen, die bereits der Suse 8.2 beiliegen, muss unter anderem noch eine aktuelle Version des J2SDK<sup>10</sup> installiert werden. Allerdings werden von den angebotenen Entwicklungswerkzeugen nur *javac*, *javadoc* und der *appletviewer* eingesetzt. (Version 1.4.2\_8)

Hinweis:

Um Komplikationen zu vermeiden, sollte zuvor noch das aktuelle JRE von Blackdown mit Hilfe von *Yast* entfernt werden. Bei dieser Gelegenheit bietet es sich an, den GCJ zu installieren, da dieser beim Einsatz von JNI in der JamaicaVM benötigt wird.

### 3.1.3 eclipse

Um effizient mit einer Programmiersprache arbeiten zu können, bedarf es einer professionellen Entwicklungsumgebung. Die Wahl fiel auf eclipse<sup>11</sup>, da diese frei erhältlich ist und insbesondere für Java sehr viel Komfort bietet. Darüber hinaus wird von der aicas GmbH ein Plug-in angeboten, welches die Arbeit mit deren Produkt vereinfacht. (Version 3.0)



### 3.1.4 RTEMS cross development environment



Um Anwendungen für das Echtzeitbetriebssystem RTEMS zu erstellen muss ein entsprechendes *RTEMS cross development environment*, so wird dort die Entwicklungsumgebung bezeichnet, eingerichtet werden. Dabei wird in großen Umfang auf die freien GNU Tools zurückgegriffen. (Version 4.6.2)

<sup>10</sup> [www.java.sun.com](http://www.java.sun.com)

<sup>11</sup> [www.eclipse.org](http://www.eclipse.org)

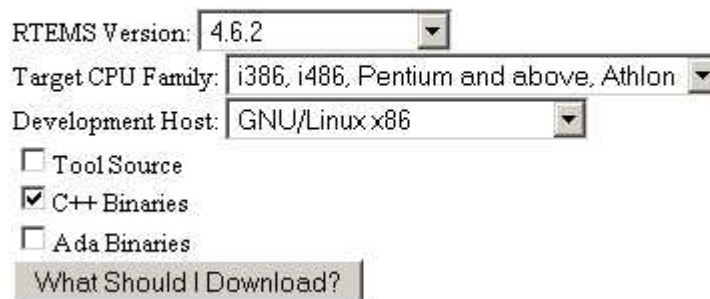
### 3.1.4.1 Installation

Im Gegensatz zu den bisherigen SW-Komponenten gestaltet sich dessen Installation kompliziert. Daher wird detaillierter darauf eingegangen<sup>12</sup>.

#### Prebuild Toolset

Bevor das *RTEMS cross development environment* einrichtet werden kann, müssen mehrere Tools installiert werden. Diese sind in Module zergliedert, wobei zwischen plattformübergreifenden *base* Modulen und jenen, die nur für bestimmte Ziel-Plattformen und Programmiersprache notwendig sind, unterschieden wird. Da es eine Reihe solcher Module gibt, wird unter [www.rtems.com](http://www.rtems.com) ein sog. *Download-Helper* angeboten. Dieser sucht, anhand der Angaben des Users, die entsprechenden Tools heraus und bietet diese zum Download an.

## RTEMS Download Helper



RTEMS Version: 4.6.2  
Target CPU Family: i386, i486, Pentium and above, Athlon  
Development Host: GNU/Linux x86  
☐ Tool Source  
☒ C++ Binaries  
☐ Ada Binaries  
What Should I Download?

Abbildung 37: Screenshot vom Download Helper auf [www.rtems.com](http://www.rtems.com)

Für die hier verwendete Kombination sind folgenden Tools vorgeschrieben:

- **BINUTILS**  
Bei den GNU BINUTILS handelt es sich um eine Sammlung von binary tools. Diese umfasst unter anderem den GNU *linker* und den GNU *assembler*
- **GCC**  
Bei GCC handelt es sich um die GNU *Compiler Collection*. Eine Sammlung von Compilern für C, C++, Java, etc.
- **NEWLIB**  
Die NEWLIB ist eine spezielle C Bibliothek für Embedded-Systems
- **GDB**  
GDB ist der GNU *Project Debugger*

---

<sup>12</sup> Die Installation ist in [started.pdf](#) ausführlicher beschrieben

## Evaluation der Realtime Specification for Java anhand einer Robotersteuerung

Diese liegen bereits als RPM Dateien vor und können daher einfach installiert werden. Folgende Reihenfolge sollte dabei eingehalten werden, weil es sonst zu Komplikationen kommen könnte, da zwischen den Paketen Abhängigkeiten bestehen.

```
rpm -i rtems-4.6-rtems-base-binutils-2.13.2.1-2.i686.rpm
rpm -i rtems-4.6-i386-rtems-binutils-2.13.2.1-2.i686.rpm
rpm -i rtems-4.6-rtems-base-gcc-gcc3.2.3newlib1.11.0-4.i386.rpm
rpm -i rtems-4.6-i386-rtems-c++-gcc3.2.3newlib1.11.0-4.i686.rpm
rpm -i rtems-4.6-i386-rtems-gcc-gcc3.2.3newlib1.11.0-4.i686.rpm
rpm -i rtems-4.6-rtems-base-gdb-5.2-1.i686.rpm
rpm -i rtems-4.6-i386-rtems-gdb-5.2-1.i686.rpm
```

Damit diese Tools auch bei Bedarf gefunden werden, muss ihre Position in einer Umgebungsvariable, hier PATH, aufgenommen werden. Um zu verhindern, dass ältere Versionen dieser Tools verwendet werden, sollte man den aktuellen Pfad an erster Stelle aufnehmen.

```
export PATH=<INSTALL_POINT>/bin:${PATH}
```

### configure

Nachdem die Tools, die für den Bau notwendig sind, installiert wurden, kann das *RTEMS cross development environment* gebaut werden. Dazu müssen die Sourcen, die in *rtems-4.6.2.tar.bz2* enthalten sind entpackt und das Script *configure* von einem neuen Verzeichnis aus, in das es installiert werden soll, ausgeführt werden. Dieses konfiguriert und compiliert die Entwicklungsumgebung für RTEMS.

```
../rtems-4.6.2/configure --target=i386-rtems
```

### make

Mit Hilfe von *make* wird anschließend das *RTEMS cross development environment* zusammengestellt und installiert.

```
make install all
```

Für die unterschiedlichen Plattformen werden folgende BSPs angeboten:

Plattform	Beschreibung
pc386	BSP spezielle für i386
i386ex	BSP für uti386ex
pc386dx	BSP spezielle für i386dx
ts_386ex	BSP für Technologic Systems TS-1325
pc486	BSP spezielle für i486
pc586	BSP spezielle für Pentium
pc686	BSP spezielle für Pentium II
pck6	BSP spezielle für AMD K6

*Tabelle 2: BSP für Standard PC*

Hinweis:

Es ist ratsam nur die notwendigen BSPs zu installieren, da *configure* und *make* relative lange Zeit in Anspruch nehmen und jedes einzelne BSP Festplattenspeicher benötigt.

### 3.1.4.2 Einsatz

Nachdem das *RTEMS cross development environment* eingerichtet ist, könnte man direkt Applikationen in C/C++ schreiben und compilieren. Da RTEMS aber in Verbindung mit dem Jamaica Toolset genutzt werden soll, muss diesem noch die Position der Entwicklungsumgebung mitgeteilt werden. Dazu muss, nach dessen Installation, die Datei *jamaica.conf*<sup>f13</sup>, im *etc* Verzeichnis von Jamaica, an den folgenden 3 Positionen angepasst werden.

- *include.rtems-i386*
- *XCFLAGS.rtems-i386*
- *XLDFLAGS.rtems-i386*

Hinweis:

RTEMS wird im Moment noch nicht in der aktuellen Version 2.6.3 der JamaicaVM unterstützt. Es muss daher auf die Version 2.4.4 zurückgegriffen werden.

### 3.1.5 Jamaica Toolset

Das eigentliche Herzstück des Programmiergeräts stellt aber das Jamaica Toolset dar. Dieses setzt sich aus der JamaicaVM, dem Jamaica Builder, dem Plug-in für eclipse und eine Reihe kleiner Tools zusammen. Es wird von der aicas GmbH aus Karlsruhe entwickelt und wurde mit dem *embedded Award 2005* ausgezeichnet.



#### 3.1.5.1 JamaicaVM

Bei der JamaicaVM handelt es sich um eine Umsetzung der *Java Virtual Machine Specification*, inklusive der Erweiterung durch die RTSJ. Sie ist Laufzeitumgebung für Applikationen die für das Java 2 Environment geschrieben wurden. Wie bei einer gewöhnlichen JVM gliedert sich eine Anwendung (blau) in einen Teil, der nur Java, und einen Optionalen, der native Code, mittels JNI, nutzt. Um Garantien bezüglich des Zeitverhaltens geben zu können, ist die JamaicaVM auf ein Echtzeitbetriebssystem (grau) angewiesen.

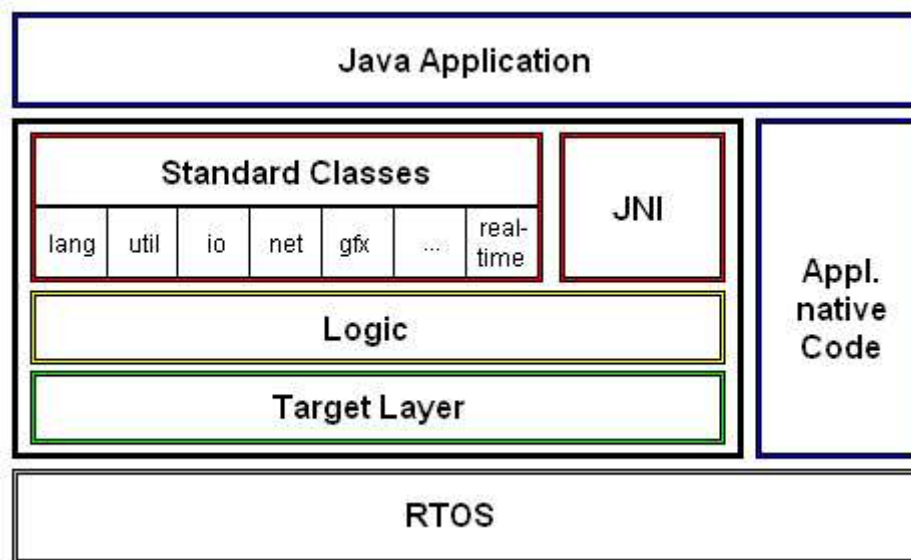


Abbildung 38: Architektur der JamaicaVM

Der Aufbau der eigentlichen JVM ist dabei an eine 3-Schichten Architektur angelehnt, wobei man die Schnittstelle zu den Anwendungen, den plattformunabhängigen Teil der VM und die Schnittstelle zum RTOS unterscheiden kann.



## Applikation Interface

Applikationen greifen auf die Funktionalität dieser Schnittstelle zurück um von den Vorteilen einer VM zu profitieren.

### Library

Dazu steht, wie es bei Java üblich ist, eine standardisierte und umfangreiche Klassenbibliothek zu Verfügung. Die der JamaicaVM ist allerdings erst auf dem Stand des J2SDK 1.3, wobei die Unterstützung von Sound, Grafik und CORBA noch komplett ausstehen. Ebenfalls ist die API der RTSJ noch nicht vollständig implementiert. Hier fehlen noch die Klassen *ImmortalPhysicalMemory*, *LTPhysicalMemory* und *VTPhysicalMemory*. Im Moment wird gerade an einer Umsetzung von *javax.swing.\** gearbeitet, doch der Fokus liegt auf den im Embedded-Bereich notwendigen Modulen und Klassen. Aus diesem Grund wird auch das package *java.lang.\** fast vollständig in Version 1.4 unterstützt.

### JNI

Des weiteren hat der Entwickler die Möglichkeit mit Hilfe von JNI, das entsprechend der J2SDK V1.2 unterstützt wird, native Code aus Java heraus aufzurufen. Damit können bestehende Treiber und Bibliotheken weiter genutzt werden, so dass die Umstellung auf Java geringeren Aufwand bedeuten würde, oder kritische Abschnitte in einer anderen Programmiersprache, beispielsweise C oder Assembler, umgesetzt werden. Dabei zeichnet sich die JamaicaVM dadurch aus, dass die GC, durch die Präsenz von native Code, nicht beeinflusst wird und weiter exakt arbeitet.

## Logic

Dieser Teil (gelb) der JamaicaVM bietet das eigentliche JRE an, wobei er nur die Funktionalität, die im die Target Layer bietet, nutzt, so dass sie selbst unabhängig von der Plattform ist. Das hat den Vorteil, dass die VM weniger fehleranfällig ist, da diese Komponenten in allen Systemen verwendet werden.

### Memory Management

Zu ihren Aufgaben zählt unter anderem die automatische Speicherverwaltung. Das Besondere an der JamaicaVM ist, dass dabei ein echtzeitfähiger GarbageCollector zum Einsatz kommt. Damit kann der Entwickler von Anwendungen für harte Echtzeitsysteme auch von den Vorzügen, die eine automatische Speicherverwaltung bietet, profitieren. Der eingesetzte Algorithmus leitet sich dabei von dem in herkömmlichem Java verwendeten *Mark and Sweep* Algorithmus ab.

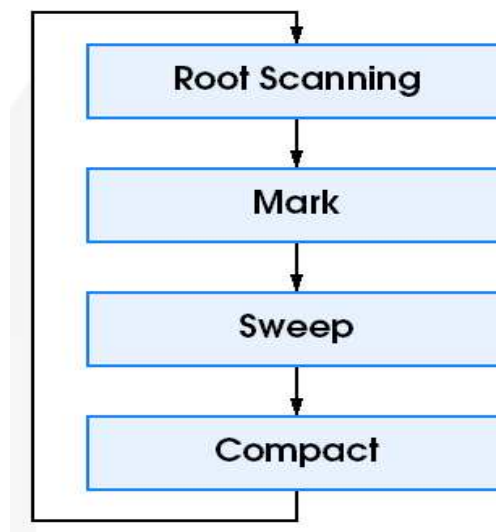


Abbildung 39: Mark and Sweep Algorithmus  
[JVM S.9]

- 1) Root Scanning  
Beim Root Scanning werden die aktiven Objekte ermittelt, indem man ausgehend von einem Wurzel-Verzeichnis alle Referenzen abläuft.
- 2) Mark  
Mark kennzeichnet alle referenzierten Objekte.
- 3) Sweep  
Sweep setzt die Markierung in den Objekten zurück und gibt den Speicher der unreferenzierten Objekte wieder frei.
- 4) Compact  
Durch Compact wird der Speicher defragmentiert. Das bedeutet, dass die Objekte so verschoben werden, dass der belegte Speicher einen zusammenhängenden Block bildet.

Dieser Algorithmus kann, wie bereits besprochen, harten Echtzeitanforderungen nicht standhalten, da er weder unterbrochen noch gesteuert werden kann. Somit finden Aussagen, die das Zeitverhalten der Threads beschreiben, in einem solchen System nicht die nötige Beachtung.

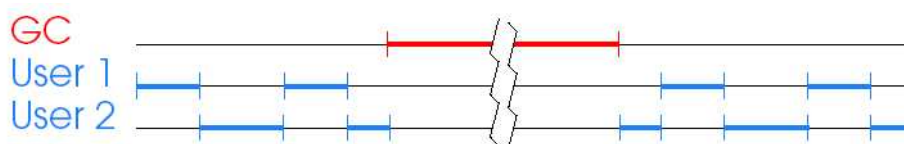


Abbildung 40: GC in Java [JVM S.6]

Um auf Basis eines solchen Algorithmus dennoch Echtzeitgarantien geben zu können, müsste man, wie es anfänglich in der RTSJ gemacht wurde, einfach Speicherbereiche schaffen, die von der GC ausgeschlossen sind. Diese beheimaten jene Echtzeitprozesse, die selbst die GC unterbrechen können.

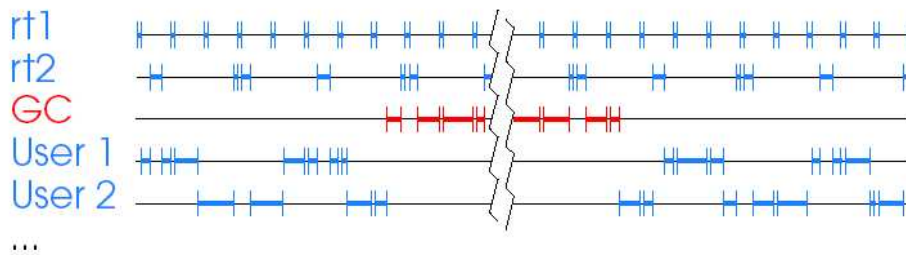


Abbildung 41: GC in der RTSJ [JVM S.7]

Dieses Verfahren hat allerdings zwei Nachteile. Zum einen muss strikt zwischen dem Echtzeit-Teil und dem normalen Java-Teil unterschieden werden und zum anderen geht damit die automatische Speicherverwaltung, welche ein herausragendes Merkmal von Java ist, im Echtzeit-Teil verloren.

Daher beschäftigt man sich schon seit längerem mit neuen Algorithmen, die den folgenden Anforderungen von Echtzeitsystemen gerecht werden.

- 1) Für die GC gelten, wie für jede andere Operation auch, deadlines
- 2) Die GC muss in dieser Zeit erfolgreich abgeschlossen werden
- 3) Alle Operationen, auch die GC, müssen sehr schnell unterbrochen werden können
- 4) GC sollte sehr schnell sein ( $\mu\text{sec}/\text{nsec}$ )

Die JamaicaVM nutzt daher für ihre RTGC den folgenden optimierten Algorithmus.

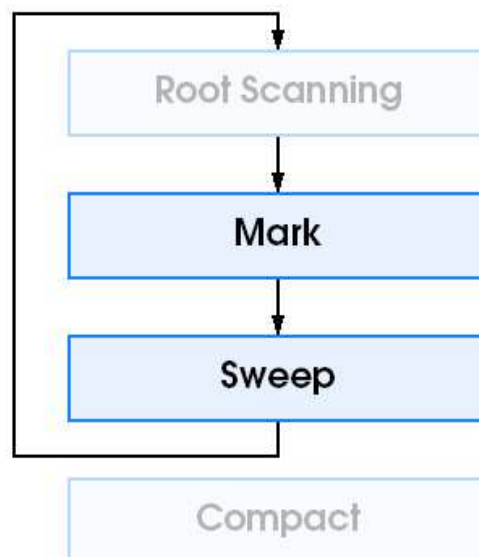


Abbildung 42: Verbesserter Mark and Sweep Algorithmus [JVM S.10]

### 1. Root Scanning

Das Root Scanning, dass in der RTGC die meisten Probleme bereitet, kann zwar nicht gänzlich entfallen, doch das aufwändige rekursive abschreiten jeder Referenz wurde durch eine automatische Verwaltung aktiver Objekte ersetzt. So werden in der JamaicaVM Referenzen auf alle aktiven Objekte zusätzlich im Heap gespeichert. Damit entfällt dieser Schritt während der eigentlichen GC.

### 2. Mark & Sweep

Auch die Phasen *Mark* und *Sweep* wurden in der JamaicaVM optimiert. So hält man an dem 3-Farben Schema, bei dem man zwischen aktiven, inaktiven und ungeprüften Blöcken unterscheidet, fest. Aber trägt gleichzeitig dafür Sorge, dass nicht mehr der gesamte Speicher der VM aufgeräumt wird. Stattdessen stellt er nur so viel Speicher bereit, wie angefordert wurde. Damit wird die GC in viele kleine Schritte unterteilt und erst bei einer Speicheranforderung aktiviert.

### 3. Compact

Zusätzlich kann die Phase *Compact* entfallen, da es zu keiner Fragmentierung mehr kommen kann. Dies hängt damit zusammen, dass die Speicherverwaltung nicht mehr direkt mit absoluten Adressen, sondern nur noch mit Blöcken (i.d.R. 32 byte) arbeitet.

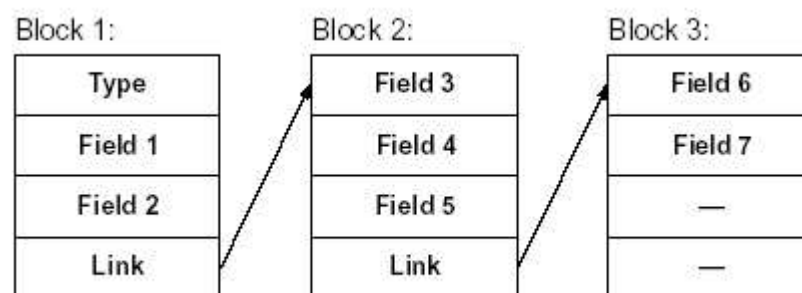


Abbildung 43: Objekte im Speicher der JamaicaVM [RTGC S.11]

Dabei belegt jedes Objekt mindestens einen solchen Block. Alle weiteren Blöcke werden, wie es von Listen her bekannt ist, einfach mit dem Vorgänger verlinkt. Damit wird erreicht, dass die Objekte nicht mehr am Stück im Speicher liegen müssen.

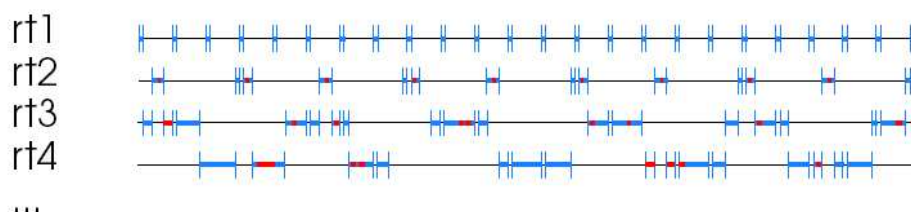


Abbildung 44: GC der JamaicaVM [JVM S.8]

Mit diesem Algorithmus ist es möglich für alle Bereiche Echtzeitgarantien zu geben, da man nun zeitliche Obergrenzen für die GC bestimmen kann. Diese sind allerdings von einer Reihe von Faktoren abhängig. Dazu zählen beispielsweise die Größe des Heap, die Anzahl der angeforderten Blöcke, die Plattform abhängigen max. Dauer für einen GC Schritt und viele mehr. Einige dieser Faktoren lassen sich im Zuge der Optimierung, der ein eigenes Kapitel gewidmet ist, bei der JamaicaVM einstellen.

### Sonstige

Die automatische Speicherverwaltung ist aber nur ein Teil des JRE. Zu ihren Aufgaben zählt auch noch das Überwachen der Ausführung, um beispielsweise das Überlaufen eines Wertes oder den Zugriff auf einen ungültigen Index eines Arrays zu verhindern. Ebenso muss der Byte-Code interpretiert und unter Umständen dynamisch geladen werden. Alle diese Aufgaben werden in traditioneller Weise von der VM übernommen, so dass nicht näher darauf eingegangen wird.

### **Target Layer**

Die Target Layer (grün) kapselt, wie der Name schon andeutet, die plattformabhängigen Teile der JVM. Diese stellt die Basisfunktionalität bereit, indem sie die spezifische Funktionen des jeweiligen RTOS nutzt.

Zu ihren Aufgaben gehört:

- Abbilden der Java-Threads auf Threads der Plattform
- Schnittstelle zu den native Libraries und Treibern
- Abbilden der Datentypen
- ...

Zum Vorteil gereicht diese Architektur, falls man die JamaicaVM portieren möchte, da man nur diese Schicht ersetzen muss.

### 3.1.5.2 Jamaica Builder

Das zweite wichtige Mitglied des Jamaica Toolsets ist der Jamaica Builder. Dieser baut aus der JamaicaVM und den für die Applikation notwendigen Klassen ein unabhängiges Executable. Das ist insbesondere für Embedded-Systems von Interesse, da diese nur über geringen Speicher verfügen und unter Umständen kein geeignetes File-System bereitstellen. Dass die Möglichkeit, Klassen dynamisch zu Laden, auf diese Weise verloren geht, wird in diesem Zusammenhang nicht als Nachteil gesehen, da es zu Lasten der Performance geht.

#### Arbeitsweise

Die Arbeit des Jamaica Builder lässt sich dabei in 3 Schritte untergliedern. Dabei werden auch Standard GNU Tools – Linker und Compiler – eingesetzt, was zu mehr Sicherheit führt, da diese sehr ausgereift sind.

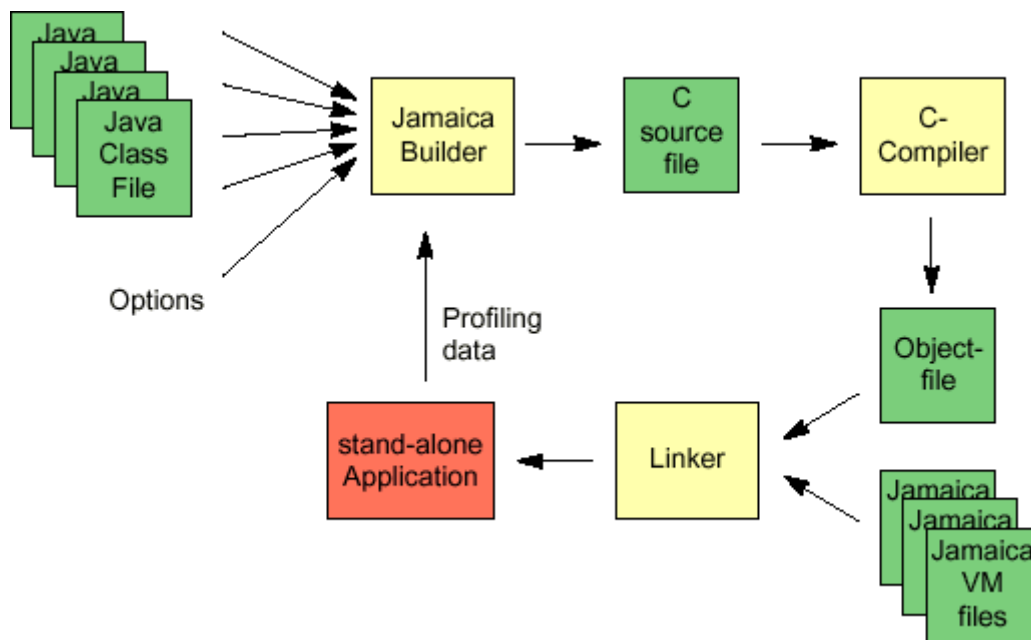


Abbildung 45: Einsatz des Jamaica Builder [JVM S.14]

1. Jamaica Builder  
Der Jamaica Builder nimmt die class-Files, die für die Applikation notwendig sind, her und generiert daraus ein plattformunabhängiges C-File.
2. C-Crosscompiler  
Ein native C-Crosscompiler erzeugt daraus ein Object-File für die entsprechende Zielplattform
3. Linker  
Der Linker erzeugt aus diesem Object-File und den teilweise plattformspezifischen JamaicaVM-Files das Executable. Dieses enthält alle notwendigen Daten um das Java Programm auszuführen.

Hinweis:

Im Falle von RTEMS werden auch noch die entsprechenden Files des OS mit eingebunden.

## Optimierung

Um den harten Anforderungen, die man an Anwendungen für den Embedded-Bereich stellt, gerecht zu werden, besteht die Möglichkeit diesen Prozess mit einer Reihe an Parametern zu beeinflussen und an die eigenen Bedürfnisse anzupassen.

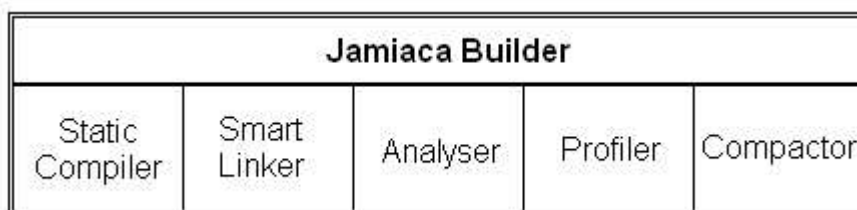


Abbildung 46: Tools für die Optimierung

### Smart Linker & Compactor

Mit dem Smart Linker und dem Compactor lässt sich der Speicherbedarf, der für das Executable benötigt wird, reduzieren. Da die JamaicaVM nur ~128kB benötigt, wird der meiste Speicher von den class-Files belegt. Aus diesem Grund setzen beide Tools auch an dieser Stelle an. Während der Compactor, der generell aktiviert ist, den Byte-Code optimiert und die Größe aller vorhandenen class-Files um mehr als 50% reduziert, muss der Smart Linker explizit aktiviert werden. Dieser sorgt dafür, dass die ungenutzten class-Files aussortiert werden. Auf diese Weise lassen sich sogar Einsparungen von bis zu 90% erzielen.

Hinweis:

Der Einsatz des Smart Linker ist allerdings nicht ganz ohne Risiken. Das hängt damit zusammen, dass er Klassen, die nur indirekt eingebunden werden, auch aussortiert. Um dennoch von den Vorteilen des Smart Linker Gebrauch machen zu können, besteht die Möglichkeit Klassen, die andere indirekt nutzen, mit Hilfe des Parameters `-notSmart=<classes>` aus dieser Optimierung herauszunehmen.

### Byte-Code Interpretation

Java erzeugt beim Compilieren einen so genannten Byte-Code. Dieser muss zu einem späteren Zeitpunkt in Maschinenbefehle umgesetzt werden. Dabei ist zu berücksichtigen, dass der Byte-Code kompakter als die entsprechenden Maschinenbefehle ist, während deren Übersetzung zur Laufzeit zu Lasten der Geschwindigkeit geht.

Folgende Techniken werden zur Zeit verwendet:

1. Interpreter  
Der Interpreter setzt den Byte-Code erst zur Laufzeit in die entsprechenden Maschinenbefehle um
2. Static Compiler  
Der Static Compiler setzt den Byte-Code direkt, beim Übersetzen aus dem Quellcode, in Maschinenbefehle um. Dabei geht allerdings die Plattformunabhängigkeit verloren
3. Just-in-Time Compiler (JIT-Compiler)  
Ein JIT-Compiler übersetzt vor der ersten Ausführung des Programms den Byte-Code

Alle diese Techniken offenbaren vor allem in performancekritischen Echtzeitsystemen ihre Schwächen. Während von einem JIT-Compiler auf Grund des problematischen Zeitverhaltens zu Beginn abgerückt werden muss, ist der reine Interpreter einfach zu langsam und der Code, der von einem reinen Static Compiler erzeugt wird, einfach zu groß. Aus diesem Grund setzt man in der JamaicaVM auf eine Mischung zwischen Interpreter und Static Compiler.

### Manuelle Konfiguration

Zur manuellen Konfiguration und Optimierung einer Applikation, kann der Entwickler dem Jamaica Builder folgende Parameter übergeben:

1. *-compile*  
Der Parameter *-compile* sorgt dafür, dass der Compiler sich wie ein reiner Static Compiler verhält und daher alle Klassen und Methoden direkt in Maschinenbefehle übersetzt.
2. *-optimize=size* bzw. *speed*  
Mit diesem Parameter kann dem C-Compiler mitgeteilt werden, ob die Größe der Applikation oder die Ausführungsgeschwindigkeit wichtiger ist. Die resultierenden Optimierungen sind Compiler spezifisch.
3. *-inline=<n>*  
Mit Hilfe des Parameters *-inline=<n>* lässt sich einstellen, in welchem Umfang Aufrufe von Methoden durch den entsprechenden Methodenrumpf ersetzt werden. Auf diese Weise lässt sich die Ausführungsgeschwindigkeit auf Kosten des Speicherbedarfs erhöhen. Dabei werden 11 Stufen, von 0 (kein inlining) bis 10 (aggressives inlining), unterschieden.



### Automatische Konfiguration

Da aber statistische Beobachtungen gezeigt haben, dass Code, bei dem 10% der Klassen statisch kompiliert sind, fast das gleiche Laufzeitverhalten aufweist, wie Code der zu 100% statisch kompiliert wurde und es viel Erfahrung benötigt die kritischen Methoden zu finden, bietet die JamaicaVM den Profiler an. Dieses Tool ermittelt wie viele Byte-Code Instruktionen von jeder Methode ausgeführt wurden. Damit lassen sich Rückschlüsse ziehen wie lange man sich während einer Ausführung in einer Methode aufgehalten hat. Auf Basis dieser Informationen kann die Applikation später automatisch optimiert werden.

Der Einsatz des Profiler erfolgt dabei in 3 Schritten:

#### 1. Bau einer Profiler-Applikation

Als erstes muss eine Applikation, die später vom Profiler untersucht wird, gebaut werden. Zu diesem Zweck startet man den Builder mit dem Parameter *-profile*.

```
jamaica -smart -numThreads=10 -profile TimedExample
```

#### 2. Ausführen der Profiler-Applikation

Wird nun die unter Punkt 1 erstellte Applikation ausgeführt läuft automatisch der Profiler mit. Das Ergebnis wird in einer Datei (\*.prof) gespeichert.

```
./TimedExample
```

#### 3. Optimieren der Applikation

Auf Basis der Daten, die während den Ausführungen gesammelt wurden, kann die Applikation nun optimiert werden. Zu diesem Zweck bindet man die gesammelten Informationen mittels *-useProfile=<profilelist>* ein. Optional kann noch die Anzahl der Methoden, die statisch kompiliert werden sollen, mit Hilfe von *-percentageCompiled=<n>* angegeben werden.

```
jamaica -smart -numThreads=10 -useProfile=TimedExample.prof TimedExample
```

Als Ergebnis dieser Optimierung erhält man eine Applikation die sich durch ein sehr gutes Laufzeitverhalten bei einem leicht erhöhten Speicherbedarf auszeichnet.

### Analyser

Bei vielen Embedded-Systems ist allerdings der Speicherbedarf im RAM, während der Ausführung, viel wichtiger als die Ausführungsgeschwindigkeit und der Speicherbedarf für die Sourcen. Daher bietet die JamaicaVM den Analyser an. Mit diesem Tool lässt sich der RAM Bedarf einer Applikation in Abhängigkeit von der max. Verzögerung durch die RTGC ermitteln. Diese ist nämlich an den verfügbaren Heap geknüpft, so dass man auch hier vor einem Dilemma steht.

Der Einsatz des Analyser erfolgt dabei in 4 Schritten:

#### 1. Bau einer Analyser-Applikation

Als erstes muss eine Applikation, die vom Analyser untersucht werden soll, gebaut werden. Zu diesem Zweck wird der Jamaica Builder mit der Option `-analyse=<n>` aufgerufen. Wobei *n* die Genauigkeit des Speicherbedarfs in % angibt.

```
jamaica -smart -numThreads=10 -analyse=1 TimedExample
```

#### 2. Ausführen der Analyser-Applikation

Wird nun die unter Punkt 1 erstellte Applikation ausgeführt, dann läuft automatisch der Analyser mit. Dessen Ergebnis wird nach dem Ende der Applikation auf der Standard Ausgabe angezeigt.

```
./TimedExample
```

#### 3. Auswahl der Konfiguration

Der Analyser liefert eine Tabelle an möglichen Konfigurationen. Aus dieser kann sich der Entwickler die für seine Bedürfnisse geeignetste herausuchen.

### Application used at most 1242053 bytes for the Java heap (accuracy 1%).

### Non-Java heap memory used: 214847 bytes.

###

### Worst case allocation overhead:

heapSize	dynamic GC	const GC work
4407k	6	3
3716k	7	4
3273k	8	4
2973k	9	4
2753k	10	4
...		
1740k	32	10
...		
1522k	96	27
1497k	128	36
1471k	192	53
1459k	256	69
1448k	384	100

*Text 1: Ausgabe des Analyser*

Diese Tabelle ist folgendermaßen aufgebaut:

- Java heap  
Gibt die maximale Größe des Heap, die während der Nutzung des Analyser benötigt wurde, an.
- Non-Java heap  
Gibt den maximalen Speicherbedarf außerhalb des Heap, der während der Nutzung des Analyser benötigt wurde, an. Dieser setzt sich unter anderem aus dem Speicher für die VM und den native Methoden zusammen.
- heapSize  
Hier werden die möglichen Größen für den Heap, die von der JamaicaVM angeboten werden, aufgelistet.
- dynamicGC  
Gibt an wie viele GC Schritte, bei dynamischer GC, im schlimmsten Fall ausgeführt werden müssen um einen freien Block bereitstellen zu können.
- const GC  
Gibt an wie viele GC Schritte, bei konstanter GC, ausgeführt werden um einen freien Block bereitzustellen.

Hinweis:

Unter dynamische GC versteht man, dass diese nur so lange aktiv ist, bis sie die geforderte Anzahl an Blöcken bereitstellen kann. Aus diesem Grund ist sie in der Regel sehr schnell. Die konstante GC führt dagegen immer die angegebene Anzahl an Schritten durch. Daher werden durchschnittlich mehr GC Schritte gemacht, als eigentlich notwendig wären. Allerdings ist dadurch die Zeit, die für die GC benötigt wird, relativ konstant und die Anzahl der GC Schritte, die im schlimmsten Fall ausgeführt werden müssen, ist geringer als bei der Dynamischen.

Um sich nun die geeignete Konfiguration in der Tabelle zu finden, geht man wie folgt vor:

Fall a) Limitierter RAM

In diesem Fall steht der Applikation nur begrenzt RAM zur Verfügung. Um diesen möglichst effizient auszufüllen, muss die maximale Größe des Heap bestimmt werden. Dazu muss man von dem verfügbaren RAM den Speicherbedarf außerhalb des Heap abziehen.

$$\text{RAM} - \text{NoHeap} = \text{Heap}$$

- RAM  
RAM der für diese Anwendung zu Verfügung steht
- NoHeap  
Speicher außerhalb des Heap, der beispielsweise von der VM und eventuellen JNI Aufrufen belegt wird
- Heap  
Größe des Heap, der maximal für diese Anwendung zur Verfügung steht

Beispiel:

Steht für die Applikation TimedExample maximal 3,5 MB RAM zu Verfügung, dann sollte für die Größe des Heap 3273k eingestellt werden, da außerhalb des Heap ca. 210k benötigt werden.

#### Fall b) Maximales RTGC Fenster

In diesem Fall stellt die maximale Verzögerung durch die RTGC den limitierenden Faktor dar. Daher muss zunächst die max. Länge eines GC Schritts dieser Plattform ermittelt werden. Aus diesen beiden Werten folgt dann die maximale Anzahl an GC Schritten die der RTGC zur Verfügung stehen. Hat man diesen Wert berechnet, sucht man in der Tabelle die entsprechende minimale Größe des Heap heraus.

$$\text{MaxDelay} / \text{MaxRTGCStep} = \text{MaxSteps}$$

- **MaxDelay**  
Die maximale Verzögerung der Ausführung durch die RTGC
- **MaxRTGCStep**  
Die maximale Dauer eines RTGC Schritts
- **MaxSteps**  
Die maximale Anzahl an RTGC die gemacht werden dürfen

#### Beispiel:

Um für die Applikation TimedExample eine maximale Verzögerung von 16 µs zu garantieren, können bei 1,6 µs pro Schritt max. 10 GC Schritten durchgeführt werden. Aus der Tabelle folgt daraus, dass bei dynamischer GC ein Heap von min. 2753kB und bei konstanter GC von min. 1740kB bereitgestellt werden muss.

#### Hinweis:

Die Rechnung gilt nur für Objekte, die einen Block belegen. Sollten mehrere benötigt werden, dann ist die RTGC entsprechend länger.

#### Fall c) Limitierter RAM und Maximales RTGC Fenster

Ist man bezüglich beider Faktoren eingeschränkt, fällt die Suche in der Liste komplizierter aus, da es nun zwei Grenzen gibt. Diese lassen sich aber unabhängig von einander, entsprechend der Formeln aus a) und b) ermitteln. Geeignet sind in diesem Fall alle Konfigurationen, die innerhalb dieser Grenzen liegen.

#### Hinweis:

Sollte keine der Konfigurationen den Anforderungen genügen, da der verfügbare RAM nicht ausreicht, dann sollte man als erstes Versuchen den eigenen Code zu optimieren, um weniger Objekte zu benötigen. Ist dies nicht möglich, kann man noch unnötige Libraries aus der VM entfernen, die Größe der Stacks reduzieren oder die Anzahl der System-Threads vermindern. Diese Verfahren werden alle in der Dokumentation der JamaicaVM beschrieben, sollten aber nur im Notfall eingesetzt werden.

Beispiel:

Um für die Applikation TimedExample eine Verzögerung die kleiner ist als 16  $\mu$ s zu garantieren, muss, bei konstanter GC und einem Bedarf von höchstens 3,5 MB RAM, der Heap zwischen 1740kB und 3273kB liegen.

4. Bau der Applikation

Hat man eine geeignete Konfiguration gefunden, dann baut man mit Hilfe dieser Werte einfach die optimierte Applikation.

```
jamaica -smart -numThreads=10 -heapSize=1740k -constGCwork=10 TimedExample
```

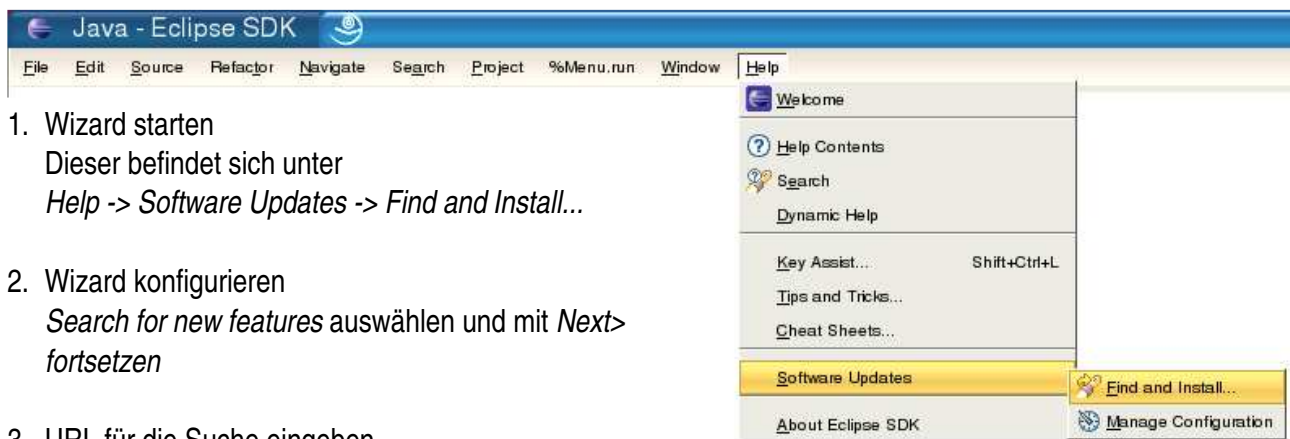
Als Ergebnis dieser Optimierung erhält man eine Applikation, bei der sowohl der max. Bedarf an RAM als auch die max. Verzögerung durch die RTGC bekannt ist.

### 3.1.5.3 eclipse Plug-in

Um ein effektives Arbeiten mit diesen Tools zu gewährleisten, wird von der aicas GmbH ein Plug-in für eclipse angeboten. Damit genießt man zum einen den Komfort einer ausgereiften Entwicklungsumgebung für Java und zum anderen ist es möglich die Tools grafisch zu konfigurieren, ihre Einstellungen zu verwalten und sie per Knopfdruck zu starten.

#### Installation

Die Installation<sup>14</sup> gestaltet sich relativ einfach, da eclipse diese automatisch vornimmt und man von einem wizard begleitet wird.



1. Wizard starten  
Dieser befindet sich unter  
*Help -> Software Updates -> Find and Install...*
2. Wizard konfigurieren  
*Search for new features* auswählen und mit *Next>*  
fortsetzen
3. URL für die Suche eingeben  
*Add Update Site* auswählen und wie im Bild ausfüllen
4. Plug-in suchen  
Auswahl der JamaicaVM für die das Plug-in  
verwendet wird und weiter mit *Next>*
5. Plug-in auswählen  
Das Plug-in auswählen und weiter mit *Next>*
6. Plug-in installieren  
Den Anweisungen des wizards folgen



Abbildung 47: Automatische Plug-in Installation – URL

#### Hinweis:

Alternativ kann dieses Plug-in auch manuell installiert werden. Im Gegensatz zu der Beschreibung in der JamaicaVM – User Documentation, kann man es aber nicht einfach im Installations-Verzeichnis von eclipse entpacken, sondern wird gezwungen es extern zu tun. Die darin enthaltenen Ordner *features* und *plugins* sollten nun nacheinander in *com.aicas.jamaica.eclipse\_2\_6\_0* umbenannt und in den entsprechenden Ordner *features* bzw. *plugins* von eclipse kopiert werden.

<sup>14</sup> JamiacaVM – User Documentation S.99

*eclipse* sollte bei einem Neustart, unabhängig von der Vorgehensweise, das Plug-in automatisch erkennen und ein entsprechendes Fenster einblenden.



Anschließend muss noch der Pfad zur JamaicaVM gesetzt werden.  
Dies ist unter *Window -> Preferences -> JamaicaVM Builder* möglich.

### Hinweis:

Zusätzlich kann noch die JamaicaVM als default JRE eingerichtet werden. Das hat den Vorteil, dass man unter Linux Applikationen, die RTSJ nutzen, direkt aus *eclipse* heraus starten kann. Dazu muss lediglich die JamaicaVM unter *Window -> Preferences -> Java -> Installed JREs* ausgewählt werden. Allerdings kann es bei herkömmlichen Applikationen zu Schwierigkeiten kommen, da die JamaicaVM noch nicht alle packages wie eine StandardVM unterstützt.



## Einsatz

Nachdem die Entwicklungsumgebung an die JamaicaVM angepasst ist, kann direkt mit der Arbeit begonnen werden. Dabei wird zunächst wie gewohnt in Java entwickelt, wobei man die API der JamaicaVM heranzieht. Die Applikationen können, wenn man die JamaicaVM als JRE eingerichtet hat, wie gewohnt auf dem Programmiergerät gestartet werden. Wenn man allerdings auf den Jamaica Builder zurückgreifen möchte, kann man in eclipse die beiden neuen Menüpunkte nutzen. Da das Erstellen einer Applikation unter Umständen relativ lange dauert, wird zwischen dem Bau und der eigentlichen Ausführung unterschieden.

## Konfiguration & Bau



Der Jamaica Builder ist ein sehr mächtiges Tool, das man über eine Vielzahl an Parametern zu konfigurieren kann. Damit man diese Parameter nicht wie bei der Konsole manuelle verwalten muss, bietet das Plug-in eine solche Verwaltung an. Diese ist über den Menüpunkt JamaicaVM Builder, der durch die Palme gekennzeichnet ist, zugänglich. Während das Anlegen, Löschen und Ausführen solcher Konfigurationen sehr übersichtlich und intuitiv gestaltet ist, kann man sich bei den Einstellungen, zumindest zu Beginn, doch etwas verloren vorkommen. Dies ist aber nur auf die Vielzahl der Parameter zurückzuführen. Um den Einstieg zu erleichtern wird hier auf die wichtigsten Einstellungen und die Anordnung eingegangen. Detailliert werden die einzelnen Parameter in der JamaicaVM – User Documentation<sup>15</sup> beschrieben.



Abbildung 48: Einstellungen - Jamaica Builder

Zunächst wird zwischen den einzelnen Zielplattformen unterschieden:

- global  
Hier können Einstellungen, die bei allen Zielplattformen gleich sind, vorgenommen werden.
- host  
Hier können Einstellungen, die nur für das Programmiergerät gelten, vorgenommen werden.
- linux-gnu-i686  
Hier können Einstellungen, die nur für die entsprechende Plattform gelten, vorgenommen werden. In diese Fall handelt es sich um Linux auf einem Standard-PC.

<sup>15</sup> JamaicaVM – User Documentation – S.11ff

Für alle diese Plattformen können, mit Ausnahme von General, Einstellungen in den darunter liegenden Kategorien vorgenommen werden. Dabei Überschreiben diejenigen die in einem Target vorgenommen werden jene, die in global festgelegt wurden:

- General  
General, das nur für global verfügbar ist, dient dazu das Projekt und die Klasse mit der *main(String[])*-Methode zu definieren. Darüber hinaus wird hier eingestellt für welche Plattform die Applikation gebaut wird, wenn man diese Konfiguration startet.
- Classes, files and paths  
Diese Einstellungen beschäftigen sich alle mit Pfadangaben. Es besteht unter anderem die Möglichkeit Jar-Files und Klassen zu importieren, den Pfad in das der Output erfolgt zu definieren oder eine Bibliothek zu nutzen.
- Smart linking  
Hier können alle Einstellungen, die für den Smart Linker von Bedeutung sind vorgenommen werden.
- Compilation  
Alle manuellen Compiler-Einstellungen, wie die Aktivierung von *inline* oder *compile*, können hier vorgenommen werden.
- Memory and threads  
Diese Kategorie bietet die Möglichkeit eine Reihe von Einstellungen, die sich vor allem mit dem verfügbaren Speicher befassen, vorzunehmen. Während die meisten Parameter aller Kategorien nur, wenn man sie explizit nutzen möchte, gesetzt werden, gibt es hier einige, die bei jeder Anwendung angetastet werden müssen.
  - ➔ numThreads  
Anzahl der Threads die maximal zum Einsatz kommen. Sollten hier zu wenige eingetragen sein, dann kann die Anwendung nicht ausgeführt werden. Dabei ist darauf zu achten, dass die JamaicaVM einen eigenen Thread als Finalizer einführt.
  - ➔ strictRTSJ  
Gibt an ob sich die JamaicaVM wie eine herkömmliche Implementierung der RTSJ verhält, oder ob man auf Grund der RTGC von diesem Verhalten abweichen und beispielsweise auch mit einem *NoHeapRealtimeThread* den Heap nutzen darf.
  - ➔ physicalMemoryRanges  
Speicherbereiche auf die direkt zugegriffen wird müssen hier eingetragen werden. Diese Option ist allerdings nur in den Targets und nicht in global möglich.
- Profiling  
Hier können alle Einstellungen, die mit dem Profiler zusammenhängen, vorgenommen werden.

- **Native Code**  
Hier müssen die native implementierten Methoden als Object-File übergeben werden. Diese Option wird ebenfalls nur in den Targets angeboten.
- **Analyzing**  
Hier können alle Einstellungen, die mit dem Analyser zusammenhängen, vorgenommen werden.
- **Miscellaneous**  
Alle sonstigen Einstellungen können hier vorgenommen werden.

### Ausführung

Das Menü für die Ausführung der, mit dem Jamaica Builder generierten, Applikation ist ähnlich wie das der Konfiguration aufgebaut. Auch hier wird die Möglichkeit angeboten Konfigurationen direkt zu starten, während die Einstellungen unter *JamaicaVM Target Site...* vorgenommen werden können.

Wie bei der Konfiguration ist auch hier die Verwaltung intuitiv und übersichtlich gestaltet. Des weiteren hat man die Möglichkeit die Plattform auf der die Anwendung ausgeführt wird in *Main* einzustellen. Falls man die Anwendung nicht auf dem Programmiergerät ausführen möchte, wird man hinsichtlich der Übertragung und der Ausführung unterstützt.

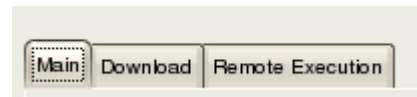


Abbildung 49: Einstellungen - Target Site

### **3.1.5.4 Additional**

Neben diesen wichtigen Tools werden noch eine Reihe Kleinerer angeboten. Diese finden allerdings nur in speziellen Situationen Verwendung.

- **numblocks**  
Mit Hilfe von *numblocks* lässt sich die Anzahl der Blöcke, die ein Objekt belegen wird, exakt bestimmen. Es kommt daher in Verbindung mit dem Analyser zum Einsatz.
- **jamaicah**  
*jamaicah* wird beim Einsatz von JNI verwendet um die Signaturen für die Methoden, die native implementiert werden, zu erstellen.
- **JB1**  
Dabei handelt es sich um eine JamaicaVM spezifische Lösung, die es erlaubt, direkt auf native Code zuzugreifen. Man kann damit zwar höchst effizienten Code erstellen, da kein Overhead wie bei JNI anfällt. Das Arbeiten mit dieser Schnittstelle erfordert jedoch höchste Genauigkeit, da nun sensible Bereiche, wie die GC, vom Entwickler gehandhabt werden müssen.

## 3.2 Steuergerät

Während das Programmiergerät eine Reihe von Aufgaben erfüllt, wird das Steuergerät in der Regel nur zur Steuerung der Prozesse eingesetzt. Für diese Aufgabe wird es meist mit spezieller HW und SW ausgestattet. In unserem Fall kommt allerdings ein Standard-PC zum Einsatz, so dass sich das Steuergerät nur durch das spezielle RTOS und einen geeigneten Bootmanager auszeichnet.

### 3.2.1 RTEMS<sup>16</sup>

RTEMS steht für *real-time executive for multiprocessor systems*. Es handelt sich also um ein RTOS für den Embedded-Bereich, das von der OAR Corporation entwickelt und betreut wird.

#### 3.2.1.1 Geschichte

Es blickt dabei auf eine lange Vergangenheit zurück, wobei man bereits den Open-Source Gedanken unterstützt hat, bevor dieser so richtig in Mode gekommen ist. Ausgangspunkt war dabei ein Projekt für die U.S. Army Missile Command. Mit dem Ziel ein portables und auf Standards basierendes RTOS zu schaffen, dass, durch Veröffentlichung der Sourcen, größtmögliche Sicherheit bieten sollte.

*Abbildung 50: Geschichte des RTEMS*

---

<sup>16</sup> [www.rtems.com](http://www.rtems.com)

Schon bald wurde klar, dass RTEMS nicht nur für rein militärische Zwecke eingesetzt werden kann, sondern auch für die Industrie von Interesse ist. Der Einsatz in diesem Umfeld führte dazu, dass der Entwicklungsprozess vereinheitlicht, weitere Standards umgesetzt und neue Features eingeführt wurden.

### **3.2.1.2 Features**

Mittlerweile unterstützt RTEMS eine Reihe von freien API und Interface Standards – POSIX 1003.1b, ITRON und RTEID (pSOS+<sup>TM</sup>, VME-exec<sup>TM</sup>) – sowie die Programmiersprachen C/C++, Java und Ada95. Darüber hinaus ist es für 13 CPU Familien und über 60 BSP verfügbar.

Es zeichnet sich dabei durch folgende Eigenschaften<sup>17</sup> aus:

- Priority-based multitasking
- Preemptability controlled on a per-task basis
- Intertask communication and synchronization
- Responsive interrupt management
- Dynamic memory allocation
- Homogeneous and heterogeneous multiprocessor support
- Priority ceiling and inheritance support
- Rate monotonic scheduling
- Filesystem support
- Networking support (IP,TCP,...)

### **3.2.1.3 Architektur**

Das Besondere an RTEMS ist allerdings, dass es zusammen mit der Applikation zu einem Executable verbunden wird. Das bedeutet, dass man kein separates Betriebssystem hat auf dem verschiedenen Anwendungen laufen, sondern dass nur diese eine Anwendung laufen kann.

---

<sup>17</sup> Joel Sherrill, Sherril.pdf – S.4

## Applikation Architektur

Dabei entkoppeln RTEMS die Anwendung und die HW von einander. Zu diesem Zweck bildet die RTEMS grundlegende Funktionen auf die Plattform ab und bietet diese der Applikation an.

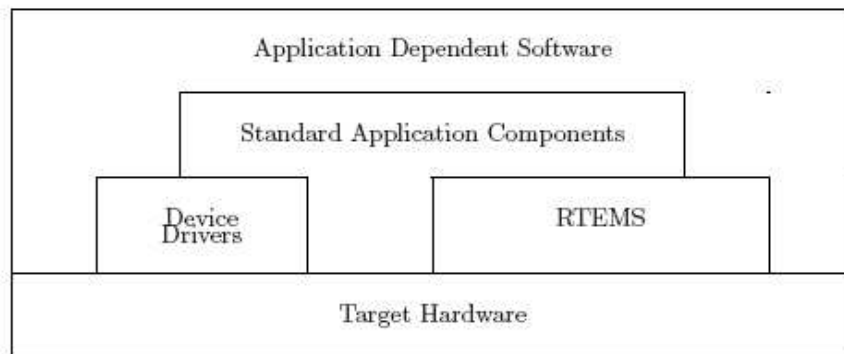


Abbildung 51: Aufbau einer RTEMS – Applikation [OAR\_1 S.7]

Diese können auch von eigens entwickelten Bibliotheken genutzt werden, so dass RTEMS auch die Wiederverwendbarkeit von Code fördert. Außerdem ermöglicht es dem Entwickler die plattformabhängigen Treiber separat zu kapseln. Um effizient auf diese zugreifen zu können wird der I/O-Interface-Manager angeboten. Der es erlaubt den Zugriff zu abstrahieren und zu vereinheitlichen, damit sich der Entwickler auf die eigentliche Anwendung konzentrieren kann.

## Internal Architektur

RTEMS als solches setzt sich dabei aus 2 Komponenten zusammen. Dem *executive core*, der die Grundlegende Funktionalität, wie beispielsweise das Scheduling, umsetzt und dabei auf die Plattform zugreifen muss, und dem *executive interface*, das dem Entwickler eine breites Spektrum an plattformunabhängigen Funktionen für die Echtzeitprogrammierung bereitstellt. Dieses gliedert sich in eine Reihe von teilweise optionalen *resource-managers*.

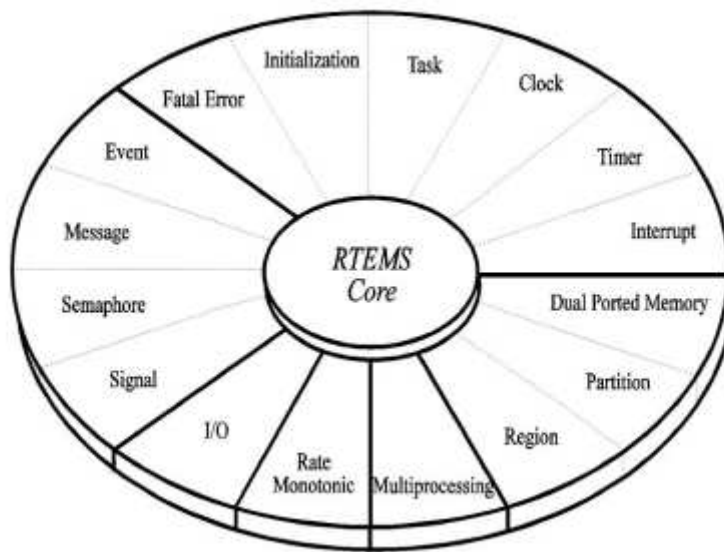


Abbildung 52: Architektur von RTEMS [OAR\_1 S.8]

- RTEMS Core  
Der *RTEMS Core*, auch *executive core* genannt, kapselt die grundlegende Funktionalität, die von den anderen Modulen genutzt wird. Dabei muss auf plattformspezifische Funktionen zurückgegriffen werden.
- Initialization  
Der *Initialization-Manager* ist verantwortlich für die Initialisierung und das Beenden von RTEMS
- Task  
Der *Task-Manager* bietet umfangreiche Verwaltungsmöglichkeiten für Tasks an.
- Interrupt  
Der *Interrupt-Manager* bietet einen Mechanismus an mit dem schnell auf externe Ereignisse reagiert werden kann.
- Clock (optional)  
Der *Clock-Manager* stellt die gesamte Funktionalität, die für den Umgang mit Zeitangaben notwendig ist, bereit.
- Timer (optional)  
Der *Timer-Manager* bietet die Möglichkeit Ereignisse zeitlich zu steuern. Er greift dabei auf das Modul Clock zurück.
- Semaphore (optional)  
Der *Semaphore-Manager* bietet Semaphore zur Synchronisation und für wechselseitigen Ausschluss an.
- Message (optional)  
Der *Message-Manager* ermöglicht Kommunikation und Synchronisation mittels Nachrichten und

Warteschlangen.

- Event (optional)  
Der *Event-Manager* ermöglicht eine sehr schnelle Kommunikation und Synchronisation von Tasks.
- Signal (optional)  
Der *Signal-Manager* ermöglicht asynchrone Kommunikation.
- Partition (optional)  
Der *Partition-Manager* bietet die Funktionalität, für das dynamische Belegen von Speicherbereichen fester Größe an.
- Region (optional)  
Der *Region-Manager* bietet die Funktionalität, für das dynamische Belegen von Speicherbereichen variabler Größe an.
- Dual Ported Memory (optional)  
Der *Dual-Ported-Memory-Manager* ermöglicht das Umrechnen von internen und externen DPMA-Adressen.
- I/O (optional)  
Der *Input/Output-Interface-Manager* bieten einen definierten Mechanismus an, mit dem auf Treiber zugegriffen werden kann. Außerdem ermöglicht er dessen Verwaltung.
- Fatal Error (optional)  
Der *Fatal-Error-Manager* ermöglicht es auf fatale oder unwiderrufliche Fehler zu reagieren.
- Rate Monotonic (optional)  
Der *Rate-Monotonic-Manager* bietet die Möglichkeit periodische Tasks einzurichten. Er greift dabei auf das Module Clock zurück.
- Multiprocessing (optional)  
Der *Multiprocessing-Manager* ist nur in Echtzeit-Systemen mit mehreren Prozessoren notwendig, da dort neuartige Probleme, wie beispielsweise der Datenaustausch zwischen den Prozessoren oder die Verwaltung globaler Ressourcen, auftreten.

Wie man sieht wurde großer Wert auf die Portabilität und Erweiterbarkeit gelegt. Daher lassen sich Plattform spezifische Komponenten wie Systemuhren, Interrupt-Controller oder I/O Geräte sehr leicht integrieren. Des weiteren ist es möglich eine breites Spektrum an Plattformen zu unterstützen, da alle abhängigen Komponenten in BSP gekapselt werden.



Hinweis:

Es wird ausdrücklich darauf hingewiesen, dass RTEMS nicht auf jedem PC läuft. Man vermutet dabei die Grafikkarte als Quelle des Problems.

Versuche haben ergeben, dass Applikationen, die mit dem BSP pck6 erstellt wurden, auf einem Chaintech NIL 7 mit AMD 2800+ und ATI 9700 ohne Ausnahme liefen. Wurden sie dagegen mit dem BSP pc686 erstellt und auf einem Asus CUV4-M mit PIII 750 und ATI Rage 128 Pro gestartet, traten sehr häufig Fehler auf. Insbesondere die Applikationen, welche die Klasse *AsyncEvent* nutzen scheinen davon betroffen. Dies legt nahe, dass die JamaicaVM diesbezüglich auch ihre Probleme hat.

### 3.2.2 GRUB

Um RTEMS-Applikation zu starten, kann bei einem Standard-PC mit GRUB<sup>18</sup> gearbeitet werden. Diese Vorgehensweise zeichnet sich vor allem dadurch aus, dass sie ohne großen Aufwand umgesetzt werden kann und auch auf einem standalone Gerät funktioniert. Dabei haben sich zwei Verfahren als besonders geeignet herauskristallisiert.

Hinweis:

Da GRUB auch, mit Hilfe von *gzip*, komprimierte Dateien verarbeiten kann, nutzt man diese Möglichkeit um Speicherplatz zu sparen.

#### 3.2.2.1 Booten von Festplatte

Um GRUB von der Festplatte starten zu können, muss dieser installiert und als Bootmanager eingerichtet sein. Das bedeutet, dass er das erste Programm ist, dass nach dem Start des BIOS ausgeführt wird. Er hat dabei nur die Aufgabe ein bestimmtes Betriebssystem zu laden und ihm die Kontrolle zu übergeben, wobei man dieses dynamisch auswählen kann.

Um eine RTEMS-Anwendung zu starten geht man nun wie folgt vor:

1. Booten abbrechen

*ESC* drücken, um zu verhindern, dass das default Betriebssystem gebootet wird.

2. Modus wechseln

Anschließend muss der Menü Modus verlassen und in den command-line Modus gewechselt werden. Ist dies erfolgt, bekommt man folgenden Tab zu sehen.

```
grub>
```

3. Kernel laden

Nun wird das RTEMS Image, das vom Jamaica Builder erzeugt wurde, geladen. Dazu wird der Befehl *kernel <file>* herangezogen.

```
grub> kernel (hd0,0)/ClockExample_RTEMS_pck6
```

Hinweis:

Mit Hilfe der Taste *Tab* wird die Autovervollständigung aktiviert.



<sup>18</sup> [www.gnu.org/software/grub](http://www.gnu.org/software/grub)

4. Applikation starten

Abschließen wird der Kernel, und damit die Applikation, mit Hilfe des *boot* Befehls, gestartet.

```
grub> boot
```

Diese Vorgehensweise ist vor allem für das Testen der Applikationen hervorragend geeignet, da GRUB bei Suse 8.2 bereits eingerichtet ist und somit keine Änderungen vorgenommen werden müssen.

### 3.2.2.2 Boot von CD

Um RTEMS auch auf einem Standard-PC, der nicht über GRUB verfügt, komfortabel starten zu können, besteht die Möglichkeit eine Boot-CD zu erstellen. Für diesen Zweck bietet GRUB ein besonderes *Stage 2-File*<sup>19</sup>, das der El Torito Spezifikation entspricht, an. Dieser Standard erlaubt es mit BIOS Funktionen von CD zu booten.

Um die Applikationen von CD zu starten wird wie folgt vorgegangen:

#### 1. CD booten

Da sich GRUB auf der CD befindet, muss das System mit deren Hilfe gebootet werden. Man findet sich daraufhin automatisch im command-line Modus wieder.

```
grub>
```

#### Hinweis:

Es ist unter Umständen notwendig, dass die Boot-Reihenfolge im BIOS angepasst werden muss.

#### 2. Kernel laden

Nun wird, in gewohnter Weise, der Kernel, mit Hilfe des Befehls *kernel <file>*, geladen. Da sich dieser in der Regel auf der CD befindet, muss diese als Laufwerk ausgewählt werden.

```
grub> kernel (cd)/ClockExample RTEMS_pck6
```

#### 3. Applikation starten

Zu guter Letzt wird der Kernel , und damit die Applikation, mit Hilfe des *boot* Befehls, gestartet

```
grub> boot
```

#### Hinweis:

Nachdem eine RTEMS Applikation beendet wurde, muss das System neu gebootet werden. Des weiteren lässt sich die Netzwerk-Unterstützung nicht abschalten. Falls dies nötig wäre, müsste man eine entsprechende Version bei der aicas GmbH bestellen. Dies wäre allerdings mit weiteren Kosten verbunden.

---

<sup>19</sup> [www.gnu.org/software/grub/manual/html\\_node/Making-a-GRUB-bootable-CD-ROM.html](http://www.gnu.org/software/grub/manual/html_node/Making-a-GRUB-bootable-CD-ROM.html)

## 4. Robotersteuerung

Um zu zeigen, dass bereits praktisch, effizient und im Rahmen größerer Projekte mit der RTSJ gearbeitet werden kann, wurde eine Steuerung für einen Industrieroboter realisiert. Ziel war es zu zeigen, dass sich die Probleme, die bei einer solchen Aufgabe auftreten, nun auch ohne Einsatz von native Code mit Java gelöst werden können und man darüber hinaus auch von den Vorteilen, die Java bietet, profitieren kann.

### 4.1 Hardware

Bei dem zu steuernden Industrieroboter handelt es sich um den RM-501 Move Master 2 von Mitsubishi. Einem kleinen, kommerziell vertriebenen Roboter aus den 80er Jahren, der beispielsweise in Laboren zum Umgang mit gefährlichen Chemikalien eingesetzt wurde. Heutzutage dient er fast ausschließlich zu Schulungszwecken, da sich an ihm die typischen Probleme, die eine Robotersteuerung mit sich bringt, sehr gut demonstrieren lassen.



Abbildung 53: Der RM-501 Move Master 2

#### 4.1.1 Arm

Der Aufbau des Arms ist entscheidend für die späteren Einsatzmöglichkeiten eines Roboters, denn Mobilität und Tragkapazität werden dadurch in hohem Maße beeinflusst. Beim RM-501 handelt es sich um einen Gelenkarm-Roboter. Genau genommen ist er ein Industrieroboter vom Typ "TRRRT". Dieses Kürzel beschreibt den prinzipiellen Aufbau des Arms, wobei nur die Gelenke, von der Basis angefangen, berücksichtigt werden. Man ersetzt dabei jedes Gelenk durch einen Buchstaben, der von dessen Typ abhängig ist.

- "T" steht für Torsionsgelenk ( $\theta$  und  $\varepsilon$ ). Bei diesen verläuft die Drehachse parallel zu den Achsen beider Glieder.

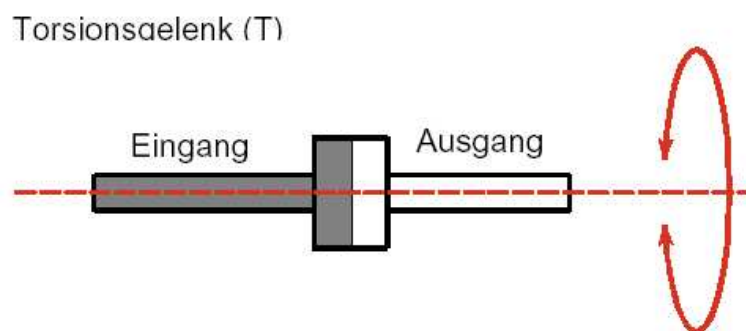


Abbildung 54: Schematischer Aufbau – Torsionsgelenk [IHME S.2]

- "R" steht für Rotationsgelenk ( $\psi$ ,  $\phi$  und  $\delta$ ). Bei diesen Gelenken steht die Drehachse im rechten Winkel zu den angeschlossenen Gliedern.

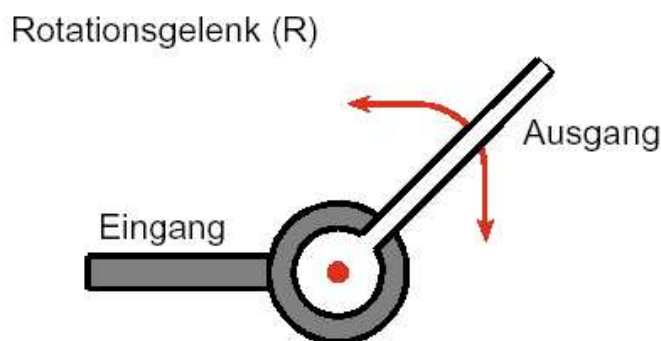


Abbildung 55: Schematischer Aufbau – Rotationsgelenk [IHME S.11]

Anhand dessen lassen sich die grundlegenden Eigenschaften eines Roboters auf einen Blick ableiten. So zeichnet sich der RM-501 durch seine große Beweglichkeit und seine geringe Tragkapazität aus. Sein Arbeitsraum, das sind alle Punkte, die er mit seinem Greifarm erreichen kann, wenn man die Grenzen der Gelenke kurz außer acht lässt, ist eine Hohlkugel. In der Realität ergibt sich aber, wie man am tatsächlichen Bewegungsraum des RM-501 sehen kann, ein etwas komplizierteres Bild.

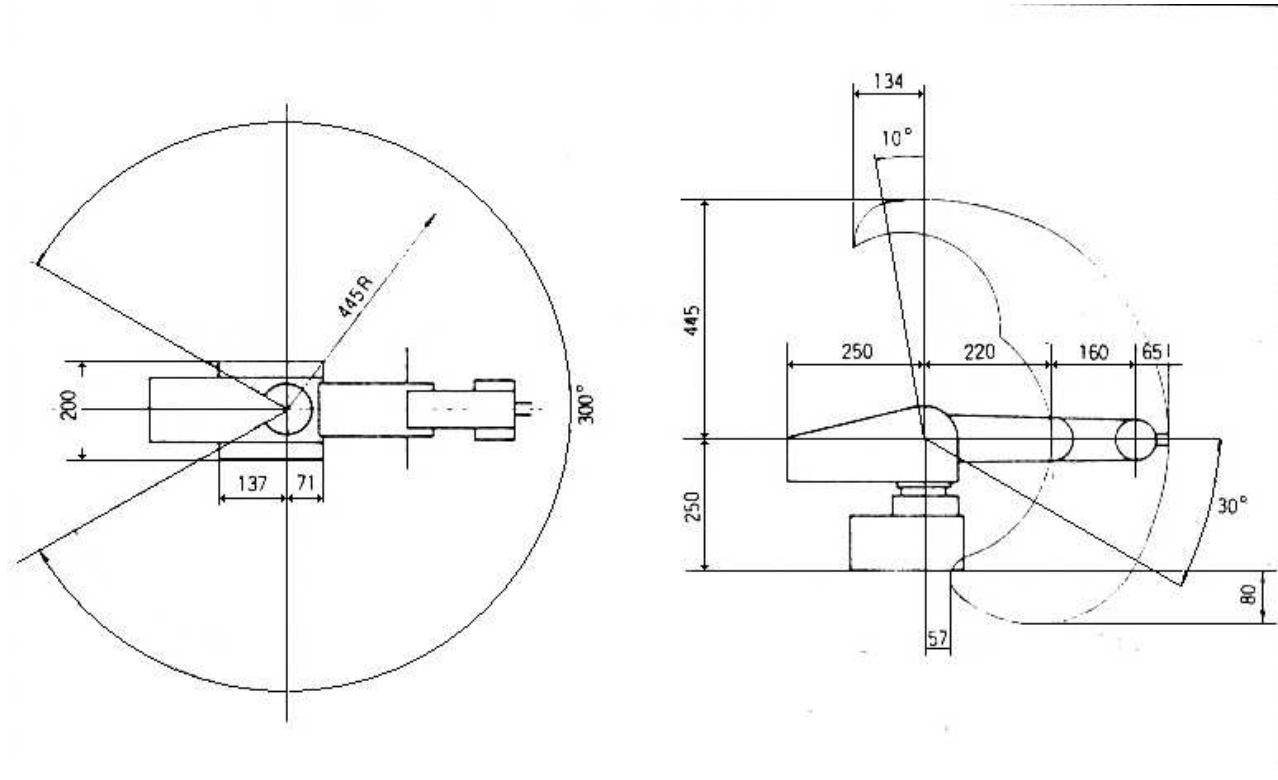


Abbildung 56: Bewegungsraum des Move Master 2

Aus diesem Grund sind diese Art von Industrieroboter universell einsetzbar. Man findet sie beispielsweise im Automobilbau zum Schweißen und Lackieren oder in der Montage von Platinen in der Elektroindustrie.

### 4.1.2 Antrieb

Auch der Antrieb des Roboters spielt für sein Einsatzgebiet eine entscheidende Rolle. Beim RM-501 wird jedes seiner Gelenke mit Hilfe von Schrittmotoren bewegt. Diese ermöglichen zwar nicht die hohen Kräfte und Geschwindigkeiten, wie es fluidische Antriebe tun würden. Doch dafür erlauben diese eine sehr genaue Ausrichtung des Arms und auf Grund ihrer Größe und dem geringen Wartungsaufwand, den Einsatz in kleinen, unzugänglichen und staubfreien Räumen. Zudem lässt sich die Steuerung relativ einfach realisieren und überwachen. Da man sich nur, für jedes Gelenke, die Anzahl an gegangenen und möglichen Schritten merken muss.

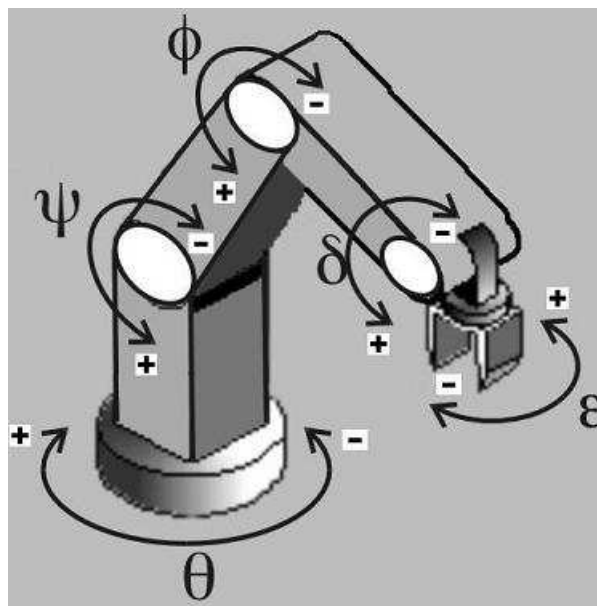


Abbildung 57: Schematischer Aufbau RM-501

Folgende Tabelle ordnet den Gelenken aus der Grafik den in der Spezifikation gebräuchlichen Namen, dessen Typen und deren Bewegungsraum zu.

Grafik	Spezifikation	Gelenktyp	Bewegungsraum
$\theta$	<b>waist</b>	<b>T</b>	-12000 ... 0
$\psi$	<b>shoulder</b>	<b>R</b>	-5200 ... 0
$\phi$	<b>elbow</b>	<b>R</b>	0 ... 3600
$\delta$	<b>wristPitch</b>	<b>R</b>	0 ... 4800
$\varepsilon$	<b>wristRotation</b>	<b>T</b>	-9600 ... 9600

Allerdings sind nicht alle Gelenke nur an einen Schrittmotor gekoppelt. Die Gelenke  $\delta$  und  $\varepsilon$  werden durch die zwei gleichen Schrittmotoren gesteuert. Das bedeutet, dass das Zusammenspiel beider Schrittmotoren für die Stellung der Gelenke verantwortlich ist.



Drehen sich beide Motoren (rot und gelb) in die entgegengesetzte Richtung, dann führt das Gelenk (grün) eine Drehbewegung aus.

Formel:  $X - Y = Z$



Abbildung 58: Schematischer Aufbau - wristRotation



Wenn sich die Motoren allerdings in die gleiche Richtung drehen, dann kippt das Gelenk (violett) auch in diese Richtung.

Formel:  $X + Y = Z$

Abbildung 59: Schematischer Aufbau - wristPitch

X = Position des einen Schrittmotors

Y = Position des anderen Schrittmotors

Z = Stellung des Gelenks

Daher setzen sich die in der Tabelle angegebenen Grenzen, für die Gelenke  $\delta$  und  $\epsilon$ , auch aus, den in den Formeln beschriebenen, Schrittkombinationen dieser beider Motoren zusammen.

### 4.1.3 Steuerung

Diese Schrittmotoren werden aber nicht direkt vom angeschlossenen PC angesteuert. Stattdessen richtet dieser seine Befehle über die parallele bzw. serielle Schnittstelle an die Steuerbox des RM-501. Diese bereitet die Befehle auf und setzt sie in die Signale, welche den eigentlichen Roboter steuern, um.

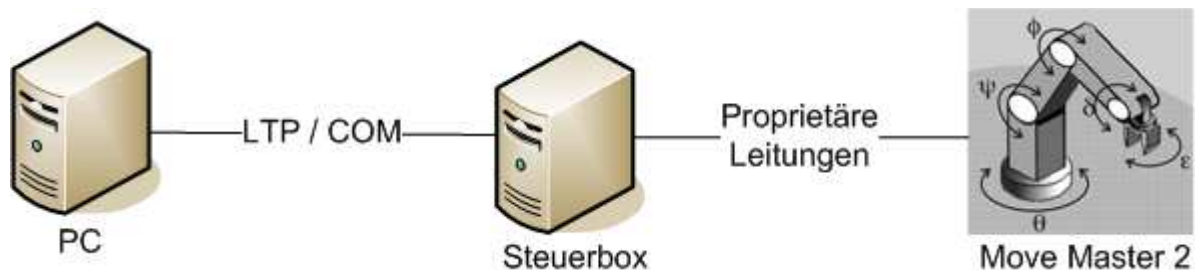


Abbildung 60: Schematischer Aufbau - Steuerung

#### 4.1.3.1 ParallelPort

Da die Steuerbox die Umsetzung dieser Befehle in Steuersignale für die Hardware übernimmt, fallen diese relativ anschaulich und für Menschen leicht lesbar aus.

Ein Befehl setzt sich dabei immer aus einem Kürzel und einer Parameterliste zusammen.

MO 629
--------

Dieser Befehl würde den Arm an die Position bewegen, die in der Steuerbox für 629 hinterlegt ist. (siehe Spezifikation Befehl Nr.5)

Damit der RM-501 diesen Befehl auch ausführt, muss diese Zeichenkette einfach über die parallele Schnittstelle an die Steuerbox geschickt werden. Diese Kommunikation erinnert stark an die Ansteuerung eines Druckers, zumal nach der vollständigen Übertragung eines jeden Befehls noch zusätzlich die beiden Zeichen „0D“ und „0A“ geschickt werden müssen. Diese symbolisieren den Abschluss der Kommunikation und werden nicht weitergereicht.

Die Übergabe eines jeden Zeichen des Befehls erfolgt dabei nach dem so genannten Centronics Handshake Verfahren:

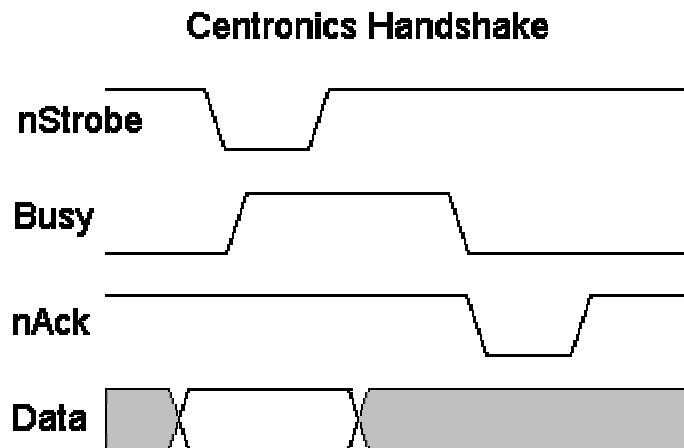


Abbildung 61: Ablauf beim Centronics Handshake [LTP S.4]

1. **BUSY**  
Als Erstes wird geprüft ob die Steuerbox bereit ist einen Befehl zu empfangen. Dazu muss BUSY abgefragt werden.
2. **Data**  
Sobald die Steuerbox bereit ist können die Daten, in unserem Fall ein Zeichen des Befehls, angelegt werden.

Hinweis:

Da die Steuerbox die übergebenen Daten als ASCII Zeichen interpretiert und Java mit UTF-16 arbeitet, müssen diese entsprechend konvertiert werden.

3. **STROBE**  
Um der Steuerbox zu signalisieren, dass Daten für sie vorliegen, wird STROBE kurzzeitig aktiviert.
4. **BUSY**  
Die Steuerbox signalisiert ihrem Kommunikationspartner durch aktivieren von BUSY, dass sie die Daten übernimmt und keine weiteren empfangen kann.

Hinweis:

ACK wird, wie es mittlerweile üblich ist, nicht abgeprüft.

## Evaluation der Realtime Specification for Java anhand einer Robotersteuerung

Diese Schritte werden solange wiederholt, bis die gesamte Zeichenkette, angefangen bei dem Kürzel des Befehls, über dessen Parameterliste bis hin zu den Zeichen, die dessen Ende signalisieren, abgearbeitet ist. Nun beginnt die Steuerbox diesen Umzusetzen. Währenddessen bleibt BUSY aktiv, so dass erst ein neuer Befehl übergeben werden kann, wenn der Alte ausgeführt wurde.

### Hinweis:

Sollte ein ungültiger Befehl übergeben werden, dann blockiert die Steuerbox und muss neu gestartet werden.

#### 4.1.3.2 SerialPort

Zusätzlich wurde ein Lichtschranke an dem Greifer angebracht. Mit dieser ist es möglich zu überprüfen, ob sich ein Objekt darin befindet. Bei dieser Erweiterung handelt es sich um eine individuelle Lösung. Deshalb erfolgt die Kommunikation zwischen der Lichtschranke und der Steuerbox nicht wie bisher über die parallele- sondern über die serielle-Schnittstelle.

Da nur ein boolescher Wert übergeben werden muss, ist auch diese Kommunikation relativ einfach aufgebaut. In diesem Zusammenhang sind nur zwei Bit von Bedeutung. Sie müssen wie folgt behandelt werden:

1. RTS aktivieren

Um der Steuerbox mitzuteilen, dass man die Lichtschranke abfragen möchte muss zunächst RTS aktiviert werden.

2. CTS auswerten

Die Steuerbox aktualisiert daraufhin das Ergebnis der Lichtschranke, welches in CTS abgelegt wird. Dieses kann nun abgefragt und ausgewertet werden.

Hinweis:

Für Detailfragen, welche die HW des Roboters und die Ansteuerung der Steuerbox betreffen, steht die Dokumentation des Roboters auf der CD im Verzeichnis *Quellen* zur Verfügung.

## 4.2 Software

Zur Steuerung des Roboters wurde eine eigene Bibliothek geschrieben. Sie bildet den Roboter ab und wird anschließend von mehreren Anwendungen genutzt um die unterschiedlichen Anforderungen umzusetzen.

### 4.2.1 Bibliothek

Sie orientiert sich an der klassischen 3-Schichten Architektur. Dabei wird zwischen der Schnittstelle zur Hardware, der Schnittstelle zu den Anwendungen und der Logikschicht unterschieden.

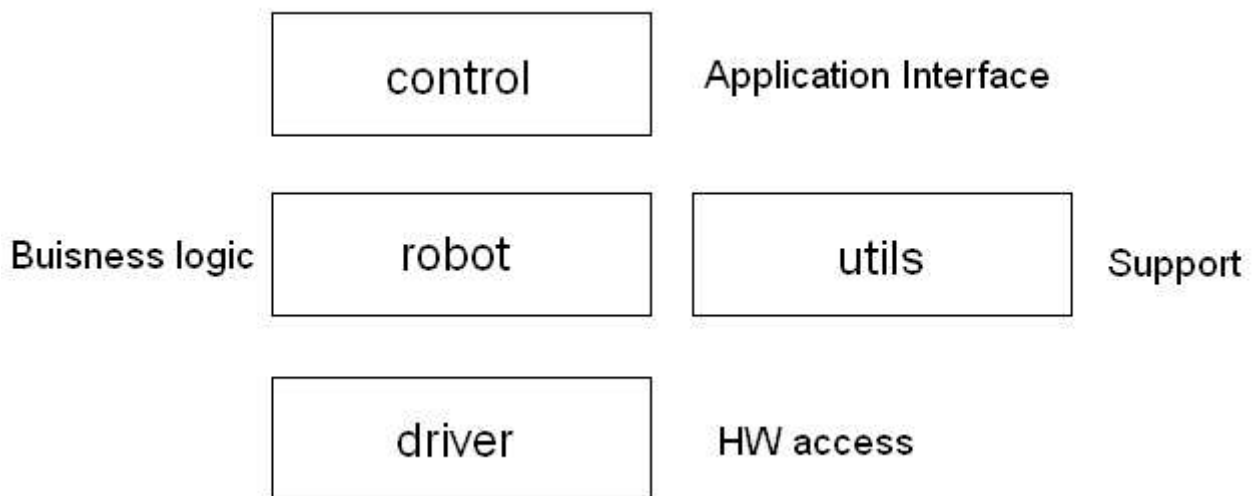


Abbildung 62: 3-Schichten-Architektur der Bibliothek

1. driver

In diesem Modul ist die Ansteuerung der parallelen und seriellen Schnittstellen gekapselt. Außerdem sind hier noch die Konstanten, wie beispielsweise die Bewegungsräume, definiert.

2. robot

*robot* enthält alle Klassen für den objektorientierten Aufbau des Roboters.

3. utils

Dieses Modul bietet Zusatzfunktionen die den Umgang mit dem Roboter erleichtern, aber nicht durch dessen Aufbau vorgeschrieben sind.

4. control

Als Schnittstelle für die Anwendungen dient das Modul *control*. Hier wird zwischen remote Anwendungen, bei denen die Steuerung von einem entfernten Rechner aus erfolgt, und lokalen Anwendungen, die direkt auf dem Steuerrechner laufen, unterschieden.

Im Folgenden wird auf den Inhalt der einzelnen Module genauer eingegangen.

#### 4.2.1.1 driver

Das Modul *driver* kapselt alle Zugriffe auf den Roboter. Es stellt somit die Schnittstelle zwischen dem Steuerrechner und der Steuerbox – sprich dem Roboter – dar. Sie umfasst folgende Klassen.

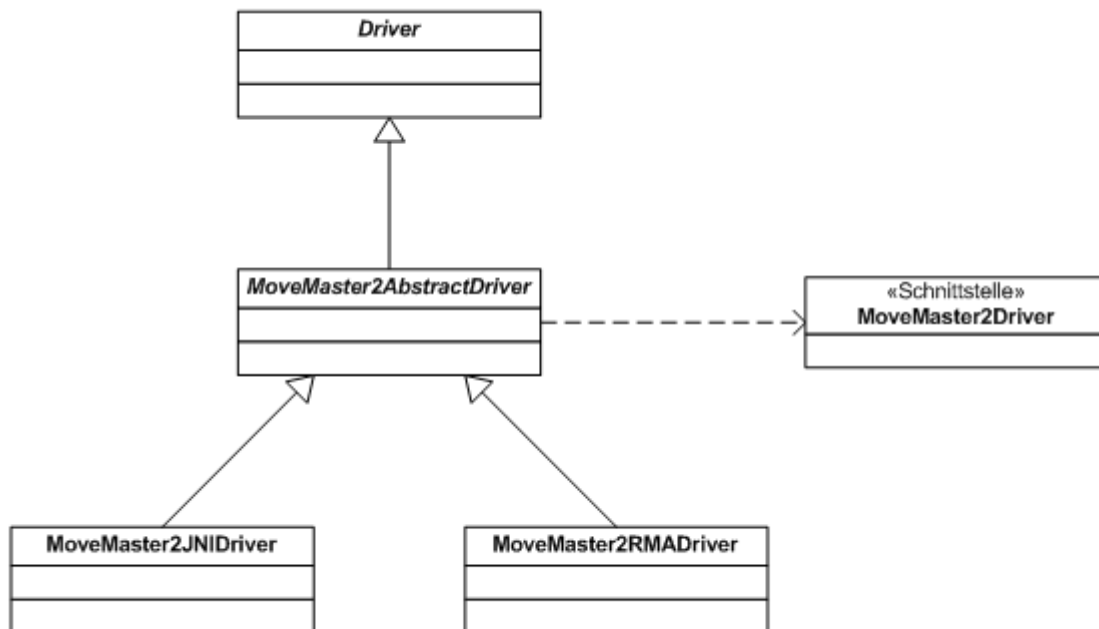


Abbildung 63: Klassenhierarchie Driver

#### Driver

Die abstrakte Klasse *Driver* erzwingt grundlegende Funktionen die jeder Treiber bieten muss. Dazu zählt beispielsweise das OS für das er geschrieben wurde und seine Version.

#### MoveMaster2Driver

Das Interface *MoveMaster2Driver* schreibt vor welche Funktionen ein Treiber für einen RM-501 Move Master 2 mindestens anbieten muss.

Dabei handelt es sich um die folgenden Methoden:

1. *public void init()*  
Synchronisiert die Roboter HW mit der SW der Steuerbox.  
(siehe Spezifikation Befehl Nr.1)
2. *public void moveRobot( int waist, int shoulder, int elbow, int wrist\_1, int wrist\_2)*  
Bewegt die Schrittmotoren des Roboters um die angegebene Anzahl an Schritten.  
(siehe Spezifikation Befehl Nr.4)

3. *public void openGrip()*  
Öffnet den Greifer  
(siehe Spezifikation Befehl Nr.12)
4. *public void closeGrip()*  
Schließt den Greifer  
(siehe Spezifikation Befehl Nr.13)
5. *public boolean checkGripper()*  
Prüft ob sich ein Gegenstand im Greifer befindet  
(nicht in der Spezifikation definiert, da es sich um eine individuelle Erweiterung handelt)

### **MoveMaster2AbstractDriver**

Die abstrakte Klasse *MoveMaster2AbstractDriver* wurde eingeführt, da die Befehle als Zeichenketten übergeben werden müssen. Sie kapselt alle Funktionen die nur diese Befehle zusammenbauen. Dies führt zwar dazu, dass der Treiber umfangreicher wird, doch dafür bietet er mehr Sicherheit und ist einfacher zu warten.

Neben den genannten Basisfunktionen, die für eine minimale Robotersteuerung notwendig sind, wurden noch folgende Befehle implementiert:

1. *public void resetPositions()*  
Löscht alle bisher gespeicherten Positionen  
(nicht in der Spezifikation definiert)
2. *public void moveHome()*  
Bewegt den Roboter zurück in die Ausgangsposition.  
(siehe Spezifikation Befehl Nr.2)
3. *public void pause(int tenthSecounds)*  
Sorgt dafür, dass der Roboter für die angegebene Zeit keine neuen Befehle akzeptiert.  
(siehe Spezifikation Befehl Nr.16)
4. *public void setSpeed(int speed)*  
Setzt die Bewegungsgeschwindigkeit des Roboterarmes  
(siehe Spezifikation Befehl Nr.15)
5. *public void setPower(int power1, int power2, int pause )*  
Setzt die Kraft mit der sich der Greifer schließt  
(siehe Spezifikation Befehl Nr.11)



6. *public void storePosition(int index)*  
Speichert die aktuelle Position an dem angegebenen Index  
(siehe Spezifikation Befehl Nr.9)
7. *public void moveToPosition(int index)*  
Bewegt den Roboterarm in die zuvor, an diesem Index, gespeicherte Position  
(siehe Spezifikation Befehl Nr.5)
8. *public void resetPositions(int lowerBound, int upperBound)*  
Löscht die gespeicherten Positionen innerhalb des angegebenen Intervalls  
(siehe Spezifikation Befehl Nr.10)
9. *private void waitForRobot()*  
Verzögert die Ausführung des Programms solange, bis die Steuerbox den Befehl abgearbeitet hat, indem sie periodisch abfragt, ob die Steuerbox noch beschäftigt ist.  
(nicht in der Spezifikation definiert)

Alle Befehl sind dabei nach dem gleichen Muster aufgebaut:

```
...
public void moveToPosition(int index)
{
    //create & execute command
    String cmd = "MO " + index + "\r\n";
    execute(cmd);
    waitForRobot();
}
...
```

1. Zeichenkette erstellen  
Die Zeichenkette wird aus dem Befehl spezifischem Kürzel, den übergebenen Parametern und dem Ende-Flag zusammengebaut.
2. Zeichenkette übertragen  
Der Befehl wird der Steuerbox mit Hilfe der *execute(String)*-Methode übergeben.
3. Warten  
Die Ausführung wird solange verzögert, bis die Steuerbox diesen Befehl abgearbeitet hat.

## Evaluation der Realtime Specification for Java anhand einer Robotersteuerung

Die Klasse *MoveMaster2AbstractDriver* implementiert nur Methoden, die nicht direkt auf die HW zugreifen. Daher schreibt er für seine Subklassen folgende Methoden vor:

1. *protected abstract void execute(String command)*  
Übergibt den Befehl mit Hilfe der parallelen Schnittstelle an die Steuerbox
2. *public abstract boolean checkGripper()*  
Fragt den Status der Lichtschranke, die im Greifer montiert wurde, über die serielle Schnittstelle ab
3. *public abstract boolean ready()*  
Prüft ob die Steuerbox den nächsten Befehl empfangen kann, indem sie das BUSY-Bit auswertet

Die Umsetzung dieser Methoden ist abhängig von der verwendeten Technologie.

### MoveMaster2RMADriver

Die Klasse *MoveMaster2RMADriver* nutzt die Klasse *RawMemoryAccess* (RMA), aus dem package *javax.realtime*, um die Befehle an den Roboter zu übergeben und die Lichtschranke abzufragen. Dabei erfolgt der Zugriff auf die entsprechenden Bits und Bytes über die Methoden *RawMemoryAccess.getBytes(long)* und *RawMemoryAccess.setByte(long, byte)*. Um zu verhindern, dass mehrere Befehle gleichzeitig ausgeführt werden, erfolgt das Lesen und Schreiben ausschließlich exklusiv, wobei die Speicherbereiche, auf die jeweils zugegriffen wird, zur Synchronisation herangezogen werden.

#### Hinweis:

In der JamaicaVM müssen Speicherbereiche, auf die mit RMA zugegriffen werden kann, explizit angegeben werden. Hierfür sind unter Linux Administratorrechte notwendig.

### MoveMaster2JNIDriver

Als Alternative steht noch die Klasse *MoveMaster2JNIDriver* zur Verfügung. Diese nutzt in traditioneller Art und Weise JNI, welches auf einen externen Treiber zugreift, um den Roboter anzusteuern. Sie wurde entwickelt bevor die Probleme mit RMA gelöst wurden.

Um JNI nutzen zu können müssen folgende Schritte durchgeführt werden:

1. Java Klasse anlegen  
Beim Anlegen der Klasse werden alle Methoden, die nicht in Java implementiert werden, mit dem Schlüsselwort *native* gekennzeichnet.

```
...  
public synchronized native boolean ready();  
...
```

## 2. Header erzeugen

Damit die JamaicaVM die native Methoden aufrufen kann, müssen diese über eine spezielle Signatur verfügen. Sie werden in unserem Fall von einem speziellen Tool, dem *jamaicah*, automatisch generiert. Es erzeugt ein Header File, dessen Name sich aus dem package und dem Namen der Klasse zusammensetzt (z.B.: *Driver\_MoveMaster2JNIDriver.h*). Auf den ersten Blick haben die einzelnen Methoden einen befremdlichen Aufbau, der jedoch typisch für JNI ist.

```
...  
jboolean JNICALL Java_Driver_MoveMaster2JNIDriver_ready(JNIEnv *env, jobject t)  
...
```

### Hinweis:

Unter keinen Umständen sollte an diesen Header Files Änderungen vorgenommen werden. Stattdessen sollte einfach ein neues File generiert werden.

## 3. Native Treiber implementieren

Auf Basis dieses Headers wird ein entsprechender Treiber verfasst. Dabei werden in unserem Fall mehrere Standard-Bibliotheken eingebunden. Diese beschreiben zum einen die Datentypen die mit Java kompatibel sind und bieten zum anderen Funktionen für den Zugriff auf physikalische Adressen an. Man bedient sich dabei der Linux spezifischen Methoden *inb(unsigned long)* und *outb(int,unsigned long)* zum Lesen und Schreiben auf physikalische Adressen, sowie der Methode *int ioperm(unsigned long from, unsigned long num, int turn\_on)* um den Zugriff auf diese Adressen zu gewähren.<sup>20</sup>

---

<sup>20</sup> [www.gmonline.demon.co.uk/cscene/CS4/CS4-02.html](http://www.gmonline.demon.co.uk/cscene/CS4/CS4-02.html)

```
...
if (ioperm(LTP1,3,1))
{
    printf("exception - ioperm alloc\n");
    return;
}

//check if Robot is busy
unsigned char busy = inb(LTP1+BUSY_BYTE_POSITION);
busy = busy & (1 << BUSY_BIT_POSITION);

//lock LTP1
if (ioperm(LTP1, 3, 0))
{
    printf("exception - ioperm free\n");
    return;
}

return busy;
...
```

Aus Sicherheitsgründen werden nach jedem Aufruf die notwendigen Adressen wieder gesperrt, damit keine andere Anwendung darauf zugreifen kann. Folglich müssen sie bei jedem erneuten Aufruf freigeschaltet werden.

#### 4. Native Treiber kompilieren

Abschließend muss der Treiber noch kompiliert werden. Dabei ist darauf zu achten, dass man ein Object-File, das erst später verlinkt wird, erzeugt. Dieses ist aber, im Gegensatz zu der Lösung mit Java, plattformabhängig.

Der Aufruf ist selbstverständlich Compiler spezifisch. Unter Linux bietet es sich jedoch an, die GCC zu verwenden.

```
gcc -O2 -I/usr/local/jamaica-2.6-3/target/linux-gnu-i686/include -c Driver_MoveMaster2JNIDriver.c
```

- -O2

Dieser Parameter gibt die Stufe der Optimierung an. Er muss gesetzt werden, da das Objekt-File sonst nicht gebaut werden kann. Dies hängt mit dem Einsatz der Methode *int ioperm(unsigned long from, unsigned long num, int turn\_on)* zusammen.

- -I <path>

Diese Angabe lässt den Compiler bestimmte Header-Files importieren. In unserem Fall wird dafür gesorgt, dass die JNI spezifischen Datentypen erkannt werden.

- -C

Damit der Linker nicht aktiviert und ein Object-File geliefert wird, muss dieser Parameter gesetzt werden.

Hinweis:

In der JamaicaVM muss der native Treiber explizit, mit `-object=<file>`, eingebunden werden. Auch hier sind unter Linux Administratorrechte notwendig.

#### 4.2.1.2 robot

Das Modul *robot* enthält die Klassen die den Aufbau des Roboters objektorientiert nachbilden. Dabei wurde der Schwerpunkt auf eine hohe Wiederverwendbarkeit gelegt.

##### IndustryRobot

*IndustryRobot* beschreibt den prinzipiellen Aufbau eines Roboters. Dieser besteht, wie man leicht der Grafik entnehmen kann, aus einem Arm, einem Werkzeug und einem Treiber.

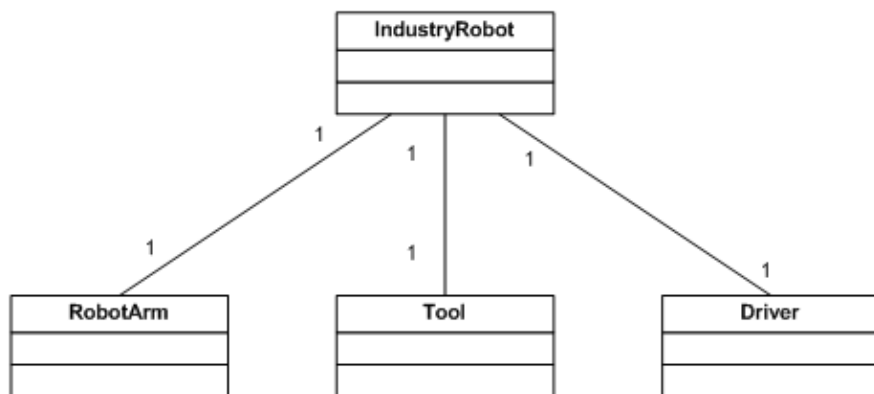


Abbildung 64: Aufbau des *IndustryRobot*

Außer dem Zugriff auf die einzelnen Komponenten bietet diese Klasse keine weitere Funktionalität an.

##### Tool

Die abstrakte Klasse *Tool* repräsentiert alle Werkzeuge die am Ende eines Roboterarms montiert werden können. Dazu zählen Schweißgeräte, Polieraufsätze, Sprühdosen zum Lackieren oder, wie in unserem Fall, ein Greifarm.

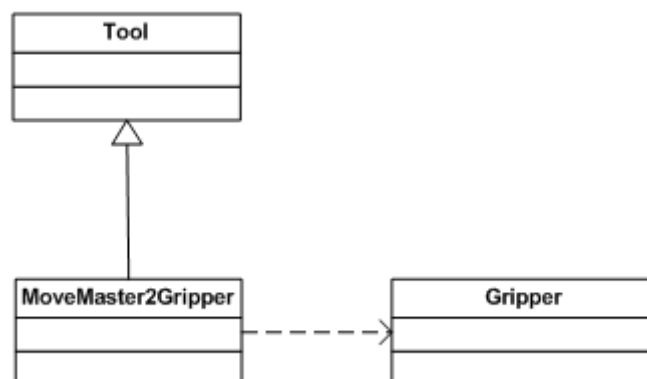


Abbildung 65: Klassenhierarchie *Tool*

Sie schreibt nur vor, wie man ein Werkzeug an einen Roboter bindet und dessen Zugehörigkeit abfragen kann.

## Gripper

Das Interface *Gripper* beschreibt die Funktionalität, die ein Greifarm mindesten bieten muss. Das sind folgende Methoden:

1. *public void openGripper()*  
Öffnen des Greifarms
2. *public void closeGripper()*  
Schließen des Greifarms
3. *public boolean getGripperState()*  
Liefert den Status des Greifarms – offen oder geschlossen – zurück.

## MoveMaster2Gripper

Um die speziellen Eigenschaften des am RM-501 montierten Greifarms umzusetzen wurde die Klasse *MoveMaster2Gripper* eingeführt.

Diese erweitert die Funktionalität des Interface *Gripper* wie folgt:

1. *public void setGripperPower(int power1, int power2, int pause)*  
Stellt die Kraft, mit welcher der Greifarm zupackt, und die Zeit, die mindestens zwischen dem Öffnen und dem Schließen verstreichen muss, ein.
2. *public int getGripperPower()*  
Liefert die aktuelle Kraft mit der der Greifarm zupackt zurück.
3. *public boolean checkGripper()*  
Fragt die Lichtschranke, welche im Greifarm montiert ist, ab.

Darüber hinaus werden für die einzelnen Grenzen separate Getter-Methoden angeboten. Insgesamt wird streng nach dem Prinzip „teile und herrsche“ direkt auf den Treiber zugegriffen. Damit ist jede Klasse für die von ihr angebotene Funktionalität voll verantwortlich. Es gibt daher auch keine überraschenden Seiteneffekte durch Methoden aus anderen Klassen, was die Wartung der Bibliothek vereinfacht.

## RobotArm

Der Arm eines Industrieroboters, der durch die Klasse *RobotArm* im System abgebildet wird, besteht seinerseits wieder aus Gliedern, Gelenken und Motoren.

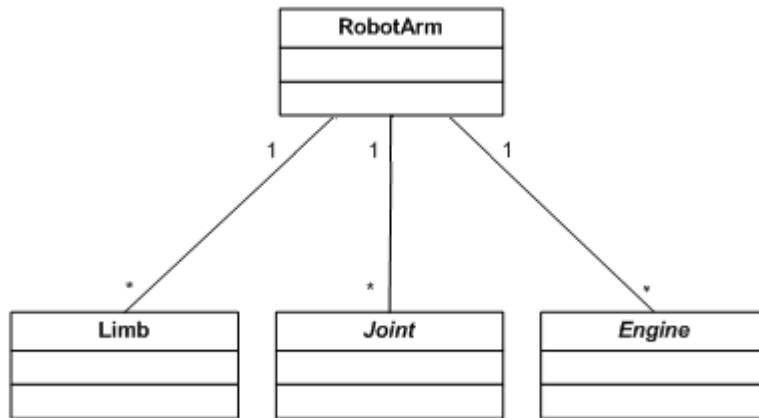


Abbildung 66: Aufbau des RobotArm

Sie ist nach dem gleichen Schema wie die Klasse *IndustryRobot* aufgebaut und bietet deshalb nur einen komfortablen und sichern Zugriff auf seine Komponenten an.

## Limb

Die Klasse *Limb* repräsentiert ein Glied eines Arms. Es zeichnet sich allein durch sein Länge aus. Im Moment ist keine Funktionalität daran gebunden. In Zukunft könnte es beispielsweise zur Bestimmung des Bewegungsraums eingesetzt werden.

## Engine

Die Klasse *Engine* ist die abstrakte Basisklasse für alle Antriebe des Roboters. Sie sorgen dafür, dass sich dessen Arm bewegt. Allen gemeinsam sind beispielsweise die Seriennummer, Betriebsstunden und Wartungsintervalle. In der Praxis wird zwischen fluidischen und elektrischen unterschieden.



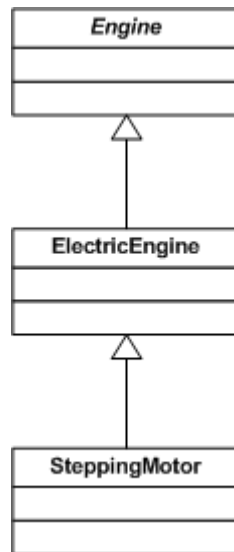


Abbildung 67:  
Klassenhierarchie Engine

Letztere werden durch die abstrakte Klasse *ElectricEngine* abgebildet. Diese zeichnen sich beispielsweise durch eine bestimmte Betriebsspannung oder einen gewissen Stromverbrauch aus. Beide Klassen werden im Moment nicht für die Funktionalität benötigt. Sie wurden ausschließlich zur Vervollständigung des objektorientierten Ansatzes eingeführt.

Allein die Klasse *SteppingMotor*, die einen Schrittmotor repräsentiert, wird produktiv eingesetzt. Diese bilden das Rückgrat der virtuellen Bewegungsräume. Da nun jeder Motor seine Schritte, die er auszuführen hat, selbstständig verwaltet, können Gelenke, die gemeinsam Schrittmotoren nutzen und deren Stellung von dem Zusammenspiel dieser abhängt, völlig unabhängig von einander betrachtet werden. Dies setzt allerdings voraus, dass jedes Gelenk die Schrittmotoren nur in einer Weise ansteuern darf, in der es ausschließlich die eigene Stellung verändert. Aus diesem Grund weichen in diesem Fall die Bewegungsräume der Schrittmotoren von denen der Gelenke ab, so dass man von Virtuellen spricht.

## Joint

Alle Gelenke eines Arms sind von der abstrakten Basisklasse *Joint* abgeleitet. Diese ordnet jenen einen Roboterarm sowie die betroffenen Antriebe zu. Zusätzlich definiert sie dessen Bewegungsraum, seine aktuelle Position und die neue Position, die das Gelenk beim Aufruf der *Moveable.move()*-Methode einnimmt. Um Gelenke, die durch Schrittmotoren bewegt werden, besser abbilden zu können, wurde das Interface *SteppingMotorJoint* eingeführt. Es schreibt die *step(int)*-Methode vor. Damit wird die Anzahl der zu gehenden Schritte vorgeschrieben. Die eigentliche Bewegung wird allerdings erst beim Aufruf von *Moveable.move()* durchgeführt.

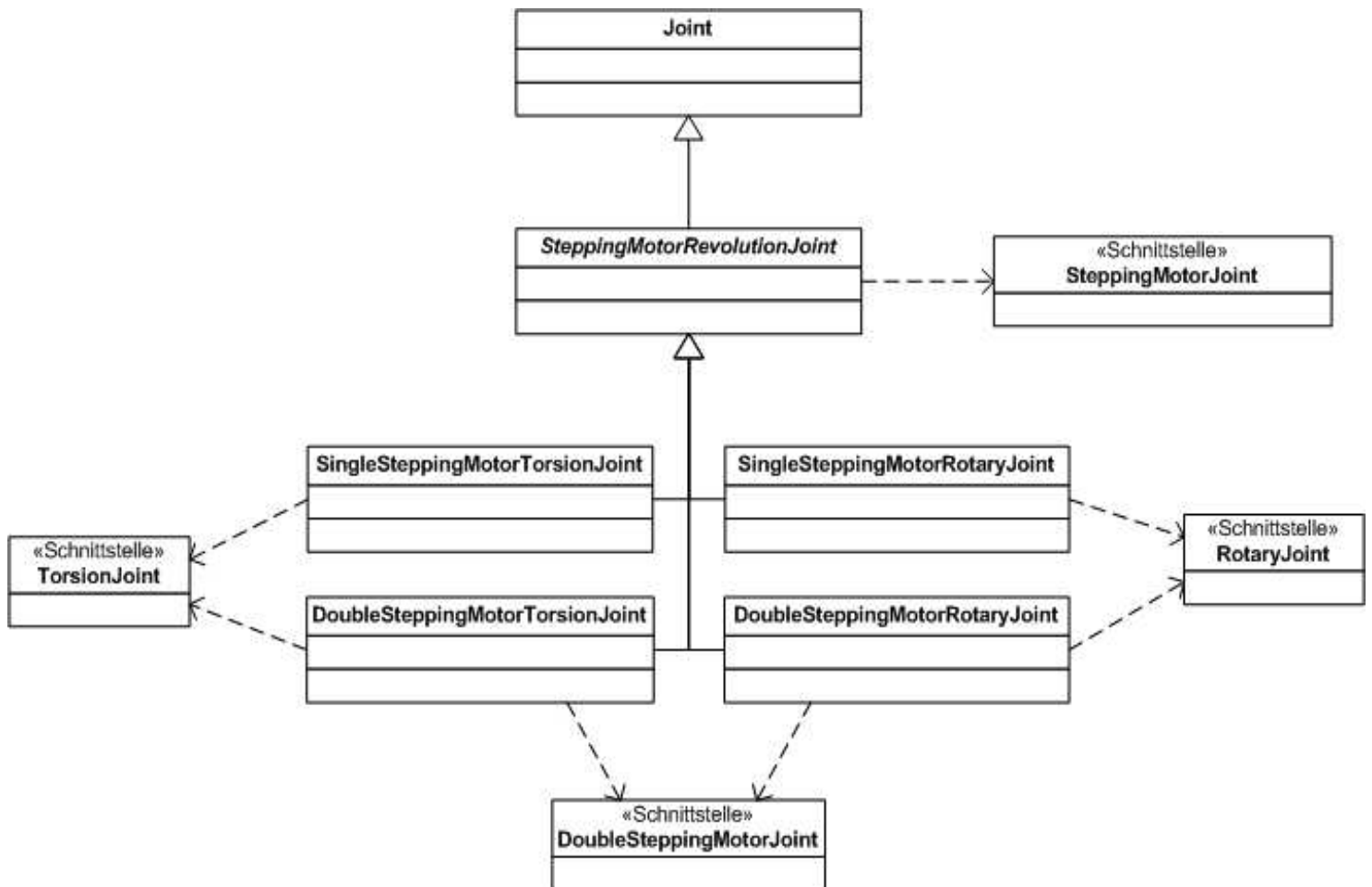


Abbildung 68: Klassenhierarchie Joint

Für Gelenke die eine Drehbewegung ausführen, wurde die abstrakte Klasse *SteppingMotorRevolutionJoint* eingeführt. Sie gibt deren grundlegenden Eigenschaften vor. Allein die *step(int)*-Methode und die *setCombineTyp(boolean)*-Methode werden in den Subklassen implementiert. Letztere ist allerdings nur für Gelenke, die mit Hilfe von 2 Schrittmotoren bewegt werden, notwendig und wird durch das Interface *DoubleSteppingMotorJoint* vorgeschrieben. Damit lässt sich festlegen, wie die beiden Schrittmotoren zusammenarbeiten. Dies ermöglicht eine von einander unabhängige Steuerung dieser Gelenke. Denn es ist ihnen nur gestattet die Motoren so anzusteuern, dass andere Gelenk nicht beeinflusst werden. Im konkreten Fall bedeutet dies, dass zum Neigen des Greifarms beide Schrittmotoren einen Schritt in die gleiche, während sie zum Drehen jeweils einen Schritt in die entgegengesetzte Richtung machen.

Klasse	Schrittmotoren	Gelenktyp	Gelenk(e)
SingleSteppingMotorTorsionJoint	1	T	$\theta$
DoubleSteppingMotorTorsionJoint	2	T	$\varepsilon$
SingleSteppingMotorRotaryJoint	1	R	$\psi$ und $\phi$
DoubleSteppingMotorRotaryJoint	2	R	$\delta$

Produktiv werden nur die in der Tabelle aufgelisteten Klassen eingesetzt. Sie bilden die Gelenke des RM-501 effizient ab und implementieren zusätzlich noch eines der beiden Interface *RotaryJoint* bzw. *TorsionJoint*. Diese geben beide keine Methoden vor, sondern ordnen nur das Gelenk einer Kategorie zu.

### MoveMaster2Arm

Um die individuellen Eigenheiten des RM-501 Arms zu kapseln, wurde die Klasse *MoveMaster2Arm* eingeführt.

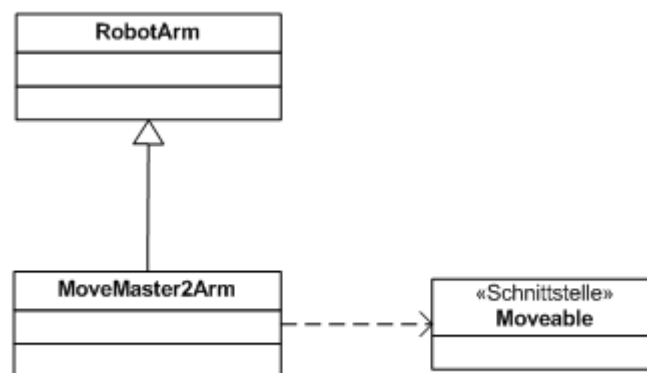


Abbildung 69: Klassenhierarchie RobotArm

Wie schon der MoveMaster2Gripper greift auch die Klasse MoveMaster2Arm direkt auf den Treiber zu. Dabei werden folgende Methoden angeboten:

- *public void move()*  
Da sich beim RM-501 immer nur der ganze Arm und nicht einzelne Gelenke bewegen lassen, implementiert er das Interface *Moveable*. Sobald die *move()*-Methode ausgeführt wird, bewegt sich der Arm in die Position, die in den Gelenken gespeichert ist
- *public void moveHome()*  
Bewegt dem Arm zurück in die Ausgangsposition
- *public void cancelMovement()*  
Da die Bewegungen erst beim Aufruf der *move()*-Methode ausgeführt werden, wird in den Gelenken neben der aktuellen Position auch die neue Position gespeichert. Letztere wird beim Aufruf der *cancelMovement()*-Methode zurückgesetzt
- *public void moveToPosition(int index)*  
Bewegt den Arm in die Position die zuvor an diesem Index gespeichert wurde
- *public void storePosition(int index)*  
Speichert die aktuelle Position an dem Index

- *public void resetPositions(int lowerBound, int upperBound)*  
Löscht alle Positionen angefangen bei *lowerBound* bis einschließlich *upperBound*
- *public void resetPositions()*  
Löscht alle Positionen
- *public void setSpeed(int speed)*  
Stellt die Geschwindigkeit ein, mit der sich der Arm bewegt
- *public void getSpeed(int speed)*  
Gibt die aktuelle Geschwindigkeit des Arms zurück
- *public void pause(int pause)*  
Sorgt dafür, dass der Roboter in der angegebene Zeitspanne keine neue Befehle entgegen nimmt
- *public int[] getUsedPositions()*  
Liefert eine Liste mit den Indices aller besetzten Positionen.
- *public String getFileName()*  
Liefert den Namen der Datei, in der auf dem Steuerrechner die Positionen archiviert werden.
- *public void setFileName(String filename)*  
Setzt den Namen der Datei auf dem Steuerrechner, der die Positionen archiviert.

Auch hier werden für die einzelnen Grenzwerte separate Getter-Methoden angeboten.

#### 4.2.1.3 *utils*

Das Module *utils* kapselt die Zusatzfunktionen für einen einfacheren Umgang mit dem Roboter, die nicht durch den Aufbau vorgeschrieben sind.

#### **PositionManager**

Im Moment zählt dazu nur die Klasse *PositionManager*. Diese archiviert die Positionen, die mit Hilfe der Methoden *MoveMaster2Arm.storePosition(int)*, gespeichert werden. Diese Datei, die sich auf dem Steuerrechner befindet, wird bei jedem Start einer Applikation ausgelesen und sorgt dafür, dass die Positionen immer, selbst nach einem Neustart, verfügbar sind. Dabei ist er vollständig im *MoveMaster2Arm* integriert, so dass nur über diesen die Möglichkeit besteht darauf zuzugreifen.

##### Hinweis:

Nachdem sich herausgestellt hat, dass der RM-501 immer den Greifer öffnet, bevor er sich in die an dem Index gespeicherte Position bewegt, wurde die Speicherverwaltung von der Steuerbox getrennt und kommt daher ohne dessen Befehle aus.

#### 4.2.1.4 control

Um dem Anwendungsentwickler einen komfortablen Zugriff auf den Roboter zu gewähren, wurde das Modul *control* eingeführt. Es bietet dem Entwickler genau eine Schnittstelle zur Steuerung des Roboters an.

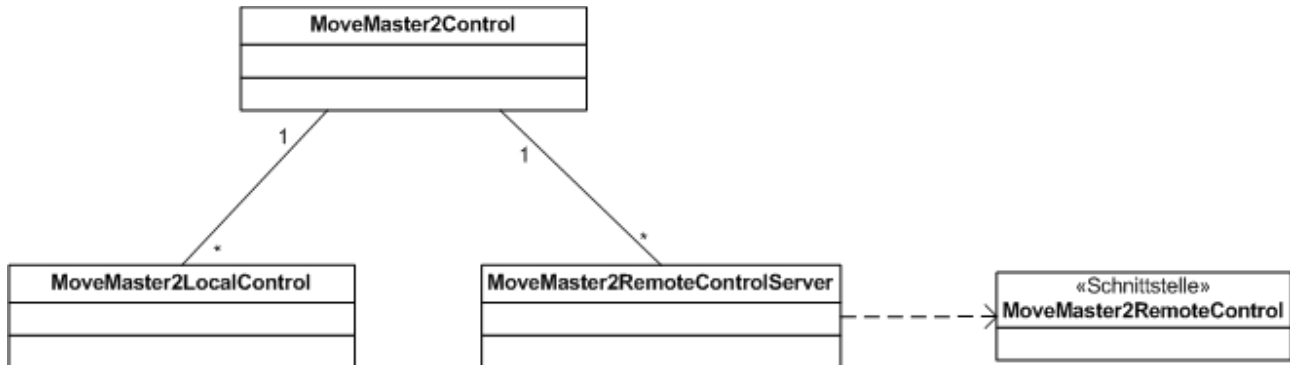


Abbildung 70: Das Modul *com.movemaster2.control*

#### MoveMaster2Control

Ihr Herzstück stellt die Klasse *MoveMaster2Control* dar. Sie ist das Bindeglied zwischen dem Roboter und den Applikationen, die diesen nutzen. Dabei übernimmt sie die folgenden 3 Aufgaben:

- **Steuerung des virtuellen Roboters**  
Die Klasse *MoveMaster2Control* bildet den Roboter ab und bietet Methoden an, die es erlauben diesen zu steuern.
- **Synchronisation**  
Da die Klasse *MoveMaster2Control* alle Methoden, die den Roboter steuern kapselt, ist sie die ideale Position für die Synchronisation. Sie sorgt dafür, dass der virtuelle und der reale Roboter nicht aus dem Tritt kommen. Als Monitore werden dabei der Roboterarm und das Werkzeug herangezogen.
- **Zugriffssteuerung**  
Um zu verhindern, dass es durch das Anlegen mehrerer Instanzen der Klasse *MoveMaster2Control* zu Inkonsistenzen kommen kann, wurde sie als Singleton realisiert. Damit ist es auch möglich mehrere Anwendungen gleichzeitig laufen zu lassen. Die Zugriffssteuerung, die jeder Anwendung dynamisch eine SessionID zuordnet, sorgt dafür, dass nur eine Anwendung Zugriff auf den Roboter erhält. Erst wenn diese den Roboter abgibt, darf die nächste diesen exklusiv nutzen.

### MoveMaster2LocalControl

Für Anwendungen, die vollständig auf dem Steuerrechner laufen, wurde die Klasse *MoveMaster2LocalControl* als Schnittstelle eingeführt. Sie übernimmt die gesamte Verwaltung, so dass sich der Entwickler auf die wesentlichen Aufgaben konzentrieren kann. Damit wird die Entwicklung von Applikationen, die den RM-501 nutzen, zu einem Kinderspiel, da man den Roboter nur wie ein einfaches Objekt behandeln muss.

### MoveMaster2RemoteControl

Um verteilte Anwendungen zu realisieren, bei denen man nicht mehr auf die VM, die auf dem Steuerrechner, der die Ansteuerung des Roboters übernimmt, beschränkt ist, wurde die Java Technologie RMI herangezogen. Damit können wichtige Programmteile, wie die Visualisierung oder sogar die Logik, in eine andere VM ausgelagert werden. Die Klasse *MoveMaster2RemoteControlServer* und das Interface *MoveMaster2RemoteControl* bilden dabei das Rückgrat dieser Architektur<sup>21</sup>.

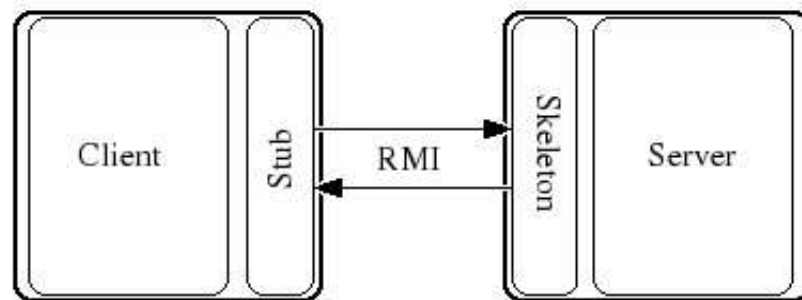


Abbildung 71: RMI - Architektur

- RMI Interface  
Das Interface beschreibt die Dienstleistungen die im Netz angeboten werden.
- Server  
Der Server implementiert jenes Interface und bietet somit diese Dienstleistungen real im Netz an.
- Client  
Der Client nutzt die angebotenen Dienstleistungen.
- Stub/Skeleton  
Während die bisher aufgeführten Klassen alle manuell implementiert werden müssen, werden der Stub und das Skeleton, die beide Schnittstellen-Klassen zur Kommunikationsplattform von RMI sind, automatisch generiert. Dazu wird das Programm *rmic* herangezogen.

Beim Einsatz der JamaicaVM unter Linux muss dazu folgender Befehl ausgeführt werden:

```
jamaicavm gnu.java.rmi.rmic.RMIC <Server>
```

<sup>21</sup> [www.ccs.neu.edu/home/kenb/com3337/rmi\\_tut.html](http://www.ccs.neu.edu/home/kenb/com3337/rmi_tut.html)

## 4.2.2 Anwendungen

Mit Hilfe dieser Bibliothek wurden mehrere Applikationen für den RM-501 geschrieben, die völlig unterschiedliche Aufgaben erfüllen und einen kleinen Eindruck vermitteln, welche Möglichkeiten die RTSJ in Verbindung mit dieser Bibliothek einem Entwickler bietet. Dabei wird, wie bereits angesprochen, zwischen verteilten und lokalen Anwendungen unterschieden.

Gezeigt wird...	Nicht gezeigt wird...
Realisierung eines Treibers in Java Interaktion der RTSJ mit anderen Modulen Lösung realer Probleme mit der RTSJ Verteilte Anwendung zur Auslagerung der GUI Kompatibilität der JamaicaVM und der VM von Sun Einsatz von Internet-Technologien	Verteilte Echtzeitanwendungen Hartes Echtzeitverhalten

### Hinweis:

Da der Zugriff auf physikalische Speicheradressen mit Hilfe von *RawMemoryAccess* unter RTEMS noch nicht möglich ist, können die Anwendungen nur unter Linux ausgeführt werden. Da es sich dabei nicht um ein RTOS handelt, wird auch kein Zeitverhalten garantiert. Für die Robotersteuerung stellt dies kein großes Problem dar, da man sowieso mit relativ großen Zeiten arbeitet und, bedingt durch die HW, große Toleranzen verkraften muss. Außerdem wurde das Zeitverhalten bereits auf RTEMS untersucht.

### 4.2.2.1 Lokale Anwendungen

Die lokalen Anwendungen nutzen dabei ausnahmslos die Klasse *MoveMaster2LocalControl* um Zugriff auf den Roboter zu bekommen. Sie demonstrieren zum einen wie die Bibliothek, mit Hilfe der RTSJ, Zugriff auf die HW erhält und zeigen zum anderen, wie man Anwendungen mit und ohne dem Einsatz von Konzepten und Klassen der RTSJ realisieren kann.

#### ConsoleControl

Um ein Gefühl für den Roboter und seinen Eigenheiten zu bekommen, wurde die Anwendung *ConsoleControl* geschrieben. Dabei handelt es sich um ein einfaches Konsolen-Programm, das die Standardein- und ausgabe als Schnittstelle zum Anwender nutzt. Es wurde zum Test der einzelnen Funktionen herangezogen und ist aus diesem Grund auch besonders gut geeignet um sich mit dem Roboter und seinen Eigenheiten vertraut zu machen. Das Menü gliedert sich in 4 logische Abschnitte, wobei die Navigation mit Hilfe von Buchstaben erfolgt. Mit *h* werden einem alle Möglichkeiten angezeigt, während man mit *x* in das übergeordnete Menü wechselt.



## Evaluation der Realtime Specification for Java anhand einer Robotersteuerung

- Greifarm-Steuerung  
Hier können alle Einstellungen die den Greifarm betreffen vorgenommen werden.
- Positionsverwaltung  
In der Positionsverwaltung können Positionen gespeichert, gelöscht und geladen werden.
- Steuerung der Gelenke  
Die Steuerung der einzelnen Gelenke erfolgt unabhängig voneinander. Hier kann für jedes Gelenk separat die aktuelle Position abgefragt und eine Neue festgelegt werden.
- Allgemeine Einstellungen  
Unter allgemeine Einstellungen fallen alle grundlegenden Befehle und jene die den ganzen Arm betreffen.

Anhand dieser Anwendung lässt sich sehr schön die Aufrufhierarchie der einzelnen Methoden zeigen. So sind beispielsweise beim Schließen des Greifarms die folgenden Methoden beteiligt:

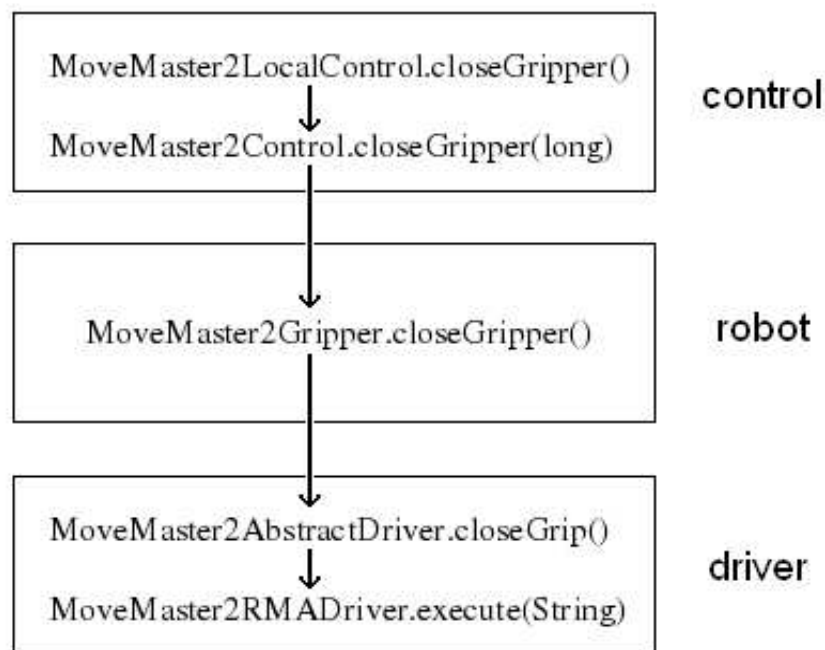


Abbildung 72: Aufrufhierarchie für des Schließen des Greifarms

1. `MoveMaster2LocalAccess.closeGripper()`  
Die Anwendung `ConsoleControl` nutzt die Methode `MoveMaster2LocalControl.closeGripper(long)` um das Schließen des Greiferarms zu initiieren.
2. `MoveMaster2Control.closeGripper()`  
Diese ruft die Methode `MoveMaster2Control.closeGripper()` auf, die dafür sorgt, dass die Session geprüft und die Methode sicher ausgeführt wird.

3. *MoveMaster2Gripper.closeGripper()*

Anschließend wird dem Tool des RM-501 mit Hilfe der Methode *closeGripper()* mitgeteilt, dass er seinen Greifarm schließen soll. Dort wird zunächst geprüft, ob dieser nicht bereits geschlossen ist.

4. *MoveMaster2AbstractDriver.closeGrip()*

Die Instanz vom Typ *MoveMaster2Gripper* nutzt daraufhin den Treiber um den Greifarm des realen Roboters zu schließen.

5. *MoveMaster2RMADriver.execute(String)*

Um den Befehl an die Steuerbox zu schicken bedient sich dieser der Methode *MoveMaster2RMADriver.execute(String)*. Diese nutzt die Klasse *RawMemoryAccess* um auf die physikalischen Adressen des LPT zugreifen zu können.

Der Treiber nutzt für die Steuerung des Roboters sowohl die serielle als auch die parallele Schnittstelle. Daher müssen diese beiden Speicherbereiche auch für den Zugriff freigeschaltet werden.

```
...
/**
 * Für den Zugriff auf den RM-501 werden für die serielle- und parallele-Schnittstelle
 * unterschiedliche Speicherbereiche verwendet.
 */
public MoveMaster2RMADriver()
{
    //Create memoryareas which are directly accessible
    memorySerialPort = new RawMemoryAccess
    (
        PhysicalMemoryManager.IO_PAGE,
        COM1,
        MEMORY_RANGE_SERIALPORT
    );

    memoryParallelPort = new RawMemoryAccess
    (
        PhysicalMemoryManager.IO_PAGE,
        LTP1,
        MEMORY_RANGE_PARALLELPORT
    );
    ...
}
...
```

Der eigentliche Zugriff auf diese Adressen ist in den Methoden *execute(String)*, *checkGripper()* und *ready()* der jeweiligen Subklassen von *MoveMaster2AbstraktDirver* gekapselt. Beim *MoveMaster2RMADriver* arbeitet man mit den Methoden *RawMemoryAccess.getBytes(long)* und *RawMemoryAccess.setByte(long, byte)*. Die Methode *ready()* fragt beispielsweise nur BUSY ab.

```
...
/**
 * Prüft anhand des BUSY-Bit ob der Roboter einen neuen Befehl entgegennehmen kann
 */
public boolean ready()
{
    byte status = memoryParallelPort.getBytes(BUSY_BYTE_POSITION);
    if((status & (0x01 << BUSY_BIT_POSITION)) == 0)
    {
        return false;
    }
    else
    {
        return true;
    }
}
...
```

Die Lösung mit *RawMemoryAccess* ist der von JNI, die bereits vorgestellt wurde, vorzuziehen, da sich diese durch einen geringeren Arbeitsaufwand, einen einheitlichen Entwicklungsprozess und der Plattformunabhängigkeit auszeichnet. Man muss nämlich nicht die gewohnte und einfache Java Umgebung verlassen und sich in andere Compiler einarbeiten. Außerdem kann man auch hier auf bekannte und bewährte Konzepte, wie beispielsweise die Synchronisation, zurückgreifen, was das Arbeiten unheimlich vereinfacht.

## Builder

Die Applikation *Builder* errichtet mit Hilfe des RM-501 aus Bauklötzen einen kleinen Turm. Im wesentlichen nimmt der Arm einen Stein an einer bestimmten Position auf und legt ihn an einer Anderen wieder ab. Damit er dabei den Turm nicht umwirft, muss er immer andere Pfade einschlagen. Diese Positionen werden mit Hilfe der Klassen *FileInputStream* und *FileOutputStream* aus dem package *java.io* automatisch archiviert und bei jedem Neustart geladen. Die zeitliche Steuerung wird dabei vom Arm selbst verwaltet. Denn er kann erst mit den nächsten Arbeitsschritt beginnen, wenn der vorhergehende abgeschlossen ist. Aus diesem Grund wurde auch auf einen *RealtimeThread* verzichtet.

```
/**
 * builds the tower
 */
public void build()
{
    //firsrt level
    control.moveToPosition(SECURE_POS_SUPPLY_1);
    control.moveToPosition(SUPPLY_1);
    control.closeGripper();
    control.moveToPosition(SECURE_POS_SUPPLY_1);
    control.moveToPosition(SECURE_POS_BUILD);
    control.moveToPosition(LEVEL_1);
    control.openGripper();

    //secound level
    control.moveToPosition(SECURE_POS_BUILD);
    control.moveToPosition(SECURE_POS_SUPPLY_2);
    control.moveToPosition(SUPPLY_2);
    control.closeGripper();
    control.moveToPosition(SECURE_POS_SUPPLY_2);
    control.moveToPosition(SECURE_POS_BUILD);
    control.moveToPosition(LEVEL_2);
    control.openGripper();

    control.moveHome();
}
```

Diese Anwendung verdeutlicht, wie einfach sich Programme mit Hilfe der Bibliothek entwickeln lassen, und wie man von bestehenden Standardbibliotheken, auch beim Einsatz der RTSJ, profitieren kann.

## MultiThreadControl

*MultiThreadControl* ist eine Anwendung bei der die RTSJ auch in der Applikation eingesetzt wird. Der RM-501 fährt dabei eine vorgegebene Strecke mit seinem Greifarm ab und sammelt alle Gegenstände (Bauklötze) die er dabei findet ein.

Sie besteht aus den Teilen:

- *MultiThreadControl*  
Die Klasse *MultiThreadControl* initialisiert die gesamte Anwendung und gibt den abzuschreitenden Weg vor.
- *RotatorThread*  
Die Klasse *RotatorThread* nutzt intern einen *RealtimeThread* um den Arm zu bewegen.
- *CheckerThread*  
Die Klasse *CheckerThread* nutzt intern einen *RealtimeThread* um die Lichtschranke periodisch abzufragen und ein *AsyncEvent* um eine Unterbrechung der Lichtschranke zu kommunizieren.
- *LightBarrierEventHandling*  
Um auf die Unterbrechung der Lichtschranke zu reagieren, wird ein *AsyncEventHandler* mit einem speziellen *Runnable* herangezogen.

Da der RM-501, Bewegungen, die er einmal begonnen hat, auch zu Ende führt, müssen diese vom *RotatorThread* in kleine Stück aufgeteilt werden. Parallel zu diesen Bewegungen prüft der *CheckerThread* die Lichtschranke ab.

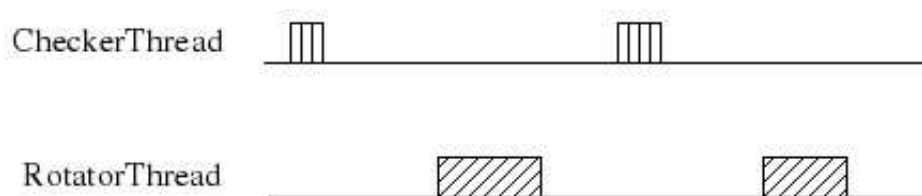


Abbildung 73: Die Threads der Anwendung *MultiThreadControl*

Sobald die Lichtschranke unterbrochen ist, löst der *CheckerThread* ein *AsyncEvent* aus. Auf dieses reagiert ein *AsyncEventHandler*, in dem er das Runnable *LightBarrierEventHandling* ausführt.

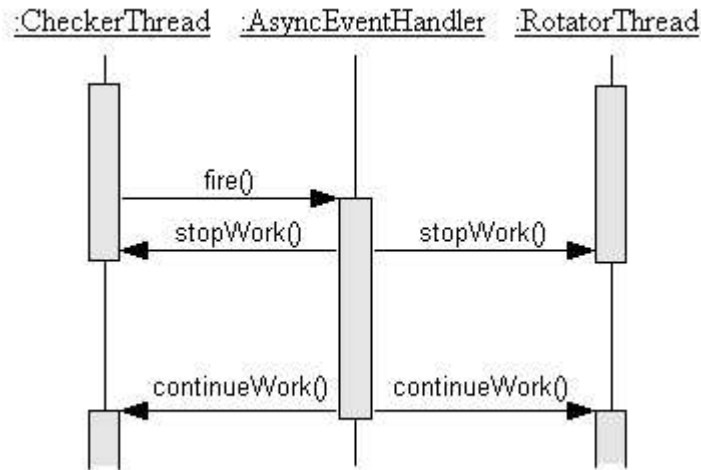


Abbildung 74: Ereignisbehandlung in der Anwendung *MultiThreadControl*

Zunächst wird die eigentliche Logik angehalten und mit der Unterbrechungslogik begonnen.

```

...
/**
 * Reaktion auf die Unterbrechung der Lichtschranke
 */
public void run()
{
    parent.stopWork();
    System.out.println("Event occurred");
    reaction();
    parent.continueWork();
}
...

```

Als Reaktion auf die Unterbrechung der Lichtschranke wird die aktuelle Position gespeichert und der Bauklotz aufgenommen. Anschließend wird der Arm in eine Position gebracht, in der er sich frei bewegen kann, um daraufhin den Stein zu entsorgen. Nachdem das Objekt beseitigt ist, fährt er zurück in die Position, an der die Lichtschranke unterbrochen wurde und aktiviert wieder die eigentliche Logik.

```

...
/**
 * Als Reaktion auf eine Unterbrechung der Lichtschranke wird...
 * - die aktuelle Position gespeichert.
 * - der Gegenstand aufgehoben
 * - der Gegenstand entsorgt
 * - das Aufräumen fortgesetzt
 */
private void reaction()
{
    synchronized(parent.getControl())
    {
        //store current position
        parent.getControl().storePosition(INTERRUPT_POSITION);

        //pick up brick
        parent.getControl().rotateWaist(-50);
        parent.getControl().move();
        parent.getControl().closeGripper();

        //throw brick away
        parent.getControl().moveToPosition(SAVE_POSITION);
        parent.getControl().moveToPosition(TRASH_POSITION);
        parent.getControl().openGripper();

        //proceed job
        parent.getControl().moveToPosition(SAVE_POSITION);
        parent.getControl().moveToPosition(INTERRUPT_POSITION);
    }
}
...

```

Die Anwendung *MultiThreadControl* zeigt, dass auch im größeren Stil mit der RTSJ gearbeitet werden kann und Java keinerlei Einbußen hinsichtlich seiner Einfachheit erleidet. Auf Grund der fehlenden Echtzeiteigenschaften des OS lassen sich aber keine Aussagen über das Zeitverhalten machen. Allerdings wurde mit der Anwendung *RTSJ\_ThreadComparison* bereits gezeigt, dass unter RTEMS alle Threads sehr genau arbeiten.

#### 4.2.2.2 Verteilte Anwendungen

Um das Zusammenspiel von RTJS mit Internet-Technologien zu untersuchen und um verteilte Anwendungen umzusetzen, wurde eine Steuerung mittels RMI realisiert.

Vor dem Start einer solchen Anwendung müssen folgende Schritte ausgeführt werden:

- Registry starten

Die Registry ist ein Programm, vergleichbar mit den Gelben Seiten, das Dienstleistungen vermittelt. Server, die eine solche anbieten, registrieren sich dort, während Clients, die einen Dienstleister suchen, dort nachfragen. Für diese Aufgabe muss sie die Lage der Stubs, die an die Clients übertragen werden, kennen.

Unter Linux kann dazu der CLASSPATH eingesetzt werden:

```
export CLASSPATH=$CLASSPATH:<path>
```

Die eigentliche Registry wird, beim Einsatz der JamaicaVM unter Linux, mit Hilfe des folgenden Befehls gestartet:

```
jamaicavm gnu.java.rmi.registry.RegistryImpl &
```

- Server starten

Nun kann der Server gestartet werden. Er meldet sich dabei mit Hilfe der Methode *Naming.rebind(String, Remote)* bei der Registry an.

- Client starten

Der Client nutzt die Dienstleistung, in dem er sie mit *Naming.lookup(String)* bei der Registry abrufen und mit Hilfe dieser Referenz mit dem Server kommuniziert. In unserem Fall handelt es sich dabei um herkömmliche Java Anwendungen, die ohne die *javax.realtime.\** auskommen. Da sie aber Klassen aus dem package *javax.swing.\** nutzen, können sie nicht in der JamaicaVM gestartet werden.

Hinweis:

RMI setzt in Verbindung mit der JamaicaVM unter Linux ebenfalls Administratorrechte voraus.



### MoveMaster2RemoteControlApplikation

Die Anwendung *MoveMaster2RemoteControlApplikation* stellt folgende GUI zur Steuerung des RM-501 bereit. Im Gegensatz zu den Anwendungen die RTSJ nutzten, läuft sie nicht auf der JamaicaVM, sondern wird auf der VM von Sun ausgeführt.

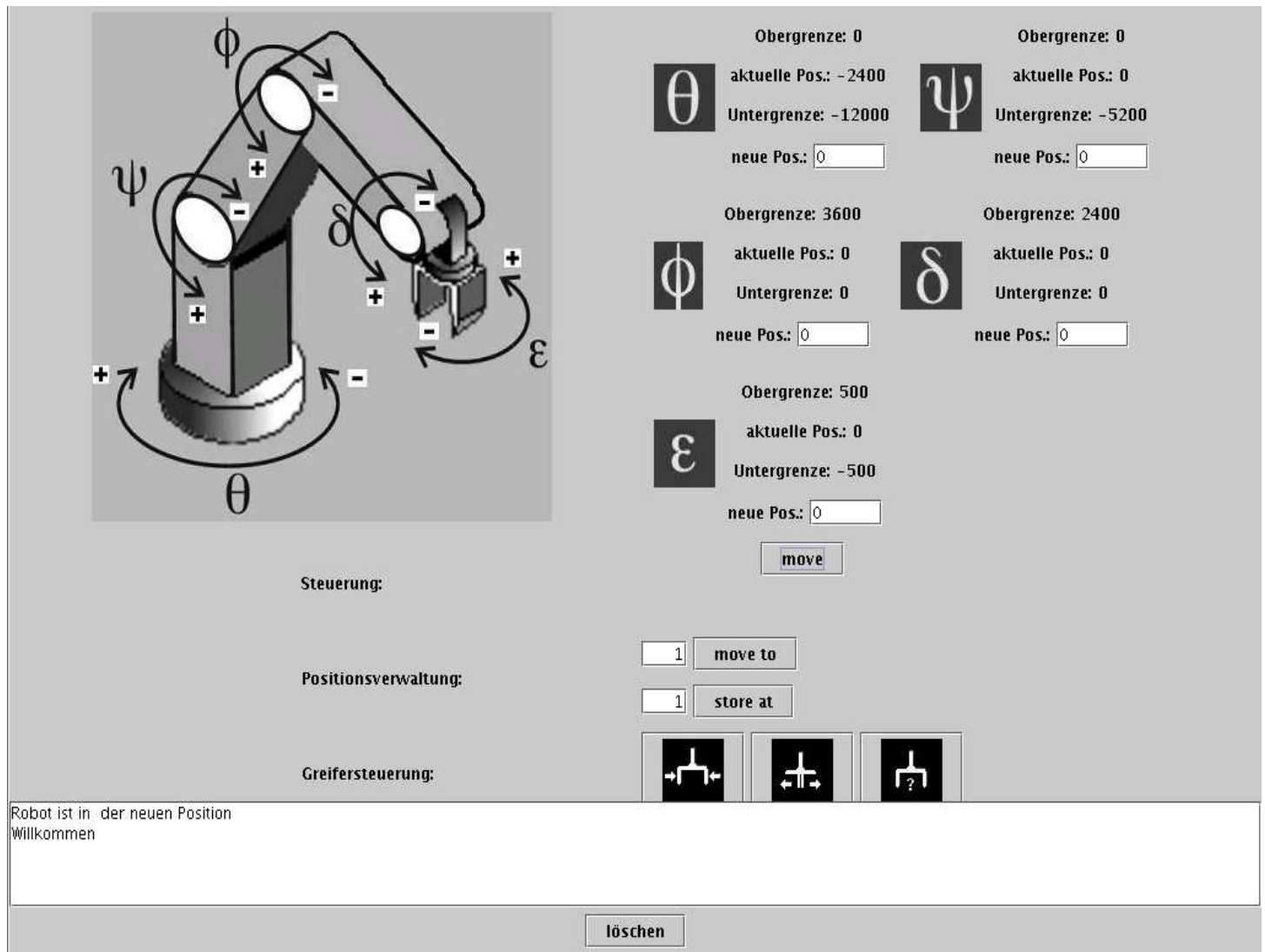


Abbildung 75: Screenshot von der GUI für die Robotersteuerung

Zur Steuerung stehen die folgenden 3 Optionen bereit:

- Direkte Steuerung  
Mit der direkten Steuerung ist es möglich die Position für jedes Gelenk individuell festzulegen. Auf Knopfdruck nimmt der Arm diese dann ein.

- Positionsverwaltung

Die Positionsverwaltung erlaubt es neue Positionen zu definieren, indem die aktuelle Position per Knopfdruck an dem eingegebenem Index gespeichert wird. Dabei ist der User auf die Speicherplätze von 1 bis 100 beschränkt, damit er nicht wichtige Positionen überschreiben kann. Diese Beschränkung gilt aber nicht für das Laden einer Position. Hier kann der Anwender die gesamte Bandbreite von 1 bis 629 ausnutzen.

- Greifarm-Steuerung

Neben der Möglichkeit den Greifarm per Knopfdruck zu öffnen und zu schließen, kann der User außerdem noch die Lichtschranke abfragen. Das Resultat wird ihm dann im Statusfeld ausgegeben.

```
...
private void closeGripper()
{
    try
    {
        //Status des Greifarms prüfen und ggf. schließen
        if(parent.getControl().remoteGetGripperState(parent.getSessionID()))
        {
            parent.getBoard().addMessage("Schließe Greifer");
            parent.getControl().remoteCloseGripper(parent.getSessionID());
            parent.getBoard().addMessage("Greifer wurde geschlossen.");
        }
        else
        {
            parent.getBoard().addErrorMessage
            ("Greifer ist bereits geschlossen.");
        }
    }
    catch(RemoteException e)
    {
        //Auf RemoteException reagieren
        parent.getBoard().addErrorMessage
        ("RemoteException while closing gripper");
        ...
    }
}
...
```

Der Aufbau ist bei allen Methoden der Gleiche. Zunächst wird der Input überprüft und anschließend die Methode des Servers, der den Roboter steuert aufgerufen. Da es dabei Kommunikationsprobleme geben kann, muss die *RemoteException* abgefangen werden.

Diese Applikation setzt nur eine kleine Anzahl der Möglichkeiten, die der RM-501 bietet, um. Doch sie macht bereits deutlich, wie man mit Hilfe von RMI, eine GUI, wie es in der Automation üblich ist, auf ein anderes System auslagern kann. Auf die gleiche Weise lassen sich auch leicht verteilte Echtzeitanwendungen, wie sie zunehmend in solchen Systemen eingesetzt werden, realisieren. Dass VMs unterschiedlicher Hersteller problemlos untereinander kommunizieren können, wurde ebenfalls demonstriert.

### MoveMaster2RemoteControlApplet

*MoveMaster2RemoteControlApplet* bietet die gleiche Funktionalität wie die obige Applikation an. Allerdings wird hier für die Visualisierung ein Applet herangezogen. Dieses ist viel flexibler als die zuvor behandelte Lösung, da es nur eines geeigneten Browsers bedarf, um auf den Roboter zuzugreifen.

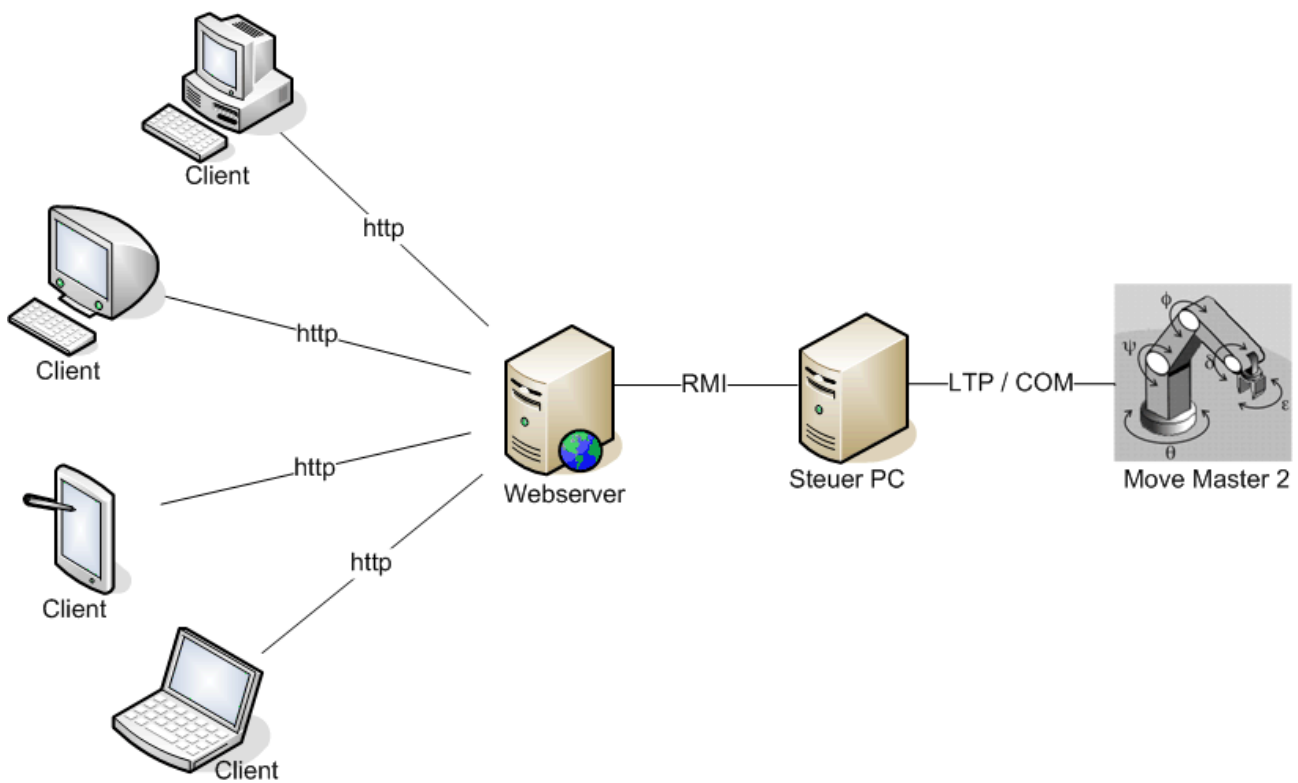


Abbildung 76: Schematischer Aufbau - Steuerung mit Hilfe eines Applet

Beispielsweise könnte man die dargestellte Architektur mit Hilfe eines Applets umsetzen. Ein Webserver bietet besagtes Applet einer Vielzahl unterschiedlicher Clients an. Er greift über RMI auf den Steuerrechner zu. Dieser übernimmt dann die eigentliche Ansteuerung des Move Master 2.

## Evaluation der Realtime Specification for Java anhand einer Robotersteuerung

Während das Applet bereits implementiert ist und mit dem *appletviewer* gestartet werden kann, stehen folgende Aufgaben dagegen noch völlig aus:

- Webserver einrichten
- Applet signieren<sup>22</sup>
- Zugriffsrechte für das Applet festlegen
- Applet in eine HTML-Seite einbinden

Das Umsetzen dieser offenen Punkte hätte den Rahmen dieser Arbeit gesprengt, da deren Fokus mehr auf der RTSJ liegt. Aber es wurde gezeigt, dass man die Internet-Technologien, die Java zu seinem Erfolg verholfen haben, auch in diesem Umfeld nutzen kann.

### Hinweis:

Zum besseren Verständnis wurde ein kleiner Film aufgenommen. Er kann unter *Movies* von CD gestartet werden.

---

<sup>22</sup> Ein Applet das RMI nutzt kann, wenn es nicht signiert ist, nur auf den Rechner auf dem es läuft zugreifen.

## 5. Fazit

Nachdem im Zuge dieser Arbeit sowohl ein größeres Projekt als auch eine Reihe kleiner Applikationen mit Hilfe der RTSJ umgesetzt wurden, hat man einen guten Überblick über diese bekommen.

### 5.1 Arbeitshinweise

Daher sollten folgende Hinweise bei der Arbeit mit der RTSJ beachtet werden. Sie stimmen im wesentlichen mit den Erfahrungen überein, die Peter Dibble [PD] gemacht hat.

- **Performance**  
Die Performance der Anwendung hängt hier, noch stärker als bei herkömmlichen Java, von der VM und dem Code des Entwicklers ab.
- **Threads & AsyncEventHandlers**  
Eine intuitive Möglichkeit um einen *Thread* oder einen *AsyncEventHandler* an die eigenen Bedürfnisse anzupassen, besteht in der Überlagerung der *run()*- bzw. *handleAsyncEvent()*-Methode.
- **Parallelität**  
Im Gegensatz zu herkömmlichen Anwendungen ist in der RTSJ fast alles auf eine parallele Nutzung ausgelegt.
- **RTGC**  
Auf Grund der RTGC können bei der JamaicaVM die Überlegungen bezüglich der Verteilung der Objekte auf die Speicherbereiche entfallen. Allerdings sind dabei die Verzögerungen zur Laufzeit zu beachten. Sollten dennoch andere Arten von Speicher als das *HeapMemory* zum Einsatz kommen, dann ist es empfehlenswert die Verteilung der Objekte schon im Vorfeld festzulegen. Dabei sollte *ImmortalMemory* sehr restriktiv Verwendung finden und möglichst Konstruktoren ohne Parameter bereit gehalten werden, da das Anlegen eines Objektes mit Parametern im *ImmortalMemory* mit großen Aufwand verbunden ist.
- **strictRTSJ**  
Diese Option, die bei der JamaicaVM verhindert, dass die Vorteile der RTGC genutzt werden und der Zugriff auf den Speicher nur nach den Regeln der RTSJ erfolgen kann, sollte nur, falls die Portabilität der Anwendung von Bedeutung ist, aktiviert werden. In diesem Zusammenhang sollte der *NoHeapRealtimeThread* nur, wenn es wirklich notwendig ist, eingesetzt werden. Sein Einsatz ist mit großem Aufwand verbunden. Er schreibt nämlich eine strikte Trennung zwischen dem Teil, der den Heap nutzen darf und jenem, der für harte Echtzeitbedingungen ausgelegt ist, vor. Auch muss auf eine sorgfältige Synchronisation geachtet werden, weil der *NoHeapRealtimeThread* sonst ausgebremst werden kann.
- **Exceptions**  
Da es sich bei den Exceptions um herkömmliche Objekte handelt, verbrauchen diese auch Speicherplatz. Dieser kann insbesondere bei einem Stacktrace recht hoch sein. Daher sollten sie möglichst nur im *HeapMemory* eingesetzt werden.

## 5.2 Schwächen

Beim Arbeiten mit der RTSJ haben sich einige Kritikpunkte herauskristallisiert. Diese lassen sich in 4 Kategorien gliedern:

### 5.2.1 Design

Einige der Klassen im package *javax.realtime.\** zeigen beim praktischen Arbeiten Schwächen im Design.

- **RealtimeThread**  
Die Klasse *RealtimeThread* unterstützt keine aperiodische Ausführung.
- **RelativeTime**  
Es fehlt die Möglichkeit eine relative Zeitangabe zu vervielfachen oder aufzuteilen.
- **ProcessingGroupParameters**  
*Scheduleable* lassen sich mit Hilfe von *ProcessingGroupParameters* zu Gruppen zusammenfassen. Allerdings lassen sich diese Gruppen nicht schachteln, was insbesondere für große Anwendungen ein Handicap darstellt. Darüber hinaus fällt die Analyse, ob ein *Scheduleable* zu einer solchen Gruppe hinzugefügt werden kann, zu rudimentär aus.
- **MemoryArea**  
Das Anlegen der Objekte in den 4 Speicherarten erfolgt jeweils auf unterschiedliche Art und Weise. Die strikte Trennung zwischen einem Teil, der harte Echtzeitbedingungen erfüllt, und einem anderen Teil, der den Komfort von Java bietet, ist auch von Nachteil.

### 5.2.2 Präzision

Die RTSJ ist für eine Spezifikation an manchen Stellen zu ungenau geraten. Dies ist insbesondere für sicherheitskritische Systeme ein untragbares Verhalten, da dort die Vorhersagbarkeit der Ausführung von entscheidender Bedeutung ist und eine Fehlinterpretation fatale Folgen haben kann.

- **PhysicalMemory**  
Das Konzept, das hinter den Klassen *VTPhysicalMemory*, *LTPhysicalMemory* und *ImmortalPhysicalMemory*, die den Zugriff auf Speicher mit speziellen Eigenschaften ermöglichen, steht, ist nicht detailliert genug beschrieben. Außerdem sind jene Eigenschaften, die diese Speicherbereiche auszeichnen, nicht als Konstanten vorgegeben, was die Plattformunabhängigkeit gefährdet.
- **AsyncEvent**  
Externe Ereignisse können mit Hilfe der Methode *bindTo(String)* intern als *AsyncEvent* abgebildet werden. Allerdings wurden auch hier keine Vorgaben gemacht, wie der *String*, der diese Ereignisse identifiziert, auszusehen hat.

- **PriorityScheduler**

Der default Scheduler der RTSJ, der *PriorityScheduler*, arbeitet mit Prioritäten um die Threads gegeneinander abzuwägen, wobei nur die Mindestanzahl der Stufen vorgegeben ist. Diese fehlende Präzision kann, wie bereits gezeigt wurde, fatale Folgen haben.

### 5.2.3 Plattformunabhängigkeit

Eine der größten Stärken von Java ist es, dass der Entwickler die darunter liegende Plattform nicht kennen muss. Doch bereits hier können einige deren Eigenschaften und Einstellungen, mit Hilfe von *System.getProperty(String)* abgefragt und behandelt werden. So kann man beispielsweise eine Applikation direkt mit der im System eingestellten Sprache starten. Da sich Embedded-Systeme aber stärker als andere Systeme unterscheiden und man dort spezielle Eigenschaften der HW nutzen kann und will, müsste das Angebot erweitert werden. Dies wurde bislang in der RTSJ versäumt.

### 5.2.4 Modularität

Ein Problem mit dem sich viele Embedded-System-Designer<sup>23</sup> konfrontiert sehen ist die Tatsache, dass die RTSJ mehr Funktionalität bietet, als sie eigentlich benötigen. Damit belegt die VM mehr Platz des kostbaren Speichers und wird komplizierter als es eigentlich nötig wäre. Sie fordern unter anderem, dass *StackedMemory* unterstützt wird, da es im Vergleich zum *ScopedMemory* weniger Ressourcen benötigt. Des weiteren würden sich diese gerne auf eine Thread Klasse und einen definierten Scheduler beschränken.

---

<sup>23</sup> [www.elecdesign.com/Articles/Index.cfm?AD=1&ArticleID=8116](http://www.elecdesign.com/Articles/Index.cfm?AD=1&ArticleID=8116)

### **5.3 Stärken**

Im Gegenzug punktet die RTSJ aber auch an vielen Stellen. Anders ist es nicht zu erklären, dass die Robotersteuerung derart komfortabel eingesetzt werden kann.

- **Integration**  
Dies ist unter anderem darauf zurückzuführen, dass die RTSJ sehr gut in Java integriert ist. Deutlich wird dies vor allem durch die Kombination mit verschiedenen bewährten Java-Technologien, wie es eindrucksvoll bei der Robotersteuerung demonstriert wurde.
- **Bewährte Konzepte**  
Weniger offensichtlich, aber mindestens ebenso bedeutsam, ist die Tatsache, dass man manchmal vergisst, dass man mit der RTSJ arbeitet. Der Übergang von herkömmlichen Java zu RTSJ geht dabei fließend, so dass man nur, wenn man wirklich auf ein befremdliches Konzept stößt, sich dieser Sache bewusst wird. Dies ist vor allem der Verdienst der sehr disziplinierten Erweiterungspolitik, die den bekannten Konzepten den Vorzug gab.
- **Vielseitigkeit**  
Auf Grund ihres Umfangs und ihrer Vollständigkeit lassen sich mit Hilfe der RTSJ fast alle Problemen lösen. Einzig und allein sehr eingeschränkte Systeme, auf denen eine VM unmöglich oder zu aufwendig wäre, fallen aus dem Rahmen.



## 5.4 Zukunft

Die Tatsache, dass es mit der JamaicaVM bereits Lösungen gibt, die ohne die problematische Aufteilung des Speichers auskommt, zeigt, dass in der RTSJ ein großes Potential steckt und das Thema Java und Echtzeit noch lange nicht abgeschlossen ist. Allerdings zeigt dies auch, dass man sich keines Falls auf den bisherigen Ergebnissen ausruhen darf. Insbesondere die größeren Projekte, die im Moment laufen und sich mit der Umsetzung größerer Systeme befassen, werden zeigen, von welcher Qualität die bisherigen Implementierungen sind und wo die RTSJ im Vergleich zu den gängigen Technologien steht. Allerdings bin ich, auf Grund der Erfahrungen, die ich mit der RTSJ gemacht habe, der Überzeugung, dass Java auch in diesem Segment seinen Siegeszug fortsetzen wird. Dabei stellt die hier eingesetzte Version in meinen Augen nur den ersten Schritt dar. Den entgültigen Durchbruch wird man vermutlich erst mit ausgereifteren Versionen, wie sie im Moment im Zuge des JSR-282<sup>24</sup> oder in der Safety bzw. Mission Critical Java Specification, von The Open Group<sup>25</sup>, erarbeitet werden, schaffen. Das hängt insbesondere damit zusammen, dass die Zielgruppe relativ konservativ ist und im Moment noch überzeugende Ergebnisse ausstehen.

---

<sup>24</sup> [www.jcp.org](http://www.jcp.org)

<sup>25</sup> [www.opengroup.org](http://www.opengroup.org)

## 6. Anhang

### 6.1 Literaturverzeichnis

#### 6.1.1 Realtime Specification for Java (RTSJ)

[RTSJ]

The Real-Time for Java Expert Group

**The Real-Time Specification for Java**

ADDISON-WESLEY, 31 Juli 2002

ISBN 0-201-70323-8

[CK]

Christos Kloukinas, Verimag

**Java & Real-Time - Advantages, Limitations & RTSJ**

29 Mai 2003

([info.in2p3.fr/page/formation/infoG/WEB\\_ECOLE-edition-2/JAVA-TR/in2p3.pdf](http://info.in2p3.fr/page/formation/infoG/WEB_ECOLE-edition-2/JAVA-TR/in2p3.pdf))

[CL]

Christophe Lizzi, Sun

**The Real-Time Specification for Java and related projects**

([www.laas.fr/~francois/STRQDS/reunions/010404/lizzi.pdf](http://www.laas.fr/~francois/STRQDS/reunions/010404/lizzi.pdf))

[PD]

Peter Dibble

**Notebook**

16 August 2005

([www.rtsj.org/docs/docs.html](http://www.rtsj.org/docs/docs.html))

[AK]

Andrzej Kononowicz

**The Realtime Specification for Java**

([www.in.tu-clausthal.de/fileadmin/fileadmin/ecker/RTSYSKap6.pdf](http://www.in.tu-clausthal.de/fileadmin/fileadmin/ecker/RTSYSKap6.pdf))

#### 6.1.2 Realtime Executive for Multiprocessor Systems (RTEMS)

[OAR\_1]

On-Line Applications Research Corporation (OAR)

**RTEMS C User's Guide for RTEMS 4.6.2**

30 August 2003

([www.rtems.com/onlinedocs/releases/rtemsdocs-4.6.2/share/rtems/html](http://www.rtems.com/onlinedocs/releases/rtemsdocs-4.6.2/share/rtems/html))

[OAR\_2]

NavIST Group - Real-Time Distributed Systems and Industrial Automation

**RTEMS 4.6.0 PC386 BSP HOWTO**

8 Mai 2003

[OAR\_3]

Joel Sherrill, On-Line Applications Research Corporation (OAR)

**The RTEMS Open Source Model: A Technical and Business Perspective**

2000

[OAR\_4]

On-Line Applications Research Corporation (OAR)

**Getting Started with RTEMS for 4.6.2**

22 Oktober 2003

([www.rtems.com/onlinedocs/releases/rtemsdocs-4.6.2/share/rtems/html](http://www.rtems.com/onlinedocs/releases/rtemsdocs-4.6.2/share/rtems/html))

### 6.1.3 JamicaVM

[RTGC]

Dr. Fridtjof Siebert, aicas GmbH

**Hard Real-Time Garbage Collection in Java Virtual Machines**

07.02.2000

([www.aicas.com/papers/jugs\\_oct01\\_slides.pdf](http://www.aicas.com/papers/jugs_oct01_slides.pdf))

[JVM]

Dr. Fridtjof Siebert, aicas GmbH

**JamaicaVM- Java for Embedded Realtime Systems**

25.10.2001

([www.aicas.com/papers/jugs\\_oct01\\_slides.pdf](http://www.aicas.com/papers/jugs_oct01_slides.pdf))

[MAN]

aicas GmbH

**JamaicaVM -- User Documentation: The Virtual Machine for Real-time and Embedded Systems**

### 6.1.4 Robotik

[IHME]

Prof. Dr Thomas Ihme, FH Mannheim

**Vorlesung Robotik SS 2005 - Teilsysteme**

2005

([www.informatik.fh-mannheim.de/~ihme/robotik\\_2205ss/index.html](http://www.informatik.fh-mannheim.de/~ihme/robotik_2205ss/index.html))

[MM2]

Mitsubishi

**Specification Move Master 2**

### 6.1.5 tech. Spezifikationen

[LPT]

Craig Peacock

**Interfacing the Standard Parallel Port**

1998

([www.beyoundlogic.org/spp/parallel.htm](http://www.beyoundlogic.org/spp/parallel.htm))

[COM]

Craig Peacock

**Interfacing the Serial / RS232 Port V5.0**

30 Januar 1998

([www.beyoundlogic.org/serial/serial1.htm](http://www.beyoundlogic.org/serial/serial1.htm))

## 6.2 Abkürzungsverzeichnis

Abkürzung	Bedeutung
ACK	ACKNOWLEDGE Eine Leitung der Parallelen-Schnittstelle, die zur Steuerung der Kommunikation eingesetzt wird. Sie kann über den <i>Status Port</i> direkt angesprochen werden. (Adresse: Base + 1, Bit 6).
AEH	Asynchronous event handling Von der RTSJ eingeführtes Konzept um schnell auf externe Ereignisse reagieren zu können. (siehe Seite 45ff)
AIE	AsynchronouslyInterruptedException Klasse die im Zuge der ATC Verwendung findet. (siehe Seite 50f)
API	Application Programming Interface Eine Schnittstelle, die ein Softwaresystem einem anderen anbietet. Dazu zählt beispielsweise eine Klassenbibliothek.
ASCII	American Standard Code for Information Interchange ASCII ist ein standardisierter Zeichensatz für Computer. Jedes Zeichen hat dabei eine Länge von 7 Bit.
ATC	Asynchronous transfer of control Von der RTSJ eingeführtes Konzept um den Programmverlauf sehr schnell und dynamisch zu ändern. (siehe Seite 49ff)
ATT	Asynchronous thread termination Die von der RTSJ vorgeschlagene Möglichkeit um Threads sicher zu beenden. (siehe Seite 54ff)
BIOS	Basic Input/Output System Ist die Software, die der Rechner direkt nach dem Einschalten ausführt. Zu seinen Aufgaben gehört ein System-Test und das Laden des Boot-Loader.
BSP	Board Support Package Ein BSP bietet die Funktionalität die für RTEMS notwendig ist um auf dieser HW zu laufen.
BUSY	BUSY Eine Leitung der parallelen Schnittstelle, die zur Steuerung der Kommunikation eingesetzt wird. Sie kann über den <i>Status Port</i> direkt angesprochen werden. (Adresse: Base + 1, Bit 7).
COM	COM In Windows-Systemen übliche Bezeichnung für die serielle Schnittstelle (RS-232). (Base: 0x3F8, 0x2F8,...)

Abkürzung	Bedeutung
CTS	Clear To Send Eine Leitung der serielle Schnittstelle, die zur Steuerung der Kommunikation eingesetzt wird. Sie kann über das <i>Modem Status Register</i> direkt angesprochen werden. (Adresse: Base + 6, Bit 4).
DMA	Direct Memory Access DMA ist eine Technik die es angeschlossenen Peripheriegeräten erlaubt direkt mit dem Arbeitsspeicher zu kommunizieren.
DPMA	dual ported memory areas Bei DPMA handelt es sich um Blöcke im RAM, die zwar einem Prozessor zugeordnet sind, aber dennoch von anderen Prozessoren genutzt werden können. Während der Prozessor, dem dieser Speicher gehört mit einer internen Adresse darauf zugreift, müssen die anderen diesen mit einer externen Adresse verwenden.
El Torito Standard	El Torito Spezifikation Die El Torito Spezifikation wurde 1995 von Phoenix Technologies und IBM herausgegeben. Sie beschreibt wie eine CD formatiert sein muss, damit Rechner direkt von CD booten kann.
GC	Garbage Collection Teil der automatischen Speicherverwaltung von Java, die dafür sorgt, dass unreferenzierter Speicher wieder freigegeben wird und es zu keiner Defragmentierung kommt.
gcc	GNU C Compiler Ein C Compiler der in der GNU Compiler Collection enthalten ist und unter der GPL vertrieben wird.
GMT	General Mean Time Uhrzeit in der Zeitzone durch die der Nullmeridian verläuft.
GRUB	Grand Unified Boot-Loader GRUB ist ein freier Boot-Loader der unter der GPL vertrieben wird. Seine Aufgabe ist es das Betriebssystem zu laden.
gzip	GNU zip Ist ein Kompressionsprogramm das von Jean-loup Gailly entwickelt wurde.
ITRON	ITRON ITRON ist in Japan der Standard für Embedded-Betriebssysteme.
J2SDK o. JDK	Java Development Kit Von Sun angebotene Sammlung von Entwicklungswerkzeugen für Java.
JamaicaVM	Jamaica Virtual Machine Eine Umsetzung der <i>Java Virtual Machine Specification</i> , inkl. der Zusätze durch die RTSJ, die von der aicas GmbH angeboten wird.

Abkürzung	Bedeutung
JB1	Jamaica Binary Interface Eine proprietäre API mit der es der JamaicaVM möglich ist native Funktionen bzw. Methoden zu nutzen. JB1 zeichnet sich im Vergleich zu JNI durch seinen geringen Overhead und seinem erhöhten Programmieraufwand aus.
JCP	Java Community Process Ein Standardisierungskomitee, das sich der Weiterentwicklung der Programmiersprache Java widmet (www.jcp.org)
JNI	Java Native Interface Eine Java Technologie mit der es möglich ist native Funktionen bzw. Methoden in Java zu nutzen und umgekehrt.
JRE	Java Runtime Environment Die JRE ist die abstrakte Plattform für die alle Java Anwendungen geschrieben werden. Eine VM emuliert diese auf der realen Plattform.
JSR	Java Specification Request Ein JSR kennzeichnet eine Änderung der <i>Java Language Specification</i> die von der JCP genehmigt wurde.
LPT	Line Printing Terminal Bezeichnung für die parallele Schnittstelle beim PC. (Base: 0x378, 0x278,...)
OAR Corporation	On-line Applications Research Corporation Ein 1978 gegründetes IT-Unternehmen aus Huntsville (USA), das die Entwicklung von RTEMS eingeleitet hat und betreut.
OS	operating system Das Betriebssystem eines Computers, das den grundlegenden Betrieb ermöglicht.
POSIX 1003.1b	Portable Operating System Interface for Unix Ist ein von der IEEE und der Open Group für Unix entwickelte standardisierte API. 1003.1b steht dabei für die Echtzeit-Erweiterung.
RAM	Random Access Memory Speicher bei dem direkt auf die einzelnen Speicheradressen zugegriffen werden kann. In dieser Arbeit ist er als Synonym für Arbeitsspeicher zu sehen.
RMI	Remote Methode Invokation RMI ist die Java eigene Umsetzung eines RPC. Mit deren Hilfe ist es möglich ein Objekt aus einer anderen VM anzusprechen. Sie zeichnet sich vor allem durch ihre gute Unterstützung und ihre Beschränkung auf Java aus.
rmic	RMI Compiler Der <i>rmic</i> ist teil des J2SDK und generiert die Klassen, die als Schnittstelle zur Kommunikationsschicht der RMI Architektur dienen.

Abkürzung	Bedeutung
RT	real-time Unter Echtzeit versteht man Systeme die synchron mit der realen Welt laufen.
RTEMS	Realtime Executive for Multiprocessor Systems Ein RTOS, das von der OAR Corporation verwaltet wird. (siehe Seite 72ff)
RTGC	Realtime Garbage Collection Eine GC welche die Anforderungen von Echtzeitsystemen erfüllt.
RTOS	real-time operating system Ein Betriebssystem das für den Einsatz in Echtzeitsystemen optimiert wurde.
RTS	Request To Send Eine Leitung der seriellen Schnittstelle, die zur Steuerung der Kommunikation eingesetzt wird. Sie kann über das <i>Modem Control Register</i> direkt angesprochen werden. (Adresse: Base + 4 ,Bit 1).
RTSJ	Realtime Specification for Java Eine Erweiterung der <i>Java Language Specification</i> und der <i>Java Virtual Machine Specification</i> für Echtzeit-Systeme im Zuge des JSR – 000001.
STROBE	STROBE Eine Leitung der parallelen Schnittstelle, die zur Steuerung der Kommunikation eingesetzt wird. Sie kann über den <i>Control Port</i> direkt angesprochen werden. (Adresse: Base + 2, Bit 0)
UTF-16	Unicode Transformation Format 16 Das UTF-16 Format ist eine optimierte Kodierung für Unicode. Jedes Zeichen hat dabei eine Länge von 16 Bit.
VM	Virtual Machine Eine virtuelle Maschine ist ein Programm, dass eine Plattform auf andersartigen System emuliert. In dieser Arbeit wird VM und JVM synonym verwendet.