GI/ITG Workshop: JAVA und Eingebettete Systeme manuscript No.
(will be inserted by the editor)

Eine JAVAVM für eingebettete 8-Bit-Systeme

Helge Böhme, Gerrit Telkamp, Ulrich Golze

Abt. Entwurf integrierter Schaltungen (E.I.S.)
TECHNISCHE UNIVERSITÄT BRAUNSCHWEIG
Gaußstraße 11-13, D-38106 Braunschweig, Germany
☎(+49) 531 391 3108, FAX (+49) 531 391 5840
e-mail: telkamp@eis.cs.tu-bs.de

26. Juni 1998

Zusammenfassung Während die meisten JAVAVM-Implementierungen 32-Bit-Prozessoren voraussetzen, haben wir eine JAVAVM für einen 8-Bit-Mikrocontroller implementiert. Damit können kostengünstige 8-Bit-Ein-Chip-Mikrocontroller in JAVA programmiert werden. Daß dabei nicht der volle JAVA-Umfang unterstützt wird, ist für unsere Anwendungen unerheblich.

0 Einleitung

Die Bedeutung von 8-Bit-Mikrocontrollern

In ihren Anforderungen unterscheiden sich eingebettete Systeme erheblich von herkömmlichen Universalsystemen. Größe, Stromverbrauch, Zuverlässigkeit und nicht zuletzt der Preis dürften die wichtigsten Kriterien sein, die bei der Konzeption eingebetteter Systeme im Vordergrund stehen. Daher wird die Hardware-Architektur meist für jeden Anwendungsfall maßgeschneidert.

In der Praxis sind je nach Leistungsbedarf Mikrocontroller mit einer Busbreite von 4, 8, 16 oder gar 32 Bit üblich. Trotz des fortwährenden Trends zu immer mehr Performance für zunehmend komplexere Anwendungen übersteigt auch heute noch die Anzahl der verkauften 4- und 8-Bit-Controller die der anderen Typen bei weitem. Das liegt vor allem an den vielen kleinen versteckten Helfern: Uhren, Telefone, Festplatten, Tastaturen, Mäuse, CD-Spieler, Videorecorder, automatische Fensterheber, Waschmaschinen usw. werden überwiegend mit 4- oder 8-Bit-Mikrocontrollern ausgestattet. Vor allem als Ein-Chip-Systeme, die RAM, ROM, EEPROM und Ein- und Ausgabekomponenten auf einem einzigen Chip vereinigen, finden Mikrocontroller dieser Art Verwendung. Daraus ergeben sich ideale Voraussetzungen für geringe Kosten, geringen Stromverbrauch und kleine Bauform sowie für größtmögliche Zuverlässigkeit.

Vorteile von JAVA für 8-Bit-Systeme

Die Vorteile von JAVA für eingebettete Systeme sind unbestritten. Die erheblich weniger fehleranfällige Programmierung, die objektorientierte und standardisierte Programmiersprache und eine Vielzahl von leistungsfähigen Entwicklungswerkzeugen sprechen dafür, Anwendungen für eingebettete Systeme mit JAVA zu entwickeln. Die JAVA-Spezifikation der Firma *Sun Microsystems* umfaßt sowohl eine Hochsprache als auch eine portable Maschinensprache, den sog. *Bytecode*. Dieser wird von der virtuellen Maschine JAVAVM, die auf dem Zielsystem abläuft, interpretiert [1]. Daraus ergibt sich neben den aufgezählten Vorteilen die größtmögliche Kompatibilität zwischen den unterschiedlichsten Systemen. Der Mikrocontrollertyp kann gewechselt werden, ohne die Software verändern zu müssen.

Außerdem kann durch die Interpretertechnik eine JAVAVM unerlaubte Aktionen von nachgeladenen Bytecode-Programmen abfangen. Insbesondere dann, wenn neue Applikationen in bestehende eingebettete Systeme nachgeladen werden sollen, ist das von großem Vorteil. Als Beispiel denke man sich eine Waschmaschine, in die ein neues JAVA-Waschprogramm geladen werden kann, das ganz gezielt die Vorteile eines weiterentwickelten Waschmittels nutzt. Um die Hardware vor möglichen Beschädigungen wie zu hoher Temperatur oder Schleuderdrehzahl zu schützen, ist es wichtig, daß die nachgeladenen Applikationen niemals unkontrollierten Zugriff auf die Peripherie haben können. Wenn die Applikationen darüber hinaus in einen nichtflüchtigen, wiederbeschreibbaren Speicher (z. B. Flash-Memory) geladen werden, lassen sich sehr oft neue Software-Versionen nachladen.

Der JAVA-Bytecode wurde für 32-Bit-Systeme konzipiert, d. h. Daten und Adressen umfassen mindestens 32 Bit. Auch wenn die Hochsprache Variablentypen von 8 oder 16 Bit Breite unterstützt, werden diese von der virtuellen Maschine intern als 32-Bit-Werte verarbeitet. Da aus den oben genannten Gründen für viele Anwendungen überwiegend 8-Bit-Mikrocontroller zum Einsatz kommen, liegt es nahe, trotz der zu erwartenden Schwierigkeiten eine JAVAVM für 8-Bit-Mikrocontroller zu implementieren.

Anwendungsgebiete

Die Anwendungsgebiete für die 8-Bit-JAVAVM sind vielfältig. Zunächst können dies alle denkbaren Anwendungen sein, die auch ohne JAVA auf diesen Controllertyp üblich sind, sofern sie nicht allzu große Anforderungen an die Rechengeschwindigkeit stellen (sie ist durch die Interpretertechnik bei JAVA prinzipbedingt geringer). Hauptdomäne sind hier reaktive Steuerungen, wie sie fast überall zu finden sind: Unterhaltungselektronik (TV-, Video und HiFi-Geräte), Weiße Ware (Geschirrspüler, Waschmaschinen, . . .), Heizungen (und Klimaanlagen), Uhren, Telefone, Alarmanlagen, etc. Der Hauptvorteil von JAVA ist hier die Flexibilität auf der Seite der Hardware (so kann z. B. aus Kostengründen in der laufenden Produktion eines Geräts der Controllertyp

gewechselt werden *ohne* die Software ändern zu müssen)[1] und die Flexibilität auf der Seite der Software (so kann sogar während des Betriebs die Software eines Geräts erneuert oder erweitert werden). Auch sind leicht zu programmierende Systeme für den Endanwender denkbar, vergleichbar z. B. mit der Basic-Briefmarke (Basic Stamp) der Firma Parallax, Inc., nur daß statt eines nicht standardisierten Basic-Dialekts eben JAVA verwendet wird.

1 Spezifikation für eine 8-Bit-JAVAVM

Eine vollständige Implementierung der von der Firma *Sun Microsystems* spezifizierten JAVAVM hat auf standardmäßigen Systemen ungefähr die Größe eines Megabytes. Dazu kommen dann noch die Klassenbibliotheken, ohne die keine Anwendungsklasse lauffähig ist (weitere Megabytes). Zur Laufzeit wird dann noch flüchtiger Speicher z. B. für Objekte benötigt, dessen maximale Größe von der Komplexität Anwendung abhängt. Unter Berücksichtigung der Ressourcen, die gewöhnlich auf einem 8-Bit Mikrocontroller vorhanden sind (der Adreßraum ist zumeist auf 64KByte begrenzt), ist die vollständige Implementierung der vorgegebenen JAVAVM nicht möglich. Für viele hier in Frage kommende Anwendungen ist diese auch nicht unbedingt sinnvoll, Funktionen für grafische Ausgaben oder für Internet-Zugriffe machen auf 8-Bit-Mikrocontrollern beispielsweise in der Regel keinen Sinn. Daher wird auf diesen Systemen nur eine Teilimplementierung der JAVAVM und der Klassenbibliotheken umgesetzt.

Der größte Einsparungseffekt hat sich bei den Klassenbibliotheken erzielen lassen, die nur zu einem kleinen Teil übernommen wurden (derzeit lediglich einige aus `java.lang.*`). Auch die JAVAVM wurde vereinfacht. Dabei ist auf einige primitive Datentypen (`int`, `long`, `float`, `double`), sowie auf alle Bytecodes, die diese Datentypen benutzen (Arithmetik- und Umwandlungsbefehle) und auf `wide`-Bytecodes verzichtet worden. Die Wortbreite, mit der die JAVAVM arbeitet, wurde von 32 auf 16 Bit verkürzt und so dem Adreßraum angepaßt. Konsequenterweise ergibt sich dadurch eine Inkompatibilität zu anderen JAVA-Implementierungen, allerdings werden für einfache Anwendungen selten Werte mit mehr als 16 Bit benötigt. Die Einschränkung ist für diese Fälle durchaus vertretbar.[2] Insgesamt konnte der ROM-Bedarf für die im Folgenden vorgestellte JAVAVM auf 20KByte beschränkt werden.

Da der flüchtige Speicher auf diesen Architekturen in der Regel sehr knapp bemessen ist (wenige KBytes), scheiden Optimierungen wie Just-In-Time-Compiler aus, denn der übersetzte Bytecode müßte zur Laufzeit dort zwischengespeichert werden. Der Bytecode wird daher direkt aus dem ROM interpretiert. Auch Optimierungen durch die sog. `_quick`-Pseudobefehle kommen deshalb nicht in Frage, da diese modifizierbaren Bytecode voraussetzen.

Abgesehen von diesen Einschränkungen wurde größtmögliche Kompatibilität zu den „großen“ JAVAVMs gewahrt. Ausgeführt werden JAVA-Klassen, die sich im ROM befinden müssen. Es findet keine Vorverarbeitung der Klassen durch Sekundärsoftware statt (im Gegensatz zu der JAVACard-Implementierung von *Sun*). Außerdem verfügt die JAVAVM für eingebettete 8-Bit-Systeme über weitere typische Elemente wie automatische Speicherbereinigung (Garbage Collection) und Multithreading.

Mikrocontroller stellen in der Regel Peripheriefunktionen zur Verfügung (z. B. Ein- und Ausgabepins, Schnittstellen, Timer, A/D- und D/A-Wandler). Um Zugriff auf diese Zusatzeinheiten zu erhalten, werden spezielle Klassen implementiert, die diese architekturunabhängig repräsentieren. Interruptgesteuerte Ereignisabarbeitung wird mittels einer mit JAVA-Objekten konfigurierbaren Zustandsmaschine realisiert.

Da JAVA von Haus aus Nebenläufigkeiten (Multithreading) zur Verfügung stellt, ist es sehr einfach, den reaktiven vom transformativen Teil (z. B. eines Reglers) zu trennen. Es ist auch möglich, einzelne Threads mit externen Ereignissen zu synchronisieren (sie werden von einem Timer- oder sonstigen Interrupt geweckt).

Als Entwicklungssystem steht jede Umgebung zur Verfügung, die gültigen JAVA Bytecode (in Form von `.class`-Files) erzeugt (z. B. `javac` von *Sun*). Ferner werden zukünftig Klassen existieren, die es ermöglichen, die Zielhardware auf dem Entwicklungsrechner auf Programmebene zu simulieren; für den Entwickler wird die Hardware grafisch dargestellt, und externe Ereignisse können dort generiert werden. Da die Software mit „großen“ JAVAVMs getestet wird, konnte auf die Bytecode-Verifikation auf dem Mikrocontroller verzichtet werden.

1.1 Einige Einschränkungen der 8-Bit-JAVAVM im Detail

1.1.1 Der Klassenlader Um einen wahlfreien Zugriff auf die Klassen, die im nichtflüchtigen Speicher abgelegt sind, zu erhalten, ist ein festgelegtes Format vorzugeben. In [3] ist dafür ein Multi-Class-File (`.mclass`) vorgesehen, da dies aber noch nicht verfügbar ist, ist als Zwischenlösung die Wahl auf das JAR-Format (JAVA-Archiv) gefallen. Dieses Format ist ebenfalls von der Firma *Sun* spezifiziert worden und ist kompatibel zu dem Zip-Standard. Der Vorteil des JAR-Formats besteht darin, daß es Plattformübergreifend ist und der (Ent-)Packer in JAVA implementiert ist. Es ist also überall dort, wo ein JAVA-Entwicklungswerkzeug installiert ist, benutzbar. Eine Einschränkung ist für das Archiv zu geben: Da auf dem Controller der Zugriff auf die Klassen direkt im JAR-Image erfolgt, ist das Archiv auf jeden Fall *ungepackt* zu erzeugen. Gepackte Archive müßten auf der Zielplattform in den flüchtigen Speicher entpackt werden, dieser ist allerdings in der Regel in noch geringerem Maß vorhanden als nichtflüchtiger Speicher und wird auch noch für Laufzeitobjekte benötigt.

1.1.2 Zugriffskontrolle und Bytecodeverifikation Auf die Zugriffskontrolle (`public`, `friendly`, `protected`, `private`) von Klassen, Datenfeldern und Methoden zur Laufzeit kann verzichtet werden. Auch werden die Klassendateien und der darin enthaltene Bytecode nicht auf Korrektheit überprüft, da dies einen nicht unerheblichen Aufwand

[1] Voraussetzung dafür ist natürlich, daß auf möglichst vielen Controllertypen JAVAVMs verfügbar sind.

[2] Ein sehr ähnlicher Optimierungsansatz wurde bei der JAVACard-Spezifikation [2] vorgeschlagen, die es in Zukunft möglich machen soll, JAVA-Programme auf Chipkarten auszuführen.

Java-(Speicher)Typ	Länge in Bits	Berechnungstyp
`boolean`	1	`short`
`byte`	8	`short`
`char`	16	`short`
`short`	16	`short`
`int`	32	`short` (!)
`long`	64	–
`float`	32	–
`double`	64	–

Tabelle 1 Speicher- und Berechnungstypen

darstellt. Da die Klassen ohnehin zuvor auf einem Entwicklungsrechner mit einer „großen“ JavaVM getestet wurden (oder gar ein externes Verifikationstool durchlaufen haben) und dort die Zugriffskontrolle und Bytecodeverifikation bestanden haben (siehe 1.3.2), ist die Notwendigkeit auch nicht gegeben. Es ist allerdings dafür Sorge zu tragen (systembedingt), daß keine ungeprüften Klassen auf den Controller gelangen und zur Ausführung kommen.

1.1.3 Native Methoden Native Methoden (das sind Methoden, die nicht in Java-Bytecode implementiert sind, sondern in dem plattformabhängigen Code für den speziellen Controller) sind zwingend notwendig, um die Bedienung der Hardware bewerkstelligen zu können oder auch nur spezielle Funktionen zur Verfügung zu stellen (Thread Management, Garbage Collection). Native Methoden sollen allerdings den Systemklassen im nicht konfigurierbaren Teil des Speichers vorbehalten bleiben und nicht den Applikationsklassen zur Verfügung stehen, um einen direkten Zugriff auf den Speicher oder die Hardware für den Applikationsprogrammierer auszuschließen.

1.1.4 Wortbreite und Datentypen Die Wortbreite bei Java wurde so festgelegt, daß Referenzen auf Objekte in einem Wort Platz finden [1] und beträgt daher normalerweise 32 Bit. Wie schon erwähnt, ist es auf Architekturen mit einem Adreßraum von 64KByte legitim, diese auf 16 Bit zu verkürzen. Dadurch ist es möglich, den Platzbedarf für Objekte auf dem Heap beinahe zu halbieren.

Verzichtet wird auf die primitiven Datentypen `int`, `long`, `float` und `double`, da für fast alle Berechnungen auf einem kleinen System 16-Bit-Festkomma-Zahlen ausreichend sind (das geht auch einher mit der Verkürzung der Wortbreite). Die Einsparung betrifft besonders die Bytecodes für Arithmetik, Typumwandlung und Load/Store, die diese Typen benutzen (etwa ein Drittel aller Bytecodes). Alle arithmetischen Berechnungen, die von einer JavaVM normalerweise über den `int`-Datentyp durchgeführt werden, finden nun als `short`-Berechnungen statt. Tabelle 1 faßt dies nocheinmal zusammen.

Felder (Arrays) sind nicht zwingend Notwendig und verbrauchen in der Regel viel Speicherplatz. In manchen Fällen ist die Verwendung von Feldern sinnvoll (z. B. Listenverwaltung etc.) und die Implementierung von eindimensionalen Feldern ist angebracht. Feldelemente des Typs `boolean` werden, im Gegensatz zu der Implementierung der Firma *Sun*, welche die Werte byteweise sichert, als Einzelbits gespeichert.

1.2 Verbotene Java Schlüsselworte

Die doch recht umfangreichen Kürzungen der JavaVM haben nur sehr begrenzt Auswirkungen auf die Programmiersprache Java. Da durch das Wegfallen einiger Bytecodes auch deren Eigenschaften verlorengehen, sind einige Schlüsselworte in Java-Quellen verboten:

`long`, `float`, `double`, (`int`).

Das Schlüsselwort `int` ist zwar weiterhin in Quellen erlaubt, die damit verbundenen Variablen und Operationen sind allerdings nicht mehr kompatibel zu der Definition von Integers (32 Bit vorzeichenbehaftete Festkommazahlen), sie arbeiten wie `short`-Variablen und -Operationen.

1.3 Neue Eigenschaften

Mikrocontroller für eingebettete Systeme haben in der Regel neben dem reinen Rechenkern noch weitere Einheiten auf dem Chip (Timer, Ein- und Ausgabepins, etc.). Andererseits fehlen auf so kleinen Systemen meist Elemente wie ein Dateisystem oder eine Konsole. Diese Umgebungsänderung hat auch Auswirkungen auf die JavaVM und die Java-Klassen. in Grenzfällen können sogar Inkompatibilitäten auftreten und es müssen fehlende Elemente ersetzt (oder modelliert) werden. Weiterhin ist die Familie der 8-Bit-Mikrocontroller sehr groß, mit sehr vielen unterschiedlichen Typen, die alle unter der gleichen Klassenbibliothek arbeiten sollen. Ein weiteres Problem ist, daß die Programme nicht auf dem Controller selbst entwickelt werden, sondern auf einem Entwicklungsrechner (PC oder Workstation) und so die Testfähigkeit der Programme in dieser Umgebung eine große Rolle spielt.

1.3.1 Die eingebettete Hardware Die Konfiguration des Prozessors (mittels seiner Hardwareregister), der Portpins (wie sind sie den Registern zugeordnet), der Interrupts, der Timer, etc. wird in Java-Klassen angegeben. Beispielsweise könnte der textuelle Benutzerdialog mit den Methoden `println` und `readLine` über die Standardkanäle `System.out` und `System.in` über eine serielle Schnittstelle abgewickelt werden. Da aber auch die *umgebende* Hardware (Aktoren und Sensoren), in die der Mikrocontroller eingebettet ist, nicht vorherbestimmt werden kann, liegt es nahe, die mit dem Controller verbundenen Einheiten (Lampen, Displays, Tasten, Relais, etc.) in Klassen festzulegen und so Programmierschnittstellen zur Verfügung zu stellen. Eine andere Möglichkeit zur Konfiguration sind Properties [5,6,7].

1.3.2 Entwicklungsumgebung Die Anwendersoftware soll mit jedem normalen Java Entwicklungswerkzeug geschrieben und übersetzt werden können. Es werden zwei Klassenpakete (mit identischen Aufbau und Schnittstellen) zur Verfügung gestellt. Eins ist zum Download auf den Controller vorgesehen (oder ist dort fest im (EP)ROM) und eins befindet sich auf dem Entwicklungsrechner, wo es exakt dasselbe Verhalten zeigt, wie das andere, zusätzlich aber noch das Systemverhalten der Hardware simuliert (dies kann z. B. mittels verschiedener Kontrollfenster geschehen).

Hier kann auch eine Kommunikation mit dem Controller implementiert werden. So kann nach erfolgreicher Simulation (und der damit verbundenen Verifikation des Bytecodes) ein Download erfolgen um die Anwendung im Zusammenspiel mit der realen Hardware zu testen. Dafür muß dann

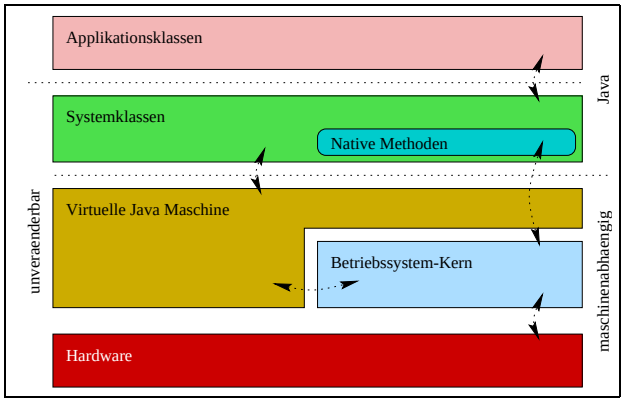

Abbildung 1 Der Aufbau des Gesamtsystems

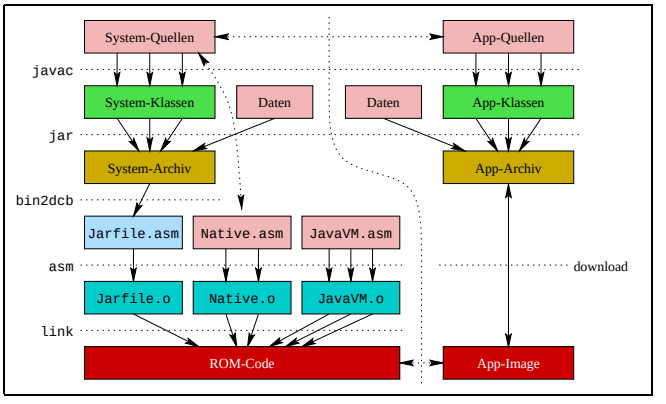

Abbildung 2 Der Entwicklungsprozeß

auf dem Controller eine Bootstrap- oder auch eine Monitor-Klasse vorhanden sein, die das Herunterladen automatisch (wenn keine Applikation vorhanden ist) oder auf externe Anfrage erledigt. Voraussetzung dafür ist natürlich wiederbeschreibbarer nichtflüchtiger Speicher.

1.4 Zusammenfassung der Spezifikation

1.4.1 Überblick auf das Gesamtsystem Zur Übersicht zeigt Abbildung 1 das Gesamtsystem. Es bestehen Unterschiede zu der von *Sun* vorgegebenen EmbeddedJava-Architektur [3]. So ist es nicht möglich, auf Applikationsebene native Methoden zu verwenden oder hardwareabhängige Klassen zu programmieren; dies aus zwei Gründen:

1. Mit nativen Methoden ist ein direkter Zugang zur Hardware und den Speicher möglich und das System wäre somit viel weniger sicher gegenüber Angriffen.
2. Der Applikationsprogrammierer soll von der Hardware soweit wie möglich abgeschirmt werden und stattdessen die hardwareunabhängigen Systemklassen verwenden (die ggf. natürlich hardwareabhängig sind, dies aber mit davon unabhängigen Schnittstellen).

Weiterhin ist die JavaVM nicht streng vom Betriebssystem getrennt. In der Tat ist des Betriebssystem ein Teil der VM, so ist im Normalfall der Thread-Manager (siehe 2.8) die oberste Instanz, der sich alle weiteren Komponenten unterzuordnen haben. Lediglich der Interrupt-Handler läuft „nebenher“ und kann Einfluß auf den Thread-Manager nehmen, wie auch umgekehrt.

1.4.2 Der Entwicklungsprozeß einer Applikation Abbildung 2 zeigt den Entwicklungsprozeß von Java-Applikationen auf der 8-Bit-JavaVM so, wie er im nächsten Kapitel implementiert ist. Gegebenenfalls wird sich dieser Prozeß im Lauf der Weiterentwicklung der JavaVM noch verändern, so ist eine Kompatibilität zur EmbeddedJava-Spezifikation von *Sun* angestrebt. Diese kann allerdings erst bei Verfügbarkeit der dazugehörigen Werkzeuge umgesetzt werden. Die Struktur ist also eher als Zwischenlösung gedacht.

Der Entwicklungsprozeß unterscheidet sich bei Systemklassen, welche sich zusammen mit der JavaVM fest im ROM befinden werden, von den Applikationsklassen, welche bei dem Download-Prozeß in Flash-Memory geschrieben werden. Enthält der Mikrocontroller keinen wiederbeschreibbaren nichtflüchtigen Speicher, so entfällt der zweite Weg und die Applikation wird ebenfalls ins ROM geschrieben. Die Prozesse im einzelnen:

Festverdrahtete Klassen: Die Klassen werden von einem normalen Java-Compiler (`javac`) oder anderen `.class`-File erzeugenden Werkzeugen erzeugt. An dieser Stelle kann nun ggf. eine Simulation des Systemverhaltens auf einem Entwicklungsrechner stattfinden, vorausgesetzt entsprechende Klassen, die den Controller und seine Umgebung emulieren, stehen zur Verfügung. Die `.class`-Files werden ggf. zusammen mit anderen Dateien in ein Java-Archiv, wie schon geschildert, geschrieben. Anschließend wird das Archiv in Assembler-Quellcode umgewandelt, damit es zusammen mit der JavaVM und den nativen Methoden zu einem ausführbaren ROM-Image für den Controller zusammengesetzt werden kann.

Applikationsklassen: Bis zur Erzeugung des Archivs wird der gleiche Prozeß verwendet. Der Weg über den Assembler wird allerdings ausgelassen, stattdessen muß der Controller in der Lage sein, auf die schon beschriebene Weise das Archiv direkt in einen nichtflüchtigen wiederbeschreibbaren Speicher zu laden.

In zukünftigen Versionen wird das JAR-Tool durch andere Programme ersetzt, die etwas differenzierter arbeiten können (z. B. `JavaCodeCompact` von *Sun*). Sie treffen eine automatische Klassenauswahl und vermindern auch die Größe von Klassen, indem nicht benutzte Teile entfernt werden. Auch ist an dieser Stelle eine Bytecodeverifikation wünschenswert (diese ist auf die *nicht* implementierten Eigenschaften der JavaVM abgestimmt), damit dies auf dem Controller entfallen kann. Ferner ist es auf der Seite der Systemklassen sinnvoll, eine Liste mit verwendeten nativen Methoden zu erzeugen, damit nur die wirklich benutzten wertvollen Speicherplatz belegen.

2 Eine Implementierung der 8-Bit-JavaVM

2.1 Der Zielprozessor

Als Zielprozessor für die erste Implementierung ist der ST7 von SGS-Thomson gewählt worden. Dieser hat einen

68HC05 Kern und ist zu diesem Code-Kompatibel. Es existiert eine große Familie von ST7 Prozessoren mit unterschiedlichen Ausbaustufen und Zusatzeinheiten. Das schließt (EP)ROM bis zu 48KByte, RAM bis zu 3KByte, EEPROM, zahlreiche serielle Schnittstellen (unter anderem auch i2c und USB), ADU, DAU, Timer, Watchdog, etc. ein. Die ST7 Familie wird von SGS-Thomson weiterentwickelt. Der zur Verfügung stehende Typ ST7285, welcher in einen In-Circuit-Emulator eingebettet ist, verfügt u. a. über 48K emuliertes ROM, 3K RAM, 62 I/O Pins, 2 Timer, ADU, i2c Interface, 4 serielle Schnittstellen und Watchdog. Er arbeitet mit einer maximalen Taktfrequenz von 8,664 MHz, welche intern um mindestens den Faktor zwei heruntergeteilt wird. Bei einem geschätzten durchschnittlichen Zyklenbedarf pro Instruktion von vier bis fünf ergeben sich ungefähr 0,8 bis 1,1 MIPS.

Die Komponenten der JavaVM

Ziel ist es die JavaVM möglichst klein und einfach zu halten, daher werden im Folgenden einige grundlegende Konzepte erläutert. Implementierungssprache ist ST7-Assembler. Die Entwicklung ist eine sgn. Clean-Room-Implementierung gemäß den Spezifikationen in [1], d. h. sie baut auf keinem Code anderer Implementierungen auf oder benutzt diesen gar. Die in Kapitel 1 vorgestellten Erweiterungen der JavaVM werden hier außer acht gelassen.

2.2 Heap

Alle zur Laufzeit erzeugten Datenstrukturen werden in einem zentralen Heap organisiert. Auf dem Heap werden die Java-Objekte (und Felder), die Laufzeitrepräsentation der Java-Klassen (Konstantenpool, Methoden- und Datenfeld-Tabellen), die Threads und die Rahmen der gerade ausgeführten Methoden abgelegt.[3] Um die Organisation des Heaps möglichst einfach zu halten, werden die verschiedenen Datenblöcke gleich behandelt. Da der Heap (und alle auf ihm abgelegten Datenblöcke) unter der Kontrolle der automatischen Speicherbereinigung (siehe 2.9) ist, wurde die Organisation des Heaps diesen Gegebenheiten angepaßt und für Objektreferenzen das Prinzip der sgn. *Handles*[4] verwendet. Der Heap besteht aus zwei Teilen:

Die Handletabelle befindet sich am oberen Ende des dem Heap zur Verfügung stehenden Speicherbereichs und wird von oben nach unten gefüllt. Die Handletabelle besteht aus Einträgen der Größe eines Wortes (16 Bit), welche die Zeiger auf die entsprechenden Datenblöcke enthalten. Die Position der Handletabelle und der darin befindlichen Zeiger „lebender“ Objekte ist für die gesamte Betriebsdauer der JavaVM fest (Handles bleiben immer gleich).

Die Datenblöcke selbst befinden sich am unteren Ende des dem Heap zur Verfügung stehenden Speicherbereichs. Neue Blöcke werden *immer* oberhalb des obersten Blocks angelegt[5], das erspart die Suche nach einem freien Block in einem fragmentierten Heap. Der Zugriff auf die Datenblöcke darf nur mittelbar über die Handles geschehen, d. h. der aus dem Handle ermittelte Zeiger darf nicht über die Grenzen einer atomaren Operation (siehe 2.5.2) hinaus benutzt werden (Zeiger können sich ändern).

Wird ein Block auf dem Heap angefordert, so wird zunächst geprüft, ob dieser in die Lücke zwischen den Datenblöcken und der Handletabelle paßt. Dafür wird neben einem Block-Typbezeichner (dieser wird vom Garbage Collector benötigt) die gewünschte Größe mitgeteilt. Zurückgegeben wird dann nur der Handle, also ein Zeiger auf den neuen Eintrag in der Handletabelle; dieser ist entweder ein „alter“ Handle eines freigegebenen Blocks, der zu Null gesetzt wurde oder ein neuer, für den die Handletabelle um einen Eintrag vergrößert wurde. Schlägt die Anforderung des Speicherblocks fehl, so wird lediglich Null zurückgegeben und es ist Sache der aufrufenden Routine, geeignet darauf zu reagieren (Garbage Collector aufrufen und wiederholt versuchen oder Fehlermeldung bzw. Exception auslösen).

2.3 Das JAR-Image

Zur Zeit gibt es nur einen Weg, die Java-Klassen auf den ST7 Mikrocontroller zu laden. Das JAR-File wird mittels des Assemblers in die JavaVM integriert (dazugelinkt). Leider bieten weder der Assembler noch der Linker die Möglichkeit, Binärdateien mit einzubinden, daher wird das JAR-File in Assembler-Quelltext umgewandelt (`dc.b` Daten). Das Umwandlungsprogramm `bin2dcb` ist in Java implementiert und daher plattformunabhängig.

2.4 Klasseninitialisierung

Ziel ist es, zur Laufzeit geeignete Datenstrukturen (im flüchtigen Speicher) zu erzeugen, um den Zugriff auf das `.class`-File zu beschleunigen. Dabei ist zwischen minimaler Zugriffszeit und minimalem RAM-Verbrauch abzuwägen. Als geeignet hat sich der Aufbau von Zugriffstabellen im RAM erwiesen (die Daten werden also nicht kopiert).

Prinzipielle Arbeitsweise Es wird eine Klassendatenstruktur (Klassenobjekt) im flüchtigen Speicher (auf dem Heap) angelegt. Möchte die JavaVM auf eine ihr unbekannte Klasse zugreifen, so wird eine Anfrage über den Klassennamen getätigt. Nun wird zunächst geprüft, ob die gesuchte Klasse bereits initialisiert worden ist (also das Klassenobjekt bereits vorhanden ist). Dafür werden alle Klassenobjekte auf dem Heap gesucht und deren Namen mit dem gesuchten verglichen. Wird die gesuchte Klasse dort nicht lokalisiert, so wird das JAR-Image nach dem entsprechenden `.class`-File durchsucht und bei Erfolg ein neues Klassenobjekt angelegt (die Klasse wird initialisiert). In beiden Erfolgsfällen wird die Klasse selektiert und der Handle des Klassenobjekts zurückgegeben.

[3] Nicht auf dem Heap befinden sich die lokalen Daten der einzelnen Assemblerroutinen, sie sind starr auf der Zeropage oberhalb der Systemregister abgelegt.

[4] Handle ist auch die Bezeichnung für Objektreferenzen innerhalb der Programmiersprache Java; Zeiger sind Java gänzlich unbekannt.

[5] Nach [5] ist diese Vorgehensweise bei JavaVMs üblich.

2.4.1 Klassenselektion Wird eine Klasse selektiert, so heißt das, daß der Handle des Klassenobjekts in einige Zeiger aufgelöst und so als aktuell gültig registriert wird. Die Daten in der Struktur werden also nachfolgenden Routinen direkt zur Verfügung gestellt (bei Objekten und Rahmen wird ähnlich verfahren). Die Folge ist ein schneller Zugriff auf die einzelnen Elemente in der Struktur. Die Selektion ist natürlich nur solange gültig, bis das Klassenobjekt vom Kompaktierer des Speicherbereinigers (siehe 2.9) verschoben wird. Es bieten sich zwei Wege an, dafür Sorge zu tragen, daß die Zeiger immer gültig sind:

1. Jede Klassenselektion nach einer atomaren Operation (siehe 2.5.2) aufheben oder zu Beginn einer atomaren Operation, die auf eine Klasse zugreifen muß, die Klasse erneut selektieren.
2. Der Kompaktierer trägt dafür Sorge, daß eine Klasse, deren Klassenobjekt verschoben wird, deselektiert wird.

Da jede Klassenselektion einen gewissen Aufwand bedeutet, ist 2 der Vorzug gewährt worden. Weiterhin wird bei jeder Rahmenselektion (siehe 2.5.1) die dazugehörige Klasse selektiert.

2.4.2 Konstantenpoolauflösung Bevor die eigentliche Konstantenpoolauflösung beginnt, wird zunächst die Tabelle der statischen Methoden im Klassenobjekt gefüllt. Eingetragen werden die Zeiger auf das `Code`-Attribut der Methoden (im `.class`-File, im ROM) [1]. Ist eine Methode `native`, so wird das entsprechende native `Code`-Äquivalent gesucht (siehe 2.7). Die statischen Variablen werden direkt im Klassenobjekt gehalten, sie wurden beim Löschen des Speicherblocks implizit mit Null initialisiert. Mit den Instanzvariablen und -methoden wird prinzipiell genauso verfahren, nur werden diese nicht im Klassenobjekt gesichert, sondern in den Objekten bzw. der Objekt-Daten-Struktur. Die Objekt-Daten-Struktur enthält alle objektspezifischen Daten, die für alle Instanzen einer Klasse gleich sind (also auch die virtuellen Methoden) und wird zusammen mit dem ersten Objekt *einmal* angelegt.

Die eigentliche Konstantenpoolauflösung geschieht nun in folgenden Schritten:

1. Durchsuchen des Konstantenpools im `.class`-File, dabei werden die Zeiger auf die jeweiligen Elemente in eine Tabelle im Klassenobjekt am entsprechenden Index eingetragen. Auf diese Weise ist später ein beinahe direkter Zugriff auf einen Eintrag im Konstantenpool mittels eines Index möglich. Diese Tabelle wird im Folgenden als *constant pool representation* (CPR) bezeichnet.
2. Die CPR wird optimiert (zahlreiche Einträge enthalten Querverweise). Diese Optimierung und auch die damit kombinierte Auflösung der symbolischen Referenzen wird von hinten nach vorne durchgeführt, da Querverweise immer auf einen weiter hinten liegenden Eintrag weisen, die hinteren also zuerst optimiert bzw. aufgelöst werden müssen. Die Auflösung symbolischer Referenzen ist entsprechend der Prinzipien Vererbung (inheritance) und Vielgestaltigkeit (polymorphism) nicht auf diese Klasse begrenzt sondern betrifft auch die Oberklassen. Externe Referenzen werden hier nur aufgelöst, falls die dazugehörige Klasse bereits initialisiert wurde, wenn nicht, so wird dies zusammen mit der Klasseninitialisierung beim ersten Zugriff erledigt.
3. Als letzter Schritt wird die Klasseninitialisierungsmethode aufgerufen (falls vorhanden). Das ist die Methode, die auf der Ebene die Programmiersprache JAVA keinen Namen hat und mit `static {...}` deklariert wird und auf der Ebene der JAVAVM (bzw. des `.class`-Files) den Namen `<clinit>` erhalten hat.

2.5 Bytecode Interpreter

Bevor auf die Funktionsweise des Bytecode Interpreters eingegangen werden kann, ist zunächst die Umgebung des Bytecodes zu beschreiben.

2.5.1 Der Rahmen Der JAVA-Bytecode befindet sich gewöhnlich im `Code`-Attribut einer Methode, definiert im `.class`-File [1] (Ausnahme: native Methoden; siehe 2.7). Dort sind auch weitere Informationen (z. B. über Speicherverbrauch zur Laufzeit oder Ausnahmen) vermerkt. Jede Methode hält, während sie ausgeführt wird, ihre Daten auf dem Heap im sgn. Rahmen (Frame).

Rahmen einrichten Soll eine Methode gestartet werden, so werden die benötigten Informationen dem `Code`-Attribut entnommen und ein dementsprechend großer Speicherbereich auf dem Heap angefordert. Die Struktur wird gefüllt, dabei wird der Handle des Rahmens, der gerade ausgeführt wird (der aufrufende Rahmen) vermerkt; dorthin wird zurückgekehrt, falls der neue Rahmen (also die Methode) beendet wird. Der virtuelle Programmzähler im Rahmen, initialisiert auf den Bytecodeanfang, wird sich bei der Ausführung ändern (er zeigt immer auf den nächsten auszuführenden Bytecode).

Rahmen selektieren und verlassen Aufgrund der nebenläufigen Struktur der JAVAVM (Multithreading; siehe 2.8) muß eine Möglichkeit geschaffen werden, die Ausführung eines Rahmens anzuhalten und bei einem anderen Rahmen fortzusetzen. Die Rahmenselektion funktioniert dabei ähnlich, wie die Klassenselektion (die folgenden Assembleranweisungen der JAVAVM greifen nun auf *diesen* Rahmen zu). Wird ein Rahmen (z. B. bei einem Threadwechsel) verlassen, so sind die aktuellen Daten (u. a. der virtuelle Programmzähler) in die Struktur zu sichern.

2.5.2 Die Hauptschleife des Bytecode Interpreters Wird ein Rahmen selektiert, so wird auch ein 8-Bit Zähler initialisiert (auf Null, entsprechend 256). Dieser wird bei jedem ausgeführten JAVA-Bytecode um mindestens eins verringert; erreicht der Zähler erneut Null, so wird der aktuelle Rahmen verlassen und die Kontrolle wieder der aufrufenden Routine übergeben (in der Regel ist das der Thread-Scheduler; siehe 2.8.1). Aufwendigere Bytecodes (z. B. Methodenaufruf, etc.) verringern den Zähler um entsprechend höhere Werte, das ermöglicht eine relativ gleichmäßige Verteilung der Rechenzeit auf verschiedene Threads.

Atomare Operationen Bei einer atomaren, also einer unteilbaren bzw. nicht unterbrechbaren Operation handelt es sich also um eine Java-Bytecode-Operation selbst. Da kein gerade in Ausführung befindlicher Bytecode unterbrochen werden kann (auch nicht durch externe Ereignisse), hat der Programmierer der JavaVM innerhalb der Bytecode-Grenzen, d. h. bei der Implementierung der Routinen für einen Bytecode, alle Möglichkeiten, die die Hardware ihm zur Verfügung stellt, ohne daß er auf eventuelle Unterbrechungen „mittendrin“ achten muß. Er muß lediglich sicherstellen, daß der Zustand beim Aufruf des Bytecodes am Ende wiederhergestellt wird (z. B. der Zustand des Stacks des ST7).

Sprungtabelle Der Bytecode Interpreter selbst funktioniert mittels einer Sprungtabelle, welche durch den Wert des Bytecodes indiziert wird [1] (siehe dafür auch Tabelle 3 im Kapitel 3.2.1). Die Beschreibung der einzelnen bytecodeimplementierenden Routinen soll hier auf nur ein paar wichtige Beispiele beschränkt bleiben:

2.5.3 Zugriff auf Datenfelder (`getfield`, `putfield`, `getstatic`, `putstatic`) Zunächst wird der referenzierte Eintrag in der CPR betrachtet. Ist der Eintrag noch in symbolischer Form vorhanden, so wird dieser jetzt aufgelöst und ggf. die dazugehörige Klasse initialisiert. Nun wird die referenzierte Klasse selektiert (bei `get-`/`putfield` wird stattdessen die Klasse des auf dem Operandenstack übergebenen Objekts verwendet). Der Index im entsprechenden Datenfeld wurde der ursprünglichen CPR entnommen (wurde eine symbolische Referenz aufgelöst wird der Index nun erst eingetragen). Erst nachdem auf diese Weise der Eintrag in der entsprechendem Struktur (im Klassenobjekt oder im Objekt) lokalisiert wurde, wird ermittelt, ob lesender oder schreibender Zugriff gewünscht war (der Bytecode wird betrachtet).

2.5.4 Methodenaufruf- und Rückgabebefehle Der Methodenaufruf funktioniert ähnlich, wie der Zugriff auf die Datenfelder. Die Aufgabe wird von zwei Routinen übernommen, getrennt für statische und virtuelle Methoden. Der Methodenaufruf gestaltet sich allerdings noch um einiges Aufwendiger, als der Zugriff auf Datenfelder:

- Ermittlung der Anzahl der Übergabeparameter. Dazu muß der symbolische Methodendeskriptor analysiert werden [1]. Das Ergebnis der Analyse wird in der CPR für spätere schnellere Zugriffe abgelegt.
- Wenn der Eintrag nur symbolisch vorhanden war, so ist zusätzlich zum Ausmachen der entsprechenden Methodenbeschreibung (inkl. Klasseninitialisierung) der Bytecode zu suchen und einzutragen. Es kann sich dabei um Bytecode im `.class`-File oder um native Methoden (siehe 2.7) handeln. Entsprechend den Regeln der Vererbung muß die Suche auch die Oberklassen einschließen.
- Ist ein Zeiger auf den `Code` der Methode ermittelt, ist ein neuer Rahmen einzurichten und zu selektieren, der Handle des aufrufenden Rahmens wird für die Rückkehr in diesem vermerkt. Der virtuelle Programmzähler der aufrufenden Methode wird auf den nächsten Bytecode gesetzt (nach `invokexxx`).
- Kopieren der Methodenparameter vom Operandenstack des aufrufenden Rahmens in die lokalen Variablen des neuen Rahmens.
- Als Letztes erfolgt die Methodensynchronisierung, d. h. der neue, jetzt aktuelle Rahmen wird zum nächstmöglichen Zeitpunkt (*vor* der Ausführung des ersten Bytecodes) verlassen und zum Thread-Scheduler zurückgekehrt (siehe 2.8.2), falls als unteilbar markierte Ressourcen belegt sind.

Die Rückkehr zur aufrufenden Methode ist einfach, es wird lediglich ihr Rahmen selektiert. Der Rahmen der verlassenen Methode wird nun nicht mehr benötigt und *explizit* vom Heap entfernt (der Garbage Collector wird so entlastet). Handelte es sich um eine synchronisierte Methode, so wird der mit dem Thread verknüpfte Monitor nun freigegeben. Bei den Bytecodes `ireturn` und `areturn` wird der Rückgabewert dem Operandenstack des verlassenen Rahmens entnommen und auf den Operandenstack des Rahmens gelegt, zu dem zurückgekehrt wird.

2.5.5 Auslösung und Behandlung von Ausnahmen In der aktuellen Implementierung der JavaVM ist ein vollständiger Ausnahmebehandler noch nicht vorhanden, so wird die `exception_table[]` einer Methode *nicht* ausgewertet. Stattdessen ist ein einfacher Fehler-Rückgabewert vorhanden, ist dieser gesetzt (ein Wert ungleich Null), so wird die JavaVM gestoppt. Bereits implementiert ist, daß im Falle eines Speicherüberlaufs der Garbage Collector aufgerufen wird (er erbt dann die Priorität des aktuellen Threads und dieser wartet auf ein Signal des Garbage Collectors und versucht es dann erneut).

Primär löst der Bytecode `athrow` eine z. Zt. unspezifizierte Ausnahme aus, alle anderen Bytecodes können ebenfalls Ausnahmen werfen, falls eine abnormale Situation aufgetreten ist. Allerdings wurde bisher auf die (recht häufige) Kontrolle von Referenzen auf Nullhandles verzichtet.

2.6 Objekte und virtuelle Methoden

Objekte werden mit einer Struktur auf dem Heap repräsentiert. Die Erzeugung eines Objekts kann nur durch den Bytecode `new` angestoßen werden, dabei werden alle Instanzvariablen zu Null initialisiert. Die Indizes in die Instanzvariablen sind in der CPR der dazugehörigen Klasse vermerkt, sie sind für alle Objekte einer Klasse gleich.

Der virtuelle Methodenaufruf gestaltet sich etwas schwieriger als der statische. Der Unterschied zwischen den beiden ist, daß bei statischen Methoden die Referenzen immer direkt in die Klasse weisen können, in denen sie definiert wurde (dezentral oder verteilt). Vererbte Methoden sind durch diesen „Querverweis“ direkt auffindbar. Instanzmethoden haben jedoch ihre Referenzen immer in ihrem Objekt, dabei geht die Information verloren, aus welcher Klasse sie stammen (die Vererbungshierarchie). Ein speicherplatzsparender Kompromiß ist es, nicht die Klassen der Methoden mit abzuspeichern, sondern lediglich wieviele Klassen zurückverfolgt werden müssen (von Klasse zur Oberklasse ...), den Vererbungsgrad. Der Overhead ist in der Regel nicht zu groß,

da tiefe Vererbungen eher selten auftreten. Wird eine Methode zum ersten mal aufgerufen, so wird der Methodenbereich der Klasse des Objekts nach dieser durchsucht (und ggf. die Methodenbereiche der Oberklassen) und die Daten in die Objekt-Daten-Struktur eingetragen (late binding).

2.7 Native Methoden

2.7.1 Die native Datenbank Es wird eine kleine Datenbank im ROM aufgebaut (sie ist Teil des Assemblerprogramms der JavaVM), sie besteht aus zwei Teilen. Zum einen existiert eine verkettete Liste, welche zunächst die Klasse bestimmt, der die angegebenen nativen Methoden zugeordnet werden sollen. Der Vergleich erfolgt immer symbolisch über den Namen der Klasse. Stimmt die Klasse überein, so wird die Liste mit den Methodenzeigern abgearbeitet, deren Einträge auf Strukturen weisen. Auch hier erfolgt der Vergleich symbolisch über den Methodennamen und den Methodendeskriptor. Der Ablauf ist, daß, wenn eine native Methode gesucht wird (z. B. bei der Klasseninitialisierung), nicht im Methodenbereich der Klasse nach dem `Code`-Attribut zu suchen, sondern die native Methoden-Datenbank zu durchsuchen. Im Erfolgsfall wird ein Zeiger zurückgegeben . Dieser Zeiger zeigt auf eine `Code`-Attribut kompatible Struktur. Die Methodenaufrufroutinen (`invokexxx`) brauchen also nicht zwischen Bytecode-Methoden und nativen Methoden unterscheiden; in beiden Fällen liegt ein `Code`-Attribut vor und ein entsprechender Rahmen wird angelegt.

2.7.2 Der `CALLNATIVE`-Bytecode Die JavaVM erwartet nun im `Code`-Attribut Bytecode und nicht controllerspezifischen Maschinencode. Das Problem wurde folgendermaßen gelöst: Von der Firma *Sun* wurden einige Bytecodes für den internen Gebrauch der JavaVM reserviert, sie dürfen von keinem Java-Übersetzter erzeugt werden und kommen daher nicht in `.class`-Files vor. Einer von diesen Bytecodes (254 oder `impdep1`) wird dazu verwendet, um den Bytecode Interpreter mitzuteilen, daß dem Bytecode nativer Code folgt, es handelt sich um einen Modus-Umschalter. Dieser Bytecode ist in der Regel der erste Bytecode, der sich in einer nativen Methode befindet (wenn auch prinzipiell hier reiner Bytecode –also *kein* Maschinencode – stehen könnte).

Das umgekehrte Äquivalent zu dem nun `CALLNATIVE` genannten Bytecode ist die Routine `leavenative`. Ihr Aufruf schaltet zurück auf den Bytecode Interpreter. Zusätzlich wird übergeben, um welchen Wert der laufende Zähler in der Hauptschleife der JavaVM verringert werden soll (das Bytecodeäquivalent), dies um einen möglichst gleichmäßigen Ablauf von mehreren Threads zu gewährleisten. Sehr lange native Methoden können mit `leavenative`–`CALLNATIVE`–Paaren mehrfach aufgeteilt werden, damit der Ablauf anderer Threads nicht zu sehr beeinflußt wird. Eine native Methode kann beendet werden, indem zunächst mit `leavenative` auf die Bytecode-Ebene gewechselt wird und dann einer der Bytecodes `RETURN`, `IRETURN` oder `ARETURN` folgt.

2.8 Threads

Auch Threads haben eine Repräsentation auf dem Heap. Um einen Thread einzurichten, wird ein Zeiger auf ein `Code`-Attribut und die gewünschte Priorität des Threads übergeben. Prioritäten sind von der Programmiersprache Java auf die Grenzen 1 (`MIN_PRIORITY`) bis 10 (`MAX_PRIORITY`) festgelegt; die normale Priorität ist 5 (`NORM_PRIORITY`). Ein Rahmen wird angelegt und zusammen mit der Priorität in der Thread-Struktur vermerkt. Finden bei der Ausführung des Threads Methodenaufrufe statt, so wird sich der Eintrag des Rahmens entsprechend ändern.

Wie werden nun die Threads ausgeführt?

2.8.1 Der Scheduler Auf dem Heap befinden sich ein oder mehrere Thread-Strukturen. Die Arbeitsweise des Thread Schedulers gliedert sich nun in folgende Schritte:

Durchsuchen des Heaps nach laufenden Threads:
: Die Threads werden nacheinander im Heap lokalisiert. Bei jedem Thread werden Flags geprüft. Bei jedem laufenden und nicht wartenden Thread wird ein Zähler um die Priorität des Threads erhöht und das Maximum aller Zählerstände ermittelt. Läuft gegenwärtig kein Thread (bzw. laufende Threads warten) so wird der ST7 in den *Wartezustand* versetzt, jetzt kann nur ein externes Ereignis (ein Interrupt) den Controller wieder zum Arbeiten bringen. Die Interruptroutine ist also angehalten, den Threads Signale zukommen zu lassen.[6] Wurde der Wartezustand verlassen, so beginnt dieser Schritt von neuem. Ist kein Thread mehr auf dem Heap, so wird der Thread Scheduler mit einer Ausnahmemeldung verlassen.

Auswahl des auszuführenden Threads:
: Der Heap wird noch einmal nach Threads durchsucht, die Suche wird jetzt bei dem Thread abgebrochen, bei dem der Zählerstand dem zuvor ermittelten Maximalwert entspricht. Damit mehrere Threads gleicher Priorität auch garantiert gleichhäufig aufgerufen werden, erfolgt diese Suche nicht von vorne im Heap, sondern beginnend vom letztausgeführten Thread zyklisch. Das ist der Übergang vom prioritätsgesteuerten Scheduler zum Round-Robin-Scheduler. Der Zähler des ausgewählten Threads wird auf Null zurückgesetzt und der entsprechende Rahmen wird selektiert. Der Bytecode Interpreter wird für die obligatorischen 256 Bytecodes gestartet. Kehrt dieser zurück, so wird der Fehlercode im Rückgabewert ausgewertet. Liegt kein Fehler vor, so wird der aktuelle Rahmen in der Thread-Struktur gesichert und wieder von vorne begonnen. Gibt der Fehlercode an, daß die oberste Methode beendet wurde, so wird der aktuelle Thread beendet, indem dessen Struktur explizit vom Heap entfernt wird. Alle anderen Fehlermeldungen beenden den Thread Scheduler (*alle* Threads werden beendet), die Fehlermeldung wird durchgereicht.

2.8.2 Synchronisierung In dieser einfachen Implementierung einer JavaVM erhalten die einzelnen Threads keine lokalen Kopien von den Daten (Klassen, Objekte und Methoden), da der flüchtige Speicher knapp bemessen ist. Alle Threads greifen auf dieselben Daten zu, die in [1] beschriebenen Aktionen und Operationen *Benutzen*, *Zuweisen*, *Speichern*, *Laden*, *Sperren* und *Freigeben* verlieren somit ihre Bedeutung. Auch ein Abgleich mehrerer lokaler Kopien von

[6] In der aktuellen Implementierung der JavaVM sind Interruptroutinen noch nicht implementiert.

Daten entfällt somit, das Datenfeld-Flag `ACC_VOLATILE` [1] wird ignoriert. Für einen deterministischen Ablauf mehrerer Threads, die dieselben Daten manipulieren, ist weiterhin der Mechanismus der Monitore erforderlich. Ein Monitor besteht aus einem Thread-Handle und einem laufenden Zähler, er kann auf Klassen (`static`) und auf Objekte angewendet werden. Monitore können auf zweierlei Weise erworben und freigegeben werden:

1. das Bytecode-Paar `monitorenter` – `monitorexit` (das Java-`synchronized(){}`-Konstrukt),
2. als `ACC_SYNCHRONIZED` [1] markierte bzw. als `synchronized` deklarierte Methoden.

Beide Mechanismen funktionieren prinzipiell auf die gleiche Weise, lediglich der Ort der Implementierung ist ein anderer (Methodeninitialisierung bzw. -Rückkehr und die erwähnten Bytecodes).

2.8.3 Threads auf Java-Ebene Mit den bisher beschriebenen Mechanismen ist lediglich der Programmierer der JavaVM in der Lage, Nebenläufigkeiten (also Threads) zu erzeugen. In der Programmiersprache Java ist es vorgesehen, auf der Applikationsprogrammebene Threads zu erzeugen und zu manipulieren [4,5,6,7]. Dazu dient (größtenteils) die Klasse `java.lang.Thread`, diese wurde übernommen und ein wenig gekürzt. Dort sind die meisten Methoden lediglich deklariert und als native Methoden implementiert worden, da die Funktionalität anders nicht realisierbar wäre (so müssen z. B. Daten manipuliert werden, die Java nicht zugänglich sind).

2.9 Garbage Collection

Implementiert wurde ein *konservativer*, *inkrementeller*, *kompaktierender*, *dreifarb*, *mark-and-sweep* Garbage Collector [8]. Die Hauptschleife existiert dabei als native Methode, d. h. der Mechanismus der nativen Methoden wurde benutzt, um den Garbage Collector einfach, unkompliziert und verträglich mit den anderen Komponenten der JavaVM zu verbinden; zu dem Garbage Collector existiert *keine* korrespondierende Java-Klasse.

Alle anderen Komponenten der JavaVM wurden bereits auf eine Zusammenarbeit mit einem Garbage Collector ausgelegt (als wichtigstes sind dabei die Handles zu nennen). Gestartet wird der Garbage Collector, indem auf der Hauptprogrammebene der JavaVM ein Thread der Garbage Collector Methode eingerichtet wird. Dessen Priorität wird auf den kleinstmöglichen Wert (`MIN_PRIORITY`) gesetzt, um andere Threads (das Applikationsprogramm) nicht stark zu beeinflussen. Der Garbage Collector arbeitet so also unauffällig als Hintergrundthread. Um die Unauffälligkeit zu maximieren, ist das Hauptprogramm des Garbage Collectors an geeigneten Stellen mit `leavenative`–`CALLNATIVE`–Paaren versehen (gemäß den Vorgaben der nativen Methoden; siehe 2.7.2). Der Garbage Collector ist also in der Lage, seine Arbeit anzuhalten (der Thread wird verlassen) und später an derselben Stelle fortzufahren, da er also in kleinen voneinander unabhängigen Einheiten arbeitet, nennt er sich inkrementell. Zu beachten ist, daß der Heap in den Pausen von anderen Threads verändert werden kann, der Garbage Collector reagiert darauf robust.

2.9.1 Dreifarb Markierung Der Garbage Collector verwendet für das Mark-and-Sweep Verfahren den sgn. Dreifarb-Markierungs-Algorithmus [8]. Im Folgenden werden die einzelnen Arbeitsschritte des Garbage Collectors beschrieben:

Vorbereiten: Alle Blöcke auf dem Heap werden weiß markiert (mit einer Ausnahme: siehe unten). Dazu wird die Handletabelle durchgegangen und Flags werden gesetzt, Null-Handles werden dabei ignoriert. Um den Blocktyp zu bestimmen, wird der Handle aufgelöst und der Header des Blocks betrachtet. Handelt es sich um einen Thread, so wird dieser stattdessen grau markiert. Threads dienen als Grundlage der Garbage Collection.

Durchsuchen: Dieser Schritt wird mehrmals durchlaufen. Die Handletabelle wird abermals sequentiell abgearbeitet. Alle als grau markierten Blöcke werden nun genauer betrachtet. Zunächst wird der Block selbst schwarz markiert und dessen Typ im Blockheader festgestellt. Der Blocktyp dient nun als Grundlage für gesonderte Behandlungsroutinen, sie betrachten nun die ihnen bekannten Einträge in den Blöcken und entnehmen Referenzen auf andere Blöcke.

Die Referenzen werden auf Gültigkeit überprüft, d. h. es handelt sich um gültige Einträge in der Handletabelle, und ggf. grau markiert (sie werden also beim nächsten Durchlauf ebenfalls betrachtet), dies aber nur wenn sie noch nicht schwarz markiert sind, um mehrfache Betrachtung oder gar zyklische Betrachtung auszuschließen. Nun kann es vorkommen, daß eine Referenz gar kein Handle war, sondern ein anderes Datum (z. B. eine `short`-Zahl), die *zufällig* denselben Wert hatte wie ein gültiger Handle. Der Garbage Collector hat keine Möglichkeit, dies festzustellen und muß den dazugehörigen Block *sicherheitshalber* erhalten, er heißt deshalb konservativ. Wurde kein weiterer Block grau markiert, so existieren auf dem Heap nur noch weiße (unbenutzte) und schwarze (benutzte) Blöcke und die Schleife wird verlassen.

In einer weiteren Schleife werden nun die unbenutzten Blöcke vom Heap entfernt. Handelt es sich bei dem zu entfernenden Block um ein Java-Objekt, so wird zuvor die virtuelle `finalize`-Methode aufgerufen[7].

Kompaktieren: Hierfür muß die Handletabelle ebenfalls mehrfach durchsucht werden. Zunächst wird der am weitesten unten im Heap liegende benutzte (also als schwarz markierte) Block gesucht. Liegt er nicht ganz unten an, so wird er an diese Stelle verschoben. Nun werden alle weiteren nächsthöheren benutzten Blöcke gesucht und ggf. an den darunterliegenden herangeschoben. Verschoben wird ein Block, indem sein Inhalt samt Header an die neue (weiter unten liegende) Stelle kopiert wird und die neue Adresse in der Handletabelle an der entsprechenden Stelle eingetragen wird (dies ist eine atomare Operation). Überschrieben werden dabei nur nicht mehr benutzte Blöcke bzw. der Quell-Block selbst. Zuletzt werden die Threads, die auf die Beendigung des Garbage Collectors warten, geweckt. Der Kompaktierer benötigt von

[7] Die `finalize`-Methode ist in `java.lang.Objekt` deklariert, also in *allen* Objekten verfügbar.

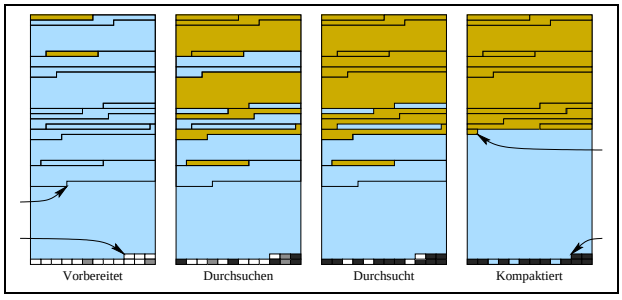

Abbildung 3 Die Arbeitsweise des Garbage Collectors

allen Teilen des Garbage Collectors die meiste Rechenzeit.

Abbildung 3 zeigt diese Schritte nocheinmal in Momentaufnahmen. Der Garbage Collector arbeitet diese Schritte immer wieder ab, er beendet sich nie. Die Robustheit des Garbage Collectors beruht auf der Tatsache, daß Blöcke, die auf dem Heap neu angelegt werden, immer schwarz markiert sind, also vom Garbage Collector nicht zerstört werden können. Weiterhin durchsucht der Garbage Collector die Handletabelle immer wieder neu, so daß Änderungen sofort auffallen bzw. keine fehlerhaften Funktionen auslösen können.

2.10 Das Hauptprogramm

Das Hauptprogramm der JavaVM muß die notwendigen Initialisierungsaufgaben übernehmen. Zu nennen sind:

- Unterprogramme initialisieren
- Thread des Garbage Collectors erzeugen
- Die Klasse des Java-Hauptprogramms einrichten: Der Klassenname ist als Konstante definiert.
- Die `main()`-Methode in der Klasse suchen: Dies ist gewöhnlich die Methode, die eine JavaVM als erstes startet. Name und Deskriptor der Methode sind ebenfalls als Konstanten definiert und lauten „`main`“ bzw. „`([Ljava/lang/String;)V`“.
- Thread der Hauptmethode erzeugen
- Den Thread Scheduler aufrufen

3 Leistungsfähigkeit

3.1 Funktionstest

Im Folgenden wird ein kleines Testprogramm für die JavaVM vorgestellt und besprochen. Tabelle 2 zeigt die Java-Quelle der Klasse `PortTest`. Das `.class`-File befindet sich zusammen mit den außerdem benötigten Klassen `controller.PortPins`, `java.lang.Object` und `java.lang.Thread` in einem JAR-Image, das auf die beschriebene Weise (siehe 2.3) in den nichtflüchtigen Speicher des ST7 geschrieben wurde.

`PortTest` erzeugt zwei Threads mit Lauflichtern und besteht aus folgenden Komponenten:

Die statische Methode `main()`: Da die Klasse bei der JavaVM als diejenige registriert ist, die gestartet werden soll, wird diese Methode zuerst ausgeführt.

```
// Copyright 1998 Helge Böhme

import controller.*;
import java.lang.*;

public class PortTest extends Thread{
  static void main(String args[]){
    PortPins.portmode((byte)0,(byte)0x40);
    PortTest thread1=new PortTest((byte)0,(short)50);
    PortTest thread2=new PortTest((byte)4,(short)40);
    thread1.start();
    thread2.start();
  }
  private byte num;
  private short anz;
  PortTest(byte num,short anz){
    this.num=num;
    this.anz=anz;
  }
  public void run(){
    while(true){
      for(short i=0;i<anz;i++){
        for(byte j=num;j<(byte)num+4;j++){
          PortPins.setpin(j,true);
          PortPins.setpin(j,false);
        }
        for(byte j=(byte)(num+3);j>=num;j--){
          PortPins.setpin(j,true);
          PortPins.setpin(j,false);
        }
      }
      { int i=getPriority();
        if(i==MAX_PRIORITY) i=MIN_PRIORITY;
        else i++;
        setPriority(i);
      }
    }
  }
}
```

Tabelle 2 `PortTest.java`

Der Konstruktor `PortTest()`: Dieser wird bei (ordnungsgemäßer) Erzeugung von Instanzen der Klasse `PortTest` aufgerufen. Es werden die Methodenparameter in Instanzvariablen geschrieben.

Die Instanzmethode `run()`: Da die Klasse `PortTest` die Klasse `Thread` ableitet, stehen deren Methoden für Threaderzeugung und -verwaltung zur Verfügung. Threads werden beim Erzeugen einer Instanz der Klasse eingerichtet und mit der Methode `start()` beginnen sie mit der Arbeit. Neue Threads beginnen immer mit der Methode `run()`.

Fazit Der Funktionstest war erfolgreich. Das relativ kurze Programm hat folgende Komponenten der JavaVM getestet:

Heap: Es wurden zahlreiche Blöcke unterschiedlicher Größe auf dem Heap erzeugt und ggf. explizit entfernt (Verlassen eines Rahmens, Beenden eines Threads). Der Zugriff über Handles funktionierte.

JAR-Image: Klassendateien konnten über ihren Klassennamen lokalisiert werden.

Klasseninitialisierung: Klassen wurden samt ihrer Oberklassen korrekt initialisiert. Klasseninitialisierungsmethoden wurden ausgeführt. klassenübergreifende statische Querverweise wurden gefunden und vermerkt. Symbolische Referenzen konnten aufgelöst werden.

Objekte: Instanzen einer Klasse wurden ordnungsgemäß erzeugt. Der Zugriff auf Instanzvariablen war möglich und Instanzmethoden wurden richtig referenziert.

Native Methoden wurden in der Datenbank ordnungsgemäß lokalisiert. Der Wechsel zwischen dem Bytecode- und dem nativen Modus funktionierte.

Methodenaufruf: Methoden konnten ordnungsgemäß lokalisiert werden, dies gilt auch für virtuelle Methoden. Rahmen wurden korrekt eingerichtet und verlassen. Die Übergabe von Aufrufparametern funktionierte ebenso wie die Übergabe von Rückgabewerten.

Threads: Es war möglich neue Threads zu erzeugen und zu manipulieren. Der Wechsel zwischen den Threads (Context-Switch) funktionierte. Threads wurden entsprechend ihrer Priorität richtig ausgewählt. Die Flags wurden richtig beachtet.

Bytecode Interpreter: Die Auswahl der entsprechenden Routine über die Sprungtabelle funktionierte.

Speicherbereinigung: Der Garbage Collector ist in der Lage zwischen noch benutzten Blöcken und unbenutzten zu unterscheiden. Er arbeitet unauffällig im Hintergrund. Der Heap wird ohne Auswirkung auf die übrigen Threads ordnungsgemäß kompaktiert.[8]

Sicherlich kann das Programm nicht alles testen und das, was getestet wird, nicht bis in alle Einzelheiten und Sonderfälle. Es zeigt dennoch, daß eine JavaVM auf einem solch kleinen System, wie dem ST7, lauffähig ist und – mit entsprechenden Java-Klassen versehen – auch sinnvolle Aufgaben übernehmen kann.

3.2 Geschwindigkeitsabschätzungen

Das zuvor gezeigte Testprogramm läuft mit akzeptabler Geschwindigkeit. Beim Betrieb des ST7 mit 8,664 MHz (also einer Core-Frequenz von 4,332 MHz) ist mit bloßem Auge lediglich ein Flackern der Leuchtdioden auszumachen, die Hin-und-Her–Bewegung ist verschwommen. Voraussetzung dafür ist natürlich, daß der dafür zuständige Thread eine ausreichend hohe Priorität besitzt. Eine genauere Betrachtung der Ausführungsgeschwindigkeit folgt.

3.2.1 Interpreter-Overhead Im Folgenden wird die absolute obere Schranke der Ausführungsgeschwindigkeit der JavaVM bestimmt. Folgendes wird hierfür außer acht gelassen:

- Thread-Scheduling,
- Speicherbereinigung,
- Initialisierungen aller Art,
- die Ausführungsdauer der Bytecodes selbst (es wird der Bytecode `nop` benutzt).

Tabelle 3 zeigt den Assemblerquellcode der Bytecode Interpreter Hauptschleife mit der Anzahl der Taktzyklen, die die jeweiligen Instruktionen benötigen [12]. Weiterhin findet Beachtung, zu welchem Anteil ein Pfad durchlaufen wird; es

[8] Obwohl nicht dauernd neue Objekte erzeugt und wieder vergessen werden, der Heap also aktiv gefüllt wird, ist der Garbage Collector für einen ordnungsgemäßen Betrieb der JavaVM nötig. Es kann bei Thread-Wechseln vorkommen, daß dadurch die notwendige Schachtelung von erzeugten und zerstörten Rahmen nicht eingehalten werden kann und der Heap dadurch fragmentiert wird und sich füllt. Ohne den Garbage Collector würde er langsam überlaufen.

Zyklen	Anteil	Quellcode		
5	1	`loopinterpr:`	`dec`	`vmcount.b`
3	1		`jreq`	`leavethread`
6	1		`callr`	`interpret`
3	1		`add`	`a,jpc+1.b`
4	1		`ld`	`jpc+1.b,a`
3	1		`jrnc`	`loopinterpr`
5	1/256		`inc`	`jpc.b`
3	1/256		`jra`	`loopinterpr`
6	1	`interpret:`	`ld`	`x,[jpc]`
7	1	`interpret2:`	`ld`	`a,(bytecode1,x)`
3	1		`push`	`a`
7	1		`ld`	`a,(bytecode,x)`
3	1		`push`	`a`
6	1		`ret`	
2	1	`nop:`	`ld`	`a,#1`
6	1		`ret`	
64,03125 → 67654 `nop`-Bytecodes/s (bei 4,332 MHz)				

Tabelle 3 Zyklenbedarf der Bytecode-Interpreter-Hauptschleife

wird die gewichtete Summe gebildet. Es ergeben sich etwas mehr als 64 Taktzyklen, die je Bytecode immer anfallen; dies ist der Interpreter-Overhead. Er läßt sich mit den gegebenen Techniken nicht beseitigen (JIT-Compilierung würde diesen Overhead vollständig beseitigen, scheidet aber aus den bereits diskutierten Gründen aus). Bei einer Core-Frequenz von 4,332 MHz entsprechen dem etwas mehr als 67500 `nop`-Bytecodes pro Sekunde. Diese Zahl wird durch alle hier vernachlässigten Teile verringert, um wieviel, sollen nun einige Messungen zeigen.

3.2.2 Messungen Es wurden einige kleine Testprogramme entworfen, die alle denselben Rumpf enthalten. Er enthält eine Klasse mit einer `main()`-Methode, einer statischen Methode, einer Instanzmethode, einer statischen Variable und einer Instanzvariable. Die `main()`-Methode enthält eine Schleife mit 10000 Durchläufen, dort werden in den anderen Programmen die eigentlichen Testroutinen eingetragen. Um die Rechenzeit nur der 10000 Testroutinen zu erhalten wird zunächst die Rechenzeit des Rumpfs bestimmt, die dann von allen weiteren Messungen abgezogen wird.

Getestet werden lediglich einige Eckwerte, die den Betrieb der JavaVM bestimmen. Es folgt die Auflistung der einzelnen Testroutinen:

`BenchVMsmet.java`: statischer Methodenaufruf (eine Operation, drei zusätzliche Bytecodes)

`BenchVMimet.java`: virtueller Methodenaufruf (eine Operation, vier zusätzliche Bytecodes)

`BenchVMsvar.java`: Zugriff auf statische Variable (zwei Operationen, vier zusätzliche Bytecodes)

`BenchVMivar.java`: Zugriff auf Instanzvariable (zwei Operationen, sechs zusätzliche Bytecodes)

`BenchVMarithm.java`: Arithmetische Operationen (sechs Operationen, 15 zusätzliche Bytecodes)

`BenchVMcond.java`: Konditionalkonstrukt (eine Operation, vier zusätzliche Bytecodes)

`BenchVMarray.java`: Feldzugriff (eine Operation, fünf zusätzliche Bytecodes)

`BenchVMobjgarb.java`: Objekterzeugung, Garbage Collection und Finalisation (eine Operation, drei zusätzliche Bytecodes; hier werden alle Methodenaufrufe herausgerechnet)

Testprogramm	OPs	BCs	ms	+ms	OP/s	BC/s
`BenchVM`	0	0	4285	0	–	–
`BenchVMsmet`	1	3	12389	8104	1234	3702
`BenchVMimet`	1	4	13599	9314	1074	4295
`BenchVMsvar`	2	4	6581	2296	8711	17422
`BenchVMivar`	2	6	7670	3385	5908	17725
`BenchVMarithm`	6	15	10960	6675	8989	22372
`BenchVMcond`	1	4	5499	1214	8237	32949
`BenchVMarray`	1	5	5926	1641	6094	30469
`BenchVMobjgarb`	1	3	97766	65539	153	458
`BenchVMobj`	1	3	26928	4015	2491	7472

Tabelle 4 Meßergebinsse der Benchmarks auf der 8-Bit-JavaVM

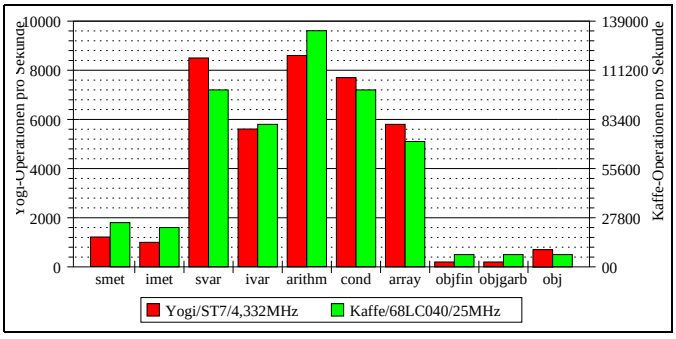

Abbildung 4 Relativer Vergleich der 8-Bit-JavaVM mit anderen JavaVMs

`BenchVMobj.java`: Dto., nur ohne Garbage Collection (eine Operation, drei zusätzliche Bytecodes)

Die Ergebnisse der Messungen faßt Tabelle 4 zusammen. Die Auswertung erfolgt getrennt nach Operationen pro Sekunde und Bytecodes pro Sekunde. Dabei ist der erste Fall eher an der Programmiersprache Java orientiert, d. h. an den im Programm gegebenen Anweisungen und der zweite Fall betrachtet die JavaVM direkt.

Zusammengefaßt ergibt sich, je nach Anwendungsfall, eine Ausführungsgeschwindigkeit von 5000 bis 8000 Operationen bzw. 15000 bis 30000 Bytecodes pro Sekunde. Reale Werte können erst ermittelt werden, wenn sinnvolle Anwendungen auf 8-Bit-JavaVM implementiert wurden.

3.2.3 Vergleich mit anderen JavaVMs Die vorgestellten Testprogramme wurden zu Vergleichszwecken auch auf einer anderen virtuellen Java Maschine ausgeführt.

Folgende Konfigurationen wurden untersucht:

1. **Yogi**, die 8-Bit-JavaVM – ST7 – 4,332 MHz
2. Kaffe V0.71 (Interpreter) – Amiga4000 – 68LC040 – 25 MHz – AmigaOS 3.0

Abbildung 4 stellt die Ergebnisse des Vergleichs dar. Dabei sind alle Werte auf die 8-Bit-JavaVM normiert worden um einen besseren Vergleich der virtuellen Maschinen zu ermöglichen. Der Geschwindigkeitsvorteil der schnelleren CPU ist also herausgerechnet worden. Da die andere JavaVM bezüglich des Garbage Collectors nicht manipulierbar ist, konnte dieser also nicht getestet werden und die Ergebnisse für `BenchVMobj` und `-objgarb` sind identisch.

Zu Beachten ist die Geschwindigkeit der Objektgeneration; sie hängt stark von der Funktion des Garbage Collectors ab. Bei den großen JavaVMs ist es denkbar, daß dort der Garbage Collector nicht so intensiv arbeiten muß, da ausreichend viel flüchtiger Speicher zur Verfügung steht, um die (nicht mehr benötigten) Objekte zu halten, sie also nicht zwingend bereinigt werden müssen.

4 Zusammenfassung und Ausblick

Unsere Implementierung hat gezeigt, daß JavaVMs mit den genannten Einschränkungen auch auf 8-Bit-Mikrocontrollern lauffähig sind. Die Performance ist für viele Anwendungen vollkommen ausreichend. Die nächsten Ziele werden es sein, die Klassenbibliotheken für Ein- und Ausgabe zu erweitern und vielfältige Peripherieelemente (LC-Displays, Tastaturen etc.) zu unterstützen. Insbesondere dann, wenn die Bytecode-Programme direkt in den nichtflüchtigen, wiederbeschreibbaren Speicher des Mikrocontrollers geladen werden können, lassen sich außerordentlich schnelle Entwicklungszyklen erreichen.

Literatur

1. Lindholm, Tim; Yellin, Frank; Java™ Die Spezifikation der virtuellen Maschine, Die offizielle Dokumentation von JavaSoft; Addison-Wesley, 1997; ISBN 3-8273-1045-8 `http://java.sun.com/docs/books/vmspec` [9]
2. JavaCard 2.0 Language Subset and Virtual Machine Specification; *Sun Microsystems*, October 1997 `http://java.sun.com/products/javacard` [9]
3. EmbeddedJava, Draft 0.1; *Sun Microsystems*, March 1998 `http://java.sun.com/products/embeddedjava` [9]
4. Java Platform 1.1 Documentation; JDK 1.1.4 `http://java.sun.com/products/jdk/1.1/docs` [9]
5. Eckel, Bruce; Thinking in Java; Revision 10a, November 1997 `http://www.EckelObjects.com/javabook.html` [9]
6. Cornell, Gary; Horstmann, Cay S.; Core Java; SunSoft Press, Prentice Hall, 1997; ISBN 0-13-596891-7
7. Campione, Mary; Walrath, Kathy; The Java™ Tutorial; *Sun Microsystems*; ISBN 0-20-163435-6 `http://java.sun.com/docs/books/tutorial` [9]
8. Wilson, Paul R.; Uniprocessor Garbage Collection Techniques; International Workshop on Memory Management, St. Malo, France, September 1992; Lecture Notes in Computer Science, number 637, pages 1–42, Springer-Verlag
9. Grafl, Reinhard; Cacao – ein 64bit – JavaVM – Just-In-Time – Compiler; Diplomarbeit, Institut für Computersprachen der Technischen Universität Wien `http://www.complang.tuwien.ac.at/java/cacao` [9]
10. `http://www.javaworld.com` [9]
11. ST7285C Data Sheet Rev. 1.0; SGS-Thomson, November 1997 `http://www.st.com/books/pdf/docs/5298.pdf` [9]
12. ST7 Family Programming Manual; SGS-Thomson, 1995 `http://www.st.com/books/pdf/docs/4020.pdf` [9]
13. Hicross Entwicklungsumgebung für den ST7; **C**-Compiler, Assembler, Linker, Maker, Simulator; Programmdokumentation, Hiware AG

[9] Im Gegensatz zum Papier ist das Internet nicht geduldig und einem ständigen Wandel unterworfen, es kann also vorkommen, daß die angegebenen Links schon nach kurzer Zeit nicht mehr gültig sind.