Eine JavaVM für Anwendungen in der Home-Automation

Helge Böhme, Gerrit Telkamp, Abteilung E.I.S., TU Braunschweig

Kurzfassung

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 (Steueraufgaben in der Home-Automation) weitgehend unerheblich. Neben den möglichen Anwendungen grenzt dieses Papier unsere Implementierung gegenüber den „marktüblichen“ Lösungen ab und geht nach einer groben Spezifikation auf einige Implementierungsdetails (*Heap*, *Garbage Collector*, *Multithreading*, *Echtzeit*) ein. Im Anschluß wird die Leistungsfähigkeit abgeschätzt.

1 Einleitung

1.1 Der vernetzte Haushalt

Seit über 2 Jahrzehnten dauert der Siegeszug der Mikroelektronik im privaten Haushalt an. Waren anfänglich nur wenige und vor allem teure Haushaltsgeräte mit integrierten Schaltkreisen ausgestattet, so wird heute fast alles mit Mikrocontrollern gesteuert und geregelt: Von der Stereo-Anlage über die Waschmaschine bis hin zum Toaster mit eingebautem Bräunungssensor, überall sind Mikrocontroller zu finden. Die Gründe für ihre Verwendung sind vor allem die Reduzierung der Produktionskosten, die Erhöhung der Zuverlässigkeit und die Schaffung neuer Funktionen. Kontinuierlich führt dieser Prozeß zu einer Vielzahl kleiner Systeme, die über den gesamten Haushalt verteilt sind. Das ermöglicht neue Anwendungen wie die Home-Automation: Vergleichbar mit der Computervernetzung kann die Effizienz und der Funktionsumfang von Haushaltsgeräten deutlich verbessert werden, wenn sie imstande sind, zu kommunizieren und untereinander Informationen auszutauschen. Dazu muß jedes Gerät mit einer Kommunikationseinheit ausgestattet werden, die als Schnittstelle zum gemeinsamen Hausgeräte-Netzwerk dient.

Was anfänglich nach übertriebenem Technik-Fanatismus klingen mag, hat durchaus sehr sinnvolle Anwendungen, wovon zwei hier kurz erwähnt werden sollen. Zunächst zum Energiemanagement: Bei Geräten mit höherer, nicht-kontinuierlicher Stromaufnahme kann der gesamte Spitzenverbrauchswert drastisch reduziert werden, wenn die Stromverbrauchsintervalle zeitlich verschachtelt werden. Praktisch bedeutet das, daß der Kühlschrank und der Gefrierschrank sich nicht mehr gleichzeitig anschalten, sondern stets nacheinander. Insbesondere für die Versorgung aus regenerativen Energiequellen (z. B. Solaranlage) ist das ein Vorteil, denn zu Zeiten des Spitzenverbrauchs muß die Energie sonst meistens von außen dazugekauft werden. Eine weitere Anwendung ist die Steuerung der Hausgeräte von einer zentralen Station aus, die z. B. mit einer Spracherkennungssoftware ausgestattet ist. Damit könnten Menschen, die an das Bett oder an den Rollstuhl gebunden sind, in komfortabler Weise die Heizung einstellen oder Lampen schalten.

1.2 Kommunikationseinheiten

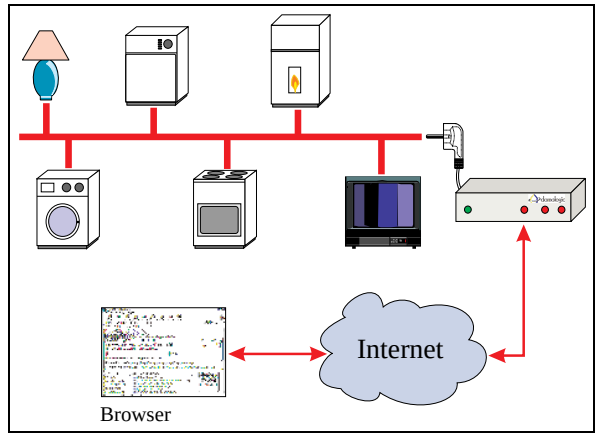

Abbildung 1: Ein kleines Home-Automation Netzwerk

Abbildung 1 zeigt ein mögliches Szenario für ein Home-Automation-Netzwerk. Verschiedene Geräte (Lampe, Waschmaschine, Geschirrspüler, Herd, Heizkessel und Fernsehgerät) sind vernetzt, ein mögliches Übertragungsmedium dafür könnte das vorhandene Stromversorgungsnetz sein (Power-Line-Kommunikation), was die Verlegung eines zusätzlichen Netzwerks überflüssig macht. Zudem ist es denkbar, an dieses Netz ein TCP/IP-Gateway anzukoppeln, das eine Verbindung zum Internet herstellt. So können unterschiedliche Steuerungen und Abfragen von außen vorgenommen werden, z. B. die Fehlerdiagnose bei der Waschmaschine durch einen Installateur; oder die Geräte könnten den momentanen Strompreis erfragen um das

Energiemanagement-System entsprechend zu parametrisieren.

Es wurden bereits die sog. Kommunikationseinheiten kurz angesprochen, die in jedem Gerät mit Netzanschluß vorhanden sein müssen und als Schnittstellen zum Hausgeräte-Netzwerk dienen. Die Aufgabe dieser Einheiten ist es, alle Informationen vom Hausgerät, die nach außen übertragen werden sollen, in ein netzwerkkompatibles Format umzusetzen (und umgekehrt). Auf der einen Seite der Kommunikationseinheiten ist eine Modulator/Demodulator- Einheit vorgesehen, die die Informationen physikalisch auf das Datenübertragungsmedium anpaßt. Auf der anderen Seite gibt es eine Schnittstelle zum jeweiligen Hausgerät. Dazwischen arbeitet ein eigener kleiner Mikrocontroller, der vor allem die Protokollfunktionen des Netzwerks bearbeitet. Von Anwendung zu Anwendung kann die Schnittstelle zum Gerät sehr unterschiedlich sein und muß daher entsprechend modifiziert werden.

Wenn im Hausgerät ein eigener Mikrocontroller für die Steuerung vorgesehen ist, muß die Kommunikationseinheit mit diesem Daten austauschen (zumeist geschieht das über ein serielles Datenprotokoll). Ist kein eigener Mikrocontroller vorhanden (z. B. bei einer Lampe), muß die Kommunikationseinheit die angeschlossene Hardware direkt bedienen. Bei einer Lampe müßte dann bei einem Einschaltkommando vom Netzwerk ein Relais geschaltet werden. Die hardwaremäßige Anpassung der Kommunikationseinheiten ist relativ einfach, da jede mit einer gewissen Anzahl an Universal-Pins ausgestattet ist, die entweder an ein Relais oder an andere Mikrocontroller angeschlossen werden können. Die Anpassung der Software für den Mikrocontroller ist jedoch erheblich aufwendiger, denn diese muß sich zum einen die Rechenzeit mit den Netzwerk-Protokollfunktionen (dem sog. Protokollstapel) teilen, und zum anderen die anwendungsspezifischen Funktionen des Hausgerätes bedienen.

1.3 Java als Programmiersprache auch für Kommunikationseinheiten

Für die Programmierung der Kommunikationseinheiten ist die Programmiersprache JAVA aus unterschiedlichen Gründen sehr gut geeignet. Zum einen fördert JAVA aufgrund seines strikten objektorientierten Ansatzes die Programmier-Effizienz, die standardisierte Programmiersprache verkürzt die Einarbeitungszeit und die JAVA-Compiler erzeugen einen maschinenunabhängigen sgn. Bytecode, der von einer virtuellen JAVA Maschine (der JAVAVM) ausgeführt wird. Unter Einschränkungen kann der Bytecode auch von einfachsten Mikrocontrollern in den Kommunikationseinheiten interpretiert werden. Dank der Maschinenunabhängigkeit ist man dabei nicht auf eine Ziel-Architektur festgelegt sondern kann relativ problemlos z.B. auf eine andere Mikrocontroller-Familie umsteigen. Dabei ermöglicht die multithreaded Interpretertechnik eine komfortable Schnittstelle zwischen der Anwendung und dem Protokollstapel, denn der Interpreter kann jederzeit die Abarbeitung der Bytecodes unterbrechen und später an der gleichen Stelle fortsetzen, ohne daß das interpretierte Programm dadurch beeinträchtigt wird. Außerdem kann der Interpreter unerlaubte Aktionen der Anwendung vor ihrer Ausführung abfangen, so daß auch Anwendungen von außen in die Kommunikationseinheit nachgeladen werden können ohne dadurch die Sicherheit eines Gerätes zu gefährden.

Die meisten verfügbaren JAVAVM Implementierungen sind für 32-Bit-Architekturen entwickelt worden, die über mehrere Megabytes frei adressierbaren Speicher verfügen müssen. Rechner dieser Größenordnung sind jedoch noch immer viel zu teuer, als daß sie in jede Kommunikationseinheit (und damit in jedes Gerät!) eingebaut werden könnten; man denke nur an ganz einfache Geräte wie Lichtschalter oder Lampen. Wenn man außerdem berücksichtigt, daß ein Netzwerk um so attraktiver wird, je mehr kommunikationsfähige Geräte daran angeschlossen sind, würde ein Home-Automation-System auf der Basis verteilter 32-Bit-Prozessoren viel zu teuer und damit unattraktiv werden.

In dem hier vorgestellten Ansatz wurde eine JAVAVM auf einen 8-Bit-Mikrocontroller implementiert, die sich sehr gut für die Abarbeitung von Anwendungsprogrammen in Kommunikationseinheiten eignet. Dazu mußte auf einige Funktionen verzichtet werden, was die beabsichtigten Anwendungsgebiete aber nicht weiter einschränkt. Das Hauptziel bei der Entwicklung war, eine äußerst kostengünstige Realisierbarkeit der JAVAVM sicherzustellen. In der Tat ist diese Lösung sehr kostengünstig: Eine komplette JAVAVM dieser Architektur ist sogar in relativ kleinen Stückzahlen schon für unter DM 15,– realisierbar.

2 Spezifikation für die 8-Bit-JAVAVM

Eine vollständige Implementierung der von der Firma *Sun Microsystems* spezifizierten JAVAVM hat auf herkömmlichen 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 der 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 das auch nicht unbedingt sinnvoll, Funktionen für grafische Ausgaben oder für Internet-Zugriffe machen auf 8-Bit-Mikrocontrollern beispielsweise aus den genannten Gründen wenig Sinn. Daher wird auf diesen Systemen nur eine Teilimplementierung der JavaVM und der Klassenbibliotheken umgesetzt.

Die Firma *Sun Microsystems* hat mit ihrem EmbeddedJava-Konzept (neuerdings: Java 2 Micro Edition) diesen Ansatz verfolgt und *skalierbare* Klassenbibliotheken geschaffen. Auch wir haben diesen Ansatz aufgegriffen, zusätzlich aber noch die JavaVM (**Yogi**) 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 vorhandnen Adreßraum angepaßt. In der Konsequenz ergibt sich dadurch eine Inkompatibilität zu den Spezifikationen für virtuelle Java Maschinen von *Sun* [4] und somit auch für die Sprachdefinition von Java selbst (einige Schlüsselworte sind nicht erlaubt); allerdings werden für einfache Anwendungen selten Werte mit mehr als 16 Bit benötigt. Insgesamt konnte der ROM-Bedarf für **Yogi** auf etwas mehr als 20KByte beschränkt werden.

Da der flüchtige Speicher auf 8-Bit 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 Festwertspeicher auf dem Chip des Controllers interpretiert (siehe 2.1.1). Auch Optimierungen durch die sog. `_quick`-Pseudobefehle kommen deshalb nicht in Frage, da diese modifizierbaren Bytecode voraussetzen. Abgesehen von den erwähnten Einschränkungen wurde größtmögliche Kompatibilität zu den „großen“ JavaVMs gewahrt.

2.1 Systemübersicht

Zur Übersicht zeigt Abbildung 2 das Gesamtsystem. Es bestehen Unterschiede zu der von *Sun* vorgegebenen EmbeddedJava-Architektur [6]. So ist es nicht möglich, auf Applikationsebene native Methoden[1] 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 weniger sicher.

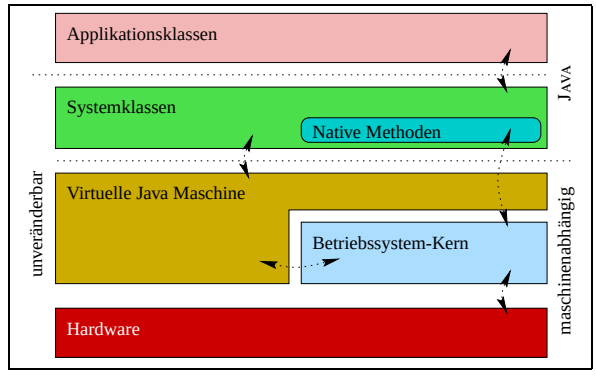

Abbildung 2: Der Aufbau der JavaVM

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 das Betriebssystem ein Teil der VM oder besser gesagt: die JavaVM *ist* das Betriebssystem. So ist im Normalfall der Thread-Manager (siehe 3.3) die oberste Instanz, der sich alle weiteren Komponenten unterzuordnen haben. Lediglich ein Interrupt-Handler kann „nebenher“ laufen und Einfluß auf den Thread-Manager nehmen (wie auch umgekehrt).

Wird die Komplexität der virtuellen Maschine **Yogi** und der verwendeten Klassen betrachtet, so ist sie zwischen der Komplexität von EmbeddedJava-Implementierungen und der JavaCard anzusiedeln.

2.1.1 Der Entwicklungsprozeß für Yogi

Von der JavaVM werden Java-Klassen (bzw. Methoden darin), die sich im ROM oder – falls vorhanden – in einem wiederbeschreibbaren Festwertspeicher (z. B. Flash-Memory) befinden, ausgeführt. Organisiert sind die Klassen dabei jeweils in dem bei Java üblichen JAR-Format[2], welches ungepackt erzeugt werden muß. Weitere Vorverarbeitungen der Klassen durch Sekundärsoftware müssen nicht stattfinden.

Der Entwicklungsprozeß unterscheidet sich bei den Systemklassen, welche sich zusammen mit der JavaVM und den nativen Methoden fest im ROM befinden, von den Applikationsklassen, welche bei einem Download-Prozeß in den wiederbeschreibbaren Festwertspeicher geschrieben werden (letztere können aber auch genauso, wie die Systemklassen ins ROM geschrieben werden, wenn keine Änderungen an der Applikation mehr nötig sind). Für den Download von Applikationen in

[1] Das sind Methoden, die nicht als Java-Bytecode, sondern im plattformabhängigen Maschinencode vorliegen

[2] Java-Archiv, kompatibel zum verbreiteten und platformübergreifenden ZIP-Format

den wiederbeschreibbaren Speicher steht im ROM eine Bootstrap-Klasse zur Verfügung, die dies erledigt. Diese Klasse wird manuell aufgerufen (z. B. kann eine an einem Port-Pin angeschlossene Taste überwacht werden) oder automatisch, wenn keine Applikation vorhanden ist. Der Download erfolgt über die serielle Schnittstelle mittels eines Terminal- oder eines JAVA-Programms auf einem Host PC.

Als Entwicklungssystem steht jede Umgebung zur Verfügung, die gültigen JAVA Bytecode (in Form von `.class`-Files) erzeugt (z. B. `javac` von *Sun*). Für die differenzierte Auswahl der für eine Anwendung benötigten Klassen wird es in Zukunft ein JAVA-Programm für den Entwicklungsrechner geben, es trifft durch Analyse der Anwendungsklassen eine automatische Klassenauswahl und vermindert auch die Größe der Klassen, indem nicht benutzte Teile entfernt werden. Auch findet an dieser Stelle die Bytecodeverifikation statt, die aus Platzgründen aus der JAVAVM auf dem Mikrocontroller entfernt wurde. Ferner ist es auf der Seite der Systemklassen sinnvoll, eine Liste mit den bnötigten nativen Methoden zu erzeugen, da diese zusammen mit der JAVAVM auf den Contoller gelangen; nur die wirklich benutzten sollen Platz im ROM belegen. Ein weiteres Problem ist, daß die Programme nicht auf dem Controller selbst entwickelt werden, sondern auf diesem Entwicklungsrechner und so die Testfähigkeit der Programme in dieser Umgebung eine große Rolle spielt. Daher werden zukünftig JAVA-Klassen existieren, die es ermöglichen, die Zielhardware auf dem Entwicklungsrechner auf Programmebene zu simulieren, für den Entwickler wird die Hardware grafisch dargestellt.

2.2 Neue Eigenschaften

Mikrocontroller für eingebettete Systeme 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. Andererseits fehlen auf derart kleinen Systemen meist Elemente wie ein Dateisystem oder eine Konsole. Diese Umgebungsänderung hat auch Auswirkungen auf die JAVAVM und die gewöhnlichen 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.

2.2.1 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 kann der textuelle Benutzerdialog mit den Methoden `println` und `readLine` über die Standardkanäle `System.out` und `System.in`[3] 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 [8].

2.2.2 Yogi und Home-Automation

Neben der Hardware, die lokal mit einem Controller verbunden ist, spielt bei der Home-Automation die Vernetzung der Geräte untereinander eine Rolle. Das Netzwerk soll abwärtskompatibel zu den vorhandenen Lösungen für die Home-Automation sein (European Home-Systems EHS, European Installation Bus EIB, etc.[4]). Die mögliche Sicht zweier Mikrocontroller mit **Yogi** auf die Harware-, Software- und Netzwerkseite zeigt Abbildung 3, dabei ist der eine Controller mit einer Lichtschalterapplikation und der andere mit einer Lampenapplikation geladen. Die Software ist dabei streng

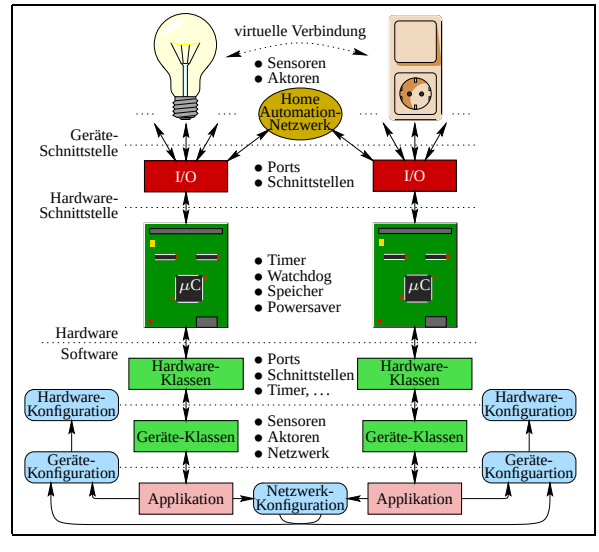

Abbildung 3: Die Abbildung der Hardware und des Netzwerks in JAVA-Klassen

hierarchisch organisiert, so existieren aus dem Blickwinkel der Applikation nur Geräte-Klassen und das Entscheidende dabei ist, daß die Geräte sowohl lokal mit dem Controller verbunden sein können als auch sich

[3] Zum Zeitpunkt der Entwicklung dieser JAVAVM diente das JAVA Development Kit von *Sun* in der Version 1.1.4 als Grundlage.

[4] In der Tat sind diese Standards zur Zeit in Bewegung, es wird ein *gemeinsames* System für alle Hersteller angestrebt, welches noch unter dem Arbeitstitel „Convergence“ existiert.

auf einem anderen Controller im Netzwerk befinden können. Das Netzwerk verhält sich dabei für die Applikation völlig transparent. *Wo* Aktionen durch bestimmte Methodenaufrufe ausgelöst werden, hängt nur von den Konfigurationen der Geräte und des Netzwerks ab, welche sich auch im laufenden Betrieb ändern können. Um im Beispiel zu bleiben, fragt die Lichtschalterapplikation den Schalter mit einer Methode ab und setzt dann den Status der Lampe entsprechend mit einer Methode, wo der Schalter (lokal mit dem Gerät verbunden) und die Lampe (über das Netzwerk angesteuert) sind, ist dabei unerheblich.

Zusätzlich zu der direkten Verknüpfung der Geräte untereinander besteht die Möglichkeit, die Geräte fernzusteuern. Dazu können die Geräte auch als Server fungieren und eine grafische Benutzerschnittstelle zur Verfügung stellen, wobei es sich im Prinzip um die von Web-Anwendungen bekannten JAVA-Applets, die eingekapselt im Browser laufen, handelt. Ein Applet kann von irgendeiner Instanz im Netzwerk angefordert werden, die JAVA- und Grafikfähig ist (das kann neben einem Browser hinter einem Home-Automation–TCP/IP-Gateway auch eine eingebettete Kontrolleinheit im Home-Automation Netzwerk selbst sein), und dann dort lokal ausgeführt werden. Es sendet dann die vom Benutzer vorgenommenen Einstellungen zu dem Gerät, von dem das Applet stammt.

Das Home-Automation Netzwerk bietet im Zusammenhang mit JAVA noch einige weitere denkbare Möglichkeiten. So sind neben dem Aufruf von Klassen über das Netzwerk und dem zur Verfügung stellen von Applets (wie oben geschildert) auch das Übermitteln von Ereignissen (Interrupts oder Exceptions) und eine verteilte Garbage Collection möglich (zur nicht verteilten Garbage Collection siehe 3.2.1). Fällt z. B. eine Lampe aus oder wird aus dem Verbund entfernt, so wird die dafür zuständige Klasse auf dem Controller entfernt und dann die über das Netzwerk damit verbundene Lichtschalterklasse auf dem anderen Controller ebenfalls. Da es bei JAVA die Möglichkeit gibt, bei der Speicherbereinigung, bestimmte Methoden (sgn. Finalizer) zu starten, kann dies auch protokolliert werden und eine Nachricht erzeugt werden, den Fehler zu beheben.

3 Eine Implementierung der 8-Bit-JAVAVM

3.1 Der Zielprozessor

Als Zielprozessor für die erste Implementierung von **Yogi** ist der ST7 von ST Microelectronics 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. Die ST7-Familie wird von ST 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, I^2C Interface, 4 serielle Schnittstellen und Watchdog. Er arbeitet mit einer maximalen Kern-Taktfrequenz von 4,332 MHz. Bei einem geschätzten durchschnittlichen Zyklenbedarf pro Instruktion von vier bis fünf ergeben sich ungefähr 0,8 bis 1,1 MIPS. Die Implementierungssprache für die JAVAVM ist ST7-Assembler; die Entwicklung ist eine sgn. Clean-Room-Implementierung gemäß den Spezifikationen in [4], d. h. sie baut auf keinem Code anderer Implementierungen auf oder benutzt diesen gar.

Einige Implementierungsdetails

Eine vollständige Implementierung einer JAVAVM umfaßt eine ganze Reihe von Komponenten. Zu nennen sind z. B. der Bytecode Interpreter, der Klassenlader (bzw. -initialisierer), welcher auch die Konstantenpoolauflösung erledigt, die Objektverwaltung, die auch für den Aufruf virtueller Methoden zuständig ist, die Verwaltung der nativen Methoden, die automatische Speicherbereinigung und der Thread-Scheduler. Aus Platzgründen werden hier nur einige wenige Komponenten kurz beschrieben.

3.2 Der Garbage Collected Heap

Alle zur Laufzeit erzeugten Datenstrukturen werden in einem zentralen Heap organisiert, welcher beinahe den gesamten RAM-Bereich einnimmt. 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.[5] 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 3.2.1) steht, wurde die Organisation des Heaps diesen Gegebenheiten angepaßt und für Objektreferenzen das Prinzip der sgn. *Handles*[6] 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

[5] Nicht auf dem Heap befinden sich die lokalen Daten der einzelnen Assemblerroutinen und der Stack des ST7 (nicht zu verwechseln mit den JAVA-Stacks, die sind in den Rahmen).

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

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[7], 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 (bei **Yogi** ist dies ein Bytecode) hinaus benutzt werden (Zeiger können sich ändern).

3.2.1 Garbage Collection

Ein wesentlicher Aspekt der Sprache Java ist die automatische Speicherbereinigung (Garbage Collection), sie nimmt dem Programmierer die Freigabe jeglicher Elemente auf dem Heap ab. Implementiert wurde ein *konservativer*, *inkrementeller*, *kompaktierender*, *dreifarb*, *mark-and-sweep* Garbage Collector [9]. 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.

Alle anderen Komponenten der JavaVM wurden bereits auf eine Zusammenarbeit mit einem Garbage Collector ausgelegt (als wichtigstes sind dabei die Handles zu nennen). So besitzt jeder Datenblock auf dem Heap ein Informationsfeld mit der Blockgröße und dem Blocktyp, welches vom Garbage Collector ausgewertet wird. Der Garbage Collector wird einmalig auf der Hauptprogrammebene der JavaVM als ein eigenständiger Thread (siehe 3.3) gestartet. Dessen Priorität wird auf den kleinstmöglichen Wert gesetzt, um andere Threads (das Applikationsprogramm) nicht zu stark zu beeinflussen, der Garbage Collector arbeitet also unauffällig im Hintergrund. Er ist in der Lage, seine Arbeit anzuhalten (der Thread wird verlassen) und später an derselben Stelle fortzufahren, er arbeitet dabei *inkrementell* in kleinen voneinander unabhängigen Einheiten. Zu beachten ist, daß der Heap in den Pausen von anderen Threads verändert werden kann; der Garbage Collector reagiert darauf robust.

Dreifarb Markierung

Der Garbage Collector verwendet für das Mark-and-Sweep-Verfahren den sgn. Dreifarb-Markierungs-Algorithmus [9]. Dabei werden die Datenblöcke entsprechend ihres Typs analysiert und darin enthaltene Referenzen auf weitere Blöcke gefunden. Abbildung 4 zeigt die einzelnen Schritte des Garbage Collectors in einigen Momentaufnahmen.

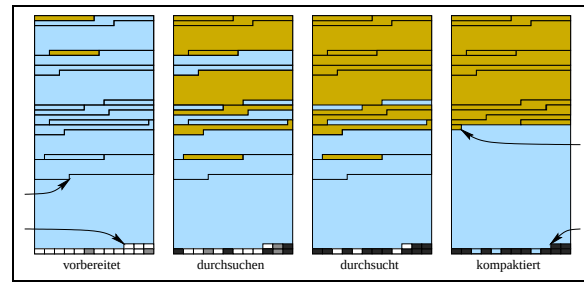

Abbildung 4: Die Arbeitsweise des Garbage Collectors

3.3 Threads

Da Java von Haus aus Nebenläufigkeiten (Multithreading) zur Verfügung stellt, ist es sehr einfach, den reaktiven vom transformativen Teil einer Anwendung zu trennen. Auch Threads haben eine Repräsentation auf dem Heap. Um einen Thread einzurichten, wird ein Zeiger auf den Bytecode-Bereich einer Methode und die gewünschte Priorität des Threads übergeben. 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.

3.3.1 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, dabei dient die Handletabelle als Ausgangspunkt. Bei jedem Thread werden die Flags (running, waiting, ...) geprüft. Bei jedem laufenden und nicht wartenden Thread wird ein Zähler um die Priorität des Threads erhöht. 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. 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 mit maximalem Zählerstand abgebrochen. Damit mehrere Threads gleicher Priorität auch garantiert

[7] Nach [8] ist diese Vorgehensweise bei JavaVMs üblich.

gleichhäufig aufgerufen werden, erfolgt diese Suche nicht von vorne im Heap, sondern beginnend vom letztausgeführten Thread zyklisch. Das ist der Übergang vom prioritätsgesteuerten Scheduler zu einem Round-Robin-Scheduler. Der Zähler des ausgewählten Threads wird auf Null zurückgesetzt und der dort eingetragene Rahmen fortgesetzt. Der Bytecode Interpreter wird für 256 Bytecodes gestartet, kehrt dieser zurück, so wird ein Fehlercode im Rückgabewert ausgewertet. Liegt kein Fehler vor, so wird 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).

3.3.2 Zeitvorgaben

Die durch die Programmiersprache JAVA vorgegebene Definition von Nebenläufgkeiten (Threads) gestattet es lediglich, Prioritäten für die Ausführung zu setzten. Dies ist für Steuerungsaufgaben nicht immer ausreichend, es besteht das Problem, daß keine Aussage darüber gemacht werden kann, wieviel *Real*zeit ein Programmteil tatsächlich benötigt (die *Rechen*zeit ist in der Regel gleich). Es kann immer vorkommen, daß ein Thread unterbrochen wird und viele andere Threads mit weniger kritischem Inhalt abgearbeitet werden. Es ist daher sinnvoll, Threads mit einer *Echtzeiterweiterung* auszustatten: Der Anwendungsprogrammierer soll in die Lage versetzt werden, einen festen Zeitrahmen für ein bestimmtes Codesegment zu setzten, es also einzuschließen; auch eine Verkettung mehrerer solcher Segmente ist möglich. Als Folge sind diese Segmente nicht von anderen Threads unterbrechbar, falls diese nicht über eine dringlichere Zeitvorgabe verfügen.

Aufgrund der starken Dynamik bei der Ausführung von JAVA-Programmen (Heap-Verwaltung, Klasseninitialisierung, etc.) scheiden exakte statische Scheduling-Verfahren aus. Dynamische Verfahren sind hingengen problemlos implementierbar, nur kann zur Entwicklungszeit keine Aussage darüber gemacht werden, ob eine gesetzte Zeitschranke *tatsächlich* eingehalten werden kann, dies ist erst zur Laufzeit einer Anwendung überprüfbar.

Echtzeit-Scheduling

Zunächst ist in das System der JAVAVM die Zeit zu integrieren. Dafür wurde einer der Timer des ST7 herangezogen und mit ihm ein freilaufender 16-Bit-Zähler im Millisekunden-Takt erhöht. Dieser Zähler ordnet nun jedem Zeitpunkt im Erfassungsbereich eine eindeutige Zahl zu.

Der vorhandene prioritätsbasierte Scheduler wurde nun um einen sgn. Earliest Deadline First (EDF) Scheduler erweitert. In der Thread-Struktur existiert neben der Priorität ein Eintrag für einen Endzeitpunkt, welcher verglichen mit dem aktuellen Zählerstand als Referenz für den Scheduler dient. Echtzeit- bzw. EDF-Threads haben immer Vorrang gegenüber prioritätsbasierten Threads. Abbildung 5 zeigt einen Beispielhaften Echtzeit-Schedule.

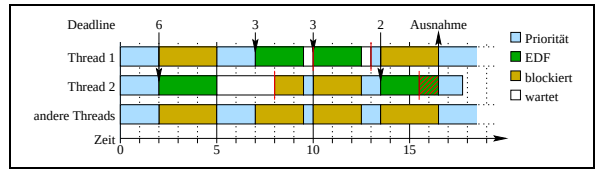

Abbildung 5: Ein Beispiel für EDF-Threads

Auf der Ebene der Programmiersprache JAVA können nun mittels der Klasse `Timer` Zeitschranken für den aktuellen Thread gesetzt und abgefragt werden (welche den Endzeitpunkt in der Struktur bestimmen); der Scheduler stellt sich dann automatisch um. Ein Beispielprogramm für eine einfache Anwendung zeigt Tabelle 1, dort werden acht einzelne Threads gestartet, deren Aufgabe ist, eine Leuchtdiode blinken zu lassen. Da für jeden Thread dabei die Zeitschranken geringfügig variieren, ergibt sich ein Lauflichteffekt; zusätzlich wird im Hauptprogramm die Blinkfrequenz kontinuierlich erhöht.

```
import controller.*;                  // PortPins, Timer, etc.
import java.lang.*;                             // classic Java

public class TimerTest extends Thread {
  private byte pin;
  private static short timeout=300;
  private static short opencnt=0;
  static void main(String args[]){          // initialisation
    try{
      PortPins.portmode((byte)0,(byte)0xc0);
      Timer.settimeconstraint((short)1000);
      for(byte i=0;i<8;i++){
        TimerTest thread=new TimerTest(i);
        thread.start();           // run 8 individual threads
      }
      while((timeout>1)&&(opencnt>0)){
        try{
          Timer.settimeconstraint((short)250);
        } catch(TimeViolationException e) {}
        timeout--;                            // increase speed
      }
      Timer.endtimeconstraint();
    } catch(TimeViolationException e) {}
  }

  TimerTest(byte pin){                          // constructor
    this.pin=pin;
    opencnt++;
  }

  public void run(){                              // a thread
    try{
      for(short i=0;i<1000;i++){
        Timer.settimeconstraint((short)(timeout+pin));
        PortPins.setpin(pin,false);
        Timer.settimeconstraint((short)(timeout+pin));
        PortPins.setpin(pin,true);
      }
      Timer.endtimeconstraint();
    } catch(TimeViolationException e) {
      PortPins.setpin(pin,false); // make exception visible
    } finally {
      Timer.wait((short)1000);              // wait 1 second
      opencnt--;
    }
  }
}
```

Tabelle 1: `TimerTest.java`

4 Leistungsfähigkeit

Abschließend soll die Geschwindigkeit der JavaVM bestimmt werden. Der ST7 wurde bei allen Tests mit einer Kern-Taktfrequenz von 4,332 MHz betrieben.

4.1 Interpreter-Overhead

Beim Interpreter-Overhead handelt es sich um die absolute obere Schranke der Ausführungsgeschwindigkeit der JavaVM. Dabei werden das Thread-Scheduling, die Speicherbereinigung, Initialisierungen aller Art sowie die Ausführungsdauer der Bytecodes selbst (es wird der Bytecode `nop` benutzt) außer acht gelassen. Bei der Analyse der Hauptschleife des Bytecode Interpreters auf Assemblerquellcodeebene ergeben sich im Mittel etwas mehr als 64 Taktzyklen, die je Bytecode immer anfallen, das entspricht etwas mehr als 67500 `nop`-Bytecodes pro Sekunde. Diese Zahl kann für alle Anwendungen auf dieser Architektur nicht überschritten werden.

4.2 Messungen

Es wurden einige kleine Testprogramme entworfen, die alle den selben 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, in der in den anderen Programmen die eigentlichen Testroutinen eingetragen werden. Um die Rechenzeit nur der 10000 Testroutinen zu erhalten, wird auch die Rechenzeit des Rumpfs bestimmt, die dann von allen weiteren Messungen abgezogen wird. In den Testroutinen werden lediglich einige Eckwerte, die den Betrieb der JavaVM bestimmen, getestet: statischer Methodenaufruf, virtueller Methodenaufruf, Zugriff auf statische Variable, Zugriff auf Instanzvariable, Arithmetische Operationen, Konditionalkonstrukt, Feldzugriff sowie Objekterzeugung, letzteres geht einher mit der Garbage Collection und der Finalisation der Objekte, welches getrennt voneinander untersucht werden konnte.

Zusammengefaßt ergibt sich – je nach Anwendungsfall – eine Ausführungsgeschwindigkeit von 15000 bis 30000 Bytecodes pro Sekunde. Reale Werte können erst ermittelt werden, wenn sinnvolle Anwendungen auf **Yogi** implementiert wurden.

5 Zusammenfassung und Ausblick

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

Literatur

[1] Helge Böhme; Konzeption und Implementierung einer virtuellen Java Maschine auf einem 8-Bit-Mikrocontroller für Steuerungsaufgaben in der Home-Automation; Diplomarbeit 8-98, Abteilung E.I.S. an der Technischen Universität Braunschweig

[2] Peter Ahlbrecht; Implementierung der Convergence-Protokollspezifikation in Java; Studienarbeit 02-99, Abteilung E.I.S. an der Technischen Universität Braunschweig

[3] Sven Tiffe; ChOP - Ein universelles Prüf- und Optimierungstool fuer Java Klassendateien; Studienarbeit 03-99, Abteilung E.I.S. an der Technischen Universität Braunschweig

[4] 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 [8]

[5] JavaCard™ 2.1 Language Subset and Virtual Machine Specification; *Sun Microsystems*, March 1999
http://java.sun.com/products/javacard [8]

[6] EmbeddedJava™ 1.0 API; *Sun Microsystems*, June 1999
http://java.sun.com/products/embeddedjava [8]

[7] Java™ Platform 1.1 Documentation; JDK 1.1.8; *Sun Microsystems*, May 1999
http://java.sun.com/products/jdk/1.1/docs [8]

[8] Eckel, Bruce; Thinking in Java; Final Version, January 1998
http://www.EckelObjects.com/javabook.html [8]

[9] 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

[8] 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.