Hardware–Zugriffe aus Java heraus

Guido Glöde Volker Hartwig
Christian Hochberger

Lehrstuhl Rechnerarchitektur, Fachbereich Informatik, Universität Rostock
Albert–Einstein–Straße 21, 18059 Rostock, Deutschland,
merlin|volker|hochberg@informatik.uni-rostock.de

Kurzfassung. Der Zugriff auf Hardware–Komponenten aus Java heraus stellt ein Problem dar, das insbesondere in eingebetteten Systemen zum Tragen kommt. Der klassische Weg über native Sprachen steht in Hardware–Implementierungen der JVM nicht zur Verfügung. In diesem Beitrag wird daher ein alternatives Konzept vorgestellt, mit dem es möglich wird, von Java aus auf Hardware–Komponenten zuzugreifen. Exemplarisch wird die Implementierung dieses Konzeptes anhand der KaffeVM unter Linux diskutiert. Viele der hierbei gewonnenen Erkenntnisse lassen sich jedoch auch auf Hardware–Implementierungen übertragen.

1 Einleitung

1.1 Motivation

Eine der ursprünglichen Intentionen beim Entwurf der Programmiersprache Java war der Einsatz in eingebetteten Systemen. Obwohl das hauptsächliche Einsatzgebiet von Java nicht in diesem Bereich liegt, gibt es doch einige Bestrebungen, eingebettete Systeme in Java zu programmieren[3],[6]. Diese Systeme sind sehr häufig durch ein spezielles, dediziertes I/O–Verhalten gekennzeichnet, zu dessen Realisierung in der Regel spezielle Hardware–Komponenten benutzt werden. Java, bzw. der Java–Bytecode enthält allerdings keine Elemente, um auf diese Komponenten direkt zuzugreifen. Weder stehen spezielle I/O–Instruktionen zur Verfügung, noch können aus dem Bytecode heraus absolute Adressen verwendet werden.

Java sieht für diesen Fall das Java Native Interface (JNI) vor, durch das es möglich wird, Code auszuführen, der in anderen Programmiersprachen erstellt wurde (also z.B. C). Ein entsprechender Hardware–Zugriff ist in C nun aber kein Problem mehr. Allerdings hat diese Vorgehensweise mehrere gravierende Nachteile, so dass alternative Konzepte von großem Interesse im Bereich eingebetteter Systeme sind:

1. JNI ist in JVM–Implementierungen für eingebettete Systeme nicht immer vorhanden (z.B. KVM von Sun). Eine zusätzliche Implementierung von JNI würde daher erhebliche Anforderungen an die Infrastruktur eines eingebetteten Systems stellen (Speicher, dynamisch nachladbare Bibliotheken). Außerdem ist es für

den Entwickler als zusätzliche Belastung aufzufassen, die Hardware–abhängigen Teile in einer anderen Programmiersprache formulieren zu müssen.

2. Die Realisierung der JVM in Hardware [4][5][1] erfordert zwingend ein alternatives Konzept, um Hardware–Zugriffe zu realisieren, da in diesem Fall keine andere Programmiersprache zur Verfügung steht. Ein solches Konzept sollte allerdings so weit wie möglich von der verwendeten Hardware abstrahieren, um möglichst universell einsetzbar zu sein.

In diesem Kontext muss auch der vorliegende Beitrag gesehen werden. Das hier vorgestellte Konzept soll anhand einer Implementierung in einer Software–JVM verifiziert werden, um dann in Hardware–Implementierungen zum Einsatz zu kommen.

1.2 Verwandte Arbeiten

Naturgegebenermaßen stehen alle Hardware–Realisierungen der JVM vor dem gleichen Problem. Der PSC1000 von Patriot Scientific wurde zwar ursprünglich nicht für Java–Bytecode entwickelt, sondern für Forth, eignet sich jedoch trotzdem sehr gut zur Implementierung der JVM. Dieser Prozessor enthält einen speziellen I/O–Subprozessor, der sehr eng mit der eigentlichen CPU verzahnt ist. Der PSC1000 enthält zur Ansteuerung dieser und anderer externer Komponenten spezielle Bytecodes, die absolute Adressierungen ermöglich. Um asynchrone externe Ereignisse verarbeiten zu können, lassen sich Interruptserviceroutinen installieren. Ein effizientes Zusammenarbeiten zwischen Haupt–CPU und I/O–Subprozessor ist jedoch nur möglich, da der I/O–Subprozessor direkt auf interne Register zugreifen kann. Das hier vorgestellte Konzept soll aber mit allgemeinen I/O–Komponenten zusammenarbeiten, weswegen eine derart starke Verzahnung von Prozessor und I/O–Komponente nicht wünschenswert, bzw. sogar störend wäre.

Die Firma ajile bewirbt seit kurzem den Prozessor aj100, der eine kommerzielle Umsetzung des Rockwell JEM2 darstellt. Bei diesem Prozessor hat der Anwender die Möglichkeit neue Bytecodes durch einen Mikroprogrammspeicher zu implementieren. Wesentliches Einsatzgebiet dieser applikationsspezifischen Bytecodes stellt die Realisierung spezialisierter I/O–Funktionalität dar. Der Mikrocode ist aber auch hier auf die Benutzung der im Prozessor integrierten Funktionalität beschränkt, so dass externe Komponenten nicht effizient in den Bytecode eingebunden werden können. Ähnlich zum hier vorgestellten Konzept ist jedoch der Ansatz, spezialisierte Bytecodes für die I/O–Komponenten zu verwenden.

Vergleichbares gilt auch für alle Implementierungen des PicoJava II Cores. Der von LG Semiconductor angekündigte Prozessor wird ebenfalls einige Peripheriekomponenten enthalten, die dann vom Prozessor aus gut ansprechbar sind. Der Zugriff auf externe Komponenten dürfte aber auch hier mit erheblichem Mehraufwand verbunden sein.

1.3 Struktur dieses Beitrags

Im zweiten Kapitel wird zunächst das Konzept der Integration von Hardware–Zugriffen in Java erläutert, wobei auch die Werkzeuge vorgestellt werden, die dazu nötig sind.

Das dritte Kapitel stellt dann die Modifikationen der JVM dar, die im wesentlichen aus neuen Bytecodes bestehen. Die Funktionalität dieser Bytecodes wird durch einen speziellen Gerätetreiber realisiert, dessen Struktur und Funktion in Kapitel 4 beschrieben werden. Die Behandlung von Interrupts erfordert in diesem Umfeld besondere Aufmerksamkeit, weswegen diesem Thema auch ein eigenes Kapitel gewidmet wird. Ein Ausblick auf zukünftige Arbeiten und eine Betrachtung der erzielten Ergebnisse schliessen diesen Beitrag ab.

2 Konzept

Um Javaprogrammen den direkten Hardwarezugriff zu ermöglichen, erfolgt eine Unterteilung in ein mehrschichtiges Modell mit strikter Trennung der einzelnen Funktionalitäten zur Erhöhung der Flexibilität und Portablilität der Entwicklung.

Als Entwicklungsplattform kommt Linux für die Prozessoren der x86–Familie zum Einsatz. Der Hardwarezugriff wird durch ein Kernelmodul realisiert. Es erscheint nach außen als ein neues Gerät, auf das über `ioctl`–Aufrufe zugegriffen wird. Die Ausführung der Schnittstelle als Pseudodatei “Device” erlaubt eine Erhöhung der Sicherheit durch die Zugriffsregelung über die Dateirechte. Die neuen Funktionen werden durch neun neue Bytecodes implementiert. Die eigentliche Funktionalität wird dem Applikationsprogrammierer durch ein schlankes Klasseninterface zur Verfügung gestellt

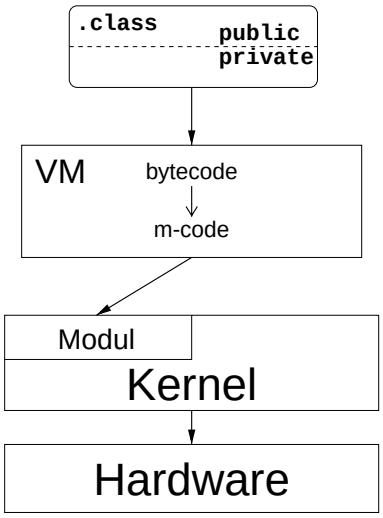


Abbildung 1. Schichtenkonzept des Hardwarezugriffs

Klassenebene. Die Kapselung der Zugriffe in einer Klasse ist von Vorteil, da sie keine Änderungen am Java–Sprachstandard zur Folge hat. Für existierende Java–Applikationen sind keine Neuübersetzungen notwendig. Dem Programmierer stellt das System ein aus einer einzelnen Klasse bestehendes API bereit, welches einfache Schreib– und Lesevorgänge von Adressen/Ports ohne zeitaufwendige Überprüfungen umsetzt.

Virtual Machine. Die VM prüft die Zugriffsmöglichkeit auf die Schnittstelle des Kernelmoduls und blockiert im Negativfall die Ausführung der auf diese Schnittstelle zugreifenden Funktionen zur Abarbeitung der zusätzlich integrierten Bytecodes. Die Übergabe der Argumente an die Funktionen der Kernelmoduls erfolgt über `ioctl`–Aufrufe. Rückgaben der Aufrufe werden auf Fehlermeldungen seitens des Kernelmoduls untersucht. Gegebenenfalls erfolgt die Auslösung von Exeptions bzw. die Ablage des Funktionswertes auf dem Stack. Die VM ist für die Benachrichtigung eines vom Anwendungsentwickler anzulegenden Threads zuständig, in dem auf Interruptmeldungen der Hardware reagiert wird.

Kernelmodul. Innerhalb des Kernelmoduls wird neben den Funktionen für Schreib–, Lese– und Bitoperationen die besonders zeitkritische Interruptserviceroutine (ISR) implementiert. Insbesondere die Interruptbehandlung erfordert eine hohe Flexibilität, damit Programmierer auch für relativ schnelle Peripheriegeräte entwickeln können. Erhalten wird diese Flexibilität durch die Auslegung der ISR als eine aus der Anwendungsschicht programmierbare Routine. Das Kernelmodul ist für die Benachrichtigung der KaffeVM bei Auslösung der ISR zuständig.

Durch den Ansatz, diese letzte Schicht als ein Modul für den Betriebssystemkern zu realisieren, erhöht sich die Flexibilität dieser Lösung im Hinblick auf Portierungen.

3 Integration des Hardware–Zugriffs in Java

3.1 Neue Bytecodes

Die typischen Operationen, um auf Hardware zuzugreifen, werden mit folgenden Bytecodes realisiert:

Nr.	Mnemonic	Bedeutung	Rückgabe	Argumente
220	`hwa_ctrl`	Reset der ISR	–	0
220		Start der Lernphase der ISR	–	1
220		Ende der Lernphase	–	2
220		Interrupt aktivieren	–	3
220		Interrupt deaktivieren	–	4
221	`hwa_open`	Öffnen des Treibers	–	–
222	`hwa_close`	Schliessen des Treibers	–	–
223	`hwa_sread`	Byte lesen	int	Offset
224	`hwa_swrite`	Byte schreiben	–	Offset, Wert
225	`hwa_bread`	Bit lesen	int	Offset, Bit#
226	`hwa_bset`	Bit setzen	–	Offset, Bit#
227	`hwa_bclear`	Bit löschen	–	Offset, Bit#
228	`hwa_isr`	gespeicherte Werte der ISR auslesen	int[]	

Die Resultate der Operationen werden auf dem Stack abgelegt.

3.2 Erforderliche Werkzeuge

Die Verwendung neuer Bytecodes in der JVM erfordert notwendigerweise umfangreiche Änderungen an den Systemen zur Bytecodegenerierung (idR. Compiler). Da keine Modifikation des Java–Sprachstandards geplant ist, sondern die Bereitstellung eines als Klasse gekapselten APIs, wird ein Werkzeug zur Herstellung einer `.class`–Datei benötigt, welches die Integration von fremden Bytecodes erlaubt. Dieses Problem kann gegenwärtig nur von Assemblern für Java gelöst werden. Nach Untersuchung der verfügbaren Assembler/Disassembler fiel die Wahl auf Ksm (Assembler) und Dis (Disassembler) aus dem unter GPL stehenden Kopi–Projekt[2] von DMS.

Mit diesen Werkzeugen kann eine Hardware–Klasse nach folgendem Schema erzeugt werden:

1. Erzeugen eines Klassengerüstes durch den Java–Compiler
2. Disassemblieren dieser Klasse
3. Einfügen der speziellen Bytecodes in die disassemblierte Klasse
4. Assemblieren dieser modifizierten Assembler–Quelle

Eine alternative Möglichkeit, `.class`–Dateien mit neuen Bytecodes zu erzeugen, wird in Abschnitt 6 beschrieben.

3.3 Modifikation der JVM

Als Grundlage für Modifikationen an einer JVM kommt die frei verfügbare Kaffe OpenVM[7] von Transvirtual in der Version 1.0.5 zum Einsatz. Diese der GPL unterliegende JVM ist auf einem stabilen Entwicklungsstand und wurde bereits auf viele Systeme portiert.

Zur Integration neuer Bytecodes müssen in der KaffeVM an mehreren Modulen Änderungen durchgeführt werden.

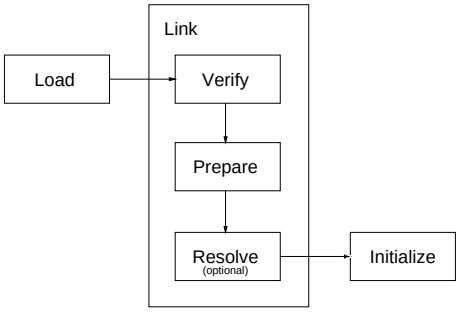


Abbildung 2. Der Linkvorgang

Der Lebenslauf einer Klasse beginnt mit dem Laden, Linken und Initialisieren (s. Abb. 2). Das Modul, das für das Laden der `.class`–Datei zuständig ist, konnte mit geringem Aufwand um die neuen Bytecodes ergänzt werden. Der *Verifier* der KaffeVM beschränkt sich auf das Zählen der Argumente. Die Änderungen umfassen die „Bekanntgabe“ neuer Bytecodes, die mit ihnen verbundenen Umgebungsänderungen (Stackgröße, VM–interne lokale Variablenmenge) und die Anpassung des Abschnitts für die Überprüfung dieser Umgebungsänderungen.

Im Kern der *ExecutionEngine* der KaffeVM befindet sich eine große Mehrfachverzweigung. Innerhalb dieser waren die größten Veränderungen notwendig. Hier finden die `ioctl`–Aufrufe des Treibers statt. Die Entwickler kapselten viele programmtechisch sehr komplexe Aufrufe in Makros, wodurch die Veränderungen der VM sehr vereinfacht wurden.

4 Generischer Hardware-Zugriff

Unter Linux gibt es zwei Möglichkeiten für Prozesse, direkt auf die Hardware zuzugreifen:

Freigabe von Portbereichen. Mit `ioperm()` und `iopl()` werden bestimmte Portbereiche freigegeben, auf die dann über `inb()` und `outb()` der Zugriff erfolgt. Diese Lösung ließe sich zwar einfach implementieren, der Prozess muss dann allerdings mit Superuser–Rechten laufen. Der größere Nachteil dieser Lösung liegt darin, dass die Behandlung von Interrupts nicht möglich ist. Diese sind jedoch zur asynchronen Ereignisverarbeitung unverzichtbar.

Gerätetreiber. Die Kommunikation der JVM mit der Hardware wird über einen Gerätetreiber abgewickelt. Dieser bildet alle Operationen auf entsprechende Funktionen ab. Der JVM–Prozess muss keine Superuser–Rechte besitzen, sondern lediglich Zugriff auf das Device–File des Treibers haben. Der Adressbereich kann eingeschränkt werden, so dass andere Geräte vor ungewünschtem Zugriff geschützt sind. Bei dieser Lösung ist die Behandlung von Interrupts möglich, was der entscheidende Vorteil gegenüber der einfachen Freigabe von Portbereichen ist.

Die Vorteile des zweiten Ansatzes überwiegen, weshalb dieser zur Realisierung kam. Dieser Abschnitt beschreibt den einfachen Zugriff auf Portadressen, mit dem prinzipiell schon alle Geräte gesteuert werden können, die nicht auf asynchronen Nachrichtenaustausch angewiesen sind. Die Behandlung von Interrupts wird auf Grund der Komplexität des Themas im Abschnitt 5 diskutiert.

4.1 Fähigkeiten des Treibers

Der Treiber soll transparente Zugriffe auf die Hardware ermöglichen, alle notwendigen Operationen für deren Steuerung anbieten und nicht an ein spezielles Gerät angepasst sein. Aus diesen Gründen war die Entwicklung eines generischen Treibers notwendig.

Der Treiber erhält beim Start den zu nutzenden I/O–Bereich und Interrupt als Parameter. Diese können, sofern sie nicht bereits von anderen Treibern reserviert wurden, frei gewählt werden.

Alle Funktionen, welche zum Zugriff auf die Hardware notwendig sind, lassen sich auf Bit– und Byte–Operationen[1] reduzieren. Daher bietet der Treiber genau diese Operationen auf den reservierten Ports an.

Theoretisch können die Bit–Operationen direkt in Java aus den Byte–Operationen zusammengesetzt werden. Dann sind jedoch für jede Bit–Operation zwei Übergänge[2] vom Nutzer– in den Kernmodus notwendig. Des weiteren liegt die Ausführungsgeschwindigkeit von C erheblich über der von Bytecode. Die Implementierung der Operationen im Treiber ist daher effizienter. Hinzu kommt, dass die gängigen Prozessoren für eingebettete Systeme direkte Opcodes zur Bit–Manipulation anbieten.

4.2 Anbindung an die JVM

Die Funktionalität des Treibers kann ein Prozess durch Zugriff auf die entsprechende Gerätedatei nutzen. Neben den eigentlichen I/O–Operationen gibt es daher auch bis

[1] Bit setzen, löschen, abfragen, Byte lesen und schreiben
[2] Byte lesen, in Java manipulieren und zurückschreiben

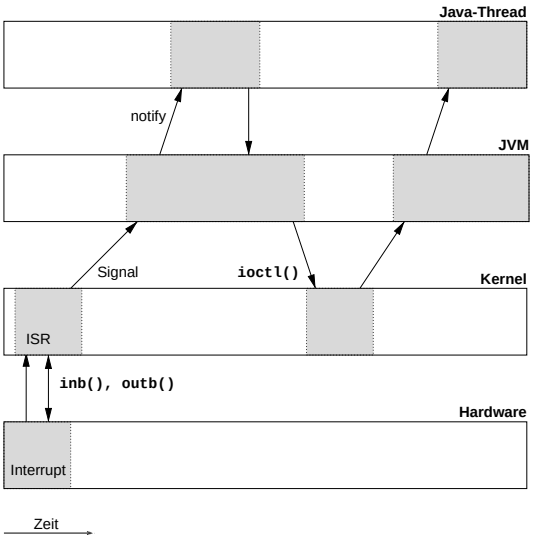


Abbildung 3. Interrupt–Behandlung

in die Java–Klasse hinauf Bytecodes zum Öffnen und Schließen des Devices. Diese finden ihre Entsprechung in den Systemaufrufen `open()` und `close()`. Auf alle anderen Funktionen wird über `ioctl()` zugegriffen. Hier wäre als Alternative die Verwendung von `read()` und `write()` möglich. Die Daten würden dann jedoch vom Kern– in den Nutzeradressraum bzw. umgekehrt kopiert, während sie im anderen Fall einfach als Parameter übergeben werden.

Bei Leseoperationen liefert der Rückgabewert das Ergebnis, im Fehlerfall ist der Wert negativ, die Fehlerbehandlung obliegt der JVM.

5 Interrupt–Behandlung

Die Interrupt–Behandlung durch einen Java–Thread und die darunter liegende JVM wird durch zwei Probleme verhindert. Zum einen kann ein Interrupt unter Linux nur im Kernmodus behandelt werden. Zum anderen wäre Java viel zu langsam, um eine vernünftige Interrupt–Behandlung gewährleisten zu können. Schnelle Interrupts – dazu zählen die meisten Hardware–Interrupts – können im allgemeinen nicht unterbrochen werden. Das System könnte also während der Abarbeitung des Interrupts keine anderen Aufgaben bearbeiten.

Die Vorgehensweise der Interrupt–Bearbeitung wird in Abbildung 3 verdeutlicht. Wenn der Interrupt–Controller einen IRQ signalisiert, ruft der Kernel die Interruptserviceroutine (ISR) des entsprechenden Treibers auf. Diese setzt den Interrupt auf gerätespezifische Weise zurück – die genaue Prozedur beschreibt Abschnitt 5.1. Alle von der Hardware gelesenen Werte werden gespeichert, da die Register während der Abarbeitung möglicherweise beschrieben bzw. durch Auslesen (UART 16550A: Ausle-

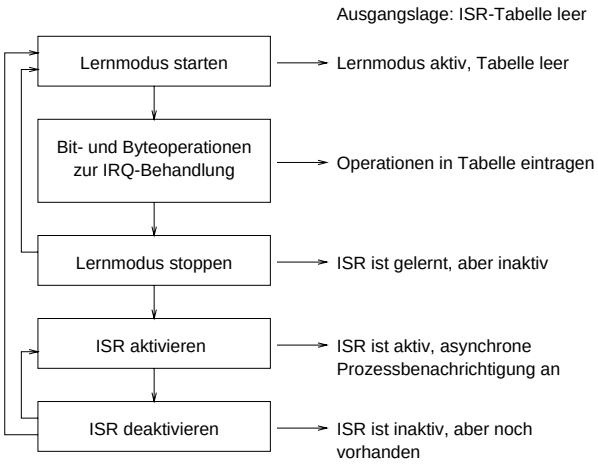


Abbildung 4. Lernen der Interruptserviceroutine

sen des Leitungsstatus–, Empfangsdaten–, Interrupterkennungs– und Modemstatusregisters) verändert werden. Anschließend wird dem Prozess, welcher die Gerätedatei des Treibers geöffnet hat, ein Signal gesendet. Sobald der Scheduler die JVM aktiviert hat, wird deren Signalhandler aufgerufen. Dieser informiert den Java–Thread mit einem `notify`, dass ein Interrupt vorlag. Jetzt kann die „Interruptserviceroutine“ der Hardware–Klasse ausgeführt werden. Die gespeicherten Statuswerte werden abgefragt — hierfür existiert ein eigener Bytecode, der seine Entsprechung wiederum in einem `ioctl()` findet.

5.1 Lernen der Interruptserviceroutine

Der Treiber besitzt weder Informationen darüber, welches Gerät sich über den reservierten Adressbereich ansprechen lässt, noch wie eine ISR für dieses Gerät aussehen muss. Es ist also notwendig, dem Treiber die für das jeweilige Gerät passende Interruptserviceroutine „beizubringen“.

Hierfür wurde in Java ein weiterer Bytecode reserviert, der wiederum auf der Systemebene mit einer `ioctl()`–Funktion realisiert ist. Dieser Bytecode ist dafür zuständig, das Lernen der ISR zu starten, zu stoppen, oder die Interrupt–Behandlung im Treiber zu de–/aktivieren.

Die zur Verfügung stehenden Operationen und deren Auswirkungen sind in Abbildung 4 zu erkennen. Im Treiber wird eine Tabelle gehalten, in dieser werden die Art der I/O–Operation, Adresse, Wert bzw. Bitposition und der Rückgabewert der Operation abgelegt. Daneben wird der Status der Interruptserviceroutine gespeichert. Zur Initialisierung ist der Interrupt deaktiviert und die Tabelle leer. Über einen `ioctl()` ist das Lernen der ISR zu starten, alle folgenden I/O–Operationen werden in der Tabelle abgespeichert und nicht ausgeführt. Nachdem das Lernen beendet ist und die Tabelle mindestens einen Eintrag enthält, ist es möglich, den Interrupt zu aktivieren.

5.2 Probleme

Ein Problem ergibt sich aus der frühzeitigen Deaktivierung des Interrupts durch den Treiber, noch bevor die JVM über den Interrupt informiert wird. Wie aus Abbildung 3 zu erkennen ist, kann vom selben Gerät ein Interrupt auftreten, bevor der Java–Thread in der Lage ist, den vorangegangenen zu bearbeiten, was aus oben genannten Gründen nicht zu verhindern ist. Um jedoch über solche „verlorenen“ Interrupts informiert zu sein, speichert der Treiber die Anzahl der seit dem letzten Abfragen der Statuswerte aufgetretenen IRQs und liefert diesen Wert zurück.

Durch Aktivieren einer ungeeigneten ISR kann das System in einen instabilen Zustand versetzt werden. Dies lässt sich prinzipbedingt mit dem hier gewählten generischen Ansatz nicht verhindern.

6 Zukünftige Arbeiten

Im gegenwärtigen Zustand ist die Verwendung der neuen Bytecodes nur über einen modifizierten Assembler möglich. Um dem Entwickler die Arbeit zu erleichtern, und ihn nicht zum Erlernen einer weiteren Sprache (Java-Assembler) zu zwingen, ist es für die Zukunft geplant, ein API ähnlich dem folgenden einzusetzen:

```
public final class HardwareAccess {
  public static void  open();
  public static void  close();
  public static void  ctlr(int mode);
  puclic static int[] readISRValues();
  public static int   byteRead( int offset );
  public static void  byteWrite( int offset, int val);
  public static int   bitRead( int offset, int pos );
  public static void  bitSet( int offset, int pos);
  public static void  bitClear( int offset, int pos);
}
```

Diese Schnittstelle bietet in Abhängigkeit von der Verfügbarkeit des JNI und des Typs der zu verwendenden JVM drei verschiedene Einsatzmöglichkeiten (Abb. 5).

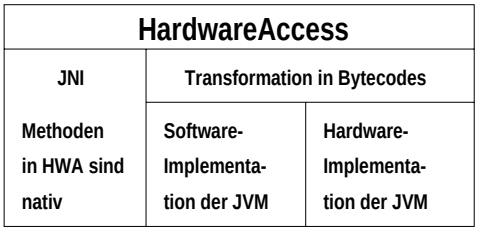


Abbildung 5. Die drei Varianten der Hardware–Ansteuerung

Enthält die JVM bereits ein Java Native Interface, so wird eine Version des APIs eingesetzt, in der alle Methoden nativ implementiert sind. Ist die Verwendung des JNI nicht möglich, erfolgt durch ein Transformationswerkzeug auf Klassendateiebene eine Umformung sämtlicher Aufrufe von Methoden obiger Klasse (in Assembler: `invokestatic HardwareAccess.*`) in die korrespondierenden Bytecodes. Die so modifizierten Klassendateien sind anschließend mit der Softwareimplementation der um die neuen Bytecodes ergänzten Java Virtual Machine, bzw. ihres in einer Hardwareausführung existierenden Pendants einsatzfähig.

Ferner besteht die Möglichkeit, diesen Transformationsschritt von einem speziellen *Classloader* ausführen zu lassen. Durch die dafür notwendigen Softwaremodule würden aber die Klassenbibliotheken der JVM vergrößert werden.

7 Zusammenfassung

In diesem Beitrag wurde ein Konzept zum Zugriff auf Hardware aus Java–Programmen heraus vorgestellt, welches nicht auf nativen Routinen beruht, sondern neue Bytecodes verwendet. Diese Vorgehensweise bietet sich insbesondere in eingebetteten Systemen und Hardware–Implementierungen der JVM an, da sie speicherschonend ist und sich nahtlos in die Bytecode–Interpretation eines Java–Prozessors integrieren lässt.

Die bisher gemachten Erfahrungen rechtfertigen das Konzept. So ist beispielsweise ein 486DX2–80 in der Lage, eine Centronics–Schnittstelle mit maximaler Geschwindigkeit zu betreiben. Eine erschöpfende quantitative Bewertung konnte bisher jedoch noch nicht durchgeführt werden.

Literatur

[1] ajile Systems. aj–100 Datasheet, 2000. http://www.ajile.com/.

[2] DMS. The Kopi Project, 2000. http://www.kopi.org/kopi/.

[3] A. Hergenhan, C. Weiler, K. Weiß und W. Rosenstiel. Internet–basierte eingebettete Systeme in der industriellen Automation. In *GI/ITG–Workshop Java und eingebettete Systeme*, S. 19–31, Karlsruhe, 1998.

[4] Patriot Scientific. PSC 1000 Microprocessor, 1998. http://www.ptsc.com/psc1000/index.html.

[5] Sun Microsystems. PicoJava II Datasheet, 1999. http://solutions.sun.com/embedded/databook/pdf/datasheets/805-4634-02.pdf.

[6] Sun Microsystems. The K Virtual Machine, 1999. http://java.sun.com/products/kvm/wp/KVMwp.pdf.

[7] Transvirtual Technologies Inc. The Kaffe Open VM, 2000. http://www.kaffe.org/.