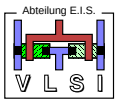
Embedded Control mit JAVA

Tagungsbeitrag *Embedded Intelligence 2001*, Helge Böhme[1], Gerrit Telkamp[2]



[1]Abteilung E.I.S., TU BRAUNSCHWEIG
Gaußstraße 11
38106 Braunschweig
ϟ h.boehme@tu-bs.de
☎ 0531/391-3108, Fax 0531/391-5840

[2]Domologic Home-Automation GmbH
Rebenring 33
38106 Braunschweig
ϟ g.telkamp@domologic.de
☎ 0531/3804-340, Fax 0531/3804-342

Kurzfassung

JControl ist eine JAVA-Realisierung, die speziell für Steuerungsaufgaben konzipiert wurde, wie sie z. B. in der Haus- und Gebäudeleittechnik (Home-Automation) vorkommen. Speziell vorgefertigte Klassenbibliotheken (APIs) unterstützen dabei die Ansteuerung von Peripheriekomponenten und Kommunikationsschnittstellen. Dadurch lässt sich der Implementierungsaufwand von Steuerungsanwendungen erheblich reduzieren, wozu auch eine eigene Entwicklungsoberfläche beiträgt. Besonderes Augenmerk lenkt dieses Papier auf die nur 20KB große virtuelle JAVA-Maschine, die über alle für Steuerungsaufgaben notwendigen Eigenschaften verfügt (*Objektorientierung*, *Garbage-Collecton*, *Multithreading*, *Echtzeit* etc.).

Motivation

Eingebettete Syteme sind zu einem festen Bestandteil unseres täglichen Lebens geworden. In einer Vielzahl einstmals rein mechanischer Geräte arbeiten heute Mikrocontroller, die die Herstellungskosten senken, die Zuverlässigkeit erhöhen und zudem den Funktionsumfang der Geräte steigern. Die Steuerung für diese Systeme hat in der Regel einen reaktiven Charakter und die Algorithmen dafür erfordern nur eine relativ geringe Rechenleistung, die bereits von *8-bit*-Mikrocontrollern erbracht werden kann. Viele Chip-Hersteller haben solche Controller im Lieferprogramm und arbeiten permanent an neuen Modellen. Es gibt ganze Produktfamilien mit zahlreichen auf dem Chip integrierten Peripherieeinheiten, z. B. Speicher, Ein-/Ausgabeports, Timer, A/D–D/A-Wandler, PWM, diverse serielle Schnittstellen und Bus-Systeme. Zumeist existiert eine Vielzahl von Varianten, die je nach Ausstattung für eine mehr oder weniger große Palette von Anwendungen einsetzbar sind.

Doch je universeller diese *Systems-On-A-Chip* sind, desto spezieller muss die Software auf die jeweilige Anwendung zugeschnitten werden. Um so ausgefallener ein Gerät ist und je geringer dessen geplante Stückzahl, um so stärker schlägt sich die Entwicklung dieser Software im Produktpreis nieder. Das gilt umso mehr dann, wenn ein Gerät mit „intelligenten“ Funktionen ausgestattet werden soll, wie z. B. komfortable Benutzerschnittstellen, selbstlernende Steuerprogramme oder umfangreiche Netzwerkfunktionen. Daher ist die Software-Entwicklung für solche Systeme zumeist deutlich aufwändiger und damit teurer als die Hardware-Entwicklung.

Weil der Markt neue Funktionen von den Geräten in immer kürzeren Abständen fordert, ist es von besonderer Bedeutung, nach effizienten Ansätzen für die Software-Entwicklung zu suchen. Für gewöhnlich ist die Beschreibung der eigentlichen Anwendung (z. B. die Implementierung

eines Regelalgorithmus) nur ein Teil des gesamten Entwicklungsaufwandes. Nicht zu vernachlässigen ist der Aufwand für die Ansteuerung der Peripheriekomponenten und der damit verbundenen weiteren Hardware, insbesondere dann, wenn der Mikrocontroller umfangreiche Funktionen anbietet und es ggf. auf ein spezielles Timing ankommt. Ein weiterer Schwierigkeitsfaktor ist die Anbindung an vorhandene Software-Module, wie z. B. an einen Protokoll-Stapel für ein spezielles Bus-System. So erfordert beispielsweise die Anbindung eines Kühlschrankes an ein Home-Automation-Bussystem inkl. der Implementierung eines einfachen Zweipunktreglers mehr als 1400 Zeilen C-Quelltext. Auch zukünftige Änderungen sind in einem solchen Programm schwierig (wenn z. B. zur Kommunikation eine andere Schnittstelle verwendet werden soll).

Das Ziel von JControl ist es, dank eines komponenten- und objektorientierten Konzepts den Entwicklungsaufwand von Software für eingebettete Systeme zu minimieren. Es stellt die Anwendung (also das eigentliche Problem) in den Vordergrund und hält sie dabei leicht änderbar und wartbar. JControl ermöglicht es, eingebettete Anwendungen in der Programmiersprache JAVA zu entwickeln und auf einem Mikrocontroller ablaufen zu lassen.

JAVA

JAVA hat sich in den letzten Jahren einen hervorragenden Stellenwert in der Software-Landschaft erobert. Die Gründe dafür dürften bekannt sein und sollen an dieser Stelle nur kurz aufgezählt werden:

- Objektorientierung,
- Robustheit,
- Plattformunabhängigkeit,
- automatische Speicherverwaltung (Garbage-Collection),
- Multithreading (Mehrläufigkeit),
- Effizienz bei der Programmcode-Erstellung,
- Verfügbarkeit von Entwicklungsplattformen.

JAVA hat sich vor allem im Desktop- und Serverbereich durchgesetzt und es gibt eine ganze Reihe von Anstrengungen, dies auf eingebettete Systeme auszudehnen. Hierbei existieren z. Zt. noch Lücken, was den Umfang, die Komplexität und die Kosten der einzelnen Systeme betrifft. Tabelle 1 vergleicht die verschiedenen JAVA-Varianten und nennt eine Größenordnung für die minimal erforderliche Speicherausstattung, Taktfrequenz und Wortbreite. Die größten Anstrengungen im EmbeddedJAVA-Bereich werden dabei von *Sun Microsystems* selbst mit der JAVA2 Micro Edition (J2ME) unternommen, die in zahlreichen Varianten angeboten wird und mit Profilen in Größe und Funktionalität einstellbar ist. Für die J2ME existiert auch eine schlanke JAVAVM, die KVM, welche heutzutage in der Lage ist, JAVA-Programme auf PDAs auszuführen und zukünftig dies auch auf Mobiltelefonen erlauben soll. Voraussetzung dafür ist aber immer noch ein relativ leistungsfähiger 16-bit-Prozessor und mindestens 128KByte zusätzlicher Speicher. Unser JControl bietet eine Lösung zwischen der J2ME und der JAVACard und ist dabei komplexitätsmäßig vergleichbar mit der JAVACard. JControl hat aber wegen der anderen Aufgabengebiete zusätzliche Eigenschaften, die von den komplexeren Implementierungen

Variante	Anwendungsgebiete	Anforderungen
JAVA2 Enterprise Edition (J2EE)	• Datenbanken • Webserver	GigaBytes Multiprozessoren @ GigaHz
JAVA2 Standard Edition (J2SE)	• Desktop-Anwendungen • Internet-Anwendungen	> MegaByte 32-bit-Prozessor > 100 MHz
JAVA2 Micro Edition (J2ME)	• Set-Top-Boxen • PDAs	< MegaByte 16–32-bit-Prozessor @ 100 MHz
JControl	• Embedded Control • Home-Automation	< 50 KiloBytes 8–32-bit-Prozessor > 8 MHz
JavaCard	• Mobile Datenbank • Kryptographie	< 50 KiloBytes 8-bit-Prozessor < 4 MHz

Tabelle 1: Die verschiedenen JAVA-Varianten

bekannt sind. Dies zeigt sich auch bei der verwendeten Hardware. Abbildung 1 zeigt z. B. eine JControl-Variante mit einer einfachen Tastatur und einem grafischen LC-Display mit 132x64 Pixel. Als Basis dient ein 8-bit-Mikrocontroller. Der Stromverbrauch beträgt bei maximaler Taktfrequenz von 16 MHz lediglich 12mA.

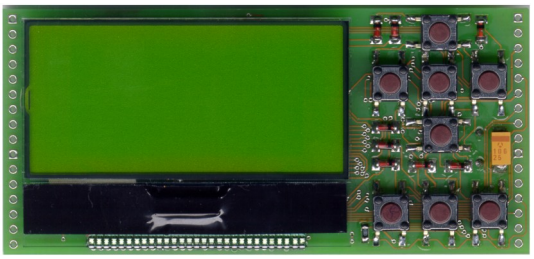


Abbildung 1: Eine JControl-Variante mit Grafik-Display

1 Die JControlVM

Der vom JAVA-Compiler generierte JAVA-Bytecode wird in der Regel nicht direkt auf einem Prozessor ausgeführt, sondern von einer virtuellen Maschine interpretiert. Andere Ansätze wie das Just-in-Time-Compiling erreichen zwar eine deutlich höhere Verarbeitungsgeschwindigkeit, benötigen dafür allerdings auch mehr Speicherplatz. Da JControl auch auf eingebetteten Systemen mit nur wenigen KByte RAM (Mikrocontroller) zum Einsatz kommt, scheiden derartige Optimierungen aus. Nebenbei vereinfacht das Interpreter-Konzept noch einige Dinge, was sich z. B. beim Thread-Scheduling noch zeigen wird (siehe 1.2).

Die JControlVM wurde im Vergleich zu Desktop-VMs um einige Funktionen reduziert. Am augenscheinlichsten ist, dass sie nicht auf ein Betriebssystem aufsetzt, sondern selbst alle notwendigen Betriebssystem-Funktionen beinhaltet. Die Vorteile sind u. a. weniger Speicherverbrauch und schnellere Reaktionszeiten der Anwendung auf externe Ereignisse. Des weiteren wurde auf Datentypen mit einer Größe von mehr als 16 Bit verzichtet, womit der RAM-Bedarf auf beinahe die Hälfte reduziert werden konnte (als Nebeneffekt reduzierte sich dabei auch die Komplexität der virtuellen Maschine, da sich die Anzahl der zu implementierenden Bytecodes

verringert). Auf die Programmierung von JAVA-Anwendungen hat dies keinen Einfluss, wenn nur die unterstützten Datentypen verwendet werden.

Im Übrigen handelt es sich um eine vollwertige VM, die über einen Mikrokernel mit lediglich 12KByte ROM-Bedarf verfügt. Mikrokernel heisst, dass lediglich die für die direkte Ausführung der Bytecodes benötigten Komponenten (*Execution-Engine*, *Heap-Manager* und *Thread-Scheduler*) vorhanden sind (siehe Abbildung 2). Das übrige „Beiwerk“ (*Garbage-*

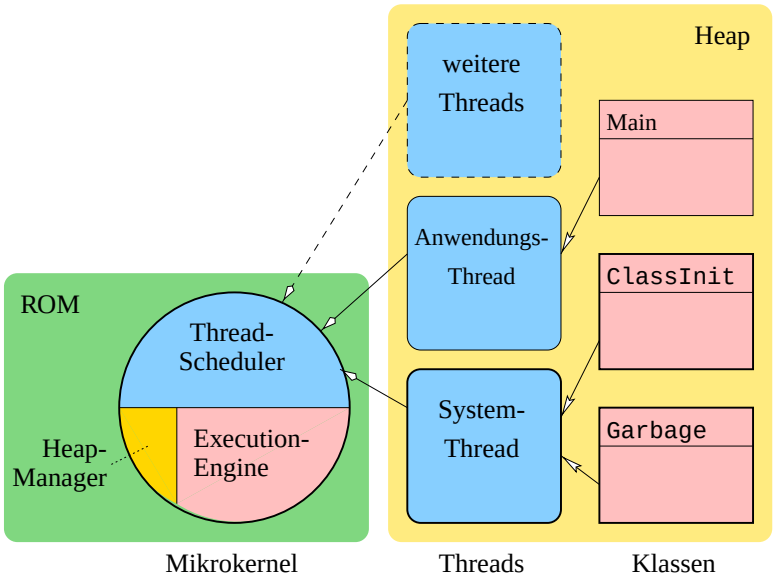


Abbildung 2: Der JControlVM-Systemaufbau

Collector, *Klassen-Initialisierung* und *ROM-Filesystem*) wurde in einen System-Thread mit 8KByte ROM-Bedarf ausgelagert. Diese Elemente laufen im Kontext der virtuellen JAVA-Maschine und können somit auch deren Möglichkeiten nutzen. Der System-Thread ist der Anwendung gegenüber gleichberechtigt und arbeitet vollkommen nebenläufig. So können der Garbage-Collector und die Klassen-Initialisierung inkrementell arbeiten, d. h. sie unterbrechen die Anwendung immer nur für einen kurzen Moment, um jeweils einen kleinen Teil ihrer Aufgaben abzuarbeiten. Dadurch wird die maximale Latenzzeit bei zeitgesteuerten Prozessen reduziert.

1.1 Speicherverwaltung

Die JControlVM ist für Systeme ab 2KByte RAM konzipiert. Im RAM werden nicht nur die JAVA-Objekte abgelegt, sondern auch weitere Laufzeit-Objekte wie z. B. Klassenzugriffstabellen und Threads. Um den knappen Speicher mit möglichst wenig Verschnitt zu nutzen, werden JAVA- und Laufzeitobjekte gleichberechtigt behandelt und in einem Unified Heap abgelegt. Dieser belegt fast das gesamte RAM und unterliegt komplett der kompaktierenden Garbage-Collection.

1.2 Thread-Scheduler

Die JAVAVM steuert selbst die Verteilung der Rechenzeit auf die einzelnen Threads. Dies hat den entscheidenden Vorteil, dass ein Wechsel *immer* nur zu einem kontrollierbaren Zeitpunkt

stattfindet (bei der JControlVM ist dies stets der Übergang zwischen zwei Bytecodes). Somit kann auf eine Variablen-Zugriffskontrolle verzichtet werden (keine lokalen Kopien je Thread), da sie immer und überall einen gültigen Wert aufweisen.

Der Scheduler arbeitet mit einer im Millisekundenraster einstellbaren Zeitscheibe. Er ermöglicht neben den bei JAVA üblichen Prioritäten auch die Verarbeitung von Zeitschranken für einzelne Threads, die dynamisch zur Laufzeit gesetzt und verkettet werden können (EDF-Scheduling; EDF = Earliest Deadline First). Das erlaubt eine rudimentäre Echtzeitabarbeitung von Anwendungen. Bei der Überschreitung einer Zeitschranke wird eine Exception ausgelöst und die Anwendung kann geeignet darauf reagieren.

Jeder Thread verfügt über einen eigenen Speicherbereich auf dem Heap, in dem ein lokaler Operandenstack verwaltet wird. Neben dem JAVA-Stack werden dort auch die Laufzeitdaten aufgerufener Methoden und deren lokale Variablen abgelegt.

1.3 ROM-Filesystem

Bei der Ausführung von JControl-Anwendungen wird auf die benötigten Class-Files zurückgegriffen. Diese können – zusammen mit der JAVAVM – im ROM des Controllers abgelegt worden sein oder sich in einem internen oder externen Flash-Speicher befinden. Für den Zugriff auf einzelne Klassen und auf andere Ressourcen dient eine verkettete Liste als Filesystem, die zusätzlich noch weitere Informationen aufnehmen kann (z. B. Binde-Informationen bei Klassen, die dann nicht zur Laufzeit der Anwendung erzeugt werden müssen; siehe auch Abschnitt 3).

Trotz dieser Optimierungen werden auch ganz normale (unkomprimierte) JAR-Files akzeptiert. Diese lassen sich mit jeder JAVA-Entwicklungssoftware (oder einem ZIP-Archivierer) erzeugen. Die Programmiersoftware für den Flash-Speicher ist dabei Teil des JControl-APIs und befindet sich im ROM des Controllers; neue Anwendungen werden z. B. über die RS232-kompatible serielle Schnittstelle programmiert.

2 Das JControl-API

JControl unterstützt eine Untermenge der gewöhnlichen JAVA-Klassenbibliothek, z. B. aus den Paketen `java.lang` (`Object`, `Thread`, `Exception`, `String`) und `java.io` (`InputStream`, `OutputStream`). Diese Untermenge garantiert die Kompatibilität zum JAVA-Sprachkonzept und stellt die von JAVA gewohnte Funktionalität zur Verfügung [3, 4]. Andere Pakete, z. B. für grafische Benutzeroberflächen (AWT, Swing), machen auf solch einem System hingegen keinen Sinn.

Zusätzlich verfügt JControl über einen Satz eigenständiger API-Klassen, die auf die speziellen Eigenschaften von JControl zugeschnitten sind. Diese API-Klassen gliedern sich in zwei Gruppen:

1. Klassen zur Erweiterung der Funktionalität der virtuellen Maschine und der Sprache JAVA, sowie als Ersatz für JAVA-API-Klassen, die aus Kompatibilitätsgründen nicht übernommen werden konnten (Datentypen),

2. Klassen zur Ansteuerung der an den Controller angeschlossenen Peripheriekomponenten.

Für JControl wurde ein Native-Interface implementiert, mit dem controller-spezifischer Maschinencode direkt in die API-Klassen einbettet werden kann. Auf diese Weise lässt sich von der API- Ebene direkt mit der virtuellen Maschine zusammenarbeiten oder die Hardware der Peripherieeinheiten mittels direktem Speicherzugriffs ansteuern. Die API-Klassen befinden sich zusammen mit der virtuellen Maschine im ROM des Controllers. Aus dem externen Flash ist es hingegen prinzipbedingt nicht möglich, nativen Code auszuführen. Alle API- Klassen befinden sich in einem eigenständigen Paket `jcontrol`.

2.1 Das JControl-System-API

Die wichtigsten System-Klassen von JControl werden an dieser Stelle kurz vorgestellt:

`jcontrol.system.TimedThread` erweitert die Thread-Funktionalität. Da Threads bei JControl erweiterte Funktionalitäten aufweisen (siehe 1.2), die die Klasse `java.lang.Thread` nicht kennt, wurden diese in einer eingenständigen Klasse zusammengefasst. Hier ist es u. a. möglich einen Thread mit einer Zeitschranke zu versehen oder den Scheduler zu steuern,

`jcontrol.system.Download` stellt die Schnittstelle zu einer Entwicklungsplattform dar. Diese Klasse wird automatisch aufgerufen wenn keine Anwendung vorhanden ist, und wechselt dann in den in Kapitel 1.3 kurz vorgestellten Download- und Flash-Programmiermodus. Die Klasse kann dann später auch explizit von der Anwendung aufgerufen werden, um eine Neu- Programmierung zu ermöglichen,

`jcontrol.lang.Math` stellt einige arithmetische Operationen zur Verfügung, die aufgrund der verkürzten Wortbreite nicht in `java.lang.Math` vorkommen.

2.2 Das JControl-Peripherie-API

Das JControl-API für die Peripherie ist hierarchisch in drei Ebenen unterteilt:

Hardware-Ebene lediglich einfachste digitale Ein- und Ausgabefunktionen (z. B. für das Setzen oder Lesen einzelner Port-Pins oder von Bussen),

komponentenorientierte Ebene anwendungsorientierte Funktionen für die Steuerung elektronischer Komponenten (z. B. Display, Tastatur, Schnittstellen),

geräteorientierte Ebene zur Steuerung komplexer Geräte (z. B. Waschmaschinen, Kühlschränke etc.).

Die höheren Ebenen setzen auf die Implementierung der unteren Ebenen auf. Dies geschieht auch auf Basis des controller-nativen Codes, wodurch sich Geschwindigkeits- und Speicherplatzvorteile ergeben. Dem Applikationsprogrammierer stehen die Klassen aller drei Ebenen zur Verfügung, nicht jedoch der direkte Zugriff auf die Hardware des Controllers.

Zu den Klassen im einzelnen:[1]

[1] Je nach Variante bzw. Ausstattung des verwendeten JControl-Modells kommt natürlich nur ein Teil dieser Klassen zur Verwendung.

jcontrol.io.Portpins ermöglicht die gezielte Ansteuerung einzelner (freier) Portpins. Damit kann z. B. die berühmte rote Leuchtdiode angesteuert werden oder auch eigene mit dem Controller verbundene Hardware,

jcontrol.io.ADC ermöglicht das Abfragen von Analogwerten an bestimmten Portpins (proportional zur anliegenden Spannung),

jcontrol.io.PWM kontrolliert die Fähigkeit einiger Pins, pulsweitenmodulierte Signale abzugeben (z. B. für DC-Motoren via Treiber),

jcontrol.io.iMelody benutzt `PWM`, um einen Ton oder eine Tonfolge gemäß der iMelody-Spezifikation [6] auf einem Piezo-Summer abzuspielen. Dies kann synchron (`run()`) oder asynchron in einem eingenständigen Thread erfolgen (`Runnable`),

jcontrol.io.IR ermöglicht die Infrarot-Kommunikation mit Consumer-Geräten,

jcontrol.io.RS232 stellt einen Kommunikationskanal über die serielle Schnittstelle zur Verfügung,

jcontrol.io.CAN ermöglicht den Datenaustausch über den CAN-Bus,

jcontrol.eib.* (Paket) bietet Zugriff auf ein Gebäudekommunikations-Netzwerk mit dem European Installation Bus.

jcontrol.io.ChipCard stellt eine Schnittstelle zu Chip-Karten her wie z. B. Telefonkarten und ermöglicht das Auslesen der Daten,

jcontrol.io.RTC stellt eine Schnittstelle zu einem Uhrenchip bereit. Hiermit ist es z. B. möglich, den Controller abzuschalten und zu einer bestimmten Zeit automatisch wieder anlaufen zu lassen,

jcontrol.io.Flash ermöglicht den Zugriff auf den Flash-Speicher, z. B. zum Ablegen von persistenten Objekten,

jcontrol.io.Keyboard fragt die Tastatur auf der JControl-Platine ab und gibt das der gedrückten Taste zugeordnete Zeichen zurück. Es kann auch auf einen Tastendruck gewartet werden,

jcontrol.io.Display steuert ein grafisches LC-Display an. Dafür werden einige Grundfunktionen zum Zeichen von Linien, Kreisen, Bit-Images und Texten zur Verfügung gestellt,

jcontrol.ui.* (Paket) stellt unterschiedliche grafische Bedienelemente zur Verfügung, die im Steuerungs-, Überwachungs- und Regelungssektor sinnvoll sind, z. B. virtuelle Zeigerinstrumente oder per Tastatur steuerbare Schieberegler,

Bei all diesen Klassen wird konsequent das objektorientierte Konzept von JAVA umgesetzt. Ein Beispiel sind Streams: Ob nun Zeichen von der Tastatur entgegengenommen werden, von der seriellen Schnittstelle oder von einer Chip-Karte, ist für die Anwendung völlig unerheblich, genauso wie das Schreiben auf ein Display (gleich welchen Typs) oder den CAN-Bus.

Die Ansteuerung und die Initialisierung der Hardware wird komplett von den API-Klassen übernommen. Bei der Erzeugung eines Objekts wird die Komponente initialisiert und bei der Freigabe (via Garbage-Collector) die entsprechende Hardware abgeschaltet. Für die Anwendung ist das völlig transparent. Möglich wird dies durch die bei JAVA üblichen Konstruktoren und Finalisatoren.

3 Die JControl-Programmierumgebung

Bei der Konzeption von JControl wurde Wert darauf gelegt, dass prinzipiell zur Entwicklung von Anwendungen keine speziellen Werkzeuge benötigt werden. Erforderlich sind lediglich ein JAVA-Compiler, ein JAR-Archivierer und ein einfaches Terminal-Programm (für das Laden der Anwendungen auf den Controller). JControl unterstützt genau ein JAR-Archiv im Flash-Memory. Die zu startende Anwendung kann dabei – wie bei JARs üblich – mittels des eingebundenen Manifest-Files festgelegt werden. Dieses Verfahren garantiert eine größtmögliche Kompatibilität zum JDK. Um den Entwicklungsprozess zu vereinfachen, zu optimieren und komfortabler zu gestalten, wurden einige unterstützende Werkzeuge implementiert. Diese stehen auf allen Plattformen zur Verfügung, die JAVA2 und das grafische Toolkit Swing unterstützen.

3.1 Anwendungs-Entwicklung für JControl

Das wichtigste Werkzeug ist die integrierte Entwicklungsumgebung VJControl (siehe Abbildung 3). Diese ermöglicht nicht nur die Eingabe und Compilierung der JAVA-Anwendungen, sondern bindet auch die übrigen JControl-Werkzeuge zum Archivieren und Upload auf den Controller in eine komfortable Benutzeroberfläche ein. So steht z. B. ein Projekt-Management zur Verfügung, das die Zusammenstellung der Klassen weitgehend automatisiert, die für eine Anwendung benötigt werden. Diese Klassen können mit wenigen Handgriffen in den Flash-Speicher des Controllers geladen werden.

Eine weitere Funktion der JControl-Werkzeuge ist die Unterstützung des speziellen JCA-Formats. Das eigenständige JControl-Archiv-Format ist nicht nur kompakter als die unkomprimierten JAR-Archive, sondern es erlaubt auch das Laden von mehreren APIs und Anwendungen auf den Controller (und ein selektives Löschen einzelner Archive), da es speziell auf die Architektur des Flash-Speichers ausgelegt ist. Der JControl-Archivierer arbeitet dabei implizit optimierend, d. h. durch Weglassen irrelevanter Information in den Class-Files werden diese verkürzt, ohne dass sie an Funktionalität verlieren. Auf dem Entwicklungsrechner kann noch eine zweite Version des Archivs (als JAR) mit kompletten Debugging- Informationen erzeugt werden. Dies kann dann für den JAVA-Compiler als Grundlage dienen, für den Fall dass später auf dem Controller geladene APIs von zukünftigen Anwendungen benutzt werden sollen.

Schließlich wird noch die Möglichkeit gegeben, die Klassen vorab um Binde-Informationen zu erweitern (Vorverlinkung). Die Klassen gewinnen dann zwar etwas an Größe und belegen mehr Flash-Speicher bzw. ROM, die Symbolreferenztabellen müssen dann aber nicht zur Laufzeit im knappen RAM abgelegt werden. Ein positiver Nebeneffekt dabei ist ein schnellerer Zugriff auf die Klassen, da die Referenzen beim ersten Zugriff nicht erst mühsam ermittelt werden müssen.

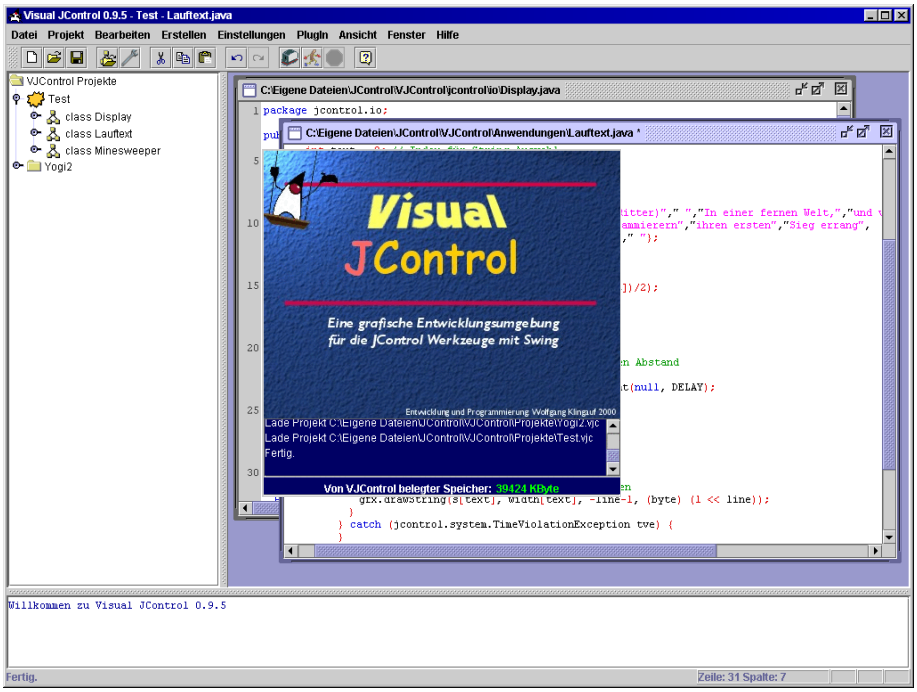


Abbildung 3: Die JControl Entwicklungsumgebung

3.2 Simulation, Verifikation und Präsentation

Ein wichtiger Aspekt bei der Entwicklung von Anwendungen für eingebettete Systeme im Allgemeinen ist die Testbarkeit auf dem Entwicklungssystem. Die Plattformunabhängigkeit von JAVA bietet hier einige Möglichkeiten. Beispielsweise stehen viele JControl-APIs auch als rein grafische Komponenten zur Verfügung, die alle Ein- und Ausgaben simulieren (Virtual JControl). So können die fertigen Anwendungen auch auf jeder anderen Plattform (sogar in einem Internet-Browser – es handelt sich um Applets) ausgeführt werden, die JAVA unterstützen. Natürlich kann auf diese Weise nicht die mit dem Controller verbundene Hardware getestet und deren Interaktion überprüft werden. Die Mensch-Maschine-Schittstelle (Tastatur, Display etc.) lässt sich so jedoch viel schneller perfektionieren, da die Test-Zyklen kürzer werden (siehe Abbildung 4). Nebenbei können diese eigentlich zur Simulation bestimmten Anwendungen auch

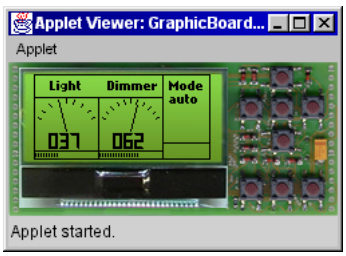


Abbildung 4: Eine simulierte JControl-Anwendung

dazu dienen, die Produkte auf Web-Pages zu präsentieren [1].

Sollen reale Peripherieeinheiten in den Simulationsvorgang einbezogen werden, bleibt nur die Ausführung der Anwendung direkt auf dem Controller. Allerdings besteht dann im Fall des Versagens der Anwendung das Problem der Fehlerlokalisation. Daher wird es zukünftig mit

JControl auch möglich sein, Remote-Debugging durchzuführen. Dabei überträgt die virtuelle Maschine kontinuierlich oder auf Anfrage ihren Status z. B. über die serielle Schnittstelle an den Host. Auf diesem befindet sich ein Debugger-Frontend, das in die VJControl- Oberfläche integriert ist und den momentanen Status der virtuellen Maschine visualisiert. Zur Verwendung kommt dabei ein reduziertes Protokoll gemäß der JPDA-Spezifikation [5].

Fazit

JControl setzt die Programmiersprache JAVA auf sehr kompakte Mikrocontroller-Systeme um. Dadurch eignet sich JControl auch für Anwendungen, bei denen es besonders auf einen günstigen Preis, kleine Ausmaße und niedrigen Stromverbrauch ankommt. Sogar Single-Chip-Systeme sind realisierbar. Trotz der extremen Kompaktheit bietet JControl aber dennoch die Vorteile der Programmiersprache JAVA. Das JControl- API realisiert eine plattformunabhängige Schnittstelle zur Implementierung von Steuerungen. Die Entwicklung von Anwendungen kann mit gewohnten Werkzeugen erfolgen, wird aber auch mit eigens dafür zugeschnittenen Werkzeugen unterstützt. Die Software ist sowohl auf dem Entwicklungsrechner wie auf dem eingebetteten System verifizierbar und kann ohne Aufwand im Internet präsentiert werden.

Literatur

[1] JControl-Homepages: http://www.jcontrol.de, http://www.jcontrol.org

[2] Böhme, Helge; Telkamp, Gerrit; Eine JAVAVM für Anwendungen in der Home Automation; 9. E.I.S.-Workshop, 22. bis 24. September 1999, Darmstadt; Tagungsband, GMM-Fachbericht 29, Seiten 107–114; VDE-Verlag; ISBN 3-8007-2485-5

[3] 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

[4] Gosling, James; Joy, Bill; Steele, Guy; The JAVA™ Language Specification; Addison-Wesley, 1996; ISBN 0-201-63451-1
http://java.sun.com/docs/books/jls

[5] The JAVA™ Platform Debugger Architecture (JPDA):
http://java.sun.com/j2se/1.3/docs/guide/jpda/index.html

[6] iMelody V1.2c Specification:
http://www.irda.org/members/docs/iMelody_Errata.pdf