JControl – Rapid Prototyping und Design Reuse mit Java

Tagungsbeitrag zum *11. E.I.S.-Workshop 2003*

Helge Böhme[1], Wolfgang Klingauf [1], Gerrit Telkamp[2]

[1]Abteilung E.I.S., TU BRAUNSCHWEIG
Mühlenpfordtstraße 23
38106 Braunschweig
ϟh.boehme@tu-bs.de, ☎ 0531/391-3108
ϟw.klingauf@tu-bs.de, ☎ 0531/391-3105

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

Kurzfassung

Bei der Entwicklung von Software für eingebettete Systeme unterstützt Java Rapid Prototyping und Design Reuse durch seinen plattformunabhängigen und objektorientierten Ansatz. In diesem Papier wird JControl vorgestellt, eine speicherplatzsparende Java-Lösung, die sogar auf sehr kleinen eingebetteten Systemen funktioniert. Neben einer Gegenüberstellung des konventionellen Entwicklungsprozesses mit dem JControl-Ansatz wird gezeigt, wie ein ausgefeiltes Konzept aufeinander abgestimmter Werkzeuge den Entwickler bei der Implementierung unterstützt und dadurch Entwicklungszeit und -kosten eingespart werden können.

Einleitung

Dieses Papier stellt die Vorgänge bei der Entwicklung von Java-Anwendungen mit JControl dar. Es werden in den ersten beiden Kapiteln kurz die Gründe, die zu der Entwicklung des Systems führten, genannt und das JControl-Konzept vorgestellt. Im anschließenden Hauptkapitel 3 wird der Design-Flow vorgestellt. Schließlich wird noch ein Fallbeispiel aus der Praxis gezeigt, das JControl zur Entwicklung einer kundenspezifischen Anwendung nutzt. Eine genaue Beschreibung der in diesem Paper erwähnten virtuellen Java-Maschine wurde bereits in früheren Arbeiten [3, 4] gegeben.

1 Motivation

Eingebettete Systeme (*Embedded Systems*) drängen mehr und mehr in unser Leben. Ob Raumthermostat, Kaffeemaschine oder Toaster, ob Set-Top-Box, Spiele-Konsole oder MP3-Player, alle diese unterschiedlich komplexen Systeme sind auf Mikroelektronik angewiesen. Bei vielen Systemen kommen neben oder anstatt anwendungsorientierten Schaltkreisen General-Purpose-Prozessoren oder -Controller zum Einsatz.

Große und mittelgroße eingebettete Systeme machen z. Zt. einen Entwicklungsprozess durch, der sie immer näher an offene Systeme bringt – sie nähern sich den PC-Architekturen an. Dies erleichert auch den Entwicklungsprozess der Geräte bzw. kommt deren Wartbarkeit und Erweiterbarkeit zugute und senkt gerade im Bereich der Massenstückzahlen die Kosten. Auch drängen immer mehr der sgn. Echtzeitbetriebssysteme (*Real Time Operating Systems*, RTOS) auf den Markt.

Auf der anderen Seite verringern sich im Bereich der kleinen Geräte mit speziellen Aufgabengebieten die Entwicklungskosten nicht gleichermaßen wie die Komplexität des Gesamtsystems. Dies liegt daran, dass es sich bei diesen Systemen um eher geschlossene Systeme handelt. Die Entwicklung von Anwendungensprogrammen kann nicht auf dem Gerät selbst erfolgen und die Tests der Anwendungen sind aufwändig und lassen sich in der Regel nur mit teurer Zusatzhardware durchführen.

1.1 Keine Standards bei kleinen Mikrocontrollern

Die Produktpalette von in diesem Bereich eingesetzten 8-Bit-Mikrocontrollern ist so groß, dass sich in der Regel für ein bestimmtes Aufgabengebiet eine perfekt passende Variante finden lässt. Viele Controller haben integrierte Peripheriekomponenten und Speicher, so dass in vielen Anwendungsfällen kaum Zusatzhardware erforderlich ist. Auf der Software-Seite sieht es allerdings anders aus. Oft liefern Hersteller für die integrierten Peripheriekomponenten keine Bibliotheken oder zumindest Code-Beispiele mit. In der Regel entstehen für jedes Produkt eigene Softwarebibliotheken von Grund auf neu, da es keine Standards für Softwarekomponenten gibt, die über Ansi-C hinausgehen.

Der Entwicklungsaufwand steigt noch mehr, wenn sich nach der Planungsphase Änderungen ergeben, z. B. wenn nun mehr Speicher erforderlich ist, als der Controller bietet, oder plötzlich vom Kunden andere Schnittstellen gewünscht werden, die der gewählte Controller-Hersteller nicht bietet. Der zu diesem Zeitpunkt bereits erstellte Code kann i. d. R. nicht oder nur mit großem (und fehlerträchtigen) Arbeitsaufwand auf die neue Plattform portiert werden.

1.2 Hoher Entwicklungsaufwand

Dass sich ggf. erst nach der Planungsphase Änderungen ergeben, hängt mit dem hohem Zeitaufwand bis zu der ersten lauffähigen Version des Produkts zusammen. Bei kleinen eingebetteten Systemen ist *Rapid Prototyping*, also das schnelle Er- und Zusammenstellen von Testversionen genauso wünschenswert, wie bei größeren Systemen, wo dies schon häufiger eingesetzt wird.

Kaum ein Entwickler erstellt zunächst unaufgefordert für den Kunden eine vereinfachte Version der Software, weil er sich davon – im Endeffekt – eine Arbeitserleichterung erhofft. Für den Kunden wäre diese Vorgehensweise allerdings eine Möglichkeit, die Fähigkeiten der späteren vollständigen Version kennen zu lernen und ggf. seine Anforderungen und Wünsche früher und genauer zu spezifizieren. Wenn aber solch eine Demoversion mit sehr geringem Arbeits- und Zeitaufwand erstellt werden könnte und diese nicht auf eine funktionsfähige Hardware der Zielplattform angewiesen wäre, dann wäre die Hemmschwelle für ihre Erstellung wesentlich geringer und die Wahrscheinlichkeit für spätere (eigentlich: zu späte) Änderungen würde sinken.

1.3 Java hilft

Wünschenswert ist also auch bei kleinen eingebetteten Systemen ein Controller-unabhängiges Software-Framework, das den Enwickler ebenso unabhängig von der verwendeten Hardware macht, wie den Kunden. Auch die üblichen Systemverwaltungsaufgaben, die ein Betriebssystem leistet, sollen ohne Ansehen der Hardware immer gleich funktionieren. Diese Anforderungen erfüllt die Programmiersprache Java.

Java ist bekannt für Plattformunabhängigkeit und Vielseitigkeit aber auch für Ressourcenhunger. Aus diesem Grund werden auf eingebetteten Systemen eigene, kleinere Profile mit eingeschränkten Klassenbibliotheken (APIs) etabliert, z. B. JBed [5] mit dem von *Sun* für die Java 2 Micro Edition (J2ME) spezifizierten CLDC-Profil [6]. Eine Lösung für besonders kleine Systeme, die für eingebettete Steuerungen eingesetzt werden sollen, ist das von uns entwickelte JControl.

Es gibt in diesem Größensektor unterhalb der J2ME nur wenige Java-Implementierungen [7, 8], alle mit derselben Grundidee, Java auf möglichst kleinen Systemen laufen zu lassen. Die Umsetzungen unterscheiden sich hingegen deutlich, gerade die Entwicklungswerkzeuge und die API-Bibliothek (falls überhaupt vorhanden) betreffend. JControl scheint hier die umfassenste und flexibelste Lösung zu bieten.

2 JControl

Bei JControl handelt es sich um ein Java-Paket, bestehend aus virtueller Java Maschine, Klassenbibliothek und Entwicklungswerkzeugen, optimiert für besonders geringen Speicherverbrauch auf dem eingebetteten System. Bei JControl wird zugunsten des Speicherverbrauchs auf mögliche Performancevorteile durch Compilierung der kompletten Java-Laufzeit und -Anwendungen in nativen Code verzichtet.

Bei Java-Bytecode handelt es sich um eine sehr kompakte Repräsentation von Algorithmen. Interessanterweise entstand der Bytecode zu einer Zeit, als sich der bis heute fortgesetzte Trend zu breiteren Prozessorarchitekturen mit 32- oder 64-Bit-aligned RISC-Instruction-Sets deutlich abzeichnete. Diese haben auf Systemen mit schnell angebundenem Speicher ihre Vorteile, da die Dekodierung der Opcodes und der Speicherzugiff selbst vereinfacht wird. Somit kann Bytecode mit seinen variablen Parametergrößen, selbst wenn man von dem Interpreter-Overhead absieht, auf diesen modernen Architekturen nicht ohne eine Übersetzung in Native-Code effizient ausgeführt werden. Anders sieht dies bei kleinen 8-Bit-Systemen aus. Untersuchungen haben gezeigt, dass – je nach Anwendungsfall – die meisten verwendeten Opcode-Parameter (Immediate, Displacement, Branch) mit einem Byte auskommen [10, Kap. 3.4ff].

Dass Java-Class-Files in der Regel doch relativ groß werden, liegt selten an umfangreichen Algorithmen im Bytecode, sondern an dem mit im Class-File enthaltenen Link-, Debug- und Symbol-Informationen. Diese werden für die Ausführung von Java-Anwendungen nicht benötigt und können entfernt werden.

2.1 Eine speicherplatzsparende Java-Implementierung

Um zu verstehen, warum wir bei JControl gerade diesen Weg gewählt haben, ist ein Blick auf die bei den angestrebten Zielsystemen vorhandene Speicherausstattung nötig. Eins der verwendeten Systeme verfügt über lediglich 2,5 KByte RAM und 60 KByte ROM zuzüglich Flash-Speicher von 64 KByte bis 256 KByte Größe. Sollen hier komplexe Anwendungen ablaufen, darf keine Verschwendung stattfinden. Der kompakte Bytecode findet unmodifizert im Festwertspeicher Platz und bietet gegenüber dem nativem Befehlssatz des Mikrocontrollers eine Kompression – je nach Anwendungsfall – um einen Faktor bis zu sechs.[1] Die Bytecode-Interpretierung bietet also eine Dekompression von Algorithmen zur Laufzeit ohne eine Zwischenspeicherung im RAM zu benötigen. Das RAM bleibt frei für Laufzeitdaten.

Die objektorientierte Struktur von Java erfordert zur Laufzeit Zugriffstabellen für Methodenaufrufe, die bei gewöhnlichen interpretierenden Java-Systemen beim Laden einer

[1] Dieser Faktor ergibt sich bei der Umsetzung des Algorithmus `y=(short)(i*x+i+i-i/x>>2)`, welcher in 15 Bytes Java-Bytecode umgesetzt wird oder eine Assembler-Sequenz von 90 Bytes erfordert, für die Division wurde eine Unterroutine aufgerufen.

Klasse erzeugt werden. Die Erfahrung hat gezeigt, dass dieser Ansatz beim vorhandenen RAM lediglich Anwendungen mit (sehr) wenigen Klassen zulässt, was das objektorientierte Konzept ad absurdum führt. Da eingebettete Controller in der Regel nach der Anwendungserstellung nicht mehr verändert werden, ist kein Nachladen von Klassen zur *Laufzeit* nötig. Es ist also möglich, diese Zugriffstabellen bei der Herstellung der Anwendung auf einem Entwicklungsplatz zu erzeugen und fest in das Gerät einzubauen, als Teil der Anwendung im ROM oder Flash-Speicher. In diesem Fall wird das RAM nur noch von den Anwendungsdaten (Java-Objekten) beansprucht.

2.2 Entwickeln mit JControl

Das Erzeugen dieser ROM- oder Flash-Images wird von einem speziell auf JControl zugeschnittenem Entwicklungswerkzeug, dem JCManager durchgeführt. Dieses Werkzeug unterstützt den Entwickler auch bei der Zusammenstellung der nötigen Klassen und weiterer Ressourcen einer Anwendung. Es kennt die auf einem JControl-Gerät vorhandenen API-Klassen und kann fehlende Klassen des JControl-APIs automatisch ergänzen und zusammen mit der Anwendung in den Flash-Speicher des Controllers laden.

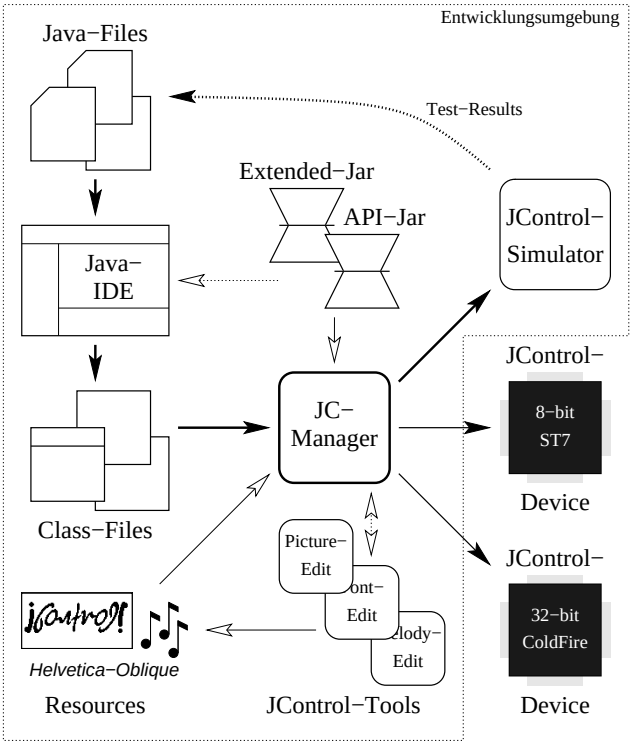

Bild 1: Der JControl Design-Flow

3 Die JControl-Entwicklungsumgebung

Das Ziel bei JControl war ein einfacher und schneller Entwicklungszyklus, welcher in **Bild 1** skizziert wird. Abgesehen von der zentralen Komponente des JCManagers, die den Übergang der Software zur Hardware verwaltet, unterscheidet dieser sich nur wenig von üblicher Java-Entwicklungsarbeit. Im Folgenden soll auf die einzelnen Komponenten der JControl-Entwicklungsumgebung genauer eingegangen werden.

3.1 Die JControl-Klassenbibliothek

Die Verfügbarkeit einer virtuellen Java Maschine auf einem eingebetteten System alleine führt nur zu einer geringen Vereinfachung der Programmierung. Entscheidend ist, dass die Programmiersprache selbst durch das objektorientierte Konzept dem Programmierer ein einfaches Werkzeug in die Hand gibt, die Anwendung sinnvoll zu gliedern und allgemeinere Module leicht für einen späteren Gebrauch in anderen Anwendungen einzukapseln. Diese Module können in eine Klassenbibliothek aufgenommen werden.

Bei JControl werden drei Typen von Klassen unterschieden.

1. interne API-Klassen,
2. externe (extended) API-Klassen und
3. externe Anwendungs-Klassen.

Zu 1. Die internen API-Klassen befinden sich zusammen mit der virtuellen Maschine auf dem ROM des Mikrocontrollers und stellen die Grundfunktionaltät zur Verfügung. Dies sind zunächst die bekannten Erweiterungen der Programmiersprache Java, die Threads, Exceptions, Strings etc. zur Verfügung stellen. Dafür existiert bei JControl ein Subset der Klassen des Pakets `java.lang`, die ihrerseits etwas gekürzt wurden. In den Paketen `jcontrol.lang` und `jcontrol.system` befinden sich weitere JControl-spezifische Spracherweiterungen, z. B. *Earliest-Deadline-First*-(EDF-) Scheduling für Anwendungen mit erhöhten Anforderungen an die Echtzeit-Verarbeitung.

Zusätzlich zu diesen üblichen Klassen befinden sich die Klassen mit auf dem Controller, die dem Programmierer den Zugriff auf die Hardware erlauben. Die Zusammenstellung dieser Klassen hängt von den auf dem benutzten Controller vorhandenen Hardware-Komponenten ab. Davon unabhängig bleiben allerdings die Schnittstellen dieser Klassen, so dass ein Austausch des Controllers bei unveränderten Anwendungen möglich wird. **Tabelle 1** stellt einige dieser Klassen vor. Damit diese Klassen möglichst speichereffizient funktionieren, wurde auf eine tiefe Vererbungshierarchie verzichtet und es wurden insbesondere bei I/O-Klassen *Interfaces* verwendet, um einen universellen Zugriff zu ermöglichen [2].

Klasse	Beschreibung
Paket `java.lang.`	
`Object`	Objektverwaltung und -synchroniserung
`Thread`	mehrere Programmfäden erzeugen und verwalten
`Exception`	Fehlerverwaltung
`String`	Umgang mit Zeichenketten
Paket `jcontrol.lang.`	
`Deadline`	aktiviert den EDF-Scheduler
`Math`	einfache mathematische Festkommafunktionen
Paket `jcontrol.system.`	
`Management`	Controller- und VM-Kontrolle (Neustart etc.)
`Download`	laden von neuen Anwendungen (startet automatisch)
Paket `jcontrol.io.`	
`Portpins`	zum Erzeugen und Auslesen von elektr. Signalen
`ADC`	Analog-Digital-Converter des Controllers benutzen
`PWM`	zum Erzeugen von Pulsweitenmodulierten Signalen
`I2C`	Zugriff auf die gleichnamige Schnittstelle
`Display`	grafische Ausgabefunktionen
`Keyboard`	Tastenabfrage
`Resource`	indirekter Zugriff auf Daten in ROM oder Flash
`Flash`	blockweises Lesen und Schreiben
Paket `jcontrol.comm.`	
`RS232`	Zugriff auf die gleichnamige Schnittstelle
`CAN`	Zugriff auf die gleichnamige Schnittstelle

Tabelle 1: Einige interne API-Klassen

Zu 2. Nur die internen Klassen können mit native Code auf die Hardware des Controllers zugreifen. Dies schirmt den Anwendungsprogrammierer von der Hardware ab, da er selbst die internen Klassen nicht verändern kann. Die internen Klassen schöpfen die Fähigkeiten des Controllers (und der ggf. damit verbundener Hardware) aus und werden aus Platzgründen einfach gehalten. Komfortable Erweiterungen zu diesen Klassen können aus einer Bibliothek externer API-Klassen entnommen werden, deren Struktur **Tabelle 2** darstellt.

Paket	Beschreibung
Paket `java.util.`	einige der Java-Erweiterungen
Paket `jcontrol.storage.`	komfortablerer Zugriff auf Flash-Speicher (und anderen) mittels Streams und Tags
Paket `jcontrol.ui.`	ein einfaches Framework für grafische Benutzerschnittstellen, benutzt `Display` und `Keyboard`
Paket `jcontrol.ext.`	diverses, z. B. das Abspielen von Melodien

Tabelle 2: Einige externe API-Pakete

Der Anwendungsentwickler muss zwischen diesen beiden Typen von API-Klassen nicht unterscheiden, er benutzt sie einfach. Die externen API-Klassen werden später – je nach Bedarf – zusammen mit der Anwendung auf den Controller geladen.

3.2 Compilieren

Ziel war es, die Entwicklung von Anwendungen für JControl genauso einfach zu machen, wie die Entwicklung von gewöhnlichen Java-Anwendungen. Jeder Java-Entwickler hat sich im Laufe der Zeit an eine Entwicklungsumgebung gewöhnt, die er nun mit JControl weiterverwenden kann, um eingebettete Anwendungen zu entwickeln. Damit die Integration möglichst einfach ist, wird jede mit JControl ausgestattete Hardware mit entsprechenden JARs[2] ausgeliefert, die in den Java-Classpath eines beliebigen Compilers aufgenommen werden. Nun ist eine Compilierung per Mausklick oder Tastendruck möglich.

3.3 Zusammenstellen

Mit der einfachen fehlerfreien Compilation ist es bei eingebetteten Anwendungen allerdings noch nicht getan. Die entstandenen Class-Files müssen zu einem Archiv zusammengefasst werden, ähnlich einem gewöhnlichen JAR, das auch auf PC-Plattformen Java-Anwendungen zusammenstellt. Diese Zusammenstellung übernimmt der JCManager. Dieses Programm ist mehr als ein gewöhnlicher Archiver, es vermittelt zwischen der Anwendung, dem Zielsystem und weiteren Werkzeugen zur Ressourcen-Erstellung.

Eine Anwendung besteht gewöhnlich nicht nur aus Code, sondern auch aus Daten, dies können z. B. Grafiken oder Zeichensätze sein, die auf dem Display dargestellt werden sollen, oder Melodien, die über einen Summer ertönen sollen. Diese Ressourcen können direkt dort erstellt werden, wo sie gebraucht werden, also bei der Zusammenstellung der Anwendung – und sie können erstellt werden *wie* sie gebraucht werden, also in dem Format, die die Zielarchitektur benötigt. Der JCManager verfügt über eine Datenbank mit Geräteprofilen, d. h. er kennt in Abhängigkeit vom Zielsystem die benötigten Zielformate und die vorhandenen API-Klassen. **Bild 2** zeigt eines der Werkzeuge bei der Erstellung von Ressources für ein Display. Ist solch eine Archivzusammenstellung (Job) für eine Anwendung einmal erstellt worden, dann sind die beiden folgenden Schritte ebenso leicht aufzurufen, wie zuvor der Compiler.

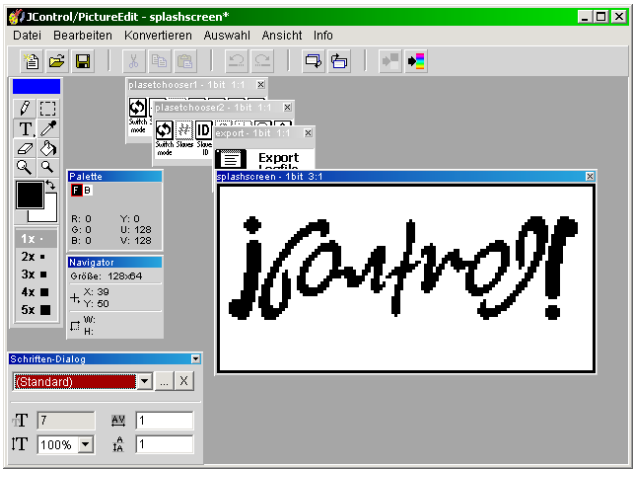

Bild 2: Ressource-Erstellung mit JControl-Picture-Edit

[2] JARs sind übliche Archive für Java-Klassen

3.4 Simulation auf dem Entwicklungsplatz

Jedem Job kann eine Zielarchitektur zugeordnet werden, die auch der JControl Simulator auswertet und eine passende Simulationsumgebung zur Verfügung stellt. So kann ein Großteil der Interaktion der Anwendung mit dem Benutzer und den Komponenten des Zielsystems simuliert werden, **Bild 3** zeigt ein Beispiel. Es kann also schon nach relativ kurzer Entwicklungszeit geprüft werden, wie die eingebettete Anwendung *aussehen* wird und wie sie sich bei der Bedienung *anfühlt*, dies ggf. sogar bevor eine entsprechene Hardware gebaut wurde oder weitere im Hintergrund ablaufende Steuerungs- und Regelungsalgorithmen fertiggestellt sind.

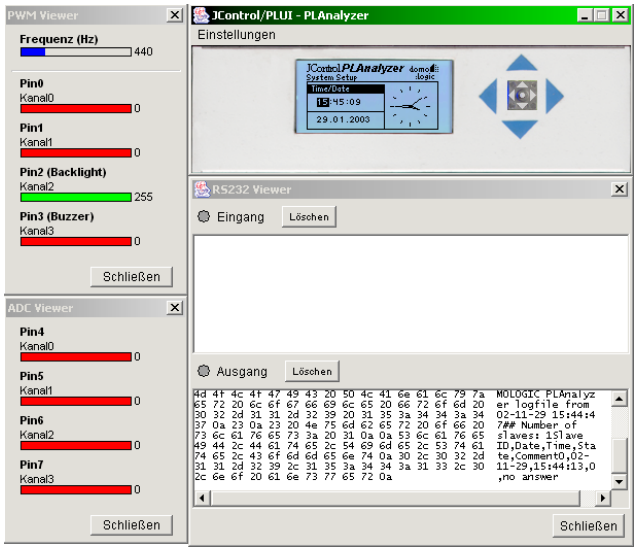

Bild 3: Simulation einer eingebetteten Anwendung

Der Simulator läuft auf der normalen JavaVM des Entwicklungsrechners. Er verfügt über einen eigenen Satz der JControl-APIs, der die Schnittstellen (Display, Tastatur, RS232 etc.) dort in einem Fenster nachbildet. Der JCManager stellt beim Aufruf für den Simulator ein spezielles JAR der Anwendung zusammen.

3.5 Programmieren der Anwendung in das eingebettete System

Die Programmierung einer Anwendung in den Flash-Speicher des eingebetteten Systems erfolgt genau so wie der Aufruf des Simulators. Die JControl-Hardware muss dazu mit dem Entwicklungsrechner verbunden sein (z. B. über die serielle Schnittstelle, es ist kein externes Programmiergerät nötig). Der JCManager kann nun die genaue Version der dort vorhandenen APIs feststellen und passt sich bei der Archivierung der Anwendungs-Klassen und -Ressourcen entsprechend an, ggf. werden an dieser Stelle Inkonsistenzen festgestellt. Es wird ein spezielles auf das System zugeschnittene Archivformat erzeugt, welches von irrelevanten Informationen befreit ist und die bereits erwähnten Zugriffstabellen enthält (siehe 2.1). Die Zugriffstabellen werden dabei mit den auf der Hardware vorhandenen Tabellen der internen API-Klassen abgeglichen.

Die Verwendung dieser JControl-Werkzeuge und der JControl-Designflow soll nun anhand eines Praxisbeispiels erläutert werden.

4 Fallbeispiel: Power-Line-Analyzer

PLAnalyzer steht für *Power-Line-Analyzer* und ist ein verteiltes Messsystem zur Untersuchung der Qualität von Kommunikationslösungen, die auf der Power-Line-Technologie basieren. Es wurde im Rahmen einer Machbarkeitsstudie für einen Hausgerätehersteller entwickelt und sollte in möglichst kurzer Zeit fertiggestellt werden. Eine auch von technischen Laien bedienbare Benutzerschnittstelle war gefordert, um einen flexiblen Einsatz des Messsystems in verschiedenen Testszenarien zu erlauben.

4.1 Hardware des PLAnalyzer

Ein PLAnalyzer-Modul ist in **Bild 4** dargestellt. Es basiert auf einem JControl-System mit einer 8-Bit-CPU, 60 KByte ROM für die JavaVM, 2,5 KByte RAM und 256 KByte Flash-Speicher für Anwendungsprogramm und Messprotokolle. Zur Interaktion mit dem Benutzer kommen ein hintergrundbeleuchtetes grafisches LC-Display mit 128×64 Pixel, ein in vier Richtungen beweglicher Taster und ein Piezo-Summer zum Einsatz.

Bild 4: Ein PLAnalyzer-Modul

Das gesamte JControl-System findet inklusive eines 230V-Netzteils zur Stromversorgung auf einer $11{,}5 \times 5{,}5\,\text{cm}^2$ großen Platine platz. Zum Senden und Empfangen von Datagrammen über das Stromversorgungsnetz wird ein Power-Line-Modem auf Basis einer 8052-CPU verwendet, welches in Form einer zweiten Platine in das PLAnalyzer-Gehäuse integriert ist. Die Kommunikation zwischen JControl-System und Power-Line-Modem erfolgt seriell über das standardisierte FT1.2-Protokoll. Bei aktivier-

ter Hintergrundbeleuchtung beläuft sich die Stromaufnahme des Gesamtsystems auf ca. 0,3 Watt.

4.2 Software des PLAnalyzer

Ziel der Softwareentwicklung für das PLAnalyzer-Projekt war ein intuitiv bedienbares System, welches die unkomplizierte Konfiguration beliebiger Testszenarien unter Verwendung von zwei bis neun Messmodulen erlaubt. Die Gerätekommunikation findet im Master-Slave-Verfahren statt.

Die Einstellung der verschiedenen Systemparameter wie Master- bzw. Slave-Modus, Netzwerk- und Hausadresse, Geräte-ID, Analyseverfahren, Testintervall etc. sollte über eine grafische Benutzeroberfläche ermöglicht werden, welche für das eingebettete System zu entwickeln und zu implementieren war. Ferner musste für die Power-Line-Analyse selbst eine entsprechende Softwarelösung gefunden werden.

4.3 Implementierung mit JControl

Die Software für den PLAnalyzer wurde komplett in Java implementiert. Dies umfasst sowohl die grafische menügeführte Benutzeroberfläche als auch das Senden und Empfangen der Power-Line-Datagramme einschließlich der Test- und Analysealgorithmen. Der Zugriff auf die Hardwarekomponententen des JControl-Moduls (Display, Taster, Summer, Schnittstellen, ggf. Sensoren und Aktoren) erfolgt komplett über die zur Verfügung stehenden API-Komponenten, es waren für den Programmierer also keine Kenntnisse über die genaue Beschaffenheit der Hardware dieser Komponenten (und deren Ansteuerung) nötig.

Ein weiterer Aspekt wird durch die Verwendung der objektorientierten Hochsprache Java begünstigt: Während der Programmierung der beschriebenen Anwendung sind gewissermaßen nebenbei neue *extended*-API-Klassen zur Ansteuerung der Hintergrundbeleuchtung über eine Puls-Weiten-Modulation sowie zum Stream-basierten Lesen und Schreiben auf den Flash-Speicher entstanden. Diese können nun der bestehenden extended-API-Bibliothek hinzugefügt werden und bei zukünftigen Projekten dienlich sein.

4.4 Ergebnisse des Fallbeispiels

Die Software-Entwicklung für das Power-Line-Messsystem unter Verwendung der JControl-Technologie wurde von einem Programmierer mit Java-, aber nicht JControl-Erfahrung durchgeführt und dauerte einschließlich einer ausführlichen Evaluations- und Testphase vier Wochen. Der entstandene Quellcode belegt in kompilierter Form 43 KByte Flash-Speicher, hinzu kommen weitere 21 KByte für extended-API-Klassen und Ressourcen wie Bilder und Schriftarten. Das System reagiert zügig auf Benutzereingaben und bietet eine komfortable grafische Oberfläche. Durch den Einsatz von *Multithreading* ist die Modifikation von Systemparametern sogar während der Power-Line-Analyse möglich, worunter die Latenzzeit bei der Beantwortung von Netzwerkdatagrammen nicht merklich leidet.

5 Fazit

Java bietet deutliche Vorteile für die Entwicklung eingebetteter Systeme hinsichtlich Code Reuse, Portierbarkeit sowie Evolutionsfähigkeit und damit die besten Voraussetzungen für einen effizienten Softwareentwicklungsprozess. JControl ermöglicht den Einsatz von Java auf besonders kleinen Systemen, wie sie für Aufgaben der Steuer- und Regelungstechnik benötigt werden. Es unterstützt den Entwickler durch eine speicherplatzsparende JavaVM mit einfachen Echtzeitfähigkeiten, einer erweiterbaren Bibliothek für den Hardwarezugriff und verschiedenen Entwicklungswerkzeugen. Dazu gehört auch die Möglichkeit, das Produkt am PC vollständig simulieren zu können, sogar wenn die zugehörige Hardware noch nicht fertiggestellt ist.

Literatur

[1] JControl-Homepage:
`http://www.jcontrol.org`

[2] *Böhme, Helge; Telkamp, Gerrit*; JControl – ein speichereffizienter Java-Ansatz; *Embedded Intelligence 2002*, 19. Februar 2002, Nürnberg

[3] *Böhme, Helge; Telkamp, Gerrit*; Embedded Control mit Java; *Embedded Intelligence 2001*, 14. bis 16. Februar 2001, Nürnberg

[4] *Böhme, Helge; Telkamp, Gerrit*; Eine JavaVM für Anwendungen in der Home Automation; *9. E.I.S.-Workshop*, 22. bis 24. September 1999, Darmstadt

[5] JBed
`http://www.esmertec.com/p.html`,
`http://www.esmertec.com/n_papers.html`

[6] JAVA™ 2 Platform, Micro Edition (J2ME)
`http://java.sun.com/j2me/`

[7] pico Virtual Machine (pVM) for Java
`http://www.charis.com/pvm/`

[8] muvium (PIC Java VM)
`http://www.muvium.com/`

[9] *Lindholm, Tim; Yellin, Frank*; JAVA™ Die Spezifikation der virtuellen Maschine, Die offizielle Dokumentation von JavaSoft; *Addison-Wesley*, 1997
`http://java.sun.com/docs/books/vmspec`

[10] *Hennessy, John L.; Patterson, David A.*; Computer Architecture. A Quantitative Approach; *Morgan Kaufmann Publishers*, 1996; ISBN 1-558-60329-8