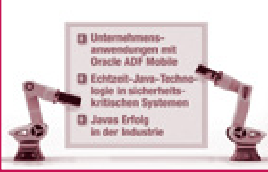

Quantensprung für Embedded-Java

Weshalb Java heute zunehmend Verbreitung in der Industrie findet

Jürgen Heck, Alexandre Leprieult, Markus Bauer

Die Größe und Komplexität eingebetteter Software wächst beständig, oft getrieben von dem Wunsch, die Systeme mit einer anspruchsvollen grafischen Benutzerbedienoberfläche auszustatten. Dies erhöht den Bedarf an verbesserter Produktivität und Qualität bei der Softwareentwicklung. Java ist bekannt dafür, solche Verbesserungen zu ermöglichen. Die Akzeptanz von Java im Embedded-Markt war jedoch in der Vergangenheit durch Bedenken bezüglich der Verwendung knapper Systemressourcen eingeschränkt. Diese wurden in modernen Embedded-Java-Toolkits adressiert. Heute erschließen Java-Anwendungen immer neuere Anwendungsbereiche in der Industrie und die zugrunde liegende Technologie wird zu einem echten wirtschaftlichen Vorteil.

Der folgende Artikel verdeutlicht, welche Anstrengungen notwendig waren, um die virtuelle Java-Maschine (JVM) mit adäquaten Bibliotheken und Konstrukten zu erweitern, damit sie für Embedded-Anwendungen auf ressourcenarmen Systemen erfolgreich eingesetzt werden kann.

Anschließend werden ein agiler Entwicklungsprozess und eine Reihe von Entwicklungswerkzeugen beschrieben, die eine effiziente Umsetzung von Embedded-Projekten ermöglichen. Es werden die Vorteile der Virtualisierung besprochen und aufgezeigt, wie sie anhand von auf Applikationsbedürfnisse zugeschnittenen Java-Plattformen umgesetzt werden kann.

Codegenerierung und Datenaustausch zwischen Anwendung und hardwarenahen Strukturen anhand von Schnittstellen zwischen Java und C werden beleuchtet, bevor der technische Teil mit Erwägungen zu echtzeitfähigen JVMs abschließt. Schließlich wird eine Reihe von mit Embedded-Java entwickelten Produkten nebst den technischen Parametern zum Ressourcenverbrauch und zu den Systemkonfigurationen aufgelistet.

Voraussetzungen für Embedded-Java

Objektorientierte Sprachen haben sich seit Langem als erste Wahl bei der Entwicklung von Software durchgesetzt. Die mächtigeren Ausdrucksmöglichkeiten für die bessere Beherrschung von abstrakten und komplexen Problemen, sowie die Vorteile einer sicheren, qualitativ hochwertigen und wartbaren Software werden nicht mehr infrage gestellt. Wieso hat dann Java bisher nur eine mäßige Verbreitung im Embedded-Bereich gefunden?

Bekannte mit Java entwickelte Embedded-Lösungen sind meist größere Systeme, die auf Linux oder einem RTOS (Real Time Operating System) laufen, mehrere Megabytes RAM-Speicher benötigen und für kleine Produktserien bestimmt sind. Betrachtet man jedoch den Markt der 32-Bit-Mikrokontroller mit geringen Speicher- und CPU-Ressourcen, die oft in Produkten mit großer Stückzahl eingesetzt werden und wo ein Preisunterschied von ein paar Euros für die Elektronik eine Rolle spielt, scheinen Java-Lösungen bisher kaum zu existieren. Weshalb?

Erstens bestand kein wirklicher Bedarf an objektorientierten Lösungen in diesem Marktsegment, da viele dieser Systeme nur eine begrenzte Komplexität beinhalten, die den Einsatz eines objektorientierten Ansatzes nicht unbedingt zu einem großen Vorteil macht. Zweitens standen keine industriell tauglichen Java-Umgebungen zur Verfügung, die die Verbreitung solcher Lösungen ermöglicht hätten.

Die Lage hat sich inzwischen geändert: Kleine, schnelle und flexible Java-Plattformen sind jetzt verfügbar und Killeranwendungen für Embedded-Java sind entstanden, z. B. durch die wachsende Verbreitung von Smartphone-ähnlichen Benutzerschnittstellen. Niemand möchte mehr auf sie verzichten, sowohl zur Bedienung von Haushaltsgeräten (Waschmaschine, Kühlschrank, Brotbackmaschine) als auch von Steuerkonsolen in der Industrie. Andere Beispiele sind Applikationen aus dem Machine-to-Machine-Bereich (M2M) und solche, die eine starke Kommunikationskomponente beinhalten und zum Bereich des „Internet der Dinge“ gehören.

Wie schafft man es aber, eine in Java entwickelte Smartphone-ähnliche Benutzerschnittstelle mit 7“ WVGA-Farbdisplay (800*480*16 Bits) zusätzlich zur üblichen Steuerungstechnik auf ein Board mit Cortex M3 Prozessor (120 MHz, 176 KB interner RAM, 1 MB Flash-Speicher) zu pressen?

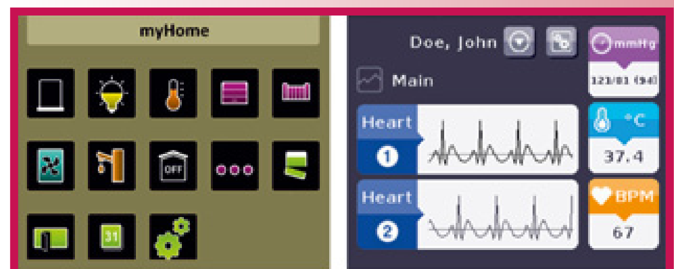

Abb. 1: Beispiele von Benutzerschnittstellen, die heute für die Steuerung von mobilen Geräten eingesetzt werden

Es hat lange gedauert, bevor einige Softwareanbieter in der Lage waren, die Java-Technologie für den Einsatz auf Mikrokontrollern und kleinen Prozessoren zu optimieren. Eine der Lösungen, die aktuell auf dem Markt Verbreitung findet, besteht aus:

- der virtuellen Maschine MicroJvm (abgekürzt MicroJvm VM) mit echtzeitfähigem Garbage Collector basierend auf dem J2ME-Standard in CLDC-Konfiguration [JSR139] (ca. 30 KB);
- optimierten Java-Bibliotheken zur erweiterten Speicherverwaltung (B-ON), zur Verwaltung von Plattformen und Peripherie (PAP), zur Erstellung grafischer 2D-Objekte (MicroUI) und für Widgets (MWT), einer Native-Schnittstelle (SNI) (insgesamt ca. 60 KB) [ESR];
- einem kompakten, echtzeitfähigen Betriebssystem, welches entweder direkt auf der Zielhardware (baremetal) oder neben einem existierenden RTOS, Linux, Android oder iOS läuft (ca. 15 KB) [IS2T].

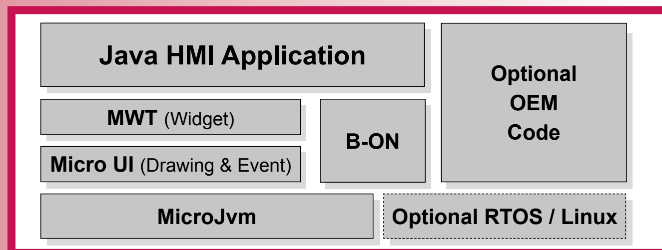

Abb. 2: Layer-Modell einer Embedded-Java-Applikation mit Laufzeitumgebung

Somit liegt der Speicherbedarf für z. B. eine mittelgroße Embedded-Java-Anwendung mit QVGA-Bildschirm unter 1 MB, inklusive aller Komponenten. Das Ziel der Hersteller, die Hardware so passgenau auf das Produkt zu bemessen wie möglich, um Kosten zu sparen, kann erreicht werden.

Neben dem begrenzten Speicherumfang und der Ausführungsgeschwindigkeit ist bei vielen Embedded-Applikationen die Startzeit der Gesamtanwendung ein KO-Kriterium: Sobald der Stromschalter gedrückt ist, muss die Applikation vorliegen und das Touch-Panel auf Kontakt reagieren. Bei auf MicroJvm VM ausgeführten Applikationen liegt die Startzeit bei 30 Millisekunden.

Entwicklungsumgebung und -prozess

Neben den technischen Voraussetzungen müssen dem Entwickler von Embedded-Anwendungen zusätzliche Instrumente in die Hand gegeben werden, die ihn während eines Entwicklungsprozesses unterstützen. Dabei sollte es ihm ermöglicht werden, die verschiedenen Entwicklungsschritte aus einer Arbeitsumgebung heraus durchführen zu können. Diese Umgebung sollte neben grafischen Editoren für Benutzerschnittstellen auch Unterstützungsfunktionen zur Messung des zukünftigen RAM/Flash-Speicherverbrauchs, zur Überprüfung der Qualität des Codes (Code-Abdeckung, automatisierte Testroutinen), zur Konfiguration verschiedener Zielplattformen, Debug-Tools zum Scheduling und Eventing der Threads und für das Java Heap Monitoring beinhalten.

Vom Entwicklungsprozess her hat sich eine agile Herangehensweise mit einem TDD-Ansatz (test-driven development) bewährt: Basierend auf einer Beschreibung von Testfällen werden eingeschränkte Prototypen erstellt, die die gestellte Problematik erfüllen müssen, aber bereits Funktionsblöcke der zukünftigen Anwendung darstellen. Zwischenstände der Anwendung können mit der Simulation oder – falls bereits verfügbar – frühzeitig auf der Zielhardware getestet werden. Die von der Hardware diktierten Einschränkungen können so frühzeitig festgestellt und im weiteren Projektverlauf berücksichtigt werden (z. B. Abstimmung der grafischen Effekte auf die maximale Anzeigegeschwindigkeit des LCDs).

Bei den verschiedenen Entwicklungsschritten werden Produkt-Owner, Designer/Ergonomen, Hardware- und Softwareentwickler gleichermaßen einbezogen, um eine höchstmögliche Endkundenakzeptanz zu gewährleisten und Produkteinführungsrisiken zu mindern.

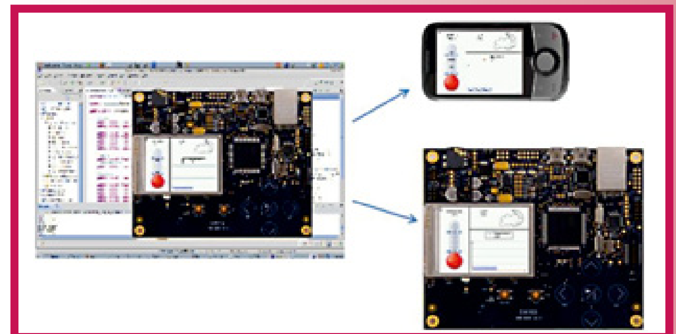

Abb. 3: Die in der Entwicklungsumgebung simulierte Anwendung kann auch für unterschiedlichste Zielhardware generiert werden, eine entsprechende JVM-Plattform auf diesen Geräten vorausgesetzt

Virtualisierung

Experten sind sich einig, dass die Möglichkeit der Abstrahierung der Hardware-Ebene heute der zukunftsweisende Weg in der Embedded-Welt ist, die Unternehmen ganz neue Möglichkeiten eröffnet [Helmerich05]. Ein um die Hardware gelegter Wrapper macht sie flexibel, verformbar und befähigt sie, ganz neue Dinge zu tun [SmNa05].

Java bringt die Möglichkeit der Virtualisierung von Haus aus durch die virtuelle Maschine mit. Im hart umkämpften Embedded-Bereich der Mikrokontroller ist es unbedingt notwendig, den Prozess der Virtualisierung so weit zu industrialisieren, dass die Erstellung von auf die spezifischen Eigenheiten elektronischer Boards angepassten Java-Plattformen ökonomisch und schnell ist. Wenn bisher die Entwicklung einer JVM Jahre gedauert hat, kann diese heute von einer Person innerhalb von wenigen Monaten für einen neuen Prozessortyp nebst Peripherie erstellt werden, vorausgesetzt es wird auf eine generische Plattform aufgesetzt und es steht ein Toolkit bereit, auf das sich der Entwickler stützen kann.

Für den oben beschriebenen agilen Entwicklungsprozess braucht der Entwickler zwei Plattformen: eine für die auf dem Desktop-PC ausgeführte Simulation (SimJPF) und eine für die Ziel-Hardware (EmbJPF). Ändert sich eine Hardwarekomponente auf dem Zielsystem (z. B. Austausch des LCD-Kontrollers, Hinzufügen von Ein-/Ausgabeeinheiten), müssen meist beide Plattformen angepasst werden.

Physische Aktoren und Ereignisse können für die SimJPF entweder mit Hilfe von Soft-Mocks (z. B. die Modellierung eines Temperaturfühlers über einen programmierten Wertegenerator) dargestellt werden oder über einen über den Bildschirm des Desktop-PCs zu betätigenden grafischen Schieberegler. Wenn die Benutzerschnittstelle und die Anwendungslogik vom Kunden auf der simulierten Version abgenommen wor-

den sind, kann der gleiche Bytecode in den Flash-Speicher der Ziel-Hardware übertragen werden und läuft dort – eventuell abgesehen von einigen Feineinstellungen an den Timings – genauso wie auf dem Desktop-PC. Somit wird der Hardware-Entwicklungszyklus von dem der Software entkoppelt und Produktentwicklungen beschleunigen sich je nach Komplexität der Anwendung um mehrere Wochen oder Monate.

Es wird noch besser: Durch die Virtualisierung können auch die Lebenszyklen der Hard- und Software entkoppelt werden. Entwickelt sich die Hardware schneller als die Software, muss eine Applikation nicht mühselig für eine neue Ziel-Hardware neu entwickelt werden, sondern kann auf die dafür neu erstellte Java-Plattform übertragen werden. Dies ist hilfreich für Systeme mit langer Laufzeit (Züge, Flugzeuge, Kraftwerkssteuerung). Ganze Abteilungen europäischer Verteidigungsministerien beschäftigen sich intensiv mit diesem Thema: der Aufrechterhaltung der Betriebsfunktion von sicherheitsrelevanten Anlagen und Systemen [StanAg05].

Codegenerierung für die Zielplattform und Ausführgeschwindigkeit

Wie wird eine hohe Ausführgeschwindigkeit auf leistungsarmen 32 Bit-Mikroprozessoren erreicht? Der erste Reflex wäre, auf eine JIT-Kompilierung von rechenzeitintensiven Code-Fragmenten zurückzugreifen. Die durch eine JIT-Kompilierung eingefahrene Beschleunigung wird jedoch auf solchen Mikroprozessoren umgehend durch die dafür notwendige Rechenzeit aufgehoben und man hat nichts gewonnen. Im Gegenteil; während der Prozessor an der JIT-Kompilierung arbeitet, ist er für die eigentliche Aufgabe der Steuerung eines Systems nicht verfügbar.

Somit bleibt in solchen Umgebungen nur die ahead-of-time (AOT) Kompilierung übrig, gebündelt mit auf den Embedded-Bereich zugeschnittenen Programmiertechniken, von denen es eine ganze Menge gibt. Zum Beispiel bei einer Schleife mit Austrittsbedingung zieht man das Herabzählen einer Variablen bis auf null dem Test auf einen bestimmten Wert vor, da die Abfrage `X == 0` von Mikrokontrollern innerhalb eines Zyklus über einen bedingten Sprung an eine Zieladresse durchgeführt wird (`Jump if 0`), im Gegensatz zu der Instruktion `Jump if not equal`, für die mehrere Prozessorzyklen benötigt werden. Auch die Vermeidung von Konstrukten wie (`i=0;i<Table.length;i++`) ist angeraten, da die Länge der Tabelle bei jeder Iteration neu errechnet werden muss.

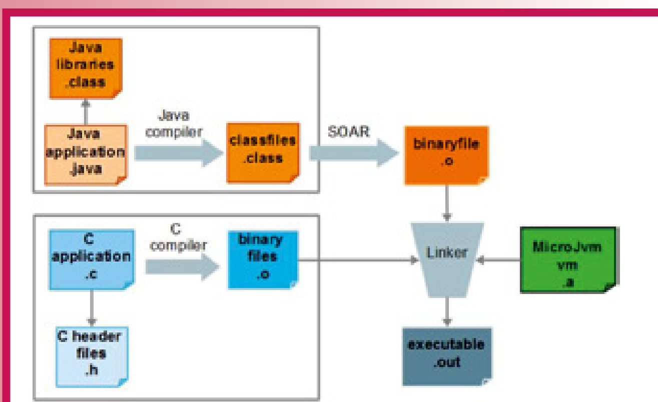

Abb. 4: Kompilierungsprozess mit der Eclipse-basierten MicroEJ-Eclipse-Workbench: Java-Bytecode wird über SOAR optimiert und mit C-Codeteilen gelinkt, um schließlich als binäre Datei auf das Ziel-Board übertragen zu werden

In unserem Beispiel der MicroJvm VM wird der Java-Bytecode auf dem Desktop-PC in binärem Maschinencode übersetzt und gleichzeitig durch den Smart-Linker SOAR von IS2T (Industrial Smart Software Technology) optimiert. Kritische Codeteile werden erkannt und zu Optimierungszwecken gesondert behandelt.

Moderne Mikrokontroller wie die ARM Cortex M-Reihe sind mit internen Matrix-Bussen ausgestattet, die sich über Software konfigurieren lassen. Eine Embedded-Anwendung kann während der Initialisierungsphase innerhalb des Mikrokontrollers Einstellungen vornehmen, die heute die Ablaufgeschwindigkeit von Kommunikationskomponenten (SRAM, SPI, Ethernet) bis zu verzehnfacht. Hier wird noch einmal klar, wie wichtig eine auf die Hardware abgestimmte, virtuelle Plattform (EmbJPF) sein kann.

C als Mittler zwischen Hochsprache und Hardware

Jede Programmiersprache hat ihre spezifischen Stärken und Schwächen. C ist in dem hier beschriebenen Vorgehensmodell am besten für das Schreiben von maschinennahen Codeteilen geeignet. Strukturelle Sprachen wie C sind wie dafür geschaffen, die Hardware von Embedded-Systemen, die aus einer Ansammlung von strukturierten Elementen besteht (Register, Speicherblöcke), abzubilden. Auch zur Optimierung von zeitkritischen Codeteilen wird C gerne hergenommen, ähnlich wie früher Assembler in Verbindung mit C.

Eine robuste und leicht verständliche Schnittstelle zwischen Java und C ist daher von elementarer Wichtigkeit. Die standardmäßig mit Java kommende JNI-Lösung (Java Native Interface) hat sich für den Embedded-Bereich als zu schwerfällig erwiesen. Alternativen dazu sind die SNI-Bibliothek und das ShieldedPlug-Konzept.

Das Simple Native Interface (SNI), das die gleichen Namenskonventionen wie das JNI verwendet, ist einfacher im Gebrauch, weniger fehleranfällig und nützt die Systemressourcen der Zielhardware, wie z. B. die DAM-Zugriffe, aus. Als Parameter werden nur Basistypen und Tabellen mit solchen übergeben, da es hier hauptsächlich darum geht, Datenblöcke zwischen der Anwendungswelt (Java) und der Hardware-Ebene (C) auszutauschen. Diese Datenblöcke definieren gemeinsame Speicherzonen, auf die von beiden Welten heraus zugegriffen werden kann; ressourcenaufwendige Kopiervorgänge werden vermieden.

Am klassischen HelloWorld-Beispiel in Listing 1 wird der Aufruf einer C-Funktion mit Parameterübergabe aus Java heraus illustriert.

Java-Code:

```
package examples;
public class Hello {
  public static void main(String[] args) {
    print HelloNbTimes(args.length);
  }
  public static native void printHelloNbTimes(inttimes);
}
```

C-Code:

```
#include <sni.h>
#include <stdio.h>
void Java_examples_Hello_printHelloNbTimes(jint times) {
  while (--times){
    printf("Hello world!\n") ;
  }
}
```

Listing 1: Beispiel für den Aufruf einer C-Funktion mittels SNI

Eine andere effiziente Technik, um zwischen einer Java-Anwendung und C-Programmen sauber zu kommunizieren, ist die Datenübergabe über den asynchronen Publish-subscribe-Mechanismus innerhalb eines ShieldedPlug-Konstrukts. Dieser ist innerhalb der sogenannten GreenThread-Ausprägung der MicroJvm VM verfügbar, die eine eindeutige Trennung zwischen der Java-Anwendung und der Legacy-C-Anwendung macht: Die gesamte Java-Anwendung läuft hier in einem einzigen RTOS-Thread, die restliche aus C-, Ada-Code usw. bestehende Anwendung läuft in anderen Threads. Die GreenThread-Ausprägung wird gerne in Projekten verwendet, wo zu einer breiten Basis von existierendem Legacy-Code eine Java-Komponente hinzugefügt werden soll, die eine spezifische Problemstellung besser als übliche Mittel löst (z. B. ein Touch-Panel). Zusätzlich ist sie einfacher einzusetzen als eine vollständig in Java geschriebene Embedded-Lösung.

Wie viel Echtzeitfähigkeit braucht man wirklich?

Viele industrielle Anwendungen kommen mit den Standardmitteln einer Java-Lösung vom Typ der MicroJvm VM aus. Da solche Umgebungen nebst dazugehörigem Software-Erstellungsprozess auf kleine optimierte Systeme zugeschnitten sind, müssen meist nur stellenweise Optimierungen stattfinden, damit das Zielsystem die funktionskritischen Programmteile in jedem Fall schnell genug und fehlerfrei durchläuft.

Obwohl dies die Ausnahme ist, kommt es immer wieder vor, dass harte Echtzeitfähigkeit (im Weiteren mit HRT abgekürzt) verlangt wird. Bei Java konzentriert sich der Aufwand zur Befriedigung der HRT-Anforderungen vor allem darauf, einen deterministischen Garbage Collector (GC) zu entwickeln. Um dies zu erreichen, tut man gut daran, erst einmal innerhalb der Java-Plattform bzw. in den systemnahen Bibliotheken (wie im B-ON) entsprechende Vorkehrungen zu treffen, um die notwendigen Voraussetzungen für ein effizientes und schnelles, auf die Topologie der Mikrokontroller zugeschnittenes Speichermanagement zu schaffen. Folgende Arten von Speicherobjekten sind in dem Zusammenhang sinnvoll:

- Persistente *immutable* Objekte (unveränderliche) sind read-only, werden also zum Startzeitpunkt der Applikation alloziert und können nicht mehr geändert werden. Sie dienen dazu, eine Embedded-Anwendung sofort nach dem Systemstart arbeitsfähig zu machen. Zudem enthalten sie Daten mit vordefinierten Werten (Konstante), die sich während der Laufzeit nicht ändern und am Besten in einem kostengünstigen Flash-Speicher abgelegt werden. Diese Objekte interessieren den GC nicht.
- *Immortal* Objekte (unsterbliche) verbleiben, einmal alloziert, immer an der gleichen Stelle im Speicher und auch sie brauchen vom GC nicht berücksichtig zu werden. Es handelt sich hier vom Speicheraufkommen her hauptsächlich um von den Treibern verwendete Pufferzonen.
- Und schließlich die *reclaimable* Objekte (einforderbare), der geläufigste Datentypus und Hauptfokuspunkt des GC, so wie es in Java standardmäßig vorgesehen ist.

Dieser Ansatz der Speichertypentrennung ist naheliegend, da es nicht sinnvoll ist, den GC über den Speicherbedarf von Objekten wachen zu lassen, die sowieso durchgehend benötigt werden oder in Bereichen liegen, die er nicht erreichen kann (z. B. Flash-Speicher). Andere Stellen der Optimierung liegen bei der Dimensionierung der Stack-Größen, die sich in der MicroJvm VM automatisch an die Anwendungsbedürfnisse anpassen; deren Speicher wird erst beim Beenden der Threads eingesammelt.

Der GC der MicroJvm VM basiert unter andrem auf dem in [Schoe06] beschriebenen Ansatz eines inhärent beschränkten Systems. Er ist in eine zyklische Task mit niedrigerer Priorität als die der anderen Tasks eingebunden. Der GC ist unabhängig von der Heap-Größe und der Anzahl der allozierten Objekte, abhängig nur von der Größe des Arbeitsspeichers („live memory"). Und da letzterer begrenzt ist, ist die Maximallaufzeit (WCET) des GC auch begrenzt und somit deterministisch.

Die Europäische Weltraumorganisation (ESA) interessiert sich seit Jahren für JVMs und hat mehrere davon auf Weltraumfahrt-Tauglichkeit überprüft. Erst kürzlich wurde die auf der MicroJvm VM basierende MicroJVM4Space-Version in Hinsicht auf Speicherverwaltungsmodelle, GC-Echtzeitfähigkeit, I/O-Handling und HRT näher untersucht und die auf einem Leon2-Prozessor getestete GC-Verwaltungsstrategie als für schlüssig befunden [RiPrPa11].

Die MicroJvm VM wird seit 2011 unter Aufsicht des französischen Verteidigungsministeriums einem den Normen ED-12C und DO-178C konformem Zertifizierungsprozess unterzogen, der im Jahr 2013 in der ersten für die Luft- und

Ressource/SW	Brotbackmaschine	Industriedrucker	Energiewandler	U-Boot-Steuerung
Prozessor	STM32 Cortex M0 66 MHz	STM32F2 Cortex M3 120 MHz	AVR32 UC3 66 MHz	AVR32 UC3 66 MHz/ PowerPC/Intel Pentium
RAM intern	48 KB	128+48 KB	64 KB	64 KB/1 MB/1 GB
SRAM extern	256 KB	2 MB	-	
Touch Display	S/W 320*240*1 Bit, Frame-Buffer 256 KB	Farbe, 800*480*16 Bit	-	-
Flash-Speicher	128 KB	1 MB	512 KB	512 KB/1 MB/250 GB HDD
JVM + OS	38 KB baremetal	42 KB baremetal	38 KB baremetal	baremetal IceOS/Linux/Linux
Nativ Treiber	2 KB	12 KB	4 KB + 12 KB Webserver	vertrauliche Daten
Anwendung	8 KB	72 KB	10 KB	

Tabelle 1: Tabelle der Ressourcen und des Speicherverbrauchs

Raumfahrt sowie für militärische Einsätze zugelassenen JVM münden soll.

Beispiele industrieller Anwendungen

Um die eingangs erwähnte Behauptung zu Javas Verbreitung in der Industrie zu belegen, seien hier vier von vielen weiteren, in den letzten Jahren erfolgreich eingeführten Produkten und Systemen angeführt:

- Bedienung und Steuerung einer Brotbackmaschine mit Schwarz-weiss-LCD-Display (ressourcenarm, hohe Stückzahl). Ansprechende grafische Benutzerführung mit kleinstmöglichen Hardware-Kosten.
- Bedienung eines Industriedruckers zur Beschriftung von Verpackungen auf Transportbändern mit 7" Farbdisplay und Touch-Panel (ressourcenarm, mittlere Stückzahl); hier lag die Herausforderung darin, mit der vorhandenen begrenzten Prozessorleistung das hohe Aufkommen der Bilddaten zu bewältigen und gleichzeitig Daten über eine serielle Schnittstelle in 3 MBit Geschwindigkeit zu verarbeiten.
- Steuerung eines Energiewandlers für Schienenfahrzeuge mit Webserver zur Nachverfolgung der Energieverteilung (sicherheitskritisch, extrem ressourcenarm, mittlere Stückzahl).
- Steuerung der Tauchneigung eines atomgetriebenen U-Boots (extrem sicherheitskritisch, kleine Stückzahl), finanziert durch die DGA/DSR4T. Hier ging es darum, ein dreifaches System mit Entscheidungsfindung im Mehrheitsverhältnisverfahren umzusetzen. Die Steuerungsanwendung ist das gleiche Java-Programm, das auf drei verschiedenen Rechner- und Betriebssystemen läuft.

Tabelle 1 beschreibt die dafür eingesetzte Hardware und die verfügbaren Ressourcen sowie den benötigten Speicherbedarf der Anwendungen.

Die Diversität der Anwendungsgebiete und die Einhaltung des stark begrenzten Bill-of-Materials zeigen, dass dem Einsatz von Embedded-Java kein e wirklichen Grenzen gesetzt sind.

Fazit und Aussicht

Die früher bestehenden technischen Einschränkungen für den Einsatz von Java im Embedded-Bereich wurden aufgehoben. Die Technologie, um solche Anwendungen industriell und ökonomisch vertretbar einzusetzen, ist heute auf dem Markt verfügbar und wurde bereits für die Entwicklung vieler Serienprodukte eingesetzt. Eine noch zu erfüllende Voraussetzung ist, dass Universitäten und Hochschulen Embedded-Java und dazugehörige Themen stärker in den Ausbildungsprogrammen aufnehmen, damit vermehrt Ingenieure mit dem notwendigen Wissen und Können auf dem Markt verfügbar sind, welches benötigt wird, um die in der Entwicklung von zukunftsweisenden Produkten steckenden Herausforderungen meistern zu können.

Die Entwicklung auf dem Embedded-Markt geht in Richtung Ubiquität intelligenter M2M-Systeme (wie z. B. dem Smart Home und Smart Metering) in Verbindung mit dem Cloud Computing für das Datenmanagement. Die Virtualisierung erlaubt die massive Verbreitung von Diensten mittels Rahmenwerken wie OSGi für das sichere Laden und Konfigurieren von Services in leistungsstarken wie auch ressourcenarmen Geräten. Damit wäre Java dann endlich dort angekommen, wofür es vor 20 Jahren konzipiert wurde.

Literatur und Links

[ESR] ESR-SPE-001-B-ON-1.2.C, ESR-SPE-002-MicroUI-1.4.F, ESR-SPE-011-MWT-1.0.C, ESR-SPE-0012-SNI-1.1.F, ESR-Consortium, www.e-s-r.net

[Helmerich05] FAST GmbH/TUM, Study of Worldwide Trends and R&D Programmes in Embedded Systems in View of Maximising the Impact of a Technology Platform in the Area, Final Report, 2005, ftp://ftp.cordis.europa.eu/pub/ist/docs/embedded/final-study-181105_en.pdf

[IS2T] IS2T-SPE-005-IceOS-1.0.I, IS2T Internal

[JSR139] Connected Limited Device Configuration 1.1, http://jcp.org/en/jsr/detail?id=139

[RiPrPa11] F. Rivard, M. Prochazka, T. Pareaud, Java for On-Board Software, in: Proceedings of the DASIA 2011 Conference, Mai 2011

[Schoe06] M. Schoeberl, Real-Time Garbage Collection for Java, in: Proceedings of the 9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing, ISORC 2006, Gyeongju, Korea, April 2006

[SmNa05] J. Smith, R. Nair, Virtual Machines: Versatile Platforms for Systems and Processes, Morgan Kaufmann, 2005

[StanAg05] NSA, Standardization Agreement for Obsolescence Management, STANAG 4597, http://www.document-center.com/standards/show/STANAG-4597

Jürgen Heck nahm Ende der achtziger Jahre in einer Start-up-Softwareschmiede teil an der Entwicklung einer objektorientierten Programmierumgebung für die Automatisierungsindustrie basierend auf Smalltalk80 bevor er bei Siemens an Java-Vorfeldthemen arbeitete. Anschließend war er lange als Projektmanager für internationale Großprojekte für namhafte Firmen tätig. Heute ist er bei IS2T verantwortlich für Vertrieb und Projektmanagement in Zentraleuropa.
E-Mail: jurgen.heck@is2t.com

Alexandre Leprieult hat 20 Jahre Erfahrung in den Bereichen der Elektronik und Software und war u. a. Software-Entwicklungsleiter für die Steuerzentrale des atomgetriebenen Unterseeboots „Le Terrible" und der Antriebssteuerung des Unterseeboots „Barracuda". Heute ist er Spezialist für Java-Plattformen in der Firma IS2T für die Bereiche Echtzeit, HMI und Embedded-Systeme

Markus Bauer ist seit 2010 Embedded-Softwareentwickler bei der Firma M-Sys GmbH. Dort ist er an der Entwicklung verschiedenster Produkte in den Bereichen Automotive, Gebäudeautomatisierung und Industriesteuerungen beteiligt. Dieses Jahr setzte er zum ersten Mal das MicroEJ-Framework in einem Embedded-Java-Projekt ein.
E-Mail: markus.bauer@msys-gmb