Lehrstuhl Systemarchitektur Universität Karlsruhe

Integration von Bluetooth in die AmbiComp Plattform für eingebettete Systeme

Malte Cornils
im Rahmen einer Studienarbeit
betreut von
Prof. Bellosa
und
Dipl.-Inform. Johannes Eickhold
Dipl.-Ing. Björn Saballus

Cornils, Malte
Rüppurrer Straße 7
76137 Karlsruhe
malte@cornils.net
Studiengang: Diplom-Informatik

Version: 1.0

Danksagung

Danken möchte ich vor allem meinen Betreuern Johannes Eickhold und Björn Saballus, die ständig ein offenes Ohr für meine Fragen und Probleme hatten – und das zu jeder möglichen und unmöglichen Tages- und Nachtzeit. Dank gebührt auch meinem fleißigen Korrekturleser Philipp Glaser.

Karlsruhe, im Februar 2008

Malte Cornils

König Harald [Blauzahn] befahl diesen Stein zu errichten, zum Gedenken an Gorm, seinen Vater, und an Thyra, seine Mutter. Der Harald, der (dem) sich ganz Dänemark und Norwegen unterwarf und die Dänen zu Christen machte.
Inschrift auf den Runensteinen von Jelling, über den Namensgeber des Bluetooth-Standards

Hiermit erkläre ich, die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Literaturhilfsmittel verwendet zu haben.

Karlsruhe, den 20. Februar 2008

Malte Cornils

Inhaltsverzeichnis

Kapitel 1

Einführung

1.1 Übersicht

Den großen Siegeszug hat die drahtlose Funktechnik schon hinter sich. International sind im privaten wie akademischen Bereich Funknetze im Einsatz. Die Vernetzung des persönlichen Umfelds geschieht zumeist mit drahtlosen Netzen nach IEEE 802.11-Standard (WLAN), geht es um die Überwindung von Distanzen zwischen Räumen, und meist mittels Bluetooth, wenn es um die Vermaschung von am Körper getragenen Geräten oder Distanzen innerhalb eines Raumes geht.

Bluetooth und seine Nachfolgetechniken wie Zigbee und Ultra Wide Band-Funk zeichnen sich durch ihre Energieeffizienz aus. Ihr Stromverbrauch liegt noch deutlich unter dem von WLAN benötigten Werten. Deshalb ist laut [Mat05, S. 50] Bluetooth eine Schlüsseltechnologie im kommenden Internet der Dinge, das Haushaltsgeräte und andere Alltagsgegenstände miteinander vernetzt. Zur Kommunikation der Geräte untereinander wird ein zumeist selbst konfigurierendes und dynamisch vermaschtes Netz aufgebaut, in dem die verschiedenen Kommunikationspartner von Stift bis Kochlöffel Informationen austauschen.

Doch die inhärenten Beschränkungen [LDB03] der Bluetooth-Technik und der eingebetteten Systeme selbst stellen die Informatik vor schwierige Aufgaben. Skalierbarkeit, Sicherheit und Sparsamkeit sind nur einige Schlagwörter, die den Entwurf von entsprechenden Systemen begleiten.

Java als *lingua franca*

Als universelle Sprache zur Programmierung von Geräten unterschiedlichster Architekturen bietet sich Sun Java als Bindeglied zwischen den per se inkompatiblen Plattformen an. Java ist nicht an eine bestimmte Hardware gebunden. Obwohl die Sprache nicht von unabhängigen Organisationen standardisiert wurde, entsteht durch Suns offene Lizenzbedingungen doch ausreichend Vertrauen darin, dass Java als zukunftssichere Wahl als Systemgrenzen überschreitende Brückensprache gelten darf.

Unglücklicherweise sind die auf dem Markt verfügbaren Lösungen nicht für eingebettete Systeme und kleinste Geräte, denen nicht die Ressourcen eines Smartphones zur Verfügung

stehen, geeignet. Selbst für ressourcenarme Geräte entworfene Java-Implementierungen nach der Java 2 Mobile Edition-Spezifikation sind allenfalls für Mobiltelefone verfügbar, denen ein deutlich höheres Maß an Prozessorgeschwindigkeit und Speicherplatz zur Verfügung steht, als für die zu vernetzenden Mikrokontroller im Internet der Dinge. Das AmbiComp-Projekt will dieses Problem lösen, indem eine Java-Laufzeitumgebung (Virtuelle Maschine) auch für solche Kleinstprozessoren erstellt wird.

Eine Brücke zur Bluetooth-Hardware

Die Ansteuerung der Bluetooth-Hardware aber ist an sich bereits aufwändig. Nur mit hardwarenaher und an begrenzte Speicher- und Prozessorressourcen angepasster Kommunikations- und Steuerungssoftware, wie die MicroBlue-Umgebung der Firma BeeCon [Gmb04], lassen sich die in den Mikrokontroller-Systemen eingebauten Bluetooth-Chips ansprechen.

Es ist Aufgabe der vorliegenden Arbeit, die Brücke zwischen dieser systemnahen Treiberumgebung und den bekannten Programmierschnittstellen (APIs) der Java-Programmiersprache zu schlagen. Dabei genügt es keineswegs, eine dünne Übersetzungsschicht zu schreiben, die die Befehlsaufrufe aus der einen Sprache in für den Treiber verständlichen Code umsetzt. Vielmehr müssen die unterschiedlichen Konzepte des hardwarenahen MicroBlue-Kommunikationsstapels (Stack), im Folgenden kurz MicroBlue, und der datenstrombasierten Schnittstelle auf der Java-Seite berücksichtigt und soweit möglich die generischen und abstrakten Java-Kommunikationsbefehle transparent und effizient auf die maschinennahe Ebene von MicroBlue umgesetzt werden.

1.2 Ziele

Die zu entwickelnde Architektur für die Implementierung einer Java-kompatiblen Programmierschnittstelle soll verschiedene Entwurfsziele berücksichtigen, die für einen erfolgreichen Einsatz in der Praxis von großer Bedeutung sind.

Die *Portabilität* auf verschiedene Plattformen ist von großer Bedeutung. Im schnelllebigen Bereich der Informationstechnik ist eine strike Festlegung auf einen bestimmten Befehlssatz, eine Entwicklungsumgebung oder einen Gerätehersteller wirtschaftlich meist unverantwortlich. Diese Portabilität bleibt keineswegs auf die zwei verschiedenen Hardware-Plattformen beschränkt, die zur Realisierung zur Verfügung standen. Ein Austausch des zugrundeliegenden Betriebssystems, der Wechsel des Mikrokontrollers oder des Bluetooth-Chips sollte möglich sein, ohne größere Anpassungen erforderlich zu machen.

Die *Berücksichtigung von Standards* ist nicht nur für die einfache Anbindung des Netzes hin zur Benutzerin oder Steuerungsrechnern entscheidend. Auch die einzelnen Knoten eines zusammengehörigen Sensor-/Aktornetzes können auf völlig unterschiedlichen Hardware-Systemen basieren, die zur Kommunikation eine einheitliche und offene Schnittstelle benötigen. Auch der Austausch von Teilkomponenten, beispielsweise des Bluetooth-Chips, wird durch die Berücksichtigung internationaler und unabhängiger Standardisie-

rung vereinfacht. Für die Java-Bluetooth-API sollte eine weitestgehende Kompatibilität mit der Java-Spezifikation JSR-82 [Mic05] erreicht werden. Ein wichtiger Teilaspekt ist auch die Vollständigkeit der Implementierung - die verwendeten Schnittstellen sind teilweise nur selektiv implementierbar, ohne Abstriche bei Codegröße zu machen. Würde die volle Bandbreite der in JSR-82 vorgesehenen Zugriffsmöglichkeiten und Optionen implementiert, müsste fast das gesamte MicroBlue eingebunden werden statt ausgewählter, kleiner Bibliotheken. Die Resourcenknappheit der eingebetteten Systeme lässt eine solche Einbindung nicht zu (siehe Abschnitt 6.4).

Robustheit stellt ein weitere Anforderung in selbst organisierenden und selbst konfigurierenden Systemen dar. Auf Fehler einzelner Sensornetzknoten muss weitgehend ohne menschlichen Eingriff reagiert werden können, damit die Verfügbarkeit trotz Verwendung von 'instabilen' Techniken wie drahtloser Funkkommunikation in hohem Maße gewährleistet werden kann. Es ist mithin nicht möglich, in jedem Fehlerfall den Quelltext zu untersuchen, die Software neu zu kompilieren und zu verteilen. Statt dessen sollten häufige Fehler zur Laufzeit erkannt, abgefangen und entweder beseitigt werden oder der schadhafte Knoten umgangen werden.

Die *Effizienz* lässt sich in einige Unteraspekte aufspalten. So ist mit den eingeschränkten Ressourcen auf einem Atmel-Mikrokontroller vorsichtig umzugehen, ein überfüllter Speicher wie beim Code Blue-Projekt (wie in [SCL+05], siehe auch Abbildung 1.1) ist eine häufige Folge von unbedachtem Software-Entwurf. Auch die Performanz des laufenden Systems wird entscheidend durch die sparsame Verwendung von CPU-Zyklen mitbestimmt. Ein weiterer Teilaspekt ist die Energieeffizienz, die bei den häufig batteriebetriebenen Netzknoten nicht zu vernachlässigen ist. Fällt ein Knoten häufig aufgrund des Aufbrauchens der vorhandenen Energiereserven aus, so sind aufwändige Korrekturmaßnahmen zu treffen. Ein Entwurf, der von vornherein auf Sparsamkeit achtet, umgeht dieses Problem weitgehend.

Ziel der Arbeit ist der Entwurf und die für eine Validierung der getroffenen Entwurfsentscheidungen ausreichende Implementierung einer Integrationsschicht, die den oben genannten Entwurfszielen so weit wie möglich entspricht.

1.3 Gliederung

Der weitere Aufbau der Arbeit stellt sich wie folgt dar. Das *zweite Kapitel* stellt kurz verwandte Arbeiten vor, auf denen aufbauend oder von denen abweichend die vorliegende Implementierung konstruiert wurde. Der Stand der Technik wird allgemein anhand ähnlicher, exemplarischer API-Implementierungen dargestellt.

Im *dritten Kapitel* widmet sich der Autor der Analyse der bei der erfolgreichen Lösung der Implementierungsaufgabe auftretenden Probleme. Die beiden zu vermittelnden, definierten Schnittstellen, MicroBlue auf der einen Seite, Javas Bluetooth-Programmierschnittstelle auf der anderen Seite werden auf Ähnlichkeiten und Unterschiede abgeklopft, die eine Überbrückung erleichten oder erschweren.

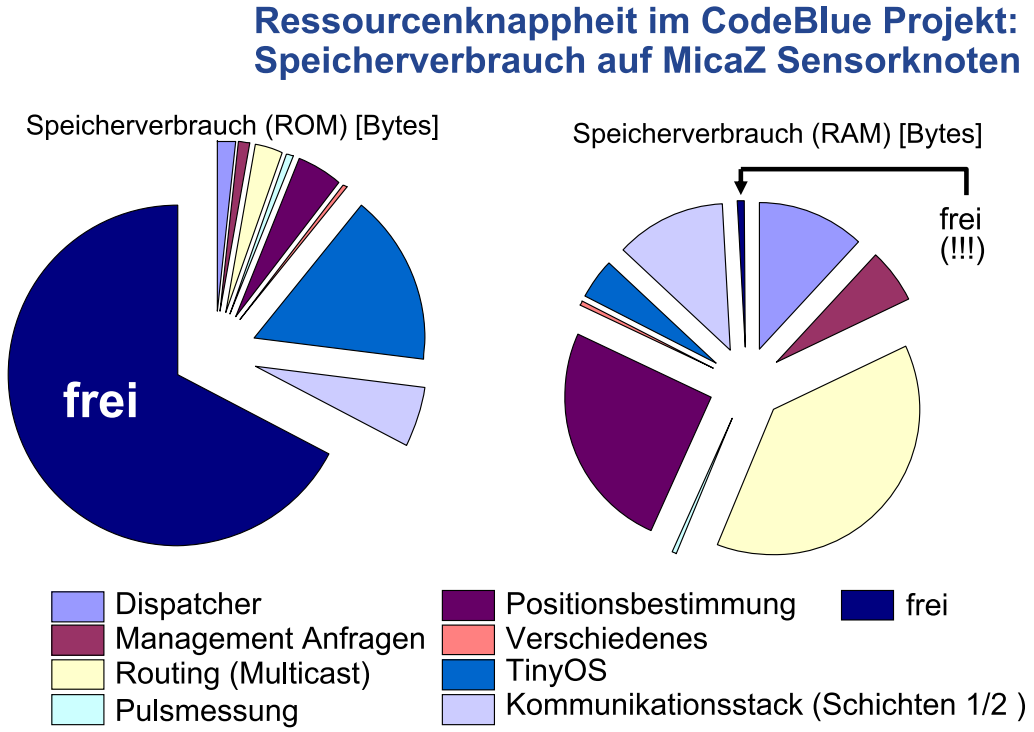

Abbildung 1.1: Typischer Speicherbedarf, aus [ZBHK07] basierend auf [SCL+05]

Demzufolge kann in *Kapitel Vier* mit dem Entwurf fortgefahren werden. Die bei der Analyse ans Licht getretenen Probleme werden betrachtet und die theoretisch beste Lösung wird erörtert. Bereits hier sollte ein Überblick über die später zu verwendende Struktur vermittelt werden. Auch die technischen Rahmenbedingungen für den Entwurf werden beleuchtet.

Im *fünften Kapitel* wird schließlich die entwickelte Implementierung von Teilen von JSR-82 aufbauend auf MicroBlue beschrieben. Die technischen Einzelheiten der gewählten Ausführung werden begründet dargestellt.

Das *sechste Kapitel* stellt die entwickelte Implementierung auf den Prüfstand, um die Einhaltung der anfangs herausgestellten Entwurfsziele zu testen. Nur auf diesem Weg lässt sich sicherstellen, dass die gestellten Voraussetzungen erfüllt wurden.

Das Fazit im *siebten Kapitel* bildet neben einer Zusammenfassung der geleisteten Arbeit und einem Resumee über das Ergebnis auch einen Ausblick, wie die exemplarische Implementierung für den täglichen Einsatz verbessert werden kann und welche Änderungen an MicroBlue wie auch an JSR-82 nötig wären, um für eine optimierte Zusammenarbeit zwischen Hardware-Plattform und der Anwendungsprogrammiererin zu sorgen.

Kapitel 2

Stand der Technik

2.1 Ähnliche Implementierungen

Für andere Systeme existieren durchaus bereits Anbindungen von Java an Bluetooth-Hardware. Inbesondere die Beschränkungen der Codegröße sorgen zwar dafür, dass die in der vorliegenden Arbeit vorgestellte Implementierung sich deutlich von den bekannten Versuchen unterscheidet, aber dennoch lassen sich einige Ansätze und Konzepte gut vergleichen.

Insbesondere die freie phoneME-Implementierung [Mic J] implementiert die auch von AmbiComp bereitzustellende CLDC-API unter Verwendung von nur wenigen nativen Methoden. So konnten viele Klassenimplementierungen sogar ohne Modifikation des Quellcodes dank der GPL-Lizensierung [Sta07] beider Komponenten übernommen werden. Die VM ist allerdings auf deutlich höhere Systemanforderungen ausgelegt: 16-bit Prozessor, 16 MHz, 160 kiB nichtflüchtiger Speicher, 192 kiB RAM [Mic94].

Einen kommerziellen Konkurrenten stellt die Firma Avetana zur Verfügung [Gmb J], unter GNU/Linux auch eine GPL-Version [Sou J]. Allerdings ist diese Implementierung auf Desktop-Systeme ausgelegt und damit um einige Größenordnungen zu groß.

Allgemeine, nicht auf Java basierende Kommunikationslösungen zu Bluetooth und Sensor-/Aktornetzen finden sich in der wissenschaftlichen Literatur [LDB03].

Als Beispiel für eine auf Java aufsetzende Kommunikationsinfrastruktur für das 'Internet der Dinge' sei auf JINI [Mic01, FM05, Kapitel 2] verwiesen.

Ähnlichkeiten mit der vorliegenden Implementierung haben auch die Arbeiten von Björn Saballus zur Anbindung von Ethernet über MicroBlue an die ACVM. Insbesondere die technische Realisierung der Anbindung der Kommunikationsfunktionen an die AmbiComp VM ist entsprechend aufgebaut.

Kapitel 3

Analyse

In diesem Kapitel werden die MicroBlue-Architektur auf der einen und die JSR-82 auf der anderen Seite auf Ähnlichkeiten und konzeptionelle Unterschiede untersucht. Ein tiefergehendes Verständnis des Aufbaus der einzelnen korrespondierenden Schichten (z.B. SPP auf MicroBlue-Seite und dessen Verwendung in JSR-82) erleichtert den Entwurf der zwischen MicroBlue und Java vermittelnden Implementierung.

3.1 Bluetooth

In dieser Arbeit werden die grundlegende Basistechniken von Bluetooth nur erklärt, insoweit es zum Verständnis der Arbeit erforderlich ist. Für einen detaillierteren Überblick, gerade auch für das theoretische Verständnis und die exakten technischen Parameter der Bluetooth-Funktechnik waren die veröffentlichten Standards der Bluetooth Special Interest Group (SIG) von herausragender Bedeutung. Das eigentliche Referenzbuch [SIG04] wird ergänzt durch einen groben Überblick, den Mettala [Met99] liefert. Für einige konkret verwendete Bestandteile des Bluetooth Protokollstapels standem dem Autor Spezialwerke zur Verfügung [SIG01b, SIG03, SIG01a].

3.2 MicroBlue

Die vorliegende Struktur des MicroBlue-Systems, einer kommerziellen Bluetooth-Implementierung für eingebettete Systeme, ist in der folgenden Abbildung 3.1 gut zu überblicken.

Der verwendete Bluetooth-'Treiber', eine von der Firma BeeCon entwickelte Implementierung eines Bluetooth-Protokollstapels, wird in [Har04] gründlich dokumentiert. Ein grober Überblick kann durch das Whitepaper [Gmb04] gewonnen werden.

Die einzelnen Bestandteile und Protokolle finden sich unter [Har04, Seite 7f] ausreichend beschrieben. Hier sei nur wiedergegeben, dass es sich bei HCI (Host Controller Interface) um eine Schicht handelt, die die Software-Protokolle und Datentypen, mit denen die oberen Schichten mit der UART-Schicht (siehe Abschnitt 4.7.2) kommunizieren, zur Verfügung stellt.

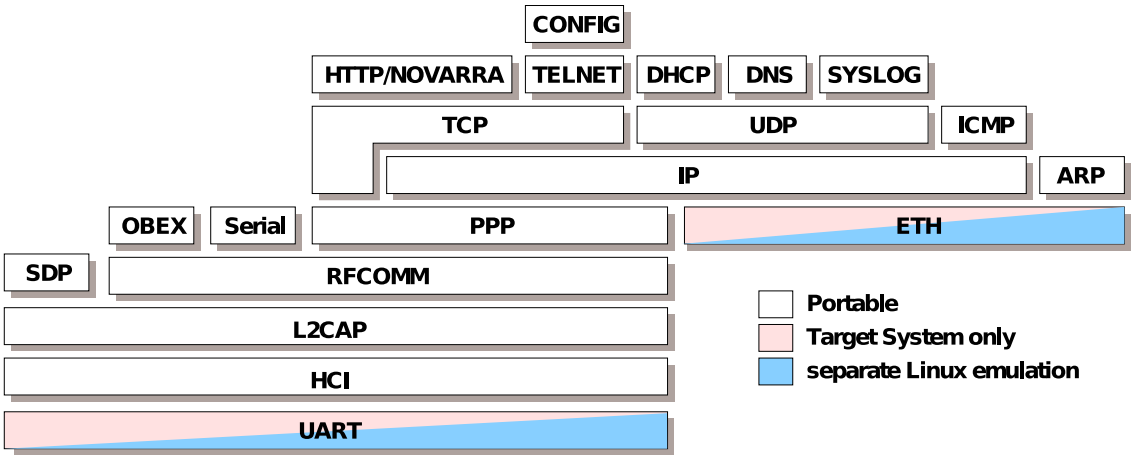

Abbildung 3.1: Aufbau von MicroBlue, aus [Har04]

Die einzelnen Protokollschichten werden auf der x86-Plattform von der Kompilierungsumgebung ('Buildsystem') in einzelne Objektdateien übersetzt, die dann zu einem lauffähigen Programm zusammengebunden werden. Dabei liegt keine klassische Bibliotheksstruktur vor. Die nötige Flexibilität lässt sich durch Auswahl von den zu verwendenden Objektdateien sicherstellen. Beispielsweise sind für ein kleines Testprogramm die in `serial.o` vorliegenden Funktionen nicht erforderlich. Die entsprechende Objektdatei wird also nicht mitgebunden.

Auch eine Abhängigkeit von externen Dateien ist bei Benutzung der MicroBlue-Funktionalität gegeben, da bei Initialisierung der MicroBlue-Funktionen die Existenz eines bestimmten Speicherabbildes in zwei Dateien vorausgesetzt wird (`eeprom.img` bzw. `eeprom.base`), damit die Handhabung von Zugriffen auf diese Speicherbereiche unter Linux ähnlich wie auf anderen Plattformen realisiert werden kann.

Für einen korrekten Zugriff auf dieses Speicherabbild auf der x86-Plattform werden während des Bindeprozesses Informationen über die Adresse gesammelt, an die das Speicherabbild gebunden wird. Diese Informationen werden im weiteren Übersetzungsprozess benötigt. Dies ist zu beachten, wenn eine Umformung in einer dynamisch gebundene Bibliothek in Erwägung gezogen wird: Die absolute Adresse, an der das Speicherabbild gebunden wird, steht erst zur Bindezeit, in diesem Fall also nach Abschluss des eigentlichen Kompiliervorgangs, fest.

Intern nimmt MicroBlue seine eigene Interrupt-Behandlung vor. Diese lässt sich also nicht von aussen beeinflussen. Deshalb sind einige Energiesparmöglichkeiten nicht verwendbar, die sich beim interruptgesteuerten Warten auf Ereignisse nutzen liessen, um die VM ebenfalls schlafen zu lassen. Vielmehr besteht die Ereignisschleife (*event loop*) MicroBlues aus einer eigenen Funktion `hci_process`. Diese wird von aussen immer wieder aufgerufen, damit gewartet werden kann und eintreffende Ereignisse abgearbeitet werden können.

3.3 JSR-82

Eine Erklärung von Java im Allgemeinen wird in dieser Arbeit nicht vorgenommen. Als Lehrbuch sei hier auf Schader [SST98] verwiesen.

Für ein genaues Verständnis der Struktur von JSR-82 ist es erforderlich, die verwendeten Java-Konzepte zu kennen. Zum Datenaustausch werden Datenströme verwendet, die über Adaptionen der `java.io`-Klassen zugreifbar sind. Während die eigentliche Datenkommunikation sich bis in die Ebene einzelner `read`- und `write`-Methodenaufrufe zurückverfolgen lässt, erlaubt die datenstrombasierte Schnittstelle die komfortable Verwaltung auch mehrerer Verbindungen.

Zur Etablierung und zum Abbau von konkreten Verbindungen sieht JSR-82 die Nutzung des Generic Connection Framework (GCF) vor, dass über `open`-Methoden und entsprechend parametrisierten Uniform Resource Locators (URLs, beschrieben in [BLMM94]) den Verbindungsaufbau unabhängig vom unterliegenden Netzwerk- und Kommunikationsmedium erledigt. Eine Umstellung von seriellen Bluetooth-Verbindungen zu Ethernetbasierten Sockets liesse sich also nur durch Änderung der URL bewerkstelligen.

Im Übrigen definiert JSR-82 eigene Klassen für Bluetooth-spezifische Funktionalität, wie der automatischen Gerätesuche (*device inquiry*) [Mic06b, Klasse `javax.bluetooth.DiscoveryAgent` etc.]. Als Umgebung für JSR-82-Programme wird zumindest ein Connected Limited Device Configuration (CLDC)-konformes Java-API vorausgesetzt. Dieses enthält bereits die erwähnten GCF- und `java.io`-Klassen. Java-APIs, die hingegen der Java 2 Standard Edition entsprechen, lassen die GCF-Klassen vermissen.

Intern arbeitet JSR-82 über verschiedene Java *interfaces*, die eine Implementierung bereit stellen muss. Eine wichtige Rolle spielen dabei Notifikator-Klassen, die asynchron eintreffende Ereignisse verwalten und zur Verarbeitung weiterleiten. Mehrfädigkeit (Multithreading) stellt ein wichtiges Element von JSR-82 dar. Im Zusammenspiel mit Notifikatoren kann so auf bestimmte Ereignisse im Hintergrund gewartet werden und auf unvorhergesehene Eingaben reagiert werden, ohne den eigentlichen Kontrollfluss des Hauptfadens abbrechen zu müssen.

Für die zur Implementierung von JSR-82 nötigen detaillierten Angaben zu technischen Details der verwendeten Java-Komponenten sei hier im Speziellen auf diejenigen Referenzwerke und Spezifikationen verwiesen, die sich im engeren Sinne mit der Java-Version für mobile Geräte, der Java 2 Mobile Edition (J2ME) und den zugehörigen API-Programmierschnittstellen befassen. Hier ist insbesondere die J2ME CLDC-Spezifikation [Mic00a] und API-Dokumentation [Mic06a] zu nennen. Für die Nutzung des Sun Java Wireless Toolkit, einer Referenzimplementierung der wichtigsten J2ME-Spezifikationen für x86-Prozessoren, war auch die Kenntnis des MID Profils [Mic00b, Mic06c] vonnöten. Die konkrete, als optionaler Bestandteil der Java 2 Mobile Edition gekennzeichnete Bluetooth-Schnittstelle ist als JSR-82 standardisiert [Mic05, Mic06b].

Kapitel 4

Entwurf

4.1 Minimale API für eine *proof of concept*-Implementierung

Um in der Kürze der für eine Studienarbeit zur Verfügung stehenden Zeit eine zielführende und die gestellten Aufgaben lösende Implementierung anzufertigen, war es zuerst nötig, die Teile der Java-Programmierschnittstelle zu identifizieren, die für einen Konzeptbeweis (*proof of concept*) des hier vorgestellten Entwurfs nötig sind.

Anhand verschiedener Beispielprogramme (siehe Seiten 31ff) wurden alle für diese Programme nötigen Klassen von JSR-82 festgestellt. Die nötigen Klassen gliedern sich in mehrere Teile. Für Grundfunktionen sowie die Datenstrom-Klassen wurden folgende Klassen der Java 2 Plattform verwendet, abgesehen von einigen Basisklassen wie `java.lang.Object`:

```
java.lang.NullPointerException
java.lang.IndexOutOfBoundsException
java.lang.String
java.io.IOException
java.io.InputStream
java.io.OutputStream
java.io.DataInputStream
java.io.DataOutputStream
```

Weiterhin werden die Klassen benötigt, die aus dem Generic Connection Framework (GCF) für den Aufbau und die Verwaltung von Verbindungen zwischen Endgeräten verwendet werden. Diese Klassen entstammen dem `javax.microedition.io`-Paket.

JSR-82 definiert nur wenige Klassen, die für eine *proof of concept*-Implementierung nötig sind: zumeist sind ausschließlich Funktionen, die über das CLDC und das GCF hinausgehen, in speziellen JSR-82-Klassen definiert. Beispielsweise sind das *pairing* und die Gerätesuche (*device discovery*) über eigene Klassen der `javax.bluetooth`-Hierarchie steuerbar. Diese Funktionalität ist aber für eine rudimentäre *proof of concept*-Implementierung nicht erforderlich. Die minimal nötigen Klassen sind Folgende:

```
javax.bluetooth.ServiceRegistrationException
```

```
javax.microedition.io.Connector
javax.microedition.io.Connection
javax.microedition.io.StreamConnection
javax.microedition.io.InputConnection
javax.microedition.io.OutputConnection
javax.microedition.io.StreamConnectionNotifier
```

4.2 Entwicklungsplattform

Eine flexible und nicht an eine bestimmte Virtuelle Maschine (VM) gebundene Implementierung der Java-Klassen entspricht den genannten Entwurfszielen der Portabilität und der Berücksichtigung von Standards. Um die Erreichung dieser Entwurfsziele sicherzustellen, wurden die zu entwickelnden *proof of concept*-Testprogramme unter verschiedenen Entwicklungsumgebungen und Plattformen entwickelt, implementiert und getestet. Die eigentliche JSR-82-Implementierung wurde ebenso parallel unter verschiedenen Plattformen vorangebracht.

Ein weiterer Grund war die anfangs noch mangelnde Vollständigkeit der AmbiComp Virtuellen Maschine (ACVM). Auf diese Art und Weise war es möglich, Unstimmigkeiten und Fragestellungen der Software-Architektur unter den vollständigeren Plattformen bereits zu lösen, während die Probleme der ACVM zeitlich nebenläufig behoben werden konnten.

Ist im Folgenden von einer Legacy Virtuellen Maschine (Legacy VM) die Rede, ist eine der von Sun implementierten VMs gemeint.

Die verschiedenen getesteten Architektur-Kombinationen, die das folgende Diagramm zeigt, werden in den folgenden Abschnitten genauer erklärt.

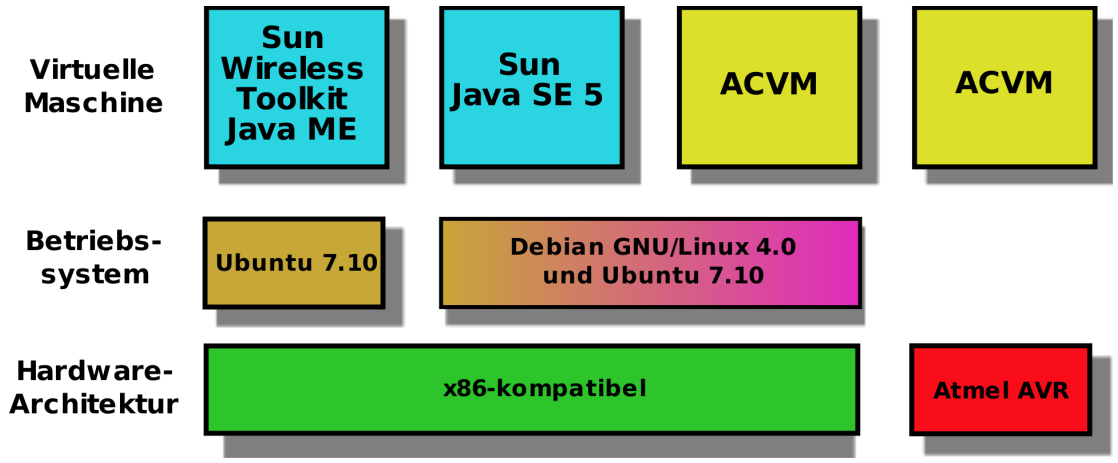

Abbildung 4.1: Architektur-Kombinationen

4.2.1 Sun Java Wireless Toolkit

Auf den ersten Blick erschien der Sun Java Wireless Toolkit (WTK) als J2ME-Implementierung für Arbeitsplatz-Betriebssysteme wie eine ideale Testplattform für die zu entwickelnden *proof of concept*-Testprogramme. Der WTK unterstützt von Haus aus die vollständige CLDC-API, das GCF sowie angebunden an plattformspezifische Treiber eine JSR-82-Schnittstelle neben einer Bluetooth-Emulation.

Der Lebenszyklus der J2ME-Anwendungen unterscheidet sich allerdings deutlich von dem klassischer Java-Programme. Die strikte Verwendung von grafischen Dialogen und Rückmeldungen, die auf Mobiltelefon-Anzeigen ausgelegt sind, macht die entstehenden Programme ungeeignet für eine spätere Verwendung auf Systemen ohne Ausgabemedium und Benutzerinnen-Interaktion, wie etwa eingebettete Systeme in Sensor-/Aktor-Netzen.

Im übrigen konnte die Implementierung der eigenen JSR-82-Anbindung an MicroBlue nicht über das WTK vorgenommen werden. Die anzusteuernden nativen Methoden aus MicroBlue und die eigene Implementierung muss dem WTK zur Verfügung gestellt werden. Das für solche Zwecke geeignete Java Native Interface (JNI) [Lia99] existiert im WTK jedoch nicht.

4.2.2 Java 2 Standard Edition

Das Vorhandensein von JNI und auch der klassischere Lebenszyklus der zu entwickelnden Programme waren bei einer anderen Programmierumgebung zu finden: die Java 2 Standard Edition (J2SE). Auch wenn die Basisklassen im Wesentlichen kompatibel zu den im CLDC spezifizierten Klassen sind, fehlt doch eine Implementierung des GCF vollständig. Zusätzlich zur Entwicklung der eigentlichen JSR-82-Anbindung an MicroBlue musste hier noch eine Implementierung dieser Klassen vorgenommen werden.

Da die einzige Alternative, die direkte Verwendung der ACVM, aus ähnlichen Gründen scheitern musste, weil auch dieser keine GCF-Implementierung zur Verfügung stand, war die Entwicklung der rudimentären GCF-Klassen zur Verwendung unter der J2SE wie auch später unter der ACVM eine effiziente Nutzung der Zeit.

Auch wenn das erklärte Ziel der ACVM die Kompatibilität mit der Connected Device Configuration (CDC), einer Obermenge des CLDC, ist, so unterscheiden sich die API-Strukturen des CLDC wenig von denen der J2SE-API. Eine Überprüfung auf Portabilität zwischen VMs war also auch mithilfe einer J2SE-Implementierung möglich, die um die wesentlichen Teile des GCF ergänzt worden ist.

4.2.3 AmbiComp Virtuelle Maschine

Die ACVM operiert nicht auf `.class`-Dateien. Statt dessen wird ein Programm namens Transcoder eingesetzt, um von Legacy-Plattformen erzeugte `.class`-Dateien in sogenannte Blob-Dateien umzuwandeln. Diese enthalten in kompakter Form nur noch die für die Ausführung in der ACVM erforderlichen Informationen.

Als Entwicklungsplattform war die ACVM anfänglich wenig nutzbar, da wichtige Basisfunktionalität fehlte. Hier sei exemplarisch die Verwendung von Zeichenketten, Zeichenfelder, die Übergabe von bestimmten Datentypen und der Mangel an Mehrfachvererbung über Schnittstellen genannt. Durch die Implementierung anhand der vollständigeren J2SE waren die benötigten und noch fehlenden Funktionen der ACVM leichter zu identifizieren. Die endgültige Portierung auf die ACVM sollte dann trotz frühem Entwicklungsstadium nicht an den zur Verfügung gestellten Funktionen der ACVM scheitern.

Die Anbindung von nativen Funktionen geschieht anders als bei den VMs mit JNI-Unterstützung über den sogennante NativeAPI-Mechanismus (siehe Abschnitt 5.3).

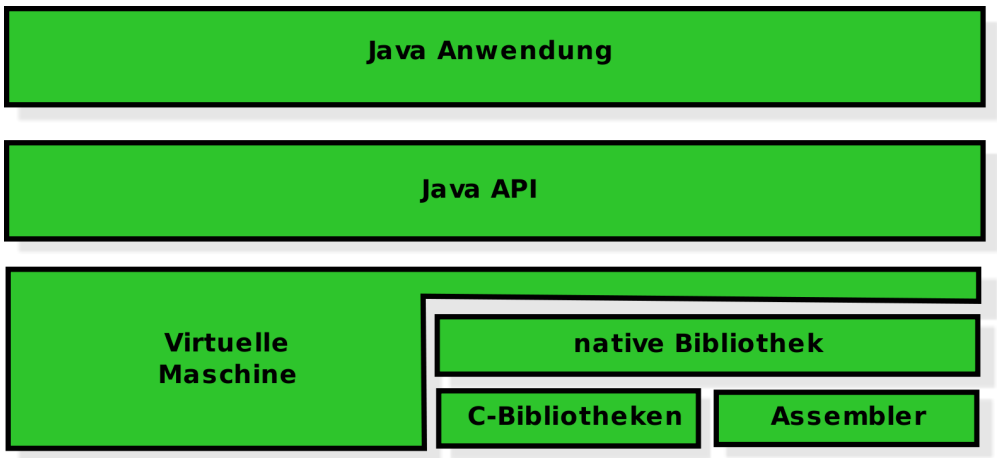

Abbildung 4.2: Aufbau einer Java-VM

4.2.3.1 Dummy-API

Als Zwischenweg zwischen Nutzung der ACVM und der Nutzung einer Legacy VM wie der J2SE gab es die Möglichkeit, die AmbiComp API-Implementierung durch eine Legacy VM ausführen zu lassen. Diese Einbindung der ACVM-API in die VM heißt 'Dummy API'. Einen kurzen Überblick über die verschiedenen Bestandteile von VMs gibt Abbildung 4.2.

4.3 Zielplattform

Als Zielplattform wird hauptsächlich GNU/Linux auf x86-Basis verwendet. GNU/Linux hat sich als wirtschaftliche Plattform für den Kommunikations- und Informationstechnik-Sektor erwiesen, wie in [G+06] geschildert. Bei IBM-kompatiblen x86-PCs handelt es sich um eine von MicroBlue unterstützte Hardware-Plattform. Um das Ziel der größtmöglichen Portabilität zu beachten, wird bei der Entwicklung Wert darauf gelegt, plattformspezifischen Code möglichst zu vermeiden.

Zweite Zielplattform sind die AVR Atmel-Chips, die in den Systemen der Firma BeeCon verwendet werden. Diese 8-Bit-Prozessoren haben enge Grenzen, was Ausführungsgeschwindigkeit und Speicherplatz angeht. Das macht sie zu einer herausfordernden Zielplattform: lässt sich die zu erstellende Implementierung auch auf diesen ressourcenbe-

grenzten Systemen verwenden, dürften sich auch 'mächtigere' eingebettete Systeme leicht einbinden lassen. Auch für diese Systemplattform existiert eine MicroBlue-Portierung. Die verwendeten GNU-Werkzeuge inklusive vollständiger Toolchain (siehe Abschnitt 5.2) stehen für die Atmel-Chips ebenfalls zur Verfügung.

4.4 Art des Bindens

Für beide Zielplattformen wurde eine Software-Bibliothek mit C-kompatiblem Aufrufstandard gewählt, um JSR-82 die MicroBlue-Funktionen zur Verfügung zu stellen. Die Art der Bibliothek variiert je nach verwendeter Entwicklungsplattform. J2SE sieht hier die Einbindung von javaexternen Bibliotheken über das Java Native Interface (JNI) vor, dass unter GNU/Linux die Verwendung eines ELF *dynamic shared objects* (`.so`-Datei) [Com95] als Bibliotheksformat verlangt [Lia99, Seite 16].

Für die Verwendung mit der ACVM wird eine statisch gebundene Bibliothek (`.a`-Datei) verwendet. Statische Bindung hat die Eigenschaft, dass die Speicheradresse, an die die Bibliotheksfunktionen gebunden wurden, zur Kompilierzeit feststeht. MicroBlue setzt diese Eigenschaft voraus und muss deshalb bei dynamischer Bindung die in Abschnitt 3.2 genannten Hilfsmittel zur Laufzeit verwenden.

4.5 Netzwerkschicht

Für die Kommunikationsschicht stellt JSR-82 zwei Optionen zur Verfügung: die L2CAP-Kommunikation und die Kommunikation mittels RFCOMM/SPP. Logical Link Control and Adaption Protocol (L2CAP) [SIG04, Volume 1, Seite 24] steuert die Verbindung zwischen zwei Endgeräten und kann zwischen mehreren Diensten auf demselben Gerät unterscheiden. L2CAP spezifiziert die *service discovery*-Dienstekennungen nicht. Für eine korrekte automatische Dienstaushandlung muss ein eigener Dienst definiert und auf MicroBlue-Ebene bekannt gemacht werden. Das ist nicht zur Laufzeit möglich, da Dienstespezifikationsdateien mit dem `sdptool`-Werkzeug erstellt werden müssen [Har04, Seite 99]. Eine Implementierung, die das in der vorliegenden Arbeit gelegte Fundament nutzt, um ohne *service discovery* arbeitende L2CAP-basierende Kommunikation anzubieten, dürfte aber nur einen geringen programmiertechnischen Aufwand bedeuten.

Die Alternative zu L2CAP sind RFCOMM-Verbindungen ([SIG03]) nach dem Serial Port Protocol (SPP) ([SIG01b]). Dabei definiert RFCOMM (Radio Frequency Communication) die eigentliche Protokollschicht, während das SPP nur eine konkrete Instanz des RFCOMM-Protokolls darstellt, dass einige Parameter vordefiniert. So ist bei SPP der für RFCOMM zu wählende Kommunikationskanal festgelegt.

RFCOMM/SPP emulieren eine serielle Verbindung zwischen Endgeräten. Für die Zwecke dieser Arbeit wurde aufgrund der vorgegebenen Dienstekennung und dem festgelegten Kommunikationskanal RFCOMM nach SPP-Vorgaben implementiert. Im Folgenden wird mit SPP die spezielle SPP-Vorgabe des RFCOMM-Protokolls gemeint, wenn nicht explizit anders angegeben.

4.6 Lizensierung

Als Lizenz wird für alle selbstgeschriebene Komponenten die durch die Open Source Initiative (`http://www.opensource.org/`) zertifizierte modifizierte BSD-Lizenz verwendet. Für Anpassungen an den Standard-APIs der ACVM ist klarzustellen, dass die Bearbeitung von existierendem GPL-Code [Sta07] einzustufende Modifikationen – zusätzlich – unter dieser Lizenz veröffentlicht werden. Code, der unter der modifizierten BSD-Lizenz steht, kann auch als Teil von GPL-Code veröffentlicht werden oder andernfalls als Teil von proprietärem Code genutzt werden. Diese Möglichkeit wurde für alle vom Autor geschriebenen Code-Fragmente vorgesehen, da Änderungen und Erweiterungen sowohl von proprietärem als auch von GPL-Code nötig waren.

Die folgende Tabelle 4.1 gibt die in dieser Arbeit modifizierten und erweiterten Software-Komponenten an und nennt die jeweilige Lizenz. Zu den Software-Komponenten sollte auch Abbildung 4.2 herangezogen werden.

Tabelle 4.1: Software-Komponenten und ihre Lizenz

Nr.	Software-Komponente	Lizenz	Lizenz der eigenen Erweiterungen
1	MicroBlue	Proprietär	mod. BSD (wahlweise GPL v2+)
2	libhal-bt.so/.a	mod. BSD	mod. BSD (wahlweise GPL v2+)
3	ACVM-NativeAPI	Proprietär	mod. BSD (wahlweise GPL v2+)
4	ACVM	Proprietär	mod. BSD (wahlweise GPL v2+)
5	Transcoder	Proprietär	mod. BSD (wahlweise GPL v2+)
6	ACVM-API	GPL v2	mod. BSD (wahlweise GPL v2+)
7	ACVM-Beispielprogramme	Proprietär	mod. BSD (wahlweise GPL v2+)
8	ACVM-Dummy API	Proprietär	mod. BSD (wahlweise GPL v2+)
9	J2SE-Beispielprogramme	GPL v2	mod. BSD (wahlweise GPL v2+)

Eine Lizensierung unter der modifizierten BSD erlaubt die Verwendung der nicht mit GPL-lizensiertem Code zusammenhängenden Programme mit proprietären Systemen, wie beispielsweise die Anbindung an den MicroBlue.

4.7 Software-Architektur

4.7.1 Gleichzeitiges Senden und Empfangen

Auf unterster Ebene verwendet Bluetooth Time Division Duplex: in ungeraden Zeitslots kann der Master senden, der Slave nur in geraden Zeitslots. Es ist auf Ebene des Funkkanal-Protokolls kein gleichzeitiges Senden und Empfangen vorgesehen. Diese Fähigkeit wird durch die ACL-Schicht (Asynchronous Connectionless Link) des HCI gewährleistet. Die auf Java-API-Ebene angesiedelten Protokolle L2CAP und SPP sind bereits vollduplexfähig.

4.7.2 Puffer

Die Kommunikation der MicroBlue-HCI-Schicht mit der eigentlichen Bluetooth-Hardware basiert auf einem seriellen Puffer für das Senden und Empfangen von Daten: einem so-

genannten *Universal Asynchronous Receiver Transmitter* (UART). Der UART hat einen Eingangspuffer der Größe 64 Bytes auf Atmel-Plattformen; auf der Linux-Plattform ist er vom BlueZ-Stack abhängig [Har04, Seiten 31, 69]. Weitere Puffer sind zur Verwendung der Bluetooth-Protokollschichten wie zum Beispiel L2CAP normalerweise nicht nötig, ausser bei Protokollen, die diese zwingend voraussetzen (beispielsweise PPP und SPP).

Es existiert ein HCI-Befehlspuffer [Har04, Seite 70], der es zulässt, dass genau ein Bluetooth-Befehl gleichzeitig verarbeitet werden kann. Ist dieser abgearbeitet, wird ein Ereignis generiert, dass dem Stack mitteilt, dass ein weiterer Befehl gesendet werden kann.

Die Verwendung von HCI-Datenpuffern ist optional. Wird anwendungsseitig sichergestellt, dass Bluetooth-Geräte nur dann Befehle senden, wenn der Empfänger dies signalisiert hat, werden HCI-Datenpuffer nicht benötigt und können in MicroBlue deaktiviert werden.

Auch auf SPP-Ebene existiert in MicroBlue immer ein Puffer, der auf Atmel-Hardware üblicherweise 16 Bytes groß ist. Dies ist ein interner Puffer, der während der Vorverarbeitung verwendet wird und nicht direkt durch MicroBlue-C-Funktionen manipuliert werden kann. Wird Echoing wie in der hier vorliegenden JSR-82-Implementierung nicht verwendet, wird keinerlei Vorverarbeitung betrieben.

4.7.3 Verbindungsstruktur

Etabliert eine der Software-Schichten von MicroBlue eine Verbindung zu einer Gegenstelle, so wird ein entsprechendes Ereignis erzeugt (wie z.B. `SPP_EVT_NOTIFY_CONNECTION`). Auf den verschiedenen Schichten werden bei Verbindungsaufbau-Ereignissen Datenstrukturen angelegt, die mit Informationen über die etablierte Verbindung gefüllt werden. Diese müssen bei Datentransfers angegeben werden. Für eine korrekte Handhabung von mehreren gleichzeitigen Verbindung zwischen Geräten oder auf der Ebene einer multiplexfähigen Schicht (wie z.B. RFCOMM/SPP) muss ausreichend Speicher für die zu allozierenden Datenstrukturen zur Verfügung stehen.

4.7.4 Flusskontrolle: `rfcomm_give_credit`

Da MicroBlue, insbesondere auf der Atmel-Hardware, nicht in schneller Folge eintreffende Bluetooth-Pakete verarbeiten kann, ist eine Flusskontrollsteuerung in MicroBlue eingebaut. Diese basiert auf Krediten, die an die jeweilige Gegenseite vergeben werden, die dann durch das Senden eines Datenpakets einen solchen Kredit aufbraucht. Stehen keine Kredite mehr zur Verfügung, werden auch solange keine weiteren Pakete mehr gesendet, bis die Gegenstelle nach abgeschlossener Verarbeitung wieder neue Kredite vergibt.

Zum Schutz vor Überläufen wird eine Anwendung, die bei Erhalt von beispielsweise RFCOMM-Daten unmittelbar wieder RFCOMM-Daten als Antwort verschicken will, Kredite erst dann wieder mithilfe der Funktion `rfcomm_give_credit` vergeben, wenn die eigene Verarbeitung abgeschlossen ist. Diese Vorgehensweise muss bei Anwendungsprotokollen, die auf andere Weise den Pufferüberläufe verhindern, nicht eingehalten werden.

Bei einer Anwendung beispielsweise, in der alle 10 Sekunden ein Messdatum an eine Gegenstelle verschickt wird, ist sichergestellt, dass keine Pufferüberläufe statt finden.

4.7.5 ACVM versus Legacy VM

4.7.5.1 Threads

Da es sich bei MicroBlue um ein System mit einer Ereignisschleife handelt, muss auf irgendeine Art und Weise sicher gestellt werden, dass die Ereignisschleife regelmäßig aufgerufen wird, wenn Bluetooth-Ereignisse vom nativen Stack bearbeitet werden müssen. JSR-82 sieht solche expliziten Aufrufe einer Ereignisschleife aber nicht vor.

Die Ereignis-Schleife muss aufgerufen werden, um eine von der Hardware empfangene Nachricht zu akzeptieren und weiterzuverarbeiten. Dies muss ausreichend regelmäßig geschehen, um Timeouts bei der Bluetooth-Kommunikation, die durch die standardkonformen Zeitabstände definiert sind, zu vermeiden. Beispielsweise existiert ein Timeout beim Verbindungsaufbau, der zwischen 0,625 s und 29 s und normalerweise bei 5 s liegt [SIG04, Volume 3, Seite 385].

Von der Java-Anwendung initiierte Kommunikation bedarf nicht des Aufrufs der Ereignisschleife. Die zugrunde liegenden MicroBlue-Funktionen wie `spp_send` erledigen alle nötigen Kommunikationsaufgaben selbst.

Regelmäßiges Polling, also ein Prüfen, ob Ereignisse vorliegen, kann durch Aufruf der Ereignisschleife immer dann durchgeführt werden, wenn auf eine Kommunikation explizit gewartet wird. Aber auch dann, wenn in der Anwendung keine andere Funktion abgearbeitet wird, sollte die Ereignisschleife aufgerufen werden.

Wird eine strikte Beschränkung auf JSR-82 gefordert, bleibt als einfachste Lösung die Verwendung von mehreren Programmfäden (*threads*). Ob diese auf mehreren Prozessoren zur gleichen Zeit ausgeführt werden oder durch zeitscheibenbasiertes Multitasking nebenläufig erscheinen, ist hierbei nicht von grundsätzlicher Bedeutung. Threadsicherheit muss bei beiden Varianten gegeben sein. Wechselseitiger Ausschluß beim Zugriff auf kritische Abschnitte/gemeinsam genutzte Ressourcen ist eine Möglichkeit, dies sicherzustellen. Diese Lösung ist mit der ACVM nicht möglich, da Threads zu Beginn der Arbeit nicht von der ACVM unterstützt wurden. Die Ereignisschleife MicroBlues muss also auf einem anderen Weg als über einen Thread an die ACVM angebunden werden.

Eine Alternative ist die Kombination der Ereignisschleife mit der Ereignisschleife der VM.

4.7.5.2 Interfaces

Das offizielle JSR-82 verwendet Schnittstellen (*interfaces*), um durch Typprüfung sicherzustellen, dass die übergebenen Kontrolldatenstrukturen, wie z.B. das `Connection`-Objekt, alle notwendigen Methoden bereit stellen. Die ACVM unterstützte zu Beginn der Arbeit noch keine Schnittstellen.

Die so mangelnde Typüberprüfung erwies sich aber als geringeres Problem als erwartet: die von Java (und dem Java-Compiler) vorgenommene Typenprüfung kann bei einem Projekt dieser Komplexität und Größe notfalls ignoriert werden. Es wird für die Laufzeit-VM nur der Zugriff auf ohnehin deklarierte Methoden verwendet, eine explizite Ausnutzung der Typenprüfung im Programm selbst findet in der vorliegenden JSR-82-Implementierung unter der ACVM nicht statt.

4.7.5.3 Strings

Strings werden so wenig wie möglich bei der Parameterübergabe verwendet. So werden für Parameter, die bei Aufrufen von C-Methoden in Java verwendet werden, nur primitive Datentypen genutzt. Selbst die MAC-Adresse wird in `int`-Werte umgewandelt und auf C-Ebene ins benötigte Format konvertiert.

JSR-82 verwendet Strings. Allerdings kann auf die explizite Verwendung von nicht konstanten `StringBuffer` in der API-Implementierung verzichtet werden.

Ein Beispiel für eine Verwendung von konstanten Strings in der API ist insbesondere der `url`-Parameter für `Connector.open`. Die ACVM unterstützte zu Beginn der Arbeit solche Zeichenketten. Strings konnten jedoch nicht als Übergabeparameter verwendet werden. Für die Lese- und Schreibkommunikation nach einem Verbindungsaufbau werden deshalb keine Strings, sondern Byte-Arrays verwendet, die noch auf Java-Seite in einzelne Aufrufe von C-Funktionen mit `char`-Variablen als Parameter umgewandelt werden. Eine direkte Verwendung von Strings als Übergabeparameter wäre ein besserer Entwurf: er würde zu deutlich lesbarerem Code führen.

Kapitel 5

Implementierung

5.1 Systemplattform

Als Hardware-Plattform sowohl für die Arbeit mit dem Sun Wireless Toolkit, der Java 2 Standard Edition als auch mit der ACVM wurden x86-kompatible 32-Bit-Prozessoren gewählt. Als Betriebssystem fand GNU/Linux Verwendung, die Tests liefen sowohl unter Debian GNU/Linux 4.0 als auch unter Ubuntu 7.10. Das Sun Wireless Toolkit lief nur unter den Ubuntu-Systemen, da das WTK nicht mit der von Debian verwendeten GNU `libc`-Version kompatibel ist.

Als Kommunikationshardware dienten handelsübliche Bluetooth-Kommunikationsgeräte, die als 'Stick' an den Universal Serial Bus (USB) [CHPI+00] angeschlossen wurden. MicroBlue ermöglicht es, die Geräte unter Verwendung des BlueZ-Stapels (siehe `http://www.bluez.org/`) über ein Programm `hci_emu` anzusprechen (siehe Anhang B, Seite 35).

Für die Tests mit dem Atmel AVR-Prozessor wurden sogenannte BtKit-Systeme verwendet, die von der Firma BeeCon geliefert wurden und über einen aktualisierbaren Flash-Speicher zur Programmierung verfügen (`http://www.btkit.com/`).

5.2 Software-Toolchain

Für die Übersetzung von MicroBlue wurde `gcc` 4.1.2 bzw. `avr-gcc` 4.1.2 des GNU-Projekts ebenso wie die in den GNU `binutils` 2.18 mitgelieferten Hilfsprogramme verwendet. Als Java-Laufzeitumgebung wurde, neben der am Institut entwickelten ACVM, Suns Java 2 Standard Edition in der Version 5.0 und das Wireless Toolkit in der Version 2.5.1 genutzt. Zum Flashen der für den Atmel entwickelten Progamme wurde `avrdude` verwendet.

Andere externe, nicht in Ubuntu oder Debian GNU/Linux vorhandene Programme fanden keine Verwendung.

5.3 Parameterübergabe-Schnittstelle

Um die Brücke zwischen den verschiedenen VMs und MicroBlue zu schlagen, waren zwei Schritte notwendig. Zum einen mussten die MicroBlue-Programme als Bibliothek um-

strukturiert werden. Zum anderen mussten die so entstandenen Bibliotheksfunktionen aus Java heraus aufrufbar gemacht werden. Für die Java 2 Standard Edition und die 'Dummy API' (siehe Abschnitt 4.2.3.1) wurde das Java Native Interface genutzt, das eine definierte Schnittstelle für den Aufruf von plattformspezifischen C-Bibliotheken bietet.

Die Nutzung von plattformspezifischen C-Funktionen in der eigentlichen AmbiComp VM erfolgt über die sogenannte 'NativeAPI', die nur rudimentäre Parameterübergabe erlaubt. Deshalb wurde sichergestellt, dass keine Zeichenketten oder Objekte als Übergabeparameter zwischen der Java-Implementierung von JSR-82 und MicroBlue vorkamen. Entsprechende Konstruktionen wurden durch Transformationsfunktionen bzw. -methoden auf Seite der MicroBlue-Bibliothek oder von Java umgestaltet.

5.4 Einschränkungen der ACVM bei der Implementierung

Für die Verwendung der ACVM waren einige Einschränkungen zu beachten, die bei der Implementierung von Bedeutung waren.

Wie im Abschnitt 4.7.5.1 dargelegt, wäre eine Anbindung der Ereignisschleife von MicroBlue über einen eigenen Thread naheliegend gewesen. Da Threads in der AmbiComp VM nicht zur Verfügung stehen, wurde diese Methode nur für die Implementierung innerhalb der J2SE verwendet. In der ACVM gibt es auch eine regelmäßig aufgerufene Funktion, die für die Verarbeitung des nächsten Opcodes zuständig ist. Ereignisschleife, die die Verarbeitung der Opcodes nacheinander erledigt.

Um die beiden Schleifen zu verquicken, wurde die MicroBlue-Ereignisschleife nach jeder abgeschlossenen Opcode-Verarbeitung aufgerufen. Falls Bluetooth-Ereignisse anstehen, werden so keine gerade verarbeiteten Opcodes unterbrochen. Innerhalb der Ereignisschleife von MicroBlue darf folglich in keinem Fall blockiert werden, um auf Ereignisse im Java-Programmfluss zu warten. Ein umgekehrtes Warten ist möglich, da während der Warteschleife des Java-Programms Opcodes abgeschlossen werden und somit auch die MicroBlue-Ereignisschleife aufgerufen wird. Während das Java-Programm sich also im *busy waiting*-Zustand befindet, kann auf MicroBlue-Ereignisse reagiert werden.

Andere Beispiele, bei denen mehrfädige Programmierung für die Nutzung von JSR-82 vorausgesetzt werden, ist das Warten auf Ereignisse wie die Suche von Geräten oder Diensten. Hier sind aber auch Timeouts vorgesehen, so dass es möglich ist, durch Nutzung der Timeouts ohne gleichzeitiges Fortschreiten des Java-Programms die Suche aufzurufen.

Schnittstellen, also die Verwendung von 'Mehrfachvererbung' durch `interface`-Deklarationen, wurden von der ACVM zu Beginn nicht unterstützt. Das Fehlen der Typprüfung ist verschmerzbar (siehe Abschnitt 4.7.5.2), doch ist es ebenfalls unmöglich, Methoden aufzurufen, die nicht direkt am eigentlichen Objekt implementiert werden, sondern an anderer Stelle der Vererbungsstruktur. Auch ein *cast* einer implementierenden Klasse in eine Klasse, die diese Implementierung enthält, ist nicht möglich. Als Beispiel möchte ich hier den Aufbau einer Verbindung darstellen:

```
StreamConnection connection =
```

```
(StreamConnection) Connector.open(url);
InputStream is = connection.openInputStream();
```

Die `openInputStream()`-Methode ist nur aufrufbar, wenn `Connector` selbst diese Methode implementiert. Das ist in der vorliegenden Implementierung der Fall. Bei der anfänglich zur Verfügung stehenden ACVM scheiterte diese Konstruktion aber schon vorher am *cast* des `Connection`-Objekts. Dieses wird von der statischen `open`-Methode zurückgegeben wird und muss in ein `StreamConnection`-Objekt konvertiert werden. Ein solcher *cast* war in der ursprünglichen ACVM nicht möglich.

Beim damaligen Stand der ACVM ließ sich also nur an dieser Beschränkung vorbei arbeiten, indem auf das Konvertieren (*casting*) von Klassen gänzlich verzichtet wird und statische Zugriffsmethoden (*accessors*) für die eigentlich durch *casts* zu beschaffenden Klassen erzeugt werden:

```
Connector.open(url);
SPPInputStream is = Connector.oIS();
```

Es ist offensichtlich, dass die objektorientierte Java-API auf diese Art und Weise kaum wiederzuerkennen ist. Die jetzigen Versionen der ACVM erlauben die Verwendung von *interfaces*, wenn die Objekte selbst die gewünschten Methoden implementieren.

5.5 Beschreibung der MicroBlue-Bibliothek

Statt einer Initialisierung der MicroBlue-Funktionen im Hauptprogramm wurden zwei Initialisierungsfunktionen geschrieben, die vor Verwendung von MicroBlue über die Bibliothek `libbt-hal` die verwendeten Variablen initialisieren (`bt_preinit`) und nach Einstellung vom Standard abweichender Parameter in den Funktionen `bt_set_passive`, `bt_set_mac_address` oder `bt_set_pts_num` die eigentliche Hardware-Initialisierung vornehmen (`bt_init_hw`).

Die übrigen exportierten Funktionen sind mit der Verbindungsverwaltung und dem Datentransfer befasst: `bt_disconnect` löst eine anwendungsseitige Verbindungstrennung aus, `bt_get_rts_state` dient als Auslöser beim blockierenden Warten auf Sendebereitschaft. `bt_read_fifo` und `bt_fifo_avail` verwalten den Lesepuffer, der bei eingehenden Datenpaketen von MicroBlue befüllt wird. `bt_send` und `bt_send_multi` schließlich dienen dem Erzeugen und Absenden von Daten an die Gegenstelle.

Die letzte exportierte Funktion, `hciProcess`, initiiert den Durchlauf der MicroBlue-Ereignisschleife. Diese Funktion wird nicht aus dem Java-Code der JSR-82-Implementierung aufgerufen, sondern mit der Opcode-Schleife der ACVM verwoben und von dort direkt aufgerufen.

Ansonsten findet sich in `fifo.c` die Implementierung des Lesepuffers, während `bt-hal.c` lediglich die Implementierung der speziell zu behandelnden Ereignisse des MicroBlue-Ereignisstapels enthält.

Auch das oben genannte `bt_set_pts_num` findet sich als Bestandteil der ACVM: das Einlesen der Kommandozeilenparameter (zumindest unter GNU/Linux) kann eine Umstellung des Pseudoterminal Device Files ('Pts num') zur Folge haben (siehe Abschnitt 5.5.1).

5.5.1 Kommunikation mit BlueZ und EEPROM-Simulation

Aus der Übersicht (siehe Abschnitt 3.2) lässt sich entnehmen, dass unter GNU/Linux MicroBlue die genannten EEPROM-Abbilddateien voraussetzt. Diese werden für die .so-Bibliothek unter J2SE und der 'Dummy API' zur Übersetzungszeit der MicroBlue-Bibliothek erstellt. Die genannten Probleme der nicht bereits zur Übersetzungszeit feststehenden Adresse im Adressraum werden durch eine Heuristik gelöst:

```
/* read the specified number of bytes from the eeprom */
void eeprom_read(u08_t *mem, u32_t addr, u16_t len) {
  FILE *efile;
  //PRINTF_MISC("base1: %lx, erb1: addr = %lx. ", eeprom_base, addr);

  if(!eeprom_base) {
    eeprom_file_check();
#ifdef EEPROM_SO
    eeprom_base=addr - 0x20;
#endif
  }
  addr -= eeprom_base;

  //PRINTF_MISC("base: %lx, erb2: addr = %lx. ", eeprom_base, addr);
  //PRINTF_MISC("eeprom_read_block: addr = %lx, len = %d\n", addr, len);
  [...]
```

Der erste Zugriff aufs EEPROM findet auf eine Speicheradresse statt, die direkt nach der Identifikations-Zeichenkette liegt. So lange die Zeichenkette nicht in der Länge geändert wird, erreicht man durch Abzug von 0x20 Bytes die Basisadresse des EEPROM-Bereichs. Eine bessere Lösung wäre hier wünschenswert.

Für die ACVM werden die EEPROM-Dateien durch ein eigenes *`make` target* beim Kompilieren der ACVM erzeugt. Änderungen in der MicroBlue-Bibliothek bedingen also ein neues Erzeugen dieser Dateien.

Die Kommunikation von MicroBlue mit BlueZ, das den Zugriff auf beliebige Bluetooth-Hardware unter GNU/Linux ermöglicht, läuft über das `hci_emu`-Programm. Siehe dazu auch Abbildung 5.1. Zur Kommunikation wird über den Pseudoterminal-Multiplexer (`/dev/ptmx`) ein neues, freies Pseudoterminal angefordert (zum Beispiel `/dev/pts/3`).

Über dieses Device laufen dann die Befehle, die auf anderen Plattformen von der HCI-Schicht an den UART übergeben werden. Diese Befehle werden statt dessen in den BlueZ-Kommandopuffer eingefügt. Das erzeugte Pseudoterminal muss der VM bekannt gemacht werden, damit die Bibliotheksfunktionen von MicroBlue per Emulator auf das korrekte Bluetooth-Gerät zugreifen.

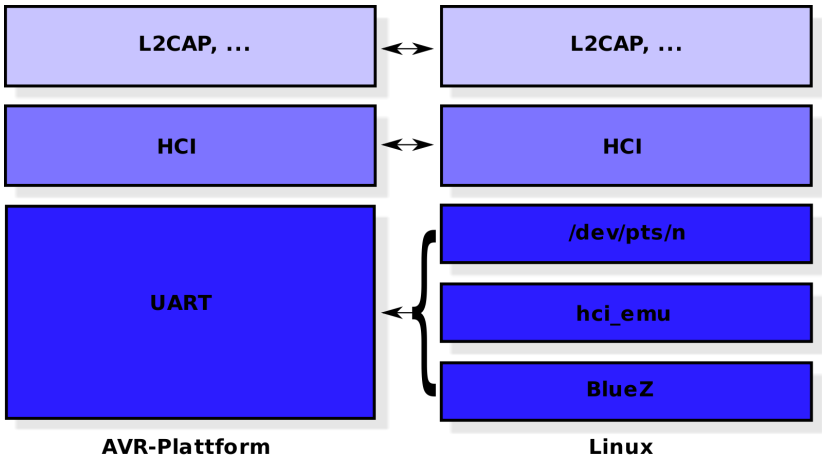

Abbildung 5.1: Kommunikation der HCI-Schicht mit dem Hardware-UART (links) bzw. mit BlueZ über den Emulator (rechts). Die Pfeile verbinden korrespondierende Schichten.

Alle genannten Kunstgriffe sind unter der Atmel-Plattform nicht nötig, da das EEPROM nicht als Datei vorliegt und keine HCI-Emulationsschicht über BlueZ verwendet wird.

5.6 Beschreibung der JSR-82-Implementierung

Die implementierten Teile der CLDC-API, wie sie im Abschnitt 3.3 genannt wurden, werden an dieser Stelle nicht beschrieben, da deren Funktionalität in der Beschreibung der CLDC-Spezifikation genau dargelegt ist und Abweichungen sich aus der internen Struktur der ACVM und weniger aus Entwurfsentscheidungen bei der Implementierung von JSR-82 begründen.

Bei den Klassen unter `javax.microedition.io` handelt es sich um eine Nachbildung des GCF, die eigentliche Bluetooth-spezifische Implementierung ist in der `Connector.open`-Funktion angesiedelt. Die Initialisierung der MicroBlue-Bibliothek erfolgt ebenso in der `Connector.open`-Funktion. Das als Teil der URL übergebene Zielgerät und die weiteren Verbindungsparameter, die für diese Initialisierung notwendig sind, stehen erst zu diesem Zeitpunkt fest.

Die Implementierung der Lese- und Schreibströme erfolgt in den `SPPInput-` und `SPPOutputStream`-Klassen. Sie sind nicht Teil der öffentlichen API und befinden sich unter `ambicomp.bluetooth`. Ebenfalls dort findet sich die Hüllklasse `Microblue`, die die Aufrufe der MicroBlue-Bibliotheksfunktionen kapselt.

Die Implementierung mithilfe der 'Dummy API' ist weitgehend analog aufgebaut. Nach dem Laden der .so-Datei im statischen Initialisierungsblock der `Connector`-Klasse beginnt

durch Aufruf der `Connector.open`-Methode ebenfalls die Bibliotheksinitialisierung. Der einzige Unterschied ist der Aufruf der JNI-Methoden statt des Aufrufs der Methoden der Klasse `Microblue`.

5.7 Beschreibung der *proof of concept*-Programme

An die Beschränkungen der ACVM angepasste Beispielprogramme, die elementare Kommunikationsprimitive der Bluetooth-Kommunikation verwenden, finden sich in den Programmen `BtServer`, `BtClient` und `BtTest` im Verzeichnis `examples/src/`: `BtServer` wartet auf eingehende Verbindungen, die MicroBlue entgegennimmt. Nach Verbindungsabbau werden einige Prüfdaten übersandt, bis die jeweils aufgebaute Verbindung nach Übertragungsende ordnungsgemäß beendet wird.

`BtClient` baut eine Verbindung zu einem solchen Server auf, der in der URL spezifiziert wird (insbesondere über die Bluetooth *device address*).

`BtTest` dient zum Testen der Basis-Funktionalität von MicroBlue. Der *low-level* Test verwendet nicht die implementierten Bibliotheksfunktionen, sondern ruft direkt die nativen MicroBlue-Methoden auf.

Im Verzeichnis `examples/src-legacy` finden sich mit `BtClient` und `BtServer` auch unter Legacy VMs wie der Java 2 Standard Edition (mit selbst implementiertem GCF) lauffähige Varianten, die eine korrekte Behandlung von 'Mehrfachvererbung' durch Java `interfaces` voraussetzen.

Kapitel 6

Evaluation

6.1 CLDC-API-Abdeckung

Die in dieser Arbeit angefertigte Implementierung stellt die für die Kommunikation essenziellen Klassen der CLDC-API zur Verfügung. Es handelt sich um vier Teile:

1. Der Verbindungsaufbau ist über die `Connector.open`-Methode implementiert. Über das URL-Objekt lässt sich spezifizieren, ob die Bluetooth-Kommunikation im Server- oder Client-Modus arbeitet. Eine URL für den Server-Modus ist nach dem Schema `"protokoll://localhost:eindeutige_id;name=Beispielserver"` aufgebaut; im Client-Modus lautet diese `"protokoll://bd_adresse:kanalnummer"` [Mic05, Abschnitt 10.3, Seite 66]. Dabei sind als Protokollmöglichkeiten `btspp` oder `btl2cap` spezifiert. Die Bluetooth Hardware-Adresse wird dynamisch aus der Verbindungszeichenkette ausgelesen und an MicroBlue weitergeleitet.

2. Die Lese-Kommunikation wird über einen `InputStream` abgewickelt. Aus Effizienzgründen ist neben der Lesemethode für einzelne Bytes eine spezielle Methode für das Einlesen mehrerer Bytes auf einmal implementiert. Damit ist es möglich, größere Objekte effizient einzulesen. Beide Methoden blockieren spezifikationsgemäß, bis Daten anliegen.

3. Auch die Schreib-Kommunikation läuft über ein Stream-Objekt, entsprechend ein `OutputStream`. Auch hier ist die Übergabe von einzelnen Bytes wie der Versand mehrerer Bytes auf einmal möglich. Dies ermöglicht den Versand von größeren Objekten, die vollständig zum gleichen Zeitpunkt im Zielpuffer der Gegenstelle eintreffen. Eine beim Lesen blockierende Gegenstelle gibt in ihrer `read`-Methode also das gesamte Objekt zurück, statt nur das erste Byte. Wechselseitiges Lesen und Schreiben wird durch die Flusskontrolle geregelt.

4. Im Rahmen der Verbindungsverwaltung ist das ordnungsgemäße Schließen der Verbindung implementiert, welches die verwendeten Objekte wieder freigibt und im Falle des Servers eine neue Verbindung ermöglicht.

6.1.1 Mehrere Geräte

Die Verwendung mehrerer Bluetooth-Geräte ist seitens MicroBlue problematisch: MicroBlue muss in diesem Fall in einem speziellen Modus kompiliert werden: der `HCI_ENABLE_PICONET`-Parameter (siehe [Har04, Abschnitt 3.4.4 und 4.5.5]) muss gesetzt sein. Dabei gibt der Parameter an, wie viele Nodes maximal am Piconetz teilnehmen können. Für jeden Node wird Speicher für die Verbindungsinformationen reserviert. Ist der Parameter gesetzt, was standardmäßig in MicroBlue und auch bei der vorliegenden Implementierung nicht der Fall ist, verlangt MicroBlue außerdem immer die Rolle als Master des Piconetzes. Da es nur einen Master in einem Piconet geben kann, ist es nicht möglich, zwei mit dem `HCI_ENABLE_PICONET` Parameter kompilierte Instanzen von MicroBlue im selben Piconetz kommunizieren zu lassen. Dies ist eine Beschränkung von MicroBlue. Die MicroBlue-Bibliothek kann in der jetzigen Form also nicht eingesetzt werden, um beispielsweise als Access Point verwendet zu werden, an dem sich Clients anmelden, die dasselbe Bibliothekskompilat verwenden. Weitere Funktionalität wie die Überbrückung von mehreren Piconets über einen im Master- und Slave-Modus alternierenden 'bridging node' ist ohne den `HCI_ENABLE_PICONET`-Parameter ebenfalls nicht möglich.

6.1.2 Discovery-Mechanismen

Für die Discovery-Mechanismen (device inquiry, Service Discovery Protocol (SDP)) wäre die Verwendung mehrerer Threads wünschenswert, damit während der Suchphase bereits erkannte Geräte und Dienste behandelt werden können. Eine Implementierung ohne Threads wäre möglich, müsste aber die Timeout-Periode abwarten, um dann sämtliche erkannte Geräte oder Dienste zurückzuliefern. Die tatsächliche Länge der Timeout-Periode ist implementierungsabhängig [Mic06b, `DiscoveryAgent.startInquiry`-Methode]. Zur Zeit ist ausschließlich die Kommunikation mit *a priori* bekannten Clients und Diensten möglich.

Ein weiteres Problem beim SDP ist die Verwendung selbst definierter Dienste. Für diese ist die Kompilierung von Dienst-Templates über das Programm `sdp_tool` in MicroBlue vorgesehen. Die Definition ist aber nicht zur Laufzeit, wie in JSR-82 vorgesehen, möglich.

6.1.3 Authentifizierung und Verschlüsselung

Diese Protokollbestandteile werden in JSR-82 über das URL-Objekt geregelt. Die Kommunikation der vorliegenden Implementierung läuft unverschlüsselt ab, sichere Kommunikation kann auf Anwendungsebene implementiert werden. Für sichere Kommunikation ist die im Bluetooth-Standard vorgesehene Verschlüsselung (die nach dem SAFER+-Algorithmus [Cor98] einen zu verwendendes gemeinsames Geheimnis aushandeln) nach [Bec07, Abschnitt 2.10, Seite 22] wegen Schwächen beim Pairing-Prozess allein kaum ausreichend.

6.1.4 btl2cap

Das L2CAP-Zugriffsprotokoll wird von der vorliegenden Implementierung nicht unterstützt. Zu den Gründen siehe Abschnitt 4.5.

6.2 Kompatibilität mit BlueZ

Um zu überprüfen, ob das Entwurfsziel der Berücksichtigung von Standards erreicht wurde, wird in diesem Abschnitt der Versuch unternommen, eine Kommunikationsverbindung mit einer alternativen Bluetooth-Implementierung aufzubauen.

Eine solche Kommunikation ist beispielsweise über Linux-Kommandozeilentools, die direkt auf BlueZ aufsetzen, möglich. Dazu wurde das BtServer-Beispielprogramm verwendet. Auf einem PC lief der BtServer, während auf einem anderen PC der BlueZ-Stack für eine RFCOMM-Verbindung vorbereitet wurde:

`hcitool scan` (ergibt Bluetooth Device Address, z.B. 00:1A:7D:00:EF:9A)
`sdptool browse` (ergibt in der Protocol Descriptor List 'RFCOMM' mit Channel 2)
`sudo rfcomm bind rfcomm0 00:1A:7D:00:EF:9A 2`

Danach wurde das Terminal-Programm Cutecom gestartet, dass auf die Parameter

- Gerätedatei `/dev/rfcomm0`
- Baudrate 57600
- 8 Data bits
- 1 Stop bit
- Parity None
- Kein Handshake

konfiguriert wurde. Mit der Option 'Apply settings when opening' muss Cutecom angewiesen werden, diese Einstellungen beim Öffnen von `/dev/rfcomm0` anzuwenden.

Während auf dem ersten PC BtServer lief, wurde mit 'Open device' die Verbindung hergestellt. Die im BtServer übersandten Zeichen `UK` wurden korrekt übermittelt.

Nachdem die Parameter der seriellen Schnittstelle durch Cutecom korrekt gesetzt wurden, funktioniert auch ein einfaches `cat /dev/rfcomm0`.

6.3 Energieeffizienz

Durch die in JSR-82 nicht vorgesehene Ansteuerung von den Energiesparmodi können nur anwendungstransparente Algorithmen verwendet werden, die basierend auf Heuristiken die Energiesparmodi aktivieren. Dies liesse sich zum Beispiel auf Seiten des Clients nach Beendigung einer Verbindung implementieren, indem dieser sich so lange in einen Schlafmodus versetzt, bis der nächste Kommunikationsvorgang ansteht.

6.4 Performance und Speicherbedarf

Durch die Verwendung von MicroBlue mit reduzierter Funktionalität und der sparsamen ACVM wird eine Codegröße von circa 197 kiB auf der AVR-Plattform erreicht, siehe Tabelle 6.1. Den größten Anteil mit circa 142 kiB hat die MicroBlue-Bibliothek. Die verschiedenen Kompilierungseinheiten sind mit ihrer anteiligen Größe aus Tabelle 6.1 ersichtlich. Wenn auf RFCOMM/SPP verzichtet würde und eine reine L2CAP-Kommunikation implementiert wäre, würde sich die Größe des Kompilats um etwa 34 kiB verringern.

Auf Ubuntu ergibt sich für die kompilierten Binärdateien und dem kaum ins Gewicht fallenden Blob eine Gesamtgröße von über 3 MiB.

Zur Laufzeit wird ein Speicherbedarf von 200 Bytes für die Puffer und die Variablen geschätzt. Der Java-Lesepuffer besteht aus 100 Bytes, während MicroBlues mehrere eigene Puffer von insgesamt etwa 40 Bytes sowie Variablen den Rest ausmachen. Der Code läßt sich auf den BtKit-Systemen *in place* ausführen, ohne dass für diesen RAM-Speicher belegt wird. Exakte Messungen waren nicht ohne größeren Aufwand durchzuführen, da kein Betriebssystem auf den AVR-Systemen vorlag, das entsprechende Meßfunktionen zur Verfügung stellt.

Tabelle 6.1: Codegröße der einzelnen Software-Bestandteile

Software-Komponente	Codegröße AVR	Codegröße Ubuntu
ACVM ohne libbt-hal.a, libc.so...	52 kiB	618 kiB
libbt-hal.a	142 kiB	267 kiB
libc.so.6 etc.	-	2370 kiB
BtServer-Blob inkl. API	3,4 kiB	2 kiB
Summe	197,4 kiB	3257 kiB

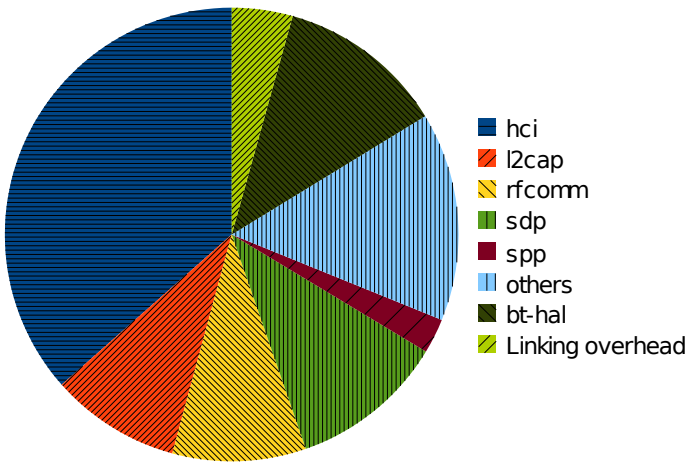

Abbildung 6.1: Anteilige Größe der einzelnen MicroBlue-Teile des `libbt-hal.a`-Kompilats, auf der AVR-Plattform.

Laufzeitmessungen haben ergeben, dass ein effektiver Durchsatz zwischen der BlueZ-Implementierung von Linux und der ACVM als Server von circa 64 Byte pro Sekunde erreicht werden kann. Dabei wurden in einer Schleife Datenpakete versandt, in denen

jeweils 8 Datenbytes enthalten waren. Größere Pakete würden den Durchsatz vermutlich deutlich steigern, würden allerdings eine Erweiterung der `bt_send_multi`-Funktion und der NativeAPI erfordern, so dass diese Funktion auch Felder beliebiger Größe als Übergabeparameter erhalten kann.

Kapitel 7

Fazit

7.1 Reusability

Die Schnittstelle zur MicroBlue-Ebene ist klein gehalten worden. Über definierte Schnittstellenmethoden in der `ambicomp.bluetooth.Microblue`-Klasse wird auf MicroBlue zugegriffen. Eine analoge Implementierung für andere Stacks wie beispielsweise das freie BlueZ müsste diese Schnittstelle implementieren, um die hier entwickelte JSR-82-Implementierung verwenden zu können. Aufgrund der fehlenden Portierung (und Portierbarkeit) des Linux-Kerns für Atmel-Prozessoren ist eine solche Implementierung für die hier verwendeten System der Firma BeeCon nicht möglich.

7.2 Zukünftige Entwicklungen

Neben der Vervollständigung der abgedeckten Schnittstellen, die den Schritt einer *proof of concept*-Implementierung zu einem vollständigen nutzbaren JSR-82 vollzieht, können die beiliegenden Beispielprogramme erweitert werden. Beispielsweise könnte ein vollständiger Echo-Server und Echo-Client als einfache Anwendung erstellt werden oder die Implementierung einer Kommandozeilen-Fernsteuerung über JSR-82. Da die Atmel-Systeme keine Nutzerinteraktivität erlauben, sollte diese Implementierung unter GNU/Linux erfolgen.

Zur Sicherung von Codequalität sollte die ACVM dringend um die vollständige Unterstützung von Java-Interfaces erweitert werden. Die hier gewählte Form der Erzeugung der speziellen Lese- und Schreibströme mit expliziten Casts ist nicht JSR-82-konform.

Die Erweiterung auf mehrere Nodes im Piconet müsste in MicroBlue vorgenommen werden, in dem die Möglichkeit hinzugefügt werden könnte, zur Laufzeit zwischen dem Piconet- und Master-/Slave-Modus umzuschalten.

Für spontane Vernetzung wäre eine verbindungslose Kommunikation von Vorteil, die allerdings im Rahmen von JSR-82 nicht angeboten wird. Durch dauerhafte Geräte- und Dienstesuche sowie kurze Übertragungen ohne langfristig etablierte Verbindungen wäre eine solche Vernetzung aber möglicherweise dennoch in anschließenden Arbeiten lösbar.

Durch Erweiterungen von JSR-82 könnten ebenfalls Mechanismen zum energiegewahren Verwalten der existierenden Verbindungen auf Anwendungsebene implementiert werden.

7.3 Entwurfsziele

Durch die Implementierung auf verschiedenen Java-Plattformen und den unterschiedlichen Hardware-Plattformen ist eine grundlegende Portabilität bereits empirisch erwiesen. Die Verwendung von plattformübergreifenden Programmstrukturen erlaubt die Annahme, dass eine Portierung auf unterschiedliche Hardware-Architekturen im Bereich des Möglichen liegt.

Die Berücksichtigung von Standards ist nur rudimentär gegeben. JSR-82 basiert fundamental auf 'mehrfach vererbten' Schnittstellen. Die verwendete ACVM müsste in dieser Hinsicht erweitert werden, um eine standardkonforme Implementierung sicherzustellen. Ansonsten wird die Anwendungsprogrammiererin nicht mit implementierungsspezifischen Aufrufen in Berührung gebracht. Die implementierungsabhängigen Aufrufe des MicroBlue-Codes sind gekapselt innerhalb der API-Implementierung.

Die Robustheit muss bezüglich Fehlerbehandlung erweitert werden. Über begrenzte Schnittstellen wie der `bt_get_rts_state`-Funktion lassen sich Statusmeldungen des Stacks auf die API-Seite zurückmelden, die dann weiter verwendet werden könnten. Beispiele dafür wären eine `ConnectionTimeoutException` und eine `ConnectionRequestDeniedException`. Eine Fehlerbehandlung ist in der vorliegenden Implementierung nur bei den in MicroBlue transparent vorhandenen Fehlerbehandlungsroutinen gegeben.

Für die Effizienz siehe Abschnitt 6.4.

7.4 Ergebnis

Insgesamt ist mit der vorliegenden Implementierung ein Fundament gelegt worden, auf dem sich die fortgeschrittenen Funktionen von JSR-82 aufbauend implementieren lassen. Die jetzt mögliche Kommunikation zwischen Bluetooth-Geräten erfüllt die Maßgaben an Code-Größe und Portabilität, für Erweiterungen sollte jedoch den Empfehlungen von Abschnitt 7.3 gefolgt werden. Dann lässt sich auch ohne die Verwendung von mehrfädiger Programmierung eine standardkonforme Implementierung gestalten. Eine Erweiterung um Funktionalitäten wie der automatischen Erkennung könnte im Rahmen von weiteren Arbeiten erfolgen; für die Verwendung in skalierbaren Sensor-/Aktor-Netzen muss aber entweder MicroBlue um eine Aktivierung des Piconet-Master-Modus zur Laufzeit erweitert werden oder in die MicroBlue-Bibliothek beide Übersetzungsvarianten integriert werden.

Anhang A: Quellcode

Beiliegende CDs

Der Quellcode sämtlicher Änderungen findet sich auf der 'Proprietary CD', während der Quellcode aller unter einer OSI-zertifizierten Lizenz stehenden Komponenten auch auf der 'Free Software CD' zu finden ist.

Im einzelnen sind diese CDs wie folgt aufgebaut:

Tabelle 7.1: Auf der beiliegenden CD befindlicher Quellcode.

Pfad	CD-Version	Beschreibung
`microblue/`	Proprietary	MicroBlue-Stapel, enthält `libbt-hal`
`libbt-hal/`	Proprietary + Free	JSR-82 anbindende Bibliothek
`dummy-api/`	Proprietary	Dummy-API für Legacy VMs
`api/`	Proprietary + Free	AmbiComp API
`transcoder/`	Proprietary	Wandelt `.class`-Datei in Blob um
`acvm/`	Proprietary	ACVM
`examples/`	Proprietary	Beispielprogramme für ACVM und Legacy VMs
`btexamples/`	Proprietary + Free	Freie Beispiele für ACVM und Legacy VMs
`latex/`	Proprietary + Free	Quellcode für die Ausarbeitung

Modifizierte Dateien

In Tabelle 7.2 findet sich eine Aufstellung aller vom Autor veränderten und erstellten Dateien in den verschiedenen Software-Komponenten.

Tabelle 7.2: Vom Autor veränderte oder *hinzugefügte* Dateien auf der CD.

Pfad	CD-Version	Beschreibung
microblue/	hci_tools/ hci_emu/	hci_emu.c, Makefile, readme.txt
microblue/	examples/unix/ spp_active/	config.h, Makefile, test.c
microblue/	examples/unix/ spp_passive/	config.h, Makefile, test.c, readme.txt
microblue/ und libbt-hal/	examples/unix/ java/	*config.h, fifo.c, fifo.h, hal-bt.c, hal-bt.h, Makefile, native_acvm.c, native_acvm.h, native.c, native.h*
microblue/ und libbt-hal/	examples/avr/ java/	*config.h, fifo.c, fifo.h, hal-bt.c, hal-bt.h, Makefile, native_acvm.c, native_acvm.h*
api/	ambicomp/ bluetooth/	*Microblue.java, PollThread.java, SPPConnection.java, SPPInputStream.java, SPPOutputStream.java*
api/	javax/bluetooth/	*ServiceRegistrationException.java*
api/	javax/ microedition/io/	*Connection.java, Connector.java, InputConnection.java, OutputConnection.java, StreamConnection.java, StreamConnectionNotifier.java*
api/	java/io/	InputStream.java, OutputStream.java, IOException.java
api/	java/lang/	IndexOutOfBoundsException.java, String.java
dummy-api/	ambicomp/ bluetooth/	*PollThread.java, SPPInputStream.java, SPPOutputStream.java*
dummy-api/	javax/bluetooth/	*ServiceRegistrationException.java*
dummy-api/	javax/ microedition/io/	*Connection.java, Connector.java, InputConnection.java, OutputConnection.java, StreamConnection.java, StreamConnectionNotifier.java*
transcoder/	src/transcoder/ serializer/ classic/	NativeClassLoader.java
acvm/	NativeAPI/	*Bluetooth.cc*, NativeMethodDispatcher.cc, NativeMethodDispatcher.hh
acvm/	vm/shared/	BootLoader.cc
examples/ und btexamples/	./	README
examples/ und btexamples/	src/ambicomp/ bluetooth/	*BtClient.java, BtServer.java, BtTest.java*
examples/ und btexamples/	src-legacy/ ambicomp/ bluetooth/	BtClient.java, BtServer.java, BtTest.java

Anhang B: README

```
0. Check out and build libbt-hal.so

svn co https://moon.ira.uka.de/subversion/
 beecon/trunk/MicroBlue b_svn
cd b_svn/examples/unix/java
make
cd ../../../..

1. Check out and build ACVM

The following packages have to be installed (using
Debian/Ubuntu):
build-essential, g++, libcppunit-dev, libtool, autoconf,
 automake

svn co https://moon.ira.uka.de/subversion/
 main/trunk/projects/JavaVM javavm_svn
cd javavm_svn
./bootstrap.sh
cd bin-x86
../configure --disable-debug --enable-bluetooth
 --with-microblue=../../b_svn/

If you leave out --disable-debug, ACVM is compiled with
debug information.

make
cd ../..

2. Check out and build ACVM-API

A Java SDK has to be installed (used here: Sun Java 5 SE)
WARNING! Classes generated by Sun Java 6 are at the time
of writing incompatible with ACVM!
Additionally, the following package has to be installed
(using Debian/Ubuntu):
ant

svn co https://moon.ira.uka.de/subversion/main/trunk/
 projects/AmbiComp/AmbiCompAPI/ api_svn
cd api_svn
```

```
ant
ant jar
cd ..

2a. Check out and build ACVM-dummy-API

A Java SDK has to be installed (used here: Sun Java 5 SE)
WARNING! Classes generated by Sun Java 6 are at the time
of writing incompatible with ACVM!
Additionally, the following package has to be installed
(using Debian/Ubuntu):
ant

svn co https://moon.ira.uka.de/subversion/main/trunk/
 projects/AmbiComp/AmbiCompAPI_dummy/ api_dummy_svn
cd api_dummy_svn
ant
ant jar
cd ..

2.1 Check out and build Transcoder

A Java SDK has to be installed (used here: Sun Java 5 SE)
Additionally, the following package has to be installed
(using Debian/Ubuntu):
ant

svn co https://moon.ira.uka.de/subversion/main/trunk/
 projects/AmbiComp/Transcoder/ trans_svn
cd trans_svn
editor properties

Here, the comp.java.api property must be set to ../api_svn
instead of ../AmbiCompAPI if you followed the path name
conventions in this README.

ant
ant jar
cd ..

3. Check out and build example programs for ACVM

A Java SDK has to be installed (used here: Sun Java 5 SE)

svn co https://moon.ira.uka.de/subversion/main/trunk/
 projects/AmbiComp/AICU/examples acex_svn
cd acex_svn/src
editor build.xml

Change the property values of "ambicomp.java.api" from "../AmbiCompAPI"
to "../api_svn" and the property value of "ambicomp.transcoder" from
```

""../Transcoder/bin/transcoder.jar" to "../trans_svn/bin/transcoder.jar".
Then, continue with

ant -Dtarget=ambicomp/bluetooth/BtServer
cd ..

3a. Check out and build example programs for testing in a
standard VM, using the acre_dummy API

Follow the instructions under 3., but change the
property values of "ambicomp.java.api" from "../AmbiCompAPI"
to "../api_dummy_svn".

4. Execute example program on ACVM

cd javavm_svn/bin-x86
./acvm ../../acex_svn/bin/ambicomp/bluetooth/BtServer.blob
To actually enable the Bluetooth functionality, see 5.

4a. Execute example program on a standard VM, using the
acre_dummy API

java -classpath ../../api_dummy_svn/acre_dummy.jar:.
 BtServer
To actually enable the Bluetooth functionality, see 5.

5. Execute Bluetooth-using program on ACVM

In one terminal, type
cd b_svn/hci_tools/hci_emu
sudo ./hci_emu

This will chain the BlueZ HCI command buffer to a pseudo-terminal device.
This device is allocated dynamically and the device number depends on
the number of other pseudo-terminals currently open (e.g. graphical terminals).
The command will display the correct device, e.g. /dev/pts/4.

Since hci_emu runs as root, the pseudo-terminal device is created with
root permissions. It is probably necessary to change the device ownership
to your user id:

sudo chown user /dev/pts/4

Then, run the Bluetooth-using program using the pts device number as an
argument (warning, this only works for 0-9):

cd javavm_svn/bin-x86
./acvm ../../acex_svn/bin/ambicomp/bluetooth/BtServer.java 4

Literaturverzeichnis

[Bec07] Becker, Andreas: *Bluetooth Security & Hacks.* Ruhr-Universität Bochum, 2007.

[BLMM94] Berners-Lee, T., L. Masinter und M. McCahill: *Uniform Resource Locators (URL).* RFC 1738, Internet Engineering Task Force, dec 1994. `ftp://ftp.isi.edu/in-notes/rfc1738.txt`. (08. Februar 2008).

[CHPI+00] Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC und Philips (Herausgeber): *Universal Serial Bus Specification, Revision 2.0.* USB Implementers Forum, 2000.

[Com95] Committee, TIS (Herausgeber): *Executable and Linking Format (ELF) Specification, Version 1.2.* Tool Interface Standard (TIS), 1995.

[Cor98] Corporation, Cylink (Herausgeber): *Nomination of SAFER+ as Candidate Algorithm for the Advanced Encryption Standard (AES).* Cylink Corporation, 1998.

[FM05] Fleisch, Elgar und Friedemann Mattern: *Das Internet der Dinge.* Springer, 2005.

[Fuh06] Fuhrmann, Thomas: *Software Engineering for Ambient Intelligence Systems.* In: *Proceedings of Workshop on Mobile and Embedded Interactive Systems (MEIS'06)*, Dresden, October 2006.

[G+06] Gosh, Rishab Aiyer et al.: *Economic impact of open source software on innovation and the competitiveness of the Information and Communication Technologies (ICT) sector in the EU.R.* Technischer Bericht, European Commission, 2006.

[Gmb04] GmbH, BeeCon (Herausgeber): *BeeCon MicroBlue Bluetooth Stack.* BeeCon GmbH, Karlsruhe, 2004.

[Gmb J] GmbH, Avetana (Herausgeber): *avetanaBluetooth JSR82 implementation for Windows, MacOS X and Linux Systems.* Avetana GmbH, o. J. `http://www.avetana-gmbh.de/avetana-gmbh/produkte/jsr82.eng.xml`. (07. Januar 2008).

[Har04] Harbaum, Till: *Microblue Manual. A bluetooth stack for embedded devices.* BeeCon GmbH, Karlsruhe, 2004.

[LDB03] Leopold, Martin, Mads Bondo Dydensborg und Philippe Bonnet: *Bluetooth and sensor networks: a reality check.* In: *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, Seiten 103–113, New York, NY, USA, 2003. ACM.

[Lia99] Liang, Sheng: *Java Native Interface: Programmer's Guide and Specification.* Prentice Hall, 1999.

[Mat05] Mattern, Friedemann: *Die technische Basis für das Internet der Dinge.* In: Fleisch, Elgar und Friedemann Mattern (Herausgeber): *Das Internet der Dinge – Ubiquitous Computing und RFID in der Praxis*, Seiten 39–66. Springer-Verlag, 2005.

[Met99] Mettala, Riku: *Bluetooth Protocol Architecture, Version 1.0.* Bluetooth SIG, Bellevue, 1999.

[Mic94] Microsystems, Sun (Herausgeber): *J2ME Connected Limited Device Configuration (CLDC); JSR 30, JSR 139 Overview.* Sun Microsystems, 1994. `http://java.sun.com/products/cldc/overview.html`. (06. Februar 2008).

[Mic00a] Microsystems, Sun (Herausgeber): *Connected, Limited Device Configuration. Specification Version 1.0a. Java 2 Platform Micro Edition.* Sun Microsystems, Palo Alto, 2000.

[Mic00b] Microsystems, Sun (Herausgeber): *Mobile Information Device Profile (JSR-37). JCP Specification. Java 2 Platform, Micro Edition, 1.0a.* Sun Microsystems, Palo Alto, 2000.

[Mic01] Microsystems, Sun (Herausgeber): *Jini Architecture Specification, Version 1.2.* Sun Microsystems, Palo Alto, December 2001. `http://www.sun.com/software/jini/specs/jini1.2html/jini-title.html`. (06. Februar 2008).

[Mic05] Microsystems, Sun (Herausgeber): *Java APIs for Bluetooth Wireless Technology (JSR 82). Specification Version 1.1. Java 2 Platform, Micro Edition.* Sun Microsystems, Palo Alto, 2005.

[Mic06a] Microsystems, Sun (Herausgeber): *CLDC Library API Specification.* Sun Microsystems, Palo Alto, 2006. `http://java.sun.com/javame/reference/apis/jsr030/`. (07. Januar 2008).

[Mic06b] Microsystems, Sun (Herausgeber): *JSR 82 Bluetooth API and OBEX API.* Sun Microsystems, Palo Alto, 2006. http://java.sun.com/javame/reference/apis/jsr082/. (07. Januar 2008).

[Mic06c] Microsystems, Sun (Herausgeber): *MID Profile.* Sun Microsystems, Palo Alto, 2006. http://java.sun.com/javame/reference/apis/jsr037/. (07. Januar 2008).

[Mic J] Microsystems, Sun (Herausgeber): *Welcome to phoneME.* Sun Microsystems, o. J. https://phoneme.dev.java.net/. (07. Januar 2008).

[SCL+05] Shnayder, Victor, Bor-rong Chen, Konrad Lorincz, Thaddeus R. F. Fulford-Jones und Matt Welsh: *Sensor Networks for Medical Care.* Technischer Bericht, Harvard University Technical Report TR-08-05, 2005.

[SEF07] Saballus, Bjoern, Johannes Eickhold und Thomas Fuhrmann: *Towards a Distributed Java VM in Sensor Networks using Scalable Source Routing.* In: *6. Fachgespraech Sensornetzwerke der GI/ITG Fachgruppe "Kommunikation und Verteilte Systeme"*, Seiten 47–50, Aachen, Germany, 2007.

[SIG01a] SIG, Bluetooth (Herausgeber): *HCI UART Transport.* Bluetooth SIG, Bellevue, 2001.

[SIG01b] SIG, Bluetooth (Herausgeber): *Serial Port Profile (SPP).* Bluetooth SIG, Bellevue, 2001.

[SIG03] SIG, Bluetooth (Herausgeber): *RFCOMM with TS 07.10.* Bluetooth SIG, Bellevue, 2003.

[SIG04] SIG, Bluetooth (Herausgeber): *Bluetooth Specification Version 2.0 + EDR.* Bluetooth SIG, Bellevue, 2004.

[Sou J] SourceForge.net (Herausgeber): *Avetana Bluetooth JSR-82 implementation.* Avetana GmbH, o. J. http://sourceforge.net/projects/avetanabt/. (07. Januar 2008).

[SST98] Schader, Martin und Lars Schmidt-Thieme: *Java. Einführung in die objektorientierte Programmierung.* Springer, Berlin, 1998.

[Sta07] Stallman, Richard M.: *GNU General Public license.* Free Software Foundation, 2007. http://www.gnu.org/copyleft/gpl.html. (06. Februar 2008).

[ZBHK07] Zitterbart, Martina, Erik-Oliver Blass, Bernhard Hurler und Andreas Kuntz: *Vorlesungsfolien Drahtlose Sensor-Aktor Netze Wintersemester 2007/2008.* University of Karlsruhe, Telematics Institute, 2007.

Index